.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3

Fecha de entrega: 24/11/2023

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

Índice

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 $B_1, B_2, \ldots, B_m$, ($B_i \subseteq A \forall i$)
- Se quiere el subconjunto $C \subseteq A$ de menor tamaño tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \emptyset \forall i$).

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 que definen la complejidad del proceso. La primera, 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 ($j \in \mathbb{N}_m$) (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 problema del Hitting-Set como NP-Completo se establece mediante la reducción polinómica de otros problemas ya catalogados como NP-Completo a este. En nuestro análisis, nos hemos propuesto abordar esta demostración de dos maneras diferentes:

- 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 `https://youtu.be/i8qhWUMC5OU?t=5100`
- Set Cover `https://youtu.be/ZvXVKWyBryg?t=8707`

3.1. Reducción Vertex Cover a Hitting-Set Problem

3.1.1. 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 $C \subseteq V$ tal que para cada arista (u, v) en E, al menos uno de los extremos, u o v, pertence al conjunto C.
- Dominating Set: Dado un grafo G, se busca un conjunto mínimo de vértices C tal que, para todo vértice $v \in G$, 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.

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

Ejemplos:

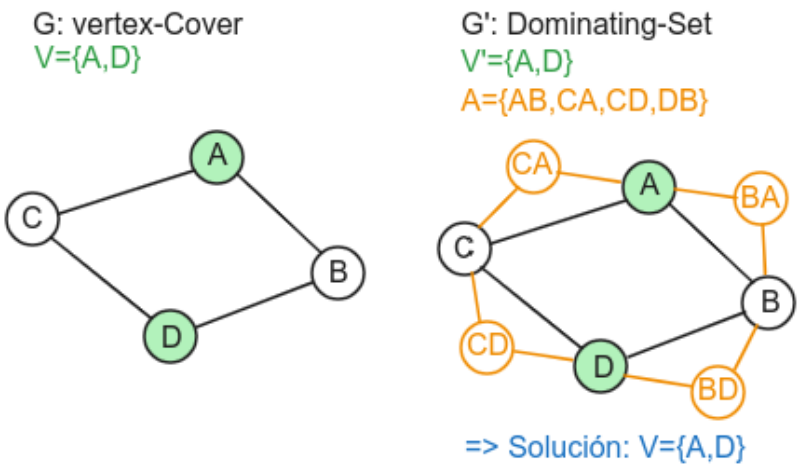

Figura 1: Ejemplo de reducción de Vertex-Cover a Dominating-Set

Si ahora, queremos un Vertex-Cover de hasta 3 vértices ($k = 3$), luego la solución proporcionada por Dominating Set puede ser la siguiente:

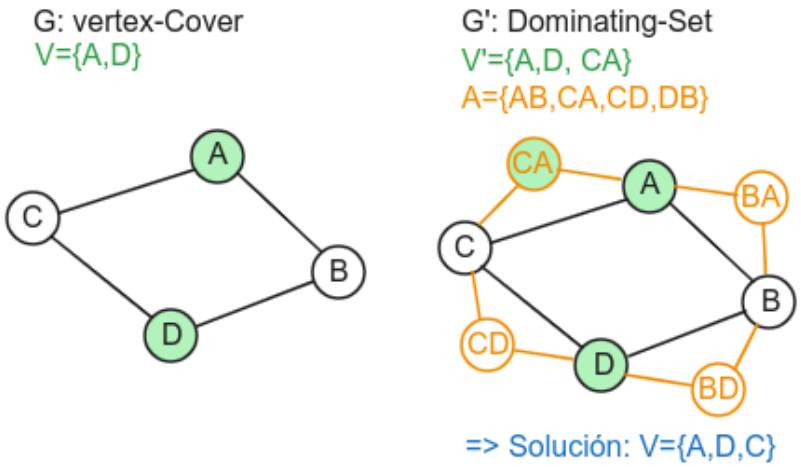

Figura 2: Ejemplo de reducción de Vertex-Cover a Dominating-Set

Notar que el conjunto solución de Dominating-Set $V' = \{CA, A, D\}$ 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 C.

3.1.2. 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. De esta manera, 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.

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

Ejemplos:

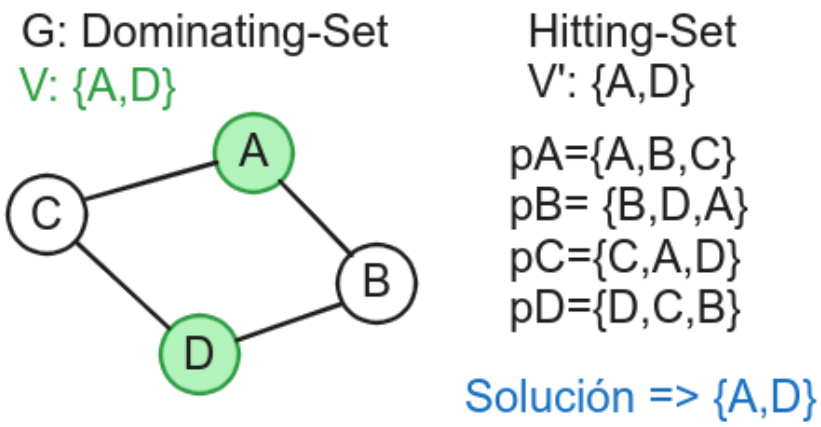

Figura 3: Ejemplo 1 de reducción de Dominating-Set a Hitting Set

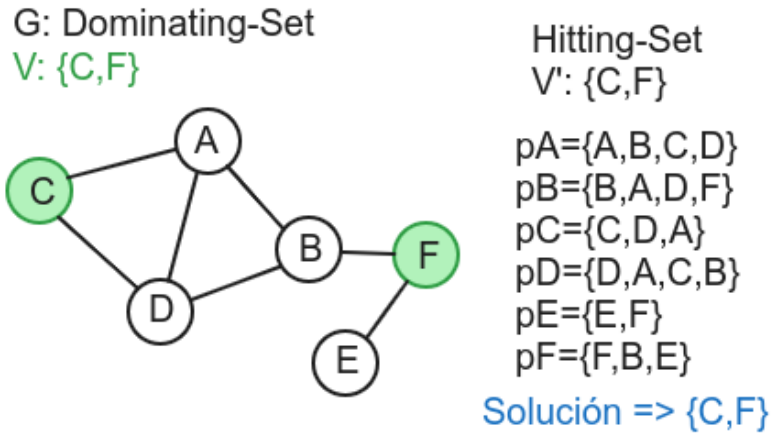

Figura 4: Ejemplo 2 de reducción de Dominating-Set a Hitting Set

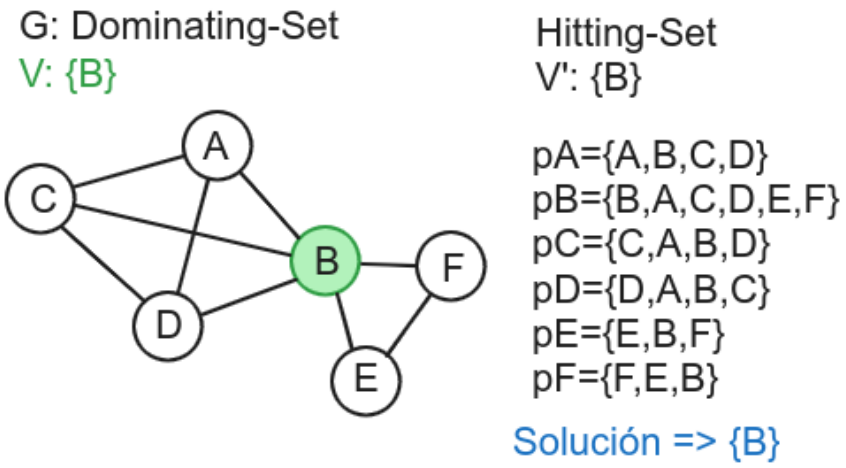

Figura 5: Ejemplo 3 de reducción de Dominating-Set a Hitting Set

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

3.1.3. Reducción Vertex Cover a Hitting-Set Problem

Finalmente:

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

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

3.2. 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 aún contiene todos los elementos del universo.

La reducción Set-Cover $\leq_p$ Hitting-Set consta en lo siguiente: Para cada elemento e_i del universo U del problema Set-Cover, se crea un conjunto E_i en el cual sus elementos van a ser cada S_i en el cual está incluido.

El análisis de la complejidad del problema del Set Cover se establece en $O(n \times u)$, donde n representa la cantidad de subconjuntos en S y u la cantidad de elementos presentes en el universo. Este costo surge 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. Luego, en cada subconjunto nuevo E_i proveniente del elemento e_i se le agrega el S_i donde está incluido, siendo esta una operación 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.

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

3.2.1. Ejemplos:

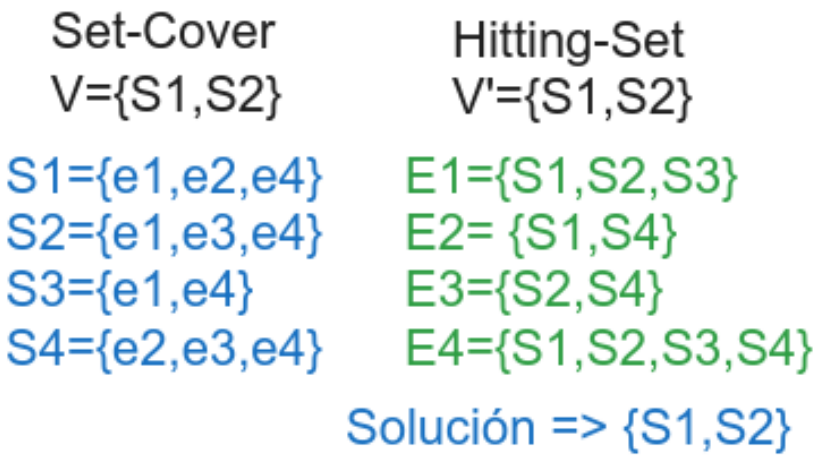

Figura 6: Ejemplo 1 de reducción de Set-Cover a Hitting Set

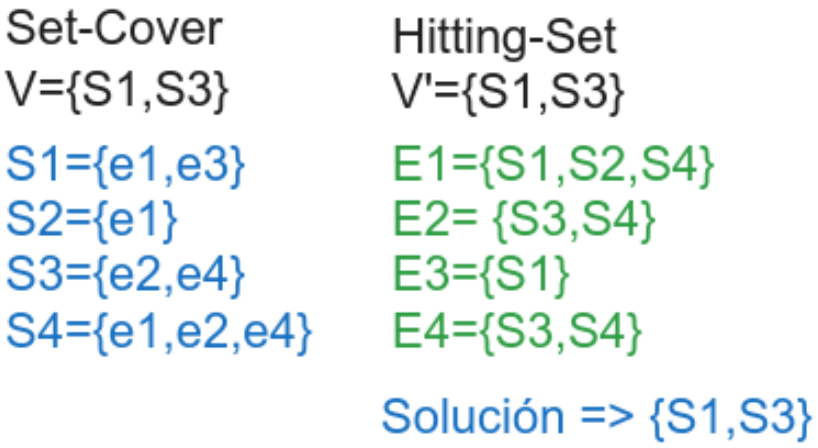

Figura 7: Ejemplo 2 de reducción de Set-Cover a Hitting Set

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. Probamos si la solucion parcial es solución y es mejor que la actual entontrada:
   a) Si lo es, actualizamos la solución actual y la devolvemos.
   b) Si no lo es, avanzamos si puedo.

2. Probamos si la solución parcial es potencial solución óptima.
   a) Si no lo es, retrocedemos y volvemos a punto 3) de la recursión anterior.
   b) Si lo es, llamamos recursivamente y seguimos en el punto 1) de la recursión siguiente.
3. Proseguimos 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 del índice i_b para verificar si coincide con la cantidad total de subconjuntos B. Se emplea esta medida para garantizar la presencia de al menos un jugador de cada conjunto incluido en la solución parcial. En caso de ser igual y si la cardinalidad de la solución parcial es inferior a la solución actual, esta es considerada la nueva solución óptima (actual).

En caso contrario, se procede a descartar la solución parcial 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.

Además, se implementó una poda adicional de gran alcance. En esta etapa, donde se examina cada jugador para determinar cuál contribuye a la solución óptima, solo se exploran aquellos que no han sido considerados previamente por los conjuntos B_j donde $j < b_i$. Esta última poda se tuvo en cuenta considerando lo siguiente: Suponiendo que se tiene $B_i = \{e_1, e_2\}$ en nivel i_b, si ya se evaluaron las alternativas con e_1, no tiene sentido revisar luego las alternativas en los conjuntos $B_k, k > i_b$ con e_2 que incluyan e_1 porque:

1. Para que eso tenga sentido, $e_1, e_2 \in C$ debe ser una alternativa superior que $e_1 \in C \land e_2 \notin C$ y $e_2 \in C \land e_1 \notin C$.
2. Si e_2 ya se estaba teniendo en cuenta en un nivel anterior $B_j, j < b_i$, ya se consideró $e_1, e_2 \in C$, por lo cual incluso en B_i no tiene sentido que se considere e_2 (ni antes e_1; ya había un jugador en B_i incluido en C).
3. La consideración de que $e_1, e_2 \in C$ es una opción mejor ya fue abordada al analizar e_1. Esto implica que ya se habría encontrado un conjunto que incluya a e_2 sin además ningún otro elemento incluido en la solución parcial. Para que $e_1, e_2 \in C$ sea una mejor alternativa, al menos un conjunto $B_k, k > i_b$ debe contener a e_2. Esto demuestra que esta alternativa ya fue contemplada al evaluar e_1 en B_i.

A continuación, incluimos el código principal del algoritmo.

```
1 def backtracking_HSP(b_array: list, sol_parcial: set, sol_actual, i_b, propuestos:
      set):
2     if i_b >= len(b_array):
3         if sol_actual == None or len(sol_parcial) < len(sol_actual):
4             return sol_parcial.copy()
5         return sol_actual
6
```

```
    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:
            continue
        sol_parcial.add(jugador)
        propuestos.add(jugador)
        propuestos_b_array.add(jugador)
        sol_actual = backtracking_HSP(
            b_array, sol_parcial, sol_actual, i_b+1, propuestos)
        sol_parcial.remove(jugador)

    propuestos.difference_update(propuestos_b_array)

    return sol_actual
```

5. Solución por Programación Lineal

5.1. Programación Lineal Entera

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

1. Para cada elemento e_i del universo $U = \{e_1, e_2, \ldots, e_n\}$, se crea una variable y_i que puede tomar los valores 0 o 1.
2. Se define una función $f_j = \sum_{i|e_i \in S_j} y_i$ 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 $O(m \times b)$, con b el cardinal del conjunto más grande, 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.

$$\text{Hitting-Set} \leq_p \text{Programación Lineal Entera}$$

El codigo de la reducción es el siguiente:

```
    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

    pulp.LpSolverDefault.msg = 0
    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, para toda variable y_i que en la solución del problema de Programación Lineal tengan un valor que exceda el umbral del $1/b$, siendo b el cardinal máximo de todos los conjuntos S_j, se incluye e_i a la solución final C del Hitting-Set Problem.

Luego, el conjunto solución C será

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

Por la mismas consideraciones aplicadas en la reducción de la Programación Lineal Entera, la complejidad asociada a la reducción de Hitting Set a Programación Lineal Continua es de $O(m \times b)$. 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:

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

    # obtenemos b = "la cantidad de jugadores que pide el periodista mas mufa"
    b = 0
    for periodista in b_array:
        if len(periodista) > b:
            b = len(periodista)

    for periodista in range(len(b_array)):
        dic_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}", lowBound=0, upBound=1, cat=pulp.LpContinuous)
                dic_jugadores[jugador] = y
            dic_p.add(y)

        p = pulp.lpSum(dic_p)
```

```
        problem += p >= 1
        problem += p <= len(dic_p)

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

    pulp.LpSolverDefault.msg = 0
    problem.solve()

    jugadores_seleccionados = [
        jugador for jugador, variable in dic_jugadores.items() if pulp.value(
    variable) > 1/b]

    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:

Consideremos dos conjuntos que representan medios diferentes, denominados S_1 y S_2. Estos conjuntos se componen de la siguiente manera:

1. S_1 con tamaño b, expresado como $S_1 = \{x_1, x_2, ..., x_{b-1}, x_b\}$
2. S_2 con tamaño $b-1$, representado como $S_2 = \{x_1, x_2, ..., x_{b-1}\}$

En el 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 S_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 $x_2 = ... = x_b = 0$.

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.

6.1. 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, maximizando así la cobertura de conjuntos dentro de la solución propuesta.

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

```
    param subconjuntos: subconjuntos de jugadores pedidos por cada prensa
    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
```

La primera operación tiene complejidad $O(b \times m)$, con b el cardinal máximo de jugadores por subconjunto y m la cantidad de subconjuntos totales. Con grupos más chicos, el algoritmo tiende a lineal ($O(m)$) y con grupos más grandes, a $O(b \times m)$. De la misma manera, la segunda operación tiene complejidad de $O(b \times m)$ ya que por cada subconjunto ($O(m)$) se deben recorrer todos sus jugadores para verificar si efectivamente están en la solución final ($O(b)$) y, en caso de que no esté ninguno, se debe encontrar el jugador con máximas apariciones $O(1)$ (esta operación se va contemplando a medida que se verifica la anterior condición). De esta manera, la complejidad total es $O(2(b \times m)) = O(b \times m)$ y resulta polinomial.

6.2. Máximo global con recálculo

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.

```
def aproximacion_greedy_maximo_global_con_recalculo(subconjuntos: list):
    """
    Obtiene la solucion por greedy.
    param subconjuntos: subconjuntos de jugadores pedidos por cada prensa
    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
```

La complejidad de la primera operación es $O(b \times m)$. La complejidad de la segunda es más difícil de acotar. La iteración del `while len ( apariciones ) != 0:` se detiene cuando todos los jugadores cuentan con 0 apariciones o recorrimos todos los jugadores. Llamaremos a esta cantidad $j \leq n$. Luego tenemos los dos `for` anidados que iteran por la cantidad de conjuntos a los que pertenece cada jugador y por todos los jugadores de cada conjunto. La primera, a la que llamaremos g está acotada superiormente por la cantidad de conjuntos m. Luego, la segunda está acotada por b, el cardinal máximo de los conjuntos. Esto resultaría en una complejidad $O(n \times m \times b)$. Sin embargo, es importante notar que el único caso en que $j = n$ es cuando $b = 1$, de complejidad $O(n \times m)$. A su vez, g no son independientes de b y m ya que, cuando más grandes son los conjuntos, a más conjuntos va a pertenecer cada jugador. Así, $g \propto^{+} m$, $g \propto^{+} b$, $g \propto^{-} n$. Por esto, la complejidad nunca alcanzaría $O(n \times m \times b)$.

7. Ejemplos

A continuación, compararemos los resultados obtenidos por nuestras dos aproximaciones Greedy en distintos ejemplos.

El primero, ilustrado en la fig. 8, es un caso feliz en el que los dos algoritmos obtienen el resultado óptimo.

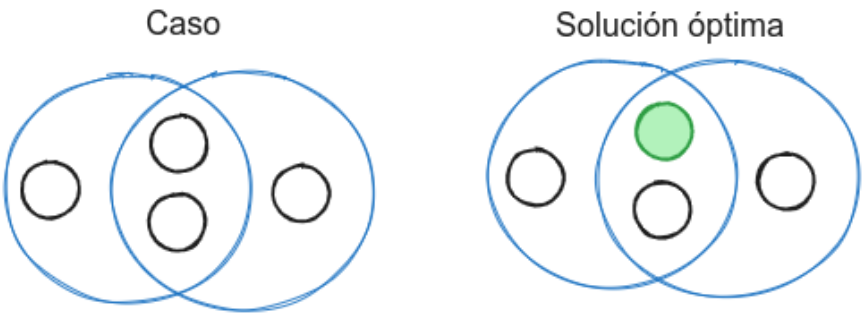

Figura 8: Ejemplo 1

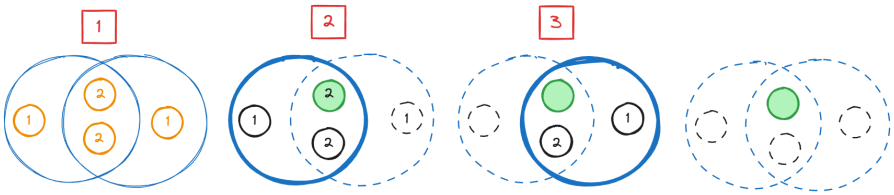

Figura 9: Ejemplo 1 resuelto por Máximo por grupo

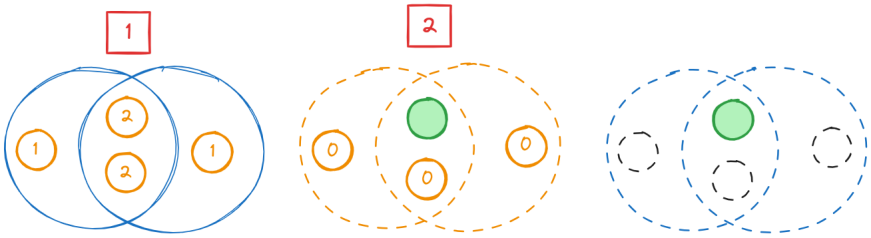

Figura 10: Ejemplo 1 resuelto por Máximo global con recálculo

El segundo ejemplo (fig. 11) podemos observar en la fig. 12 que "Máximo por grupo", de menor complejidad, no encuentra la solución óptima, pero el "Máximo global con recálculo", en la fig. 13, si.

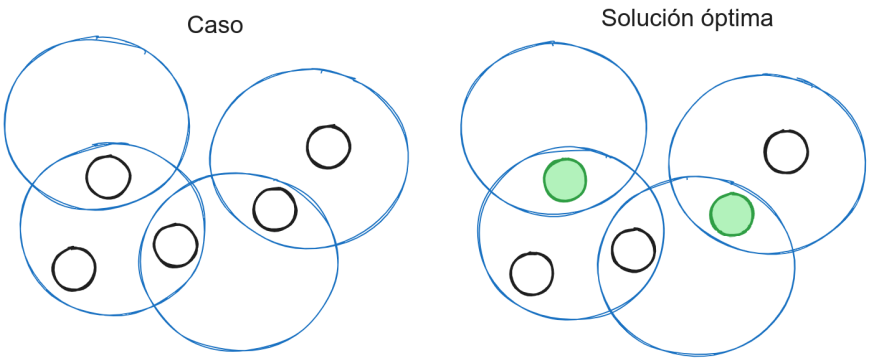

Figura 11: Ejemplo 2

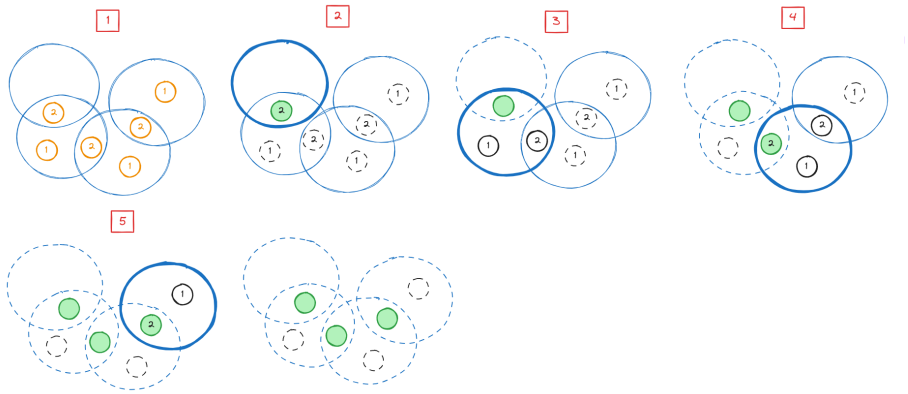

Figura 12: Ejemplo 2 resuelto por Máximo por grupo

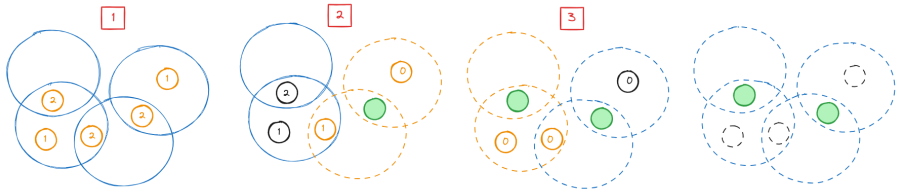

Figura 13: Ejemplo 2 resuelto por Máximo global con recálculo

Por último, en el ejemplo de la fig. 14, tanto "Máximo por grupo"(fig. 15) como "Máximo global con recálculo"(fig. 16) no llegan al la solución óptima.

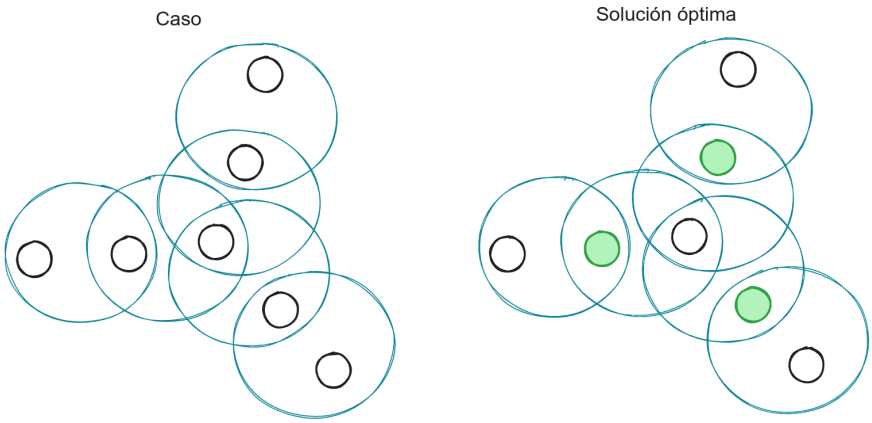

Figura 14: Ejemplo 3

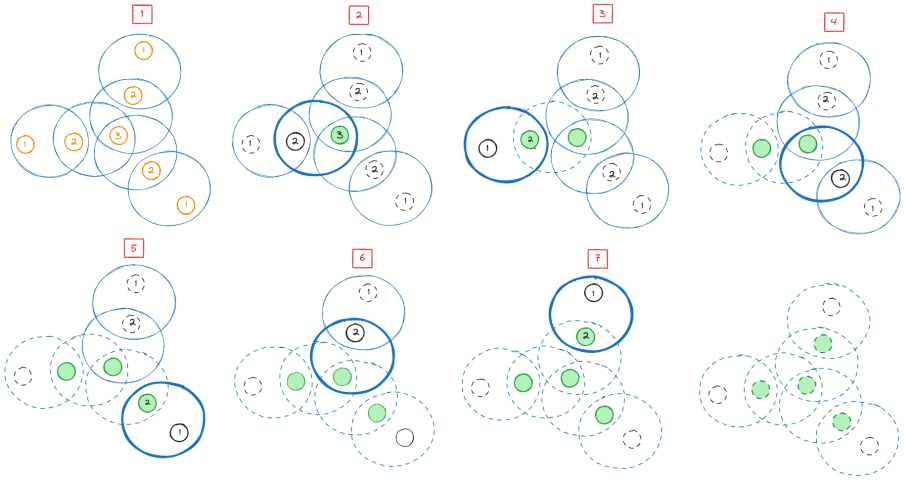

Figura 15: Ejemplo 3 resuelto por Máximo por grupo

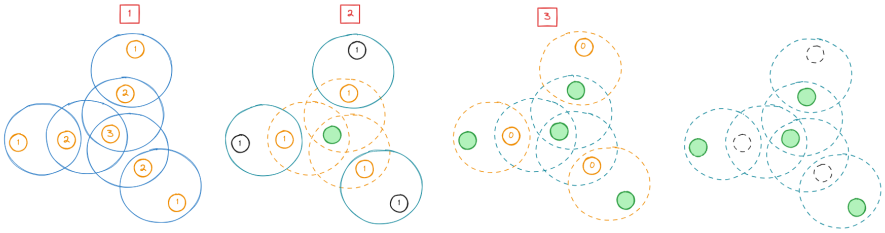

Figura 16: Ejemplo 3 resuelto por Máximo global con recálculo

8. Mediciones

Para tomar las mediciones, creamos grupos de subconjuntos de jugadores con nombres arbitrarios, variando la cantidad de subconjuntos, el tamaño máximo de los subconjuntos y el número total de jugadores.

Además, notamos que los procesos de fondo del sistema nos generaba distorciones en los tiempos de ejecución de una misma muestra. Resolvimos este problema realizando varias mediciones para el mismo escenario de subconjuntos y tomando el promedio de los tiempos obtenidos. Esta estrategia nos permitió abordar otro problema significativo: el orden en el que se evalúan los jugadores en cada subconjunto adquiere una importancia relevante en el rendimiento final del algoritmo. Esto depende especialmente de la frecuencia con la que cada jugador aparece en los demás subconjuntos.

Asimismo, al definir arbitrariamente cada escenario, puede ocurrir que en una cantidad m de subconjuntos, los jugadores incluidos en cada uno sean considerablemente diferentes o poco comunes, lo que representa un caso desfavorable. En contraste, en una cantidad de conjuntos $m+1$, podría darse una situación mucho más favorable, con una mayor similitud entre los jugadores incluidos en cada conjunto. Esto puede llevar a una diferencia significativa en los tiempos de ejecución, a pesar de que la cantidad de conjuntos varíe en una sola unidad.

Tanto en mediciones temporales como en mediciones de aproximación, comparamos el efecto de modificar una variable por vez, tanto m como b. Estas mediciones se realizaron sobre los diferentes algorítmos analizados a lo largo del informe: Backtracking, algorítmo Greedy Máximo por Grupo, algorítmo Greedy Máximo Global con Recálculo, Programación Lineal Entera y Programación Lineal Continua.

Al evaluar el algoritmo de Backtracking, podemos corroborar a través de los gráficos una clara tendencia exponencial en el tiempo de ejecución conforme se incrementa la cantidad de subconjuntos (figs. 17). Sin embargo, al observar el gráfico de la figura 18, se puede apreciar que el tiempo de ejecución disminuye a medida que se incrementa el tamaño máximo de los subconjuntos. Esto se debe a que nuestra implementación de backtracking hace grandes podas cuando se repiten muchos jugadores entre conjuntos. Un mayor tamaño de los conjuntos implica un aumento en la probabilidad de intersecciones grandes entre los mismos.

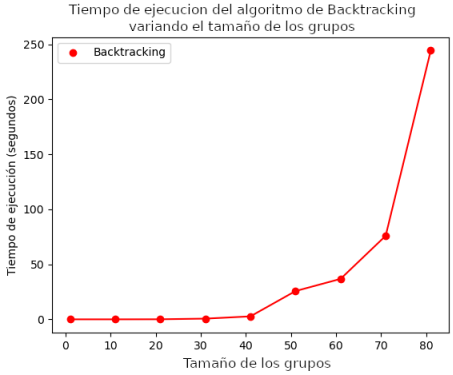

Figura 17: Backtracking: Medición de tiempos variando m.

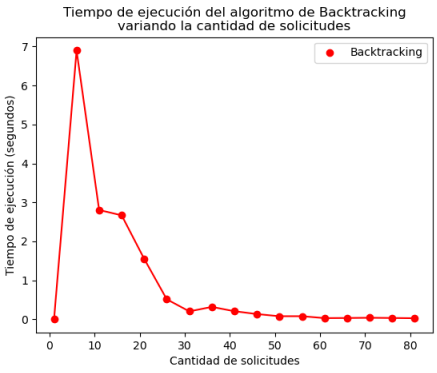

Figura 18: Backtracking: Medición de tiempos variando b.

Por otro lado, el algoritmo de Programación Lineal Entera, contrario a lo esperado, presenta una gráfica más cercana a una función lineal (figs. 19 y 20), asemejandose en tiempo de ejecución con el algorítmo de Programación Lineal Continua. Esto se debe a las optimizaciones implementadas en la librería `pulp`, las cuales reducen significativamente su tiempo de ejecución en comparación con su complejidad teórica. Es importante mencionar que la visualización de una curva exponencial en este algoritmo solo ocurre en situaciones donde las optimizaciones no influyen de manera considerable.

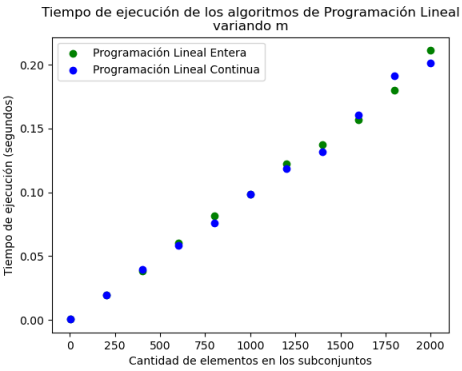

Figura 19: Programación Lineal: Medición de tiempos variando m.

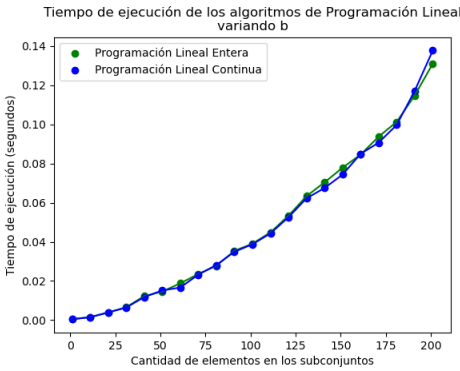

Figura 20: Programación Lineal: Medición de tiempos variando b.

En el siguiente gráfico, se ilustra el grado de aproximación, variando m, manteniendo $b = 25$, (fig. 21) y variando b, manteniendo m constante (fig. 22). La peor aproximación en la figura de la izquierda es $\approx 1{,}9 \leq b - 1$. En la figura de la derecha se puede observar que r está muy lejos de superar $b - 1$.

Por otra parte, al examinar los algoritmos que emplean la estrategia de diseño Greedy, se evidencia en la figura 23 que Máximos por Grupos tiene una tendencia lineal respecto a m, mientras

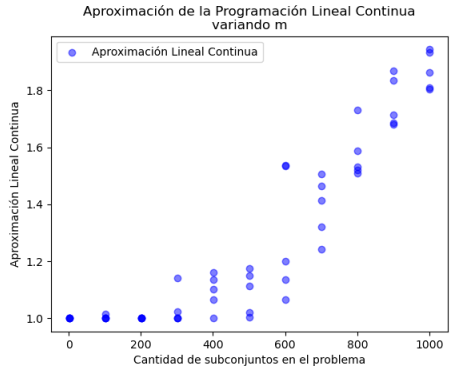

Figura 21: Greedy: Relación entre $r(A)$ y m para b constante.

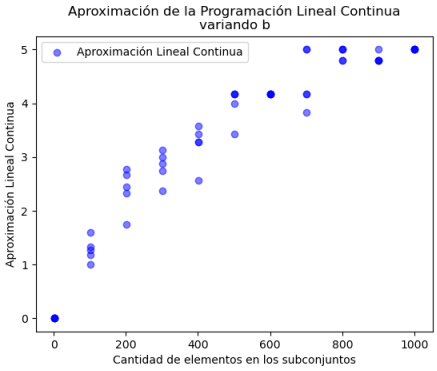

Figura 22: Greedy: Relación entre $r(A)$ y b para m constante.

que Máximo Global con Recálculo, una tendencia polinomial. En la figura 24 podemos ver que la complejidad de ambos algoritmos tiende a ser lineal para mayores valores de b como consecuencia del mismo fenomeno observado en Backtracking donde se generan mayores intersecciones entre los grupos. Asimismo, se destaca una marcada diferencia en los tiempos de ejecución entre los dos algoritmos Greedy. A pesar de que el primero es más rápido, en las figuras 25 y 26 se expone que el segundo se acerca más a la solución óptima en todos los casos estudiados.

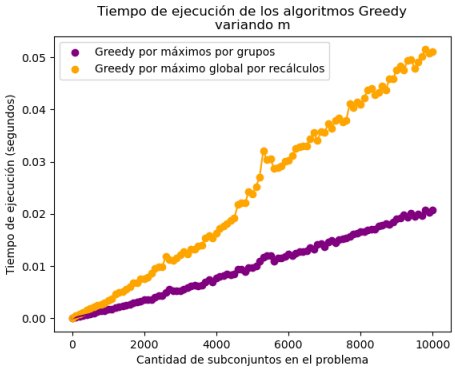

Figura 23: Greedy: Medición de tiempos variando m.

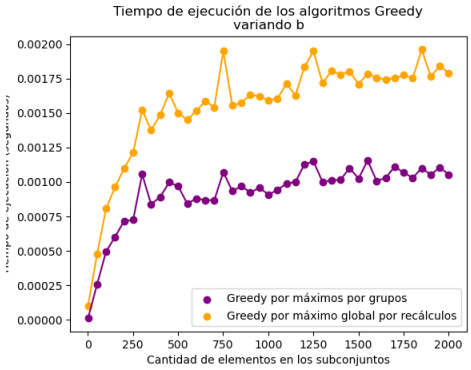

Figura 24: Greedy: Medición de tiempos variando b.

Es interesante observar como el factor de aproximación $r(A) = \frac{A(I)}{z(I)}$ varía respecto a m y a b. Para valores b pequeños constantes, que es donde los dos algorítmos Greedy tienden a performar peor, se aprecia una tendencia polinomial respecto a la variación de m (fig. 25). Sin embargo, con m constante, se puede observar una tendencia inversamente proporcional entre $r(A)$ y b (fig. 27). Esto se debe a que, con grupos más grandes, la probabilidad de que un elemento pertenezca a más de un grupo es mayor, llegando al punto de que exista uno o varios jugadores en la intersección de todos los conjuntos. Ambos algoritmos greedy obtienen siempre la solución óptima en este caso, por lo que $\lim_{b \to \infty} r(A) = 1$.

La decisiónn de qué algoritmo Greedy es el más adecuado cambia respecto a la sotiaco+pm. Por un lado, el algoritmo por grupo se destaca por su eficiencia temporal, ofreciendo tiempos de ejecución notoriamente inferiores y, por otro lado, el enfoque global muestra una mayor aproximación a la solución óptima.

Además realizamos mediciones con volumenes de datos inmanejables para los algoritmos de solución exacta, con el fin de corroborar de forma empírica la cota del algoritmo de Programacion

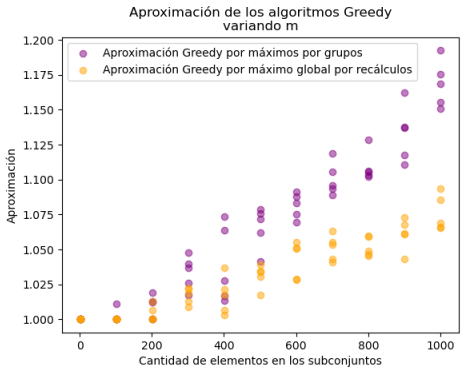

Figura 25: Greedy: Relación entre $r(A)$ y m para b constante.

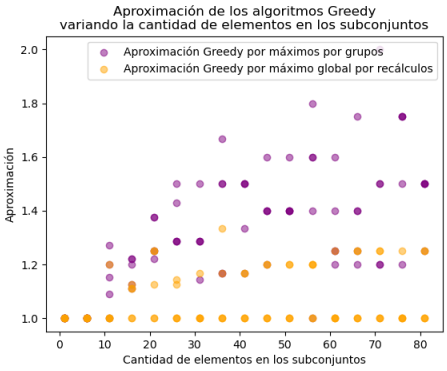

Figura 26: Greedy: Relación entre $r(A)$ y $b \in [0, 100]$ para m constante.

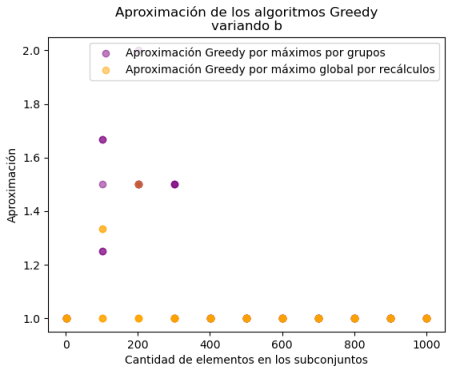

Figura 27: Greedy: Relación entre $r(A)$ y $b \in [0, 1000]$ para m constante.

Lineal Continua. Para ello implementamos un algoritmo que genera un set de conjuntos con una solución óptima menor o igual a un número k (en nuestros ejemplos $k = 25$). De esta manera, podemos corroborar dicha cota, conociendo la máxima solución óptima (decimos que es menor o igual porque pueden haber casos donde la generación aleatoria de subconjuntos implique elementos repetidos, lo que radica en una solución con un cardinal menor al impuesto por k).

En los dos primeros casos, se generaron combinando 2 y 3 letras por elemento, mientras que en el tercero se usaron solamente 2 letras. Esta variedad se buscó con el fin de ampliar las minimizar las colisiones (pero mantener suficientes como para que no sean todos los jugadores diferentes). En todos los casos, el tamaño de los subconjuntos oscila entre 10 y 12 elementos como mínimo y máximo, respectivamente.

Lo que podemos observar es que en estos casos generados (buscados meticulosamente) nuestro algoritmo de Greedy por Máximo Global con Recálculo tiene la solución óptima de las 3, con un tiempo de ejecución mediano, mientras el peor desempeño lo tiene el algoritmo de Programación Lineal Continua, con un tiempo de ejecución muy alto, y una solución muy lejana a la óptima, y el algoritmo de Greedy por Máximo por Grupo, con el tiempo de ejecución más bajo de todos, pero con una solución no siempre óptima.

En estos los casos generados, nuestro algoritmo de Greedy por Máximo Global con Recálculo ha logrado obtener la solución con cardinal más pequeño entre las tres, aunque este algoritmo alcanza un tiempo de ejecución promedio. Por otro lado, el desempeño más deficiente lo muestra el algoritmo de Programación Lineal Continua, con un tiempo de ejecución muy alto y soluciones distantes de la óptima. En contraste, el algoritmo de Greedy por Máximo por Grupo destaca por

su tiempo de ejecución más reducido, aunque su solución no siempre alcanza la óptima.

Además la cota teórica calculada para Programacion Lineal Continua se cumple en todos los casos.

5000 subconjuntos, con 2 letras, 10 mínimo b, 12 máximo b, $|C| \leq 25$

- Programación Lineal Continua:
  Cantidad mínima: 36
  Tiempo de ejecución: 336.2557799999997 milisegundos
- Greedy por Máximo Global con Recálculo:
  Cantidad mínima: 14
  Tiempo de ejecución: 13.29740200000007 milisegundos
- Greedy Máximo por Grupos:
  Cantidad mínima: 16
  Tiempo de ejecución: 6.6933350000000225 milisegundos

5000 subconjuntos, con 3 letras, 10 mínimo b, 12 máximo b, $|C| \leq 25$

- Programación Lineal Continua:
  Cantidad mínima: 36
  Tiempo de ejecución: 336.2557799999997 milisegundos
- Greedy por Máximo Global con Recálculo:
  Cantidad mínima: 14
  Tiempo de ejecución: 13.29740200000007 milisegundos
- Greedy Máximo por Grupos:
  Cantidad mínima: 16
  Tiempo de ejecución: 6.6933350000000225 milisegundos

10000 subconjuntos, con 2 letras, 10 mínimo b, 12 máximo b, $|C| \leq 25$

- Programación Lineal Continua:
  Cantidad mínima: 35
  Tiempo de ejecución: 1460.1356119999998 milisegundos
- Greedy por Máximo Global con Recálculo:
  Cantidad mínima: 8
  Tiempo de ejecución: 50.457938999999726 milisegundos
- Greedy Máximo por Grupos:
  Cantidad mínima: 10
  Tiempo de ejecución: 30.84798200000005 milisegundos

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 Continua 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 discrepar con la solución óptima.