.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 1

Fecha de entrega: 24/11/2023

- Gauler, Ian Benjamin, Padrón: 109437
- Jauregui, Melina Belén, Padrón: 109524
- Tourne Passarino, Patricio, Padrón: 108725

Índice

1 Introducción **3**
1.1 Información de la problemática 3

2 Análisis de la complejidad del problema **3**

3 Hitting-Set Problem: Un Problema NP-Completo **4**
3.1 Reducción Vertex Cover a Dominating-Set Problem 4
3.1.1 Ejemplos: 5
3.2 Reducción Dominating-Set a Hitting-Set 5
3.2.1 Ejemplos: 6
3.3 Reducción Set-Cover a Hitting-Set 7
3.3.1 Ejemplos: 8

4 Solución por backtracking **8**

5 Solución por Programación Lineal **9**
5.1 Programación Lineal Entera 9
5.2 Aproximación por Programación Lineal Continua 10

6 Aproximación por Greedy **11**
6.0.1 Máximo por grupo 12
6.0.2 Máximo global con recálculo 12

7 Ejemplos **13**

8 Mediciones **16**

9 Conclusiones **17**

1. Introducción

El propósito de este informe radica en llevar a cabo un análisis de los distintos enfoques y diseños aplicados para abordar la problemática presentada por Scaloni. Este desafío se centra en la determinación del conjunto mínimo de jugadores, denotado como C, requerido para satisfacer las demandas de cada medio, asegurando al menos la presencia de un jugador favorito por cada uno. Es importante destacar que este problema se enmarca como un caso específico del 'Hitting Set Problem'. En consecuencia, a lo largo de este informe, se realizará una análisis de múltiples soluciones con el fin de resolver este último.

1.1. Información de la problemática

El enunciado propuesto, nos da la siguiente información:

- Scaloni tiene a su disposición el conjunto A de $n = 43$ jugadores $a_1, a_2, \ldots, a_{43}$.
- Existen m medios, cada uno con un grupo de jugadores favorito $B_1, B_2, \ldots, B_m$, ($B_i \in A \forall i$)
- Se quiere el subconjunto $C \in A$ de menor tamaño tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i$)
- Scaloni necesita obtener el grupo C más pequeño de jugadores de tal forma que cada medio B_i tenga al menos un jugador en el equipo.

2. Análisis de la complejidad del problema

El problema planteado por Scaloni es un caso específico del problema del Hitting-Set, cuya definición formal se presenta de la siguiente manera: dado un conjunto A compuesto por n elementos y m subconjuntos $B_1, B_2, \ldots, B_m$ pertenecientes a A ($B_i \subseteq A \forall i \in \mathbb{N}_m$), se busca encontrar un conjunto $C \subseteq A$ tal que para cada subconjunto B_j, donde $B_j \subseteq A / \forall j \in \mathbb{N}_m, C \cap B_j \neq \emptyset$.

Además de su formulación básica, el problema del Hitting-Set también presenta una versión de decisión: dada la colección de un conjunto A con n elementos y m subconjuntos $B_1, B_2, \ldots, B_m$ ($B_i \subseteq A$ para todo i), junto con un parámetro numérico k, se plantea el interrogante sobre la existencia de un conjunto $C \subseteq A$ que cumpla con dos condiciones fundamentales:

- En primer lugar, que la cardinalidad de C sea menor o igual a k ($|C| \leq k$)
- En segundo lugar, que para cada subconjunto B_j, la intersección entre C y B_j no sea vacía ($C \cap B_j \neq \emptyset$) para todo j perteneciente al conjunto de números naturales hasta m.

Para realizar esta verificación, se llevan a cabo dos operaciones clave que definen la complejidad del proceso. La primera operación, relacionada con la verificación de la cardinalidad de C ($|C| \leq k$), tiene una complejidad constante $O(1)$, ya que implica simplemente obtener el número de elementos en C y verificar que $|C| \leq k$. La segunda operación, que implica verificar la inclusión de al menos un elemento de C en cada conjunto B_j, presenta una complejidad de $O(n \times m)$. Esto se debe a que para cada conjunto B_j (j perteneciente al conjunto de números naturales hasta m) (operación $O(m)$), se debe recorrer, en el peor de los casos, todo el conjunto C ($O(n)$) para verificar la pertenencia de al menos un elemento de este en B_j (operación con complejidad $O(1)$ si tanto B_j como C se implementan como un 'set').

De esta manera, el problema del Hitting-Set se sitúa en la clase de complejidad NP debido a su capacidad de verificación en tiempo polinomial.

3. Hitting-Set Problem: Un Problema NP-Completo

La clasificación del problema del Hitting-Set como NP-Completo se establece mediante la demostración de su reducibilidad polinómica a partir de otros problemas ya catalogados como NP-Completo.

En nuestro análisis, nos hemos propuesto abordar esta demostración de dos maneras distintas, empleando estrategias de reducción que ilustran la naturaleza NP-Completa del Hitting-Set Problem:

- Reducción de Vertex Cover a Dominating-Set Problem → Reducción de Dominating-Set Problem a Hitting-Set Problem.
- Reducción de Set Cover a Hitting-Set Problem.

Es importante destacar que la pertenencia de Vertex Cover y Set Cover a NP-Completo fue demostrada en clases anteriores.

- Vertex Cover [link]
- Set Cover [link]

3.1. Reducción Vertex Cover a Dominating-Set Problem

- Vertex Cover: Dado un grafo $G = (V, E)$, donde V es un conjunto de vértices y E es un conjunto de aristas, un Vertex Cover de G es un conjunto $C \in V$ tal que para cada arista (u, v) en E, al menos uno de los extremos de u o v esrá en C. Es decir, para cada arista en el grafo, al menos uno de sus extremos pertence al conjunto C.
- Dominating Set: Dado un grafo G, se busca un conjunto de vértices C tal que, para todo vértice $v \in G$, este esté contenido en C o existe al menos un vértice en C adyacente de v.

La reducción Vertex-Cover $\leq_p$ Dominating-Set consta en lo siguiente: Dado del grafo G con n vértices y m aristas del problema Vertex Cover, para cada par de vértices adyacentes $v - w$, se agregan ambos al grafo G' (del problema Dominating Set) junto a su arista y se agrega un tercer vértice auxiliar vw, adyacente a los otros dos. A será el conjunto de vértices auxiliares.

Luego, del conjunto V' de k vértices solución de Dominating Set, se debe agregar a V (solución de Vertex Cover) todos los vértices de V' que no estén en el conjunto de auxiliares y, para cada vértice en $V' \cap A$ se agrega a V cualquiera de sus adyacentes. La primera parte de la reducción es $O(m)$ y la segunda es $O(k), k \leq n$, por lo que la complejidad total es $O(m + k)$, que es polinomial.

$$\text{Vertex-Cover} \leq_p \text{Dominating-Set}$$

3.1.1. Ejemplos:

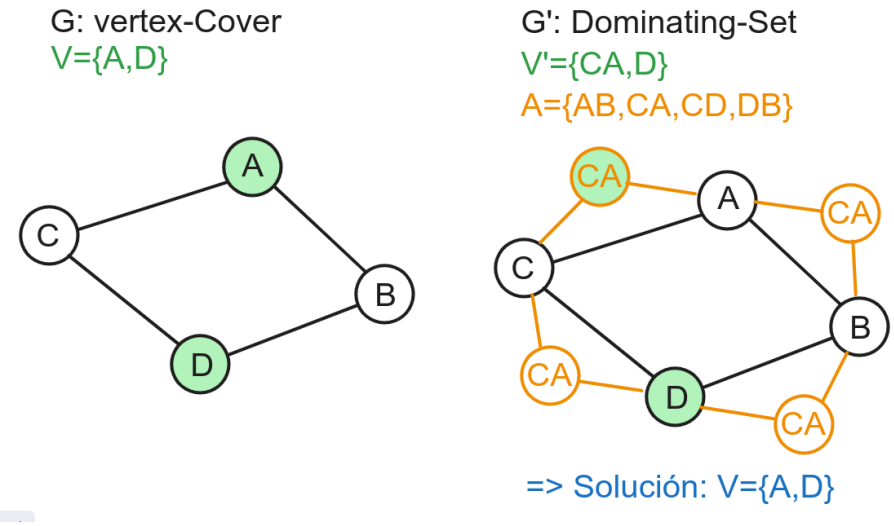

Figura 1: Ejemplo de reducción de Vertex-Cover a Dominating-Set

Notar que el conjunto solución de Dominating-Set $V' = \{CA, D\}$ se incluye un vértice que pertenece al conjunto de auxiliares $CA \in A$. Por ello, se debe agregar a V cualquiera de sus adyacentes, en este caso D.

Finalmente:

$$\begin{matrix}\text{Vertex-Cover} \leq_p \text{Dominating-Set} \\ \text{Dominating-Set} \leq_p \text{Hitting-Set}\end{matrix} \xRightarrow{\text{por transitividad}} \text{Vertex-Cover} \leq_p \text{Hitting-Set}$$

$$\Longrightarrow \text{Hitting-Set} \in \text{NP-Completo}$$

3.2. Reducción Dominating-Set a Hitting-Set

La reducción Dominating-Set $\leq_p$ Hitting-Set consta en lo siguiente: Dado un grafo G de n vértices del problema Dominating Set, para cada vértice v_i, se construye un grupo B_i con él mismo y todos sus vértices adyacentes. Esto tiene un costo temporal de $O(N \times E)$, ya que para cada vértice v_i se deben obtener los vértices adyacentes a v_i y crear los conjuntos B_i por cada uno de ellos (es decir, realizamos operaciones polinomiales para transformar nuestro problema de Dominating-Set a Hitting-Set).

A su vez, esta reducción se caracteriza por ser de equivalencia simple ya que el resultado devuelto por el algoritmo de Hitting Set coincide directamente con los resultados esperados por el algoritmo de Dominating Set. Como consecuencia, no es necesario realizar operaciones adicionales o suplementarias para obtener el resultado deseado.

La complejidad del mismo se formula a continuación: La construcción de los m conjuntos B_i es $O(N \times E)$. Ya que para cada vértice v_i se deben obtener los vértices adyacentes a v_i y crear los conjuntos B_i por cada uno de ellos.

$$\text{Dominating-Set} \leq_p \text{Hitting-Set}$$

3.2.1. Ejemplos:

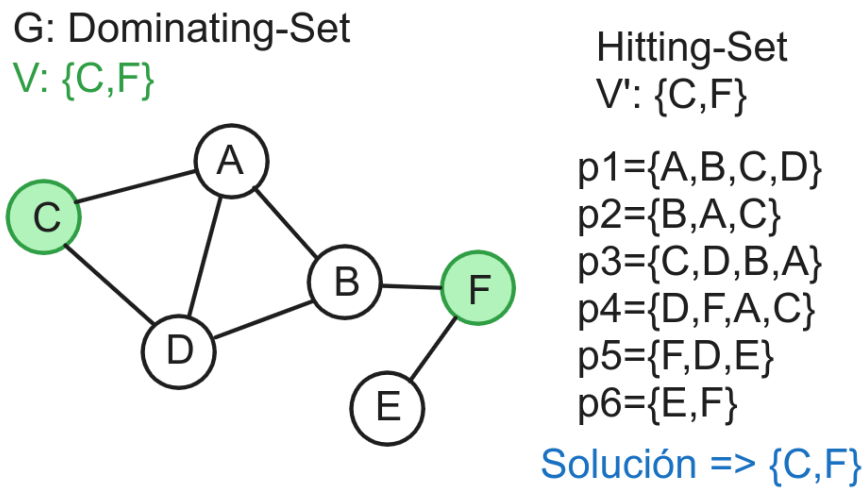

Figura 2: Ejemplo 1 de reducción de Dominating-Set a Hitting Set

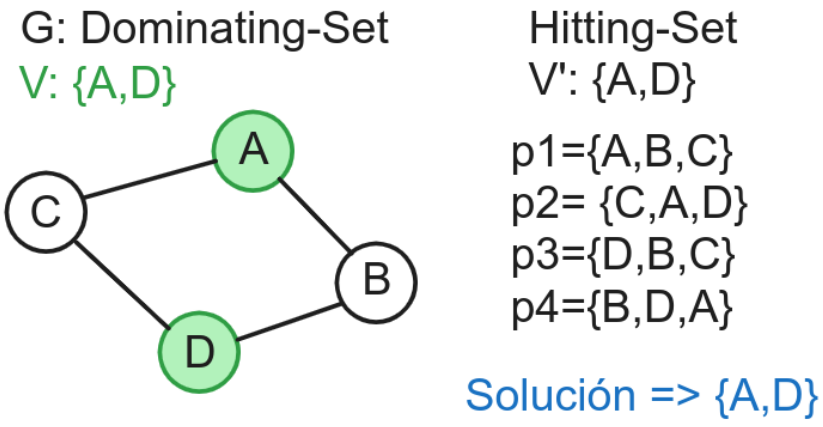

Figura 3: Ejemplo 2 de reducción de Dominating-Set a Hitting Set

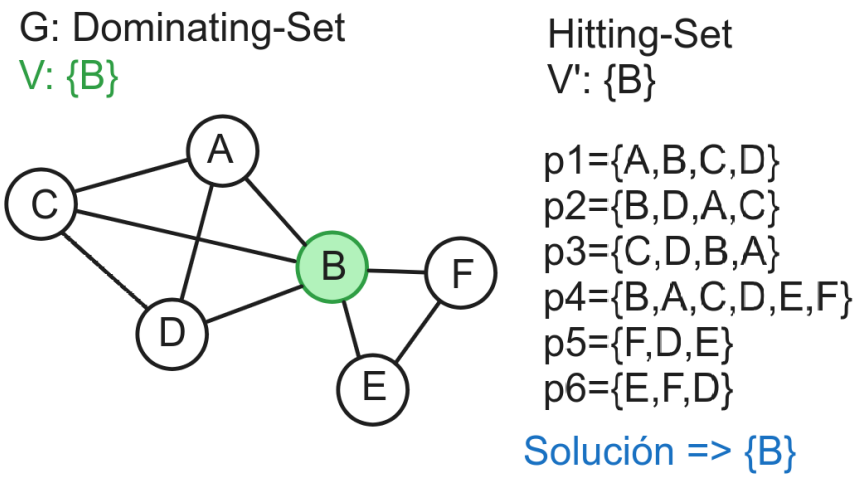

Figura 4: Ejemplo 3 de reducción de Dominating-Set a Hitting Set

En los tres casos analizados, se evidencia que la solución derivada del problema Hitting-Set representa la solución óptima para el Dominating-Set.

3.3. Reducción Set-Cover a Hitting-Set

- Set-Cover: Dado un conjunto finito U y una familia $S = S_1, S_2, ..., S_n$ de subconjuntos de U, el problema de Set Cover consiste en identificar el menor número de conjuntos cuya unión aun contiene todos los elementos del universo.

La reducción Set-Cover $\leq_p$ Hitting-Set consta en lo siguiente: Dado un grafo G de n vértices del problema Set-Cover, para cada elemento del subconjunto S_i, se construye un grupo E_i en el cual sus elementos van a ser cada S_i en el cual está incluido.

Esto tiene un costo temporal de $O(N \times U)$ (siendo N la cantidad de subconjuntos S y U la cantidad de elementos del universo), ya que para cada subconjunto S_i se debe recorrer sus elementos (en el peor de los casos todos tienen la misma cantidad que el universo) y crear los conjuntos E_i (en caso de que no esté ya creado). En cada subconjunto nuevo E_i se le agrega el S_i donde está incluido, siendo esta una operación constante.

El análisis del costo temporal asociado al problema del Set Cover se establece en O(N×U), donde N representa la cantidad de subconjuntos en S y U la cantidad de elementos presentes en el universo. Este costo se deriva de la necesidad de recorrer cada subconjunto S_i en la familia S y sus elementos correspondientes, operación que, en el peor de los casos, implica recorrer conjuntos de igual tamaño al universo.

Para cada subconjunto S_i, se realiza la creación de conjuntos E_i si no están previamente creados. En el proceso de generación de cada conjunto E_i, se añade el subconjunto S_i donde el elemento está incluido, siendo esta una operación de complejidad constante.

De esta manera, realizamos operaciones polinomiales para transformar nuestro problema de Set-Cover a Hitting-Set. Además, al igual que la reducción de Dominating Set a Hitting Set, esta reducción se caracteriza por ser de equivalencia simple ya que el resultado devuelto por el algoritmo de Hitting Set coincide directamente con los resultados esperados por el algoritmo de Set-Cover. Como consecuencia de esta correspondencia entre los resultados, no es necesario realizar operaciones adicionales o suplementarias para obtener el resultado deseado.

$$\text{Dominating-Set} \leq_p \text{Hitting-Set}$$

3.3.1. Ejemplos:

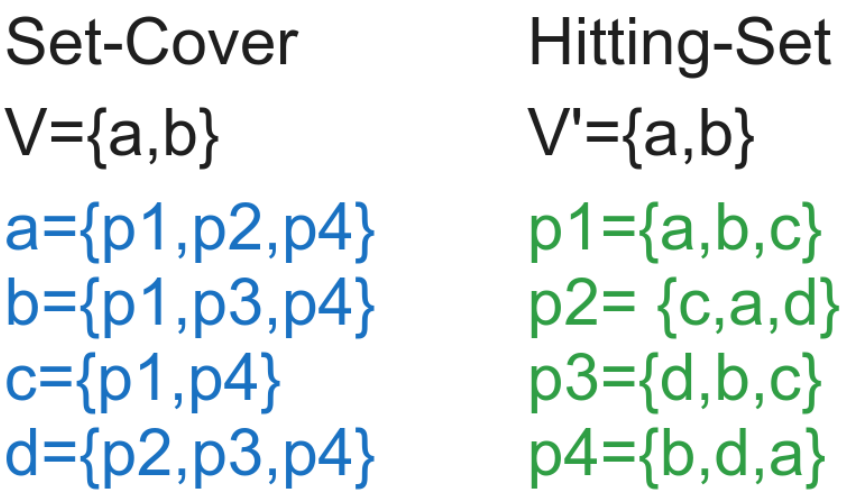

Figura 5: Ejemplo 1 de reducción de Set-Cover a Hitting Set

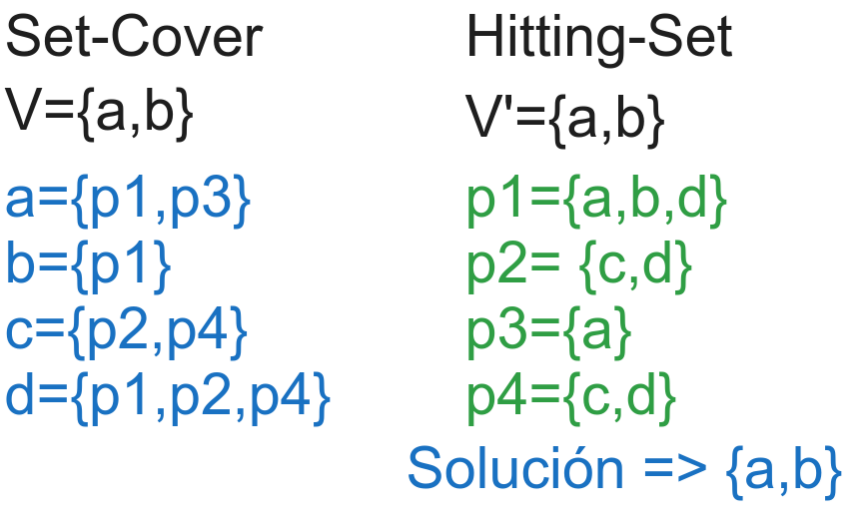

Figura 6: Ejemplo 2 de reducción de Set-Cover a Hitting Set

Al igual que la reducción de Dominating-Set a Hitting-Set, en los dos casos analizados, se evidencia que la solución derivada del problema Hitting-Set representa la solución óptima para el problema de Set-Cover.

4. Solución por backtracking

Para poder plantear el álgoritmo de backtracing procedimos a seguir el esquema proporcionado por la cátedra:

1. Pruebo si la solucion parcial es solución y es mejor que la actual entontrada:

   a) Si lo es, actualizo la solución actual y la devuelvo
   b) Si no lo es, avanzo si puedo
2. Pruebo si la solución parcial es válida
   a) Si no lo es, retrocedo y vuelvo a 3)
   b) Si lo es, llamo recursivamente y vuelvo a 1)
3. Prosigo explorando la solución parcial llamando recursivamente

Para la solución por backtracking consideramos relevantes las siguientes variables:

- Los subconjuntos B_i
- Solución parcial
- La solución actual óptima
- El índice (i_b) del subconjunto B_i en cuestión.

En cada instancia recursiva del algoritmo, se inicia con la evaluación de la compatibilidad de la solución parcial con el problema de Hitting-Set. En caso de ser compatible y si su cardinalidad es inferior a la de la solución actual, esta solución parcial se considera la nueva solución óptima (actual).

Por otro lado, si la solución parcial no es compatible con el problema planteado, se procede a descartarla únicamente si su cardinalidad es mayor o igual que el de la solución actual. En este escenario, se implementa la técnica de backtracking, regresando al llamado anterior y desechando la exploración de esta rama de solución.

En el caso específico en el que la solución parcial no es compatible pero aún puede representar una solución óptima potencial, se lleva a cabo una verificación adicional. Se examina el subconjunto B_i (donde $i = i_b$) para garantizar la inclusión de al menos un jugador en la solución final. Si este requisito no se cumple, se procede a evaluar individualmente la inclusión de cada jugador presente en B_i para determinar cuál de ellos conduce a la obtención de la solución óptima.

```
def backtracking_HSP(b_array: list, sol_parcial: set, sol_actual, i_b, propuestos: set):
    if i_b >= len(b_array):
        if sol_actual == None or len(sol_parcial) < len(sol_actual):
            return sol_parcial.copy()
        return sol_actual

    if sol_actual != None and len(sol_parcial) >= len(sol_actual):
        return sol_actual

    if esta_incluido(b_array[i_b], sol_parcial):
        return backtracking_HSP(b_array, sol_parcial, sol_actual, i_b+1, propuestos)

    propuestos_b_array = set()

    for jugador in b_array[i_b]:
        if jugador in propuestos:
```

5. Solución por Programación Lineal

5.1. Programación Lineal Entera

También se puede reducir Hitting-Set Problem a Programación Lineal de la siguiente manera:

1. Para cada elemento i del universo U, se crea una variable y_i que puede tomar los valores 0 o 1.
2. Se define una función $f_j = \sum_{e \in S_j} e$ para cada conjunto S_j, y se establece una restricción que acota a f_j en el rango $1 \leq f_j \leq |S_j|$.
3. El objetivo radica en minimizar la función $f = \sum_{i=0}^{n} y_i$.

Luego, el conjunto solución C será

$$C = \{e_i \in U \mid y_i = 1\}$$

La complejidad de esta reducción es $O(N \times U)$ ya que se debe recorrer todos los subconjuntos B_i y, por cada uno, recorrer cada elemento. Este proceso es polinomial, y la complejidad de Programación Lineal Entera es exponencial.

El codigo de la reducción es el siguiente:

```python
def hitting_set_pl_entera(b_array):
    problem = pulp.LpProblem("Scaloni", pulp.LpMinimize)
    dic_jugadores = {}

    for periodista in range(len(b_array)):
        set_p = set()
        for jugador in b_array[periodista]:
            y = None
            if jugador in dic_jugadores:
                y = dic_jugadores[jugador]
            else:
                y = pulp.LpVariable(f"y_{jugador}", cat=pulp.LpBinary)
                dic_jugadores[jugador] = y
            set_p.add(y)

        p = pulp.lpSum(set_p)
        problem += p >= 1
        problem += p <= len(set_p)

    z = pulp.lpSum(dic_jugadores.values())
    problem += z

    problem.solve()

    # Obtener jugadores seleccionados
    jugadores_seleccionados = [
        jugador for jugador, valor in dic_jugadores.items() if pulp.value(valor) > 0]

    return jugadores_seleccionados
```

5.2. Aproximación por Programación Lineal Continua

Otra estrategia para abordar la solución del Hitting-Set Problem implica el enfoque de la Programación Lineal continua. Esta aproximación se asemeja a la reducción previamente descrita, con la distinción fundamental de que las variables y_i pueden asumir valores reales dentro del intervalo 0 a 1.

Posteriormente, se lleva a cabo una consideración respecto a estos valores, donde aquellos que excedan el umbral del $1/b$, siendo b el cardinal máximo de todos los conjuntos S_j, son incluidos como componentes de la solución final.

Luego, el conjunto solución C será

$$C = \left\{ e_i \in U \mid y_i \geq \frac{1}{2} \right\}$$

Por la mismas consideraciones aplicadas en la reducción de la Programación Lineal Entera, la complejidad asociada a la reducción de la Programación Lineal Continua al Hitting Set es de O(N×U). Sin embargo, la complejidad de Programación Lineal Continua es $O(n^9)$ lo cual es polinomial, a diferencia de la Entera.

El codigo de la reducción es el siguiente:

```python
def hitting_set_pl_entera(b_array):
    problem = pulp.LpProblem("Scaloni", pulp.LpMinimize)
    dic_jugadores = {}

    for periodista in range(len(b_array)):
        set_p = set()
        for jugador in b_array[periodista]:
            y = None
            if jugador in dic_jugadores:
                y = dic_jugadores[jugador]
            else:
                y = pulp.LpVariable(f"y_{jugador}", cat=pulp.LpBinary)
                dic_jugadores[jugador] = y
            set_p.add(y)

        p = pulp.lpSum(set_p)
        problem += p >= 1
        problem += p <= len(set_p)

    z = pulp.lpSum(dic_jugadores.values())
    problem += z

    problem.solve()

    # Obtener jugadores seleccionados
    jugadores_seleccionados = [
        jugador for jugador, valor in dic_jugadores.items() if pulp.value(valor) > 0]

    return jugadores_seleccionados
```

Con el objetivo de evaluar qué tan buena aproximación es la Programación Lineal Continua respecto a la Programación Lineal Entera, se propuso el siguiente escenario que resalta las diferencias entre las soluciones obtenidas por ambos algorítmos:

Consideramos dos conjuntos que representan medios diferentes, denominados set1 y set2. Estos conjuntos se componen de la siguiente manera:

1. set1 con tamaño b, expresado como $set_1 = x_1, x_2, ..., x_{b-1}, x_b$
2. set2 con tamaño $b-1$, representado como $set_2 = x_1, x_2, ..., x_{b-1}$

En un contexto de Programación Lineal Continua (PLC), una solución posible puede ser la asignación $x_1 = x_2 = ... = x_{b-1} = 1/(b-1)$ y $x_b = 0$.

En esta situación, dado que $\frac{1}{b} < \frac{1}{b-1}$, las variables del set_2, al ser redondeadas, resultarían en 1, siendo todas incluidas en la solución final.

Por otro lado, la solución proporcionada por la Programación Lineal Entera (PLE), que representa la solución óptima, presentaría, por ejemplo, la siguiente asignación de valores: $x_1 = 1$ y $x_2 = ... = x_{b-1} = 0$.

No obstante, al obtenerse una aproximación mediante PLC que establece $x_1 = x_2 = ... = x_{b-1} = 1$ se deriva la siguiente relación de aproximación: $r(A) = \frac{A(I)}{z(I)} = \frac{b-1}{1}$

En consecuencia, $b-1$, se establece como una cota superior de la aproximación.

6. Aproximación por Greedy

Proponemos dos aproximaciones extra.

6.0.1. Máximo por grupo

Este enfoque realiza un cálculo de la frecuencia con la que cada jugador aparece en los m subconjuntos B. Posteriormente, procede a recorrer cada uno de estos subconjuntos y, si la solución parcial propuesta no incluye ningún jugador presente en un subconjunto dado, se incluye en esta solución al jugador que tiene la mayor frecuencia de apariciones dentro de dicho subconjunto.

Este algorítmo sigue la técnica de diseño greedy al enfocarse en optimizar la solución global mediante la búsqueda de óptimos locales en cada subconjunto. Se prioriza la inclusión del jugador con la mayor frecuencia de aparición en situaciones donde la solución parcial no incluye a ningún jugador del subconjunto evaluado. Este método busca mejorar la solución global al introducir jugadores de acuerdo con su frecuencia de aparición en los subconjuntos, minimizando así la cobertura de elementos dentro de la solución propuesta.

```
    return: solucion
    """
    # contamos la cantidad de apariciones de cada jugador por cada prensa
    apariciones = {}
    for subconjunto in subconjuntos:  # O(len(subconjuntos)*len(subconjunto))
        for jugador in subconjunto:
            if jugador in apariciones:
                apariciones[jugador] += 1
            else:
                apariciones[jugador] = 1

    # ordenamos los subconjuntos por cantidad de apariciones de cada jugador

    # obtenemos la solucion mediante el optimo local
    solucion = set()
    for subconjunto in subconjuntos:  # O(len(subconjuntos)*len(subconjunto))
        aparicion_max = None
        for jugador in subconjunto:
            if jugador in solucion:
                aparicion_max = jugador
                break

            else:
                if aparicion_max is None or apariciones[jugador] > apariciones[
    aparicion_max]:
                    aparicion_max = jugador

        solucion.add(aparicion_max)

    return solucion

def aproximacion_greedy_maximo_global_con_recalculo(subconjuntos: list):
    """
    Obtiene la solucion por greedy.
```

La primera operación tiene complejidad $O(k \times m)$, con k el promedio de jugadores por subconjunto ($k \leq n$). Con grupos más chicos, el algoritmo tiende a lineal ($O(m)$) y con grupos más grandes, a $O(m \times n)$. De la misma manera, la segunda operación tiene complejidad de $O(m \times n)$ ya que por cada subconjunto ($O(k)$) se deben recorrer todos sus jugadores para verificar si efectivamente están en la solución final ($O(m)$) y, en caso de que no esté ninguno, se debe encontrar el jugador con máximas apariciones $O(m)$. Entonces, la complejidad total es $O(2(k \times m)) = O(k \times m)$ y resulta polinomial.

6.0.2. Máximo global con recálculo

La ténica de diseño implementada en este algorítmo, al igual que el anterior explicado, es greedy ya que busca el óptimo local al optimizar la solución por cada subconjunto, priorizando la inclusión del jugador con la mayor frecuencia hasta que se cumpla la cobertura de por lo menos un jugador por conjuntos.

La segunda estrategia inicia de manera similar, llevando a cabo el cálculo de la frecuencia de

aparición de cada jugador en los diferentes subconjuntos. Posteriormente, se procede a añadir a la solución el jugador que presenta la mayor cantidad de apariciones entre todos, y quita las apariciones de los subconjuntos que ya cubre al resto de los jugadores. Este proceso se repite sucesivamente hasta quedarse sin jugadores restantes o hasta que el jugador con la mayor frecuencia encontrada no tenga más apariciones restantes.

La técnica de diseño empleada en este algoritmo, al igual que el anterior explicado, se caracteriza por ser greedy, ya que busca optimizar la solución al priorizar la inclusión del jugador con la mayor frecuencia en cada iteración, hasta alcanzar la cobertura total de todos los subconjuntos.

```
    return: solucion
    """
    # contamos la cantidad de apariciones de cada jugador por cada prensa
    apariciones = {}
    # O(len(subconjuntos)*len(subconjunto))
    for index, subconjunto in enumerate(subconjuntos):
        for jugador in subconjunto:
            if jugador not in apariciones:
                apariciones[jugador] = set()
            apariciones[jugador].add(index)

    # obtenemos la solucion mediante el optimo local
    solucion = set()

    while len(apariciones) != 0:  # O(len(jugadores)*len(subconjuntos)*len(
    subconjunto))
        jugador, index_subconjuntos = max(apariciones.items(
        ), key=lambda jugador_index_subconjuntos: len(jugador_index_subconjuntos
    [1]))
        apariciones.pop(jugador)
        if len(index_subconjuntos) == 0:
            break
        solucion.add(jugador)

        # O(len(subconjuntos)*len(subconjunto))
        for index_subconjunto in index_subconjuntos:
            for jugador_companiero_de_conjunto in subconjuntos[index_subconjunto]:
                if jugador != jugador_companiero_de_conjunto:
                    apariciones[jugador_companiero_de_conjunto].remove(
                        index_subconjunto)

    return solucion
```

Lla complejidad de la primera operación es $O(k \times m)$. La segunda tiene complejidad $O(j \times g \times m)$, con j la cantidad de jugadores de la solución, $j \leq m$, y g la cantidad promedio de grupos que cubre cada jugador $g \leq k$. Entonces, la complejidad total es $O(k \times m + j \times g \times m)$.

7. Ejemplos

A continuación, compararemos los resultados obtenidos por cada uno de los algoritmos con distintos ejemplos.

El primer ejemplo, ilustrado en la fig. 7, es un caso feliz en el que los tres algoritmos obtienen el resultado óptimo.

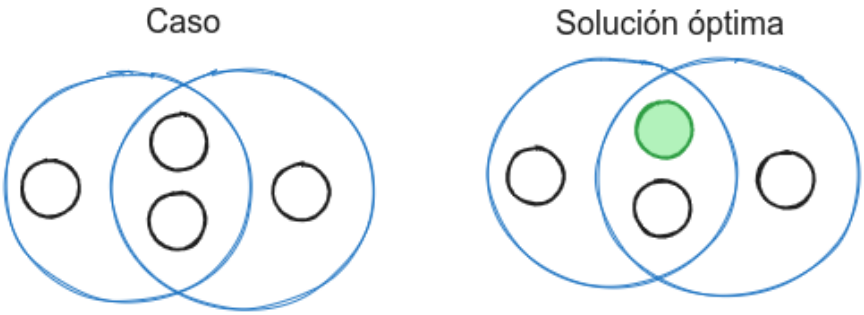

Figura 7: Ejemplo 1

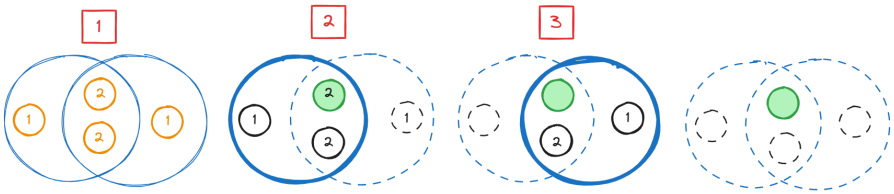

Figura 8: Ejemplo 1 resuelto por Máximo por grupo

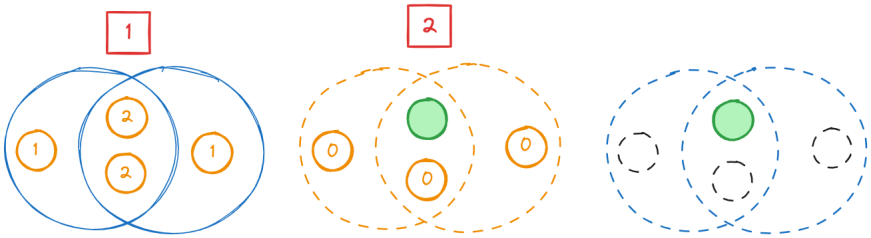

Figura 9: Ejemplo 1 resuelto por Máximo global con recálculo

El segundo ejemplo (fig. 10) podemos observar en la fig. 11 que "Máximo por grupo", de menor complejidad, no encuentra la solución óptima, pero el "Máximo global con recálculo", en la fig. 12, si.

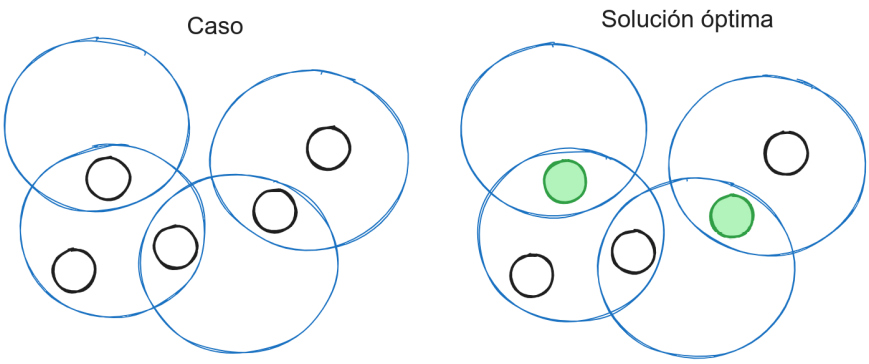

Figura 10: Ejemplo 2

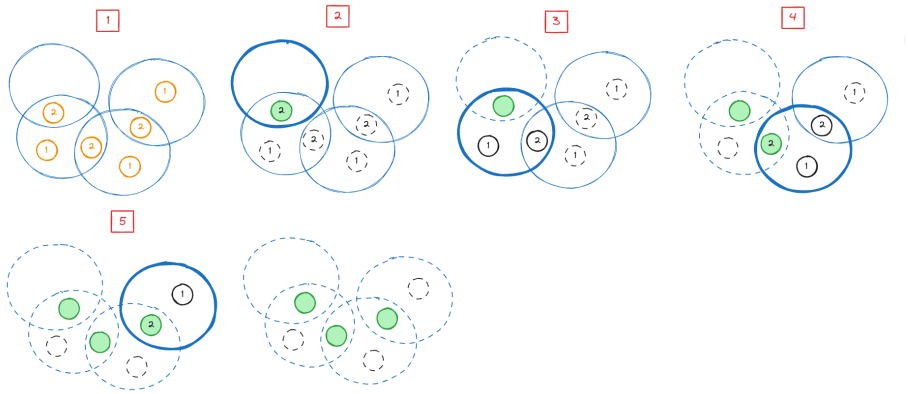

Figura 11: Ejemplo 2 resuelto por Máximo por grupo

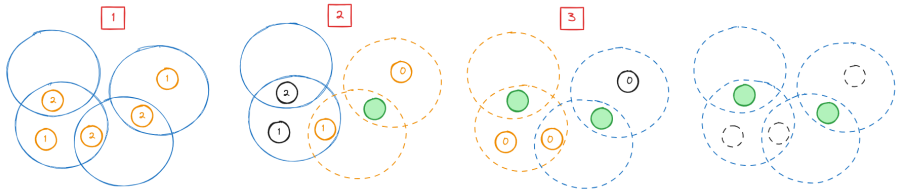

Figura 12: Ejemplo 2 resuelto por Máximo global con recálculo

Por último, en el ejemplo de la fig. 13, tanto "Máximo por grupo"(fig. 14) como "Máximo global con recálculo"(fig. 15) no llegan al la solución óptima.

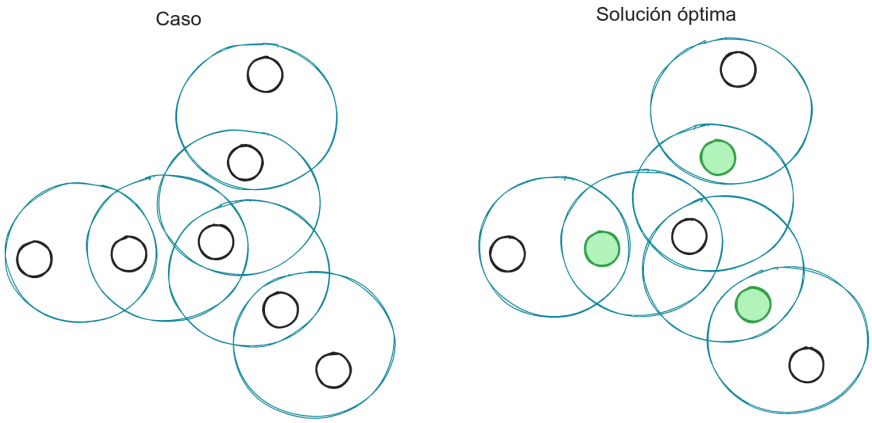

Figura 13: Ejemplo 3

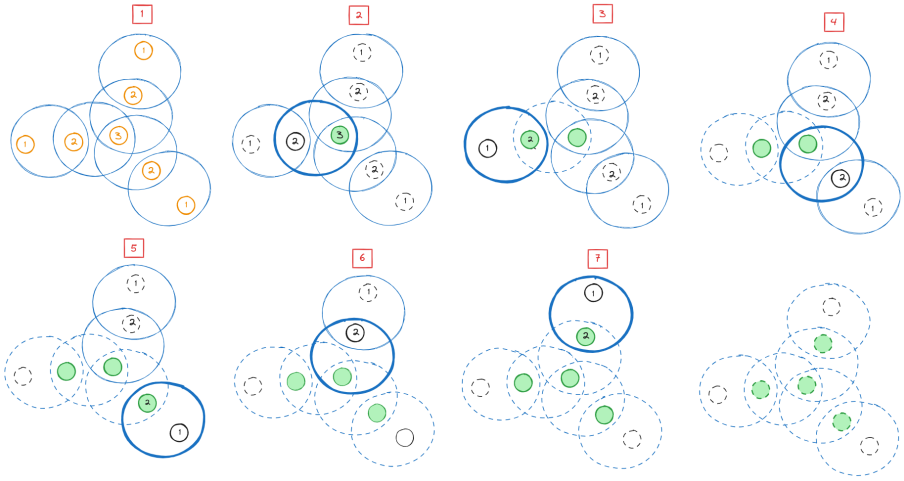

Figura 14: Ejemplo 3 resuelto por Máximo por grupo

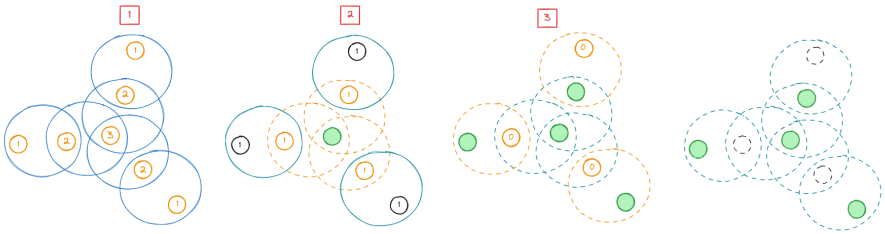

Figura 15: Ejemplo 3 resuelto por Máximo global con recálculo

8. Mediciones

Para las mediciones, creamos listas de compilados de ejemplo de forma aleatoria. Decidimos que el tiempo de análisis de un compilado fuera un valor aleatorio entre 1 y 99'999, basándonos en los datos de ejemplo provistos por la cátedra.

Medimos el tiempo de ejecución de nuestro algoritmo para ciertas cantidades de compilados.

Notamos que los procesos de fondo del sistema nos generaba distorciones en los tiempos de ejecución de una misma muestra. Atacamos este problema realizando varias mediciones para el mismo escenario de compilados y tomando el promedio de los tiempos obtenidos.

Además, como definimos cada escenario arbitrariamente, en una cantidad de compilados x, este escenario puede ser poco favorable (puede estar muy desordenado), mientras que en una cantidad de compilados $x + 1$ puede ser muy favorable. Por lo cual la diferencia de tiempos va a ser muy grande a pesar de que la cantidad de compilados solamente difiere en una unidad (TimSort es una combinación de inserción y mergeSort, por lo cual el desorden inicial toma gran relevancia en el rendimiento final del algorítmo). Para solucionar esto, decidimos hacer varios escenarios distintos para la misma cantidad de compilados.

Con el objetivo de comparar los tiempos de ejecución de nuestro algoritmo con la complejidad teórica, optamos por realizar tanto un análisis de regresión lineal como uno de regresión lineal logarítmica que se ajustara a nuestros datos. Para evaluar qué curva se ajusta mejor, usamos la raíz del error cuadrático medio (RMSE). Realizamos este análisis en un intervalo con tamaños pequeños (hasta 1'000) y en un intervalo con tamaños grandes (hasta 10'000). Ver figuras **??** y **??**

Se puede observar que en valores pequeños, el tiempo de ejecución del algoritmo sigue una clara

curva lineal logarítmica. Sin embargo, a medida que aumenta el tamaño de la muestra de compilados, su comportamiento se aproxima más a una complejidad lineal.

Esto ocurre debido a la naturaleza de la función lineal logarítmica, que presenta una pendiente logarítmica. Cuando se trabajan con valores grandes, esta pendiente se vuelve casi constante, lo que la asemeja a una función lineal. Esto se debe a que el logaritmo es una función monótona creciente, lo que significa que su derivada es siempre positiva para valores mayores que cero. Sin embargo, el crecimiento de esta función se vuelve cada vez más lento a medida que los valores aumentan, indicando que su segunda derivada es siempre negativa. En consecuencia, para valores muy grandes, la función lineal logarítmica se estabiliza y, aunque no es completamente constante, su comportamiento se asemeja a una función lineal.

De esta manera, mediante el gráfico y la validación a través del error cuadrático medio, hemos comprobado que la curva lineal logarítmica se adapta de manera más precisa a nuestros datos.

De esta forma pudimos comprobar empíricamente que la complejidad tiende a $O(n \log n)$.

Nótese que graficamos también dos curvas que demarcan una estimación de los cuantiles verticales 0,1 y 0,9. Esta estimación se realizó calculando los cuantiles para un grupo pequeño centrado en cada punto. Estas curvas nos ayudan a dimencionar cómo la varianza del tiempo de ordenamiento crece con el aumento del tamaño de la información de entrada.

Adicionalmente, graficamos la densidad de los tiempos que también brinda una visualización de cómo aumenta la varianza con el aumento del tamaño de los datos de entrada. Ver figuras **??** y **??**

9. Conclusiones

Finalmente, consideramos que la solución más adecuada para abordar la problemática presentada por Scaloni sería a través del empleo de técnicas como Backtracking o Programación Lineal Entera, siempre y cuando el número de medios no sea excesivamente alto.

No obstante, en situaciones que involucren conjuntos de medios considerablemente extensos, las aproximaciones obtenidas a través de algoritmos como los métodos greedy o la Programación Lineal pueden ofrecer una alternativa viable para obtener resultados en tiempos reducidos. Sin embargo, se debe tener en cuenta que, si bien estas soluciones pueden ser efectivas en términos de eficiencia computacional, pueden distanciarse significativamente de la solución óptima. Este fenómeno es particularmente notable en el caso de la Programación Lineal, la cual representa una (b-1)-Aproximación (siendo b el número máximo de elementos en los conjuntos de medios), siendo una diferencia significativa respecto a la solución óptima.