Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

15 de diciembre de 2023

Tomas Caporaletti
108598

Helen Chen
110195

Lucas Garcia Campos
110099

1. Introducción

Luego de haber ayudado a Scaloni y el equipo técnico a no solo ordenar los análisis de los siguientes rivales sino también elegir el mejor cronograma de entrenamientos de la selección **CAMPEONA DEL MUNDO**, Scaloni comenzó a armar la plantilla para el mundial 2026 que traerá la *4ta* a casa **(se anula todo tipo de mufa)**. A su vez, la prensa está metiendo presión y cada medio está dando su conjunto de jugadores que les gustaría ver jugar con la albiceleste. Ante esta presión, Scaloni se topó con el problema de querer encontrar la mínima cantidad de jugadores para contentar a todos. Con tener un jugador que contente a cada medio le es suficiente. Como no podía ser de otra manera, el gran Bilardo, astuto pues ya se conoce todos los problemas con la prensa, le comenta que este no es más que un caso particular del $HittingSetProblem$, el cual es: *Dado un conjunto A de n elementos y m subconjuntos $B_1, B_2, ..., B_m$ de A ($B_i \subseteq A \forall i$) , queremos el subconjunto $C \subseteq A$ de menor tamaño tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \emptyset$).* En nuestro caso, A son los jugadores convocados, los B_i son los deseos de la prensa, y C es el conjunto de jugadores que deberían jugar si o si en la selección. Scaloni solicita nuestra ayuda para ver si este subconjunto de jugadores se puede obtener eficientemente, es decir, en tiempo polinomial, o se nos complica obtener dicho subconjunto más de lo que se pretende.

Ahora bien, ¿Qué significa eso de *"tiempo polinomial"*? ¿Existe algo peor? ¿A dónde puede llegar todo esto? Vamos de a poco. En el mundo de la teoría de la complejidad computacional existe lo que se conoce como *Clases de Complejidad*. Segun Wikipedia, la definicion es la siguiente:

> En teoría de la complejidad computacional, una clase de complejidad es un conjunto de problemas de decisión de complejidad relacionada. Una clase de complejidad tiene una definición de la forma:
>
> *el conjunto de los problemas de decisión que pueden ser resueltos por una máquina M utilizando O(f(n)) del recurso R (donde n es el tamaño de la entrada).*

Existen diversas clases de complejidad, y aquí se enumeran algunas de ellas:

- `P`: problemas que se resuelven en tiempo polinomial, es decir, son bastantes eficientes.
- `NP`: problemas que sus soluciones se pueden verificar en tiempo polinomial. Con esto podemos afirmar que todo problema que está en P, también está en NP, ya que si encontramos la solución en tiempo polinomial, también la podremos validar en tiempo polinomial[1].

$$P \subset NP$$

- `NP-Completos`: problemas de decisión los cuales la respuesta pueden ser "si" o "no" dependiendo si se puede resolver el problema. Estos problemas son los más difíciles de los problemas que se pueden verificar su solución en tiempo polinomial, es decir, los más difíciles de los NP. Por lo tanto,

$$P \subset NP \subset NP - Completos$$

- `PSPACE`: problemas que para encontrar su solución se requiere un espacio polinomial de memoria.

Ahora bien, si hablamos de clases de complejidad y problemas NP-Completos, también debemos hablar de *Reducciones*. Según Wikipedia:

> En teoría de la computación y teoría de la complejidad computacional, una reducción es una transformación de un problema a otro problema. Dependiendo de la transformación usada, la reducción se puede utilizar para definir clases de complejidad en un conjunto de problemas.
>
> Intuitivamente, un problema A es reducible a un problema B si las soluciones de B existen y dan una solución para A siempre que A tenga solución. Así, resolver A no

[1] Aclarar que no vale su inversa, ya que no todo problema de NP se encuentra en P, por lo que $NP \not\subset P$.

puede ser más difícil que resolver B. Normalmente, esto se expresa de la forma $A \leq B$, y se añade un subíndice en $\leq$ para indicar el tipo de reducción utilizada. Por ejemplo, se usa la letra p como subíndice para indicar que la reducción puede realizarse en tiempo polinomial: $A \leq_p B$

En otras palabras, si reducimos un problema A a un problema B, resolver el problema A será como mucho tan difícil de resolver como el problema B. Esto va a ser de gran utilidad, ya que se ha demostrado distintos problemas que son `NP-Completos`, y muchas veces para poder demostrar esto se ha utilizado y se puede utilizar la propiedad de transitividad, que es que si tengo un problema A tal que $A \in$ `NP-Completos`, y un problema B que desconozco su dificultad pero pertenece a NP, en caso que logre reducir A a B, B también es `NP-Completos`. Es decir

$$A \in NP - C : A \leq_p B \Rightarrow B \in NP - C$$

1.1. Hitting-Set Problem

Volviendo al problema del Hitting-Set Problem, empecemos por ver si este pertenece a `NP`. Para ello, recordemos que los problemas pertenecientes a NP son aquellos los cuales se pueden verificar su solución en tiempo polinomial. Para este caso, $Hitting - SetProblem \in$ `NP` de la siguiente manera: dado un conjunto C como solución al problema y un conjunto A con todos los elementos, y subconjuntos B_i tal que $\forall B_i : B_i \subseteq A$, es sencillo verificar que la solución es válida siguiendo los siguientes pasos: para cada B_i verificar que al menos uno de sus elementos está en el conjunto C. En caso que haya un B_i donde no esté ninguno de sus elementos en C, la solución no será válida. Por lo tanto, la complejidad para verificar la solución será $\mathcal{O}(n \times m)$ siendo n la cantidad de subconjuntos B, y m la cantidad de elementos de cada subconjunto. Por lo tanto, la verificación de la solución $\subset$ `P` $\Rightarrow Hitting - SetProblem \in$ `NP`.

Supongamos que tenemos el conjunto $A = (1, 2, 3, 4, 5, 6, 7, 8, 9, 10)$, y los subconjuntos $B = ((1, 2), (3, 4), (5, 6), (7, 8), (9, 10))$, y el conjunto solución $C = (1, 3, 6, 7)$. La verificación tendrá el siguiente formato:

```
1 if len(C) > len(B): return false
2 for subconjunto in B:
3     hay_un_elemento = false
4     for elemento in subconjunto:
5      if elemento in C:
6        hay_un_elemento = true
7     if not hay_un_elemento: return false
8 return true
```

La primera línea hace referencia a la validez de que es el conjunto mínimo que abarca todos al menos un elemento de todos los subconjuntos. Ya que si tengo N subconjuntos, la cantidad máxima que debo tener como solución debe ser N elementos (esto es así ya que el peor de los casos es que todos los subconjuntos no compartan ningún elemento).

Siguiendo con el ejemplo, la verificación devolverá false ya que no hay ningún elemento en el conjunto C que representa el subconjunto $(9, 10)$.

Continuando con el problema del Hitting-Set, ¿Es posible que $Hitting - Set \in$ `NP-Completo`? Para que se cumpla eso, podemos aprovecharnos de la propiedad de transitividad y reducir polinomialmente un problema que sabemos que es `NP-Completo` a Hitting-Set. Ahora bien, tenemos que encontrar un problema el cual nos pueda facilitar esta transformación (siempre y cuando exista la posibilidad de que Hitting-Set sea NP-Completo).

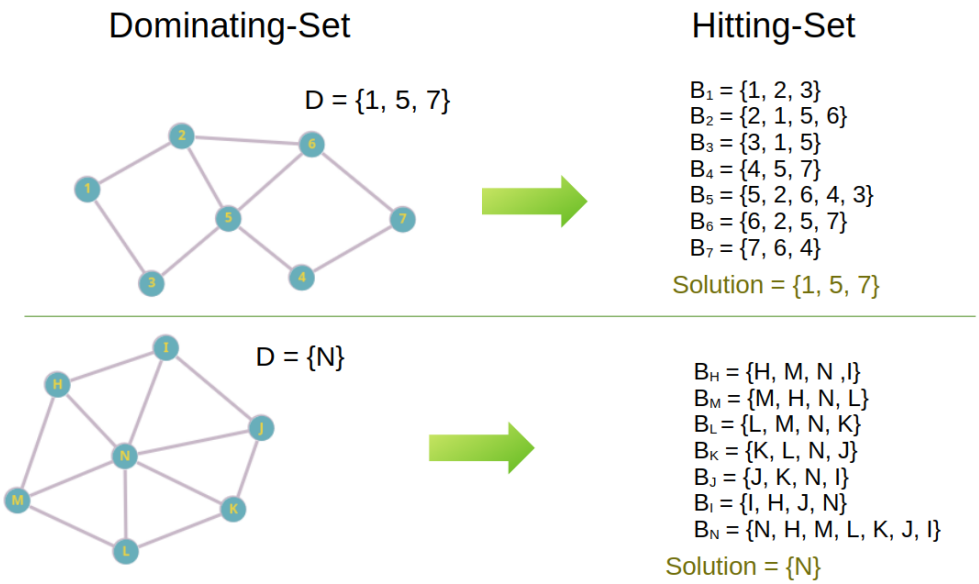
Para ello, usaremos el problema del Dominating-Set (prueba que es `NP-Completo`). Dominating-Set es un problema el cual propone que dado un grafo G, encontrar el subconjunto D de vértices de G tal que: $\forall v \in G : v \in D \vee w \in D$, siendo w un adyacente a v, y este subconjunto el más chico posible. Ahora está la pregunta, ¿Cómo podemos hacer para transformar el grafo y el problema de Dominating-Set de tal manera que haciendo uso del Hitting-Set, devuelva si existe o no una solución al problema del Dominating-Set de tamaño como mucho k?

En el Hitting-Set, lo que nos condiciona es que para cada subconjunto, haya al menos un representante, y buscar el conjunto solución tal que sea mínimo. Por el otro lado, en el Dominating-Set lo que nos condiciona es el hecho de que si o si dado un vértice v y sus adyacentes, al menos uno de dichos vértices deben estar en el subconjunto D. Esto ya nos va indicando cómo podemos hacer posible la reducción, dado que podemos empezar a ver cómo se pueden formar nuestros B_i del Hitting-Set. Empecemos por lo básico: transformar el grafo.

Ya que con lo que debemos cumplir son los subconjuntos B_i que haya al menos un elemento de cada uno, sería acorde que los vértices del grafo sean los elementos de todo el conjunto A. Falta lo importante y la parte clave: cómo relacionar estos vértices de manera que al pasar la transformación de nuestro grafo a nuestra "caja negra" que resuelve un Hitting-Set, nos devuelva que puede existir solución de como mucho k elementos, siendo k la cantidad de vértices del grafo original. La relacion sera de la siguiente manera:

- Para cada vertice v_i, se creará un subconjunto B_i el cual contendrá a v_i y todos sus adyacentes, es decir, $\forall v_i \in G : \forall w_i \in adyacentes(v_i) : v_i \in B_i \land w_i \in B_i$.
- Habrá tantos subconjuntos B_i como vértices en el grafo, y el tamaño de estos subconjuntos será la cantidad de adyacentes que tenga el vértice a partir del cual se genero el subconjunto.

Por ejemplo para los siguientes grafos, les correspondera los siguientes subconjuntos:



De esta manera, una vez creados todos los subconjuntos B_i, al pasar a la "caja negra" que resuelve el Hitting-Set nuestro conjunto y un valor k, nos devolverá si existe o no una solución la cual contenga como mucho k elementos, que "interpretado" a nuestro problema original del Dominating-Set, que existan como mucho k vértices pertenecientes al conjunto D. Una vez obtenida la salida de nuestra "caja negra" que diga si existe o no una solución al problema, podría hacerse un estilo de *Búsqueda Binaria* para encontrar el conjunto mínimo del Dominating-Set.

Al ser Dominating-Set un problema `NP-Completo`, por propiedad de la transitividad, queda demostrado que el $Hitting - Set \subset$ `NP-Completo`.

2. Algoritmos

A continuación se detallan los códigos y los pasos que se siguieron para llevar a cabo los algoritmos planteados utilizando distintas técnicas de programación.

2.1. Backtracking

A continuación se detallan el código y los pasos que se siguieron para llevar a cabo el algoritmo planteado utilizando ***Backtracking***.

2.1.1. Obtener las posibles soluciones

Una vez se obtuvieron todos los subconjuntos de jugadores para cada partido:

```
def BT2(A, jugadores, solucion, n, utilizados):

    if n == len(A):                 #Ya revise todas las listas que habian
        copia = jugadores[:]
        solucion[0] = copia
        return solucion

    if len(solucion[0]) != 0 and len(jugadores) >= len(solucion[0]):   #Hago mayor
    igual porque si esta aca es porque va a iterar devuelta
        return solucion

    if (elementosEnComun(A[n], jugadores)):
        BT2(A, jugadores, solucion, n+1, utilizados)
        return solucion

    for jugador in A[n]:
        if jugador in utilizados:
            if utilizados[jugador] < n: continue       #Ya probo todas las
    combinaciones con ese jugador.
        else:
            utilizados[jugador] = n
        jugadores.append(jugador)
        BT2(A, jugadores, solucion, n+1, utilizados)
        jugadores.remove(jugador)

    return solucion
```

siendo:

- A: Lista compuesta de m listas, las cuales representan un conjunto de jugadores para jugar un partido.
- `jugadores`: Lista con un subconjunto de jugadores que representan una posible solución actual.
- `solucion`: Lista con todos los subconjuntos de jugadores que son soluciones.
- `n`: Número que indica sobre cuál de las m listas estamos trabajando.
- `utilizados`: Diccionario cuyo par *clave-valor* es *jugador-n* donde n representa la enésima-lista en la que fue utilizado el jugador.[2]

El algoritmo persigue la exploración de diversas soluciones mediante combinaciones, las cuales se ven restringidas mediante condiciones de poda. Este enfoque se implementa con el propósito de evitar la evaluación exhaustiva de todas las posibles combinaciones evitando que se convierta en un algoritmo de *Fuerza Bruta*, optimizando así la eficiencia del proceso.

Estas condiciones de poda son:

[2] Siendo n el menor valor encontrado hasta el momento

- En caso de que la longitud de la solución actual sea igual o mayor que la longitud de la solución óptima encontrada hasta el momento, se concluye que la solución actual no mejorará la situación. La igualdad se justifica porque, al estar en esta línea de código, implica que aún quedan listas por recorrer. En estas listas, podría agregarse un elemento más, lo cual invalidaría la condición de optimalidad. En el caso de no agregarlo, no se habría encontrado una solución mejor.
- En el caso de que un elemento de la solución actual pertenezca al subconjunto bajo análisis, se concluye que no es necesario explorar soluciones adicionales que involucren elementos de dicho subconjunto. dado que nuestro objetivo es usar la menor cantidad de elementos en la solución final.
- Si un jugador ya fue utilizado en un nivel más alto del árbol de posibilidades, entonces ya cubrió todas las combinaciones que vengan en niveles inferiores. En consecuencia, si se detecta la presencia de dicho jugador en un nivel menor al que se había identificado previamente, se interrumpe la exploración de nuevas combinaciones con este elemento, dado que su participación ya ha sido exhaustivamente considerada en niveles superiores.

En cuanto a la complejidad del algoritmo, este analiza exhaustivamente todas las combinaciones posibles. No obstante, interrumpe el procesamiento de aquellas combinaciones que no conducen a una solución óptima, lo que, en consonancia con la propia naturaleza del enfoque de backtracking, implica una complejidad de $O(2^n)$, dada su relación con la exploración de todas las combinaciones posibles.

2.2. Programación Lineal

2.2.1. Variables Binarias

Con el objetivo de resolver el problema de encontrar el conjunto más pequeño de jugadores que intersectan con los subconjuntos que representan los deseos de cada periodista, definimos en primer lugar las condiciones necesarias para plantear un algoritmo utilizando *Programacion Lineal*:

1. Para encarar un problema de este tipo, comenzamos definiendo a cada jugador como una variable binaria, ya que se trata de un problema en el que tenemos que tomar una decisión con cada jugador: si formará parte o no del conjunto que jugará contra Burkina Faso. Por lo tanto, cada uno de ellos tendrá un valor que puede ser 1 (si forma parte de la solución) o 0 (si no forma parte). Para esto, se deberá cumplir con lo siguiente: $X_{player} \in 0, 1$.
2. Luego, ideamos una inecuación que representa una restricción impuesta para cada subconjunto: que al menos un jugador de este esté incluído en la solución final, para que la consigna se cumpla. Por esta razón, la suma de las variables binarias que corresponden a los jugadores de un subconjunto debe ser mayor o igual a 1, lo que se traduce en que, de manera obligatoria, uno o varios jugadores deben formar parte de la respuesta, ya que si por lo menos uno está incluído, ese jugador tiene un valor 1 y el resto 0, y en consecuencia, la suma total daría 1; sin embargo, existen posibilidades de que más de uno forme parte, por lo tanto la suma puede dar mayor a 1 también.

$$\forall subset : \sum_{player} X_{player} \geq 1$$

3. Por último, como la función objetivo es minimizar lo máximo posible la cantidad de jugadores seleccionados, la suma de todas las variables de decisión deben ser minimizadas: $min \sum X_{player}$

Se detalla a continuacion el codigo y su funcionamiento del algoritmo planteado con ***Programación Lineal***:

```python
from pulp.apis import PULP_CBC_CMD
from pulp import LpProblem, LpMinimize, LpVariable, lpSum

def hitting_set_lp(sets):
    # 1) We create binary variables for each player
    total_players = {player for s in sets.values() for player in s}
    players_vars = {player: LpVariable(f"{player}", cat='Binary') for player in
    total_players}

    # 2) We create a linear programming problem
    problem = LpProblem("HittingSet", LpMinimize)

    # 3) Constraints: each subset must have at least one player
    for _, subset_players in sets.items():
        problem += lpSum(players_vars[player] for player in subset_players) >= 1

    # 4) We minimize the total number of selected elements
    problem += lpSum(players_vars.values())
    problem.solve(PULP_CBC_CMD(msg=False))

    # 5) We extract the solution
    solution = []
    for _, player in players_vars.items():
        if player.varValue > 0:
            solution.append(f"{player}")

    return solution
```

Nuestra solución utilizando Programación Lineal se compone de la siguiente manera:

1. `Asignación de las variables`: Debido a que Scaloni nos pide hacer una selección de la mínima cantidad de jugadores que deberían jugar para satisfacer los deseos de los periodistas, decidimos utilizar como variables a todos los jugadores que fueron nombrados por ellos, guardándolos en un set previamente para evitar su repetición, y luego un diccionario cuyas claves son los nombres de los jugadores y sus valores son las variables. Estas variables son binarias (variables de decisión que pueden tomar valores 0 o 1) ya que el objetivo es decidir a quienes de todos ellos poner en la solución. Aquellas variables que tengan un valor de 1, se incluirán en la solución. En caso contrario, no se hará esto. Estas se crean utilizando *LpVariable* de *PuLP*, con su categoría en 'Binary'.

2. `Creación del problema de minimización`: Creamos un objeto problem utilizando *LpProblem* de *PuLP*, que se formula como un problema de minimización (*LpMinimize*).

3. `Restricciones`: En este paso, planteamos la inecuación lineal que define las restricciones sobre dichas variables: Para garantizar que cada subconjunto tenga al menos un jugador seleccionado, iteramos sobre los subconjuntos asegurándonos de que la suma de las variables asociadas a los jugadores en cada conjunto sea al menos 1.

4. `Minimización de la función objetivo`: Agregamos la función objetivo al problema para minimizar la suma de las variables de decisión, minimizando el número total de jugadores seleccionados.

5. `Solución`: Mediante el método *solve()*, resolvimos el problema lineal y obtuvimos una solución. Luego, procedimos a extraer la solución creando una lista que contiene los nombres de los jugadores cuyas variables de decisión tienen un valor mayor que 0 en la solución generada. Finalmente, la función retorna la solución, que es la lista de jugadores seleccionados que por lo menos jugarán contra Burkina Faso.

Con respecto a la complejidad temporal, crear las variables binarias para cada jugador cuesta $\mathcal{O}(N)$, siendo N el número total de jugadores. Sin embargo, resolver el problema lineal es la parte más costosa en cuanto a tiempo de ejecución. Esto depende en cierta parte del método de resolución utilizado por PuLP y del tamaño del modelo. La complejidad temporal en el peor de los casos llega a ser exponencial ya que analiza todos los casos, por lo que la complejidad final será de $\mathcal{O}(2^N)$.

2.2.2. Variables Reales

Para obtener la solución al problema utilizando el algoritmo propuesto por el doctor Bilardo, establecimos lo siguiente previamente:

1. En lugar de utilizar variables de decisión binarias, donde es un simplemente **si** o **no** para cada jugador, consideramos a los jugadores como variables de decisión continuas, siendo estas un número real en el rango $[0, 1]$, representando la probabilidad de seleccionar cada jugador.

$$0 \leq X_{player} \leq 1, X_{player} \in R$$

2. Para restringir las variables, utilizamos la misma inecuación que en el código anterior. Sin embargo, al tratarse de variables reales, establecimos esta misma restricción para garantizar que al menos una fracción total de un jugador en el subconjunto sea seleccionada en la solución óptima.

$$\forall subset : \sum_{player} X_{player} \geq 1$$

3. Finalmente, para minimizar la cantidad de jugadores que estarán en la solución óptima, se debe minimizar la suma de todas las variables: $min \sum X_{player}$.

Ahora bien, al resolver el problema relajando la condición de las variables nos genera un resultado distinto al anterior. Entonces, ¿cómo es posible aprovechar la solución con este método para obtener una solución válida? En el caso de que la solución con variables continuas nos indique la existencia de variables iguales a 0, estas no se incluirán en la respuesta. Por el contrario, si sus valores son iguales a 1, sí lo estarán definitivamente. Finalmente, las que se encuentran en un rango intermedio se las añadirán a la respuesta si superan o igualan un valor de redondeo calculado previamente. Este valor equivale 1/b, siendo b la cantidad del subconjunto mayor de todos los pedidos por la prensa.

```python
from pulp import LpVariable, LpProblem, LpMinimize, lpSum
from pulp.apis import PULP_CBC_CMD

def lp_approx(sets):
    # 1) We create continuous variables for each player
    total_players = {player for s in sets.values() for player in s}
    players_vars = {player: LpVariable(f"{player}", lowBound=0, upBound=1) for
    player in total_players}

    # 2) We create a linear programming problem
    problem = LpProblem("Selection", LpMinimize)

    # 3) Constraints: each subset must have at least one player
    for _, subset_players in sets.items():
        problem += lpSum(players_vars[player] for player in subset_players) >= 1

    # 4) We minimize the total number of selected elements
    problem += lpSum(players_vars.values())
    problem.solve(PULP_CBC_CMD(msg=False))

    # 5) We extract the continuous solution
    cont_solution = [player.name for player in players_vars.values() if player.
    varValue > 0]

    # 6) We calculate the rounding threshold
    b = max(len(subset) for subset in sets.values())

    # 7) Rounding the solution
    rounded_solution = [player for player in total_players if players_vars[player].
    varValue >= 1/b]

    return rounded_solution, cont_solution
```

En cuanto al código, para esta propuesta decidimos reutilizar parte del anterior haciendo unas alteraciones:

- (1) `Asignación de variables`: creamos variables continuas para cada jugador, quienes pueden tener cualquier valor de 0 hasta 1 inclusive.
- (5) `Extracción de la solución continua`: una vez resuelto el problema lineal, obtuvimos la solución continua, siendo esta una lista que incluye los jugadores seleccionados con sus valores de decisión.
- (6) `Cálculo del valor de \b"`: le asignamos a "b" el valor de la cantidad del subconjunto más largo de jugadores entre todos los que están en el conjunto.
- (7) `Solución redondeada`: obtuvimos la solución final redondeada, donde planteamos previamente que si un jugador tiene un valor mayor o igual a 1/b, se lo debe de agregar a la respuesta.

Al considerar parte de la solución a todos los valores mayores o iguales a 1/b, es posible demostrar que este valor de redondeo está bien definido ya que b al ser el subconjunto mayor, cuando se realice la sumatoria de todas las variables dentro de cualquier subconjunto $\sum_{player} X_{player} \geq 1$, además de que el resultado tenga que valer 1 o más, con dicho valor nos aseguramos que por lo menos un jugador de un subconjunto esté incluído en la respuesta final. Esto se debe a que en el peor caso de que estemos analizando el subconjunto más largo de valor b, si todas las variables tienen un valor de $1/b$ (por lo tanto la sumatoria resultaría igual a 1), deben estar todas en la solución para cumplir con la restricción impuesta. En cambio, si se elije un valor menor de redondeo, ninguna de estas variables formaría parte de la solución, lo que en este caso sería erróneo. Otra razón es que con esta restricción, algún X_{player} de cada subconjunto tomará un valor mayor o igual a $1/b$ obligatoriamente. En consecuencia, este formará parte del set para asegurarnos de que el subset esté hitteado.

Relacionando lo dicho anteriormente con la complejidad temporal de nuestro algoritmo, siendo:

- `W(S*)`: el costo de la solución óptima por programación lineal entera.
- W_{LP}: el costo de la solución continua por programación lineal.
- `W(S)`: el costo de la solución redondeada.
- `b`: largo del subconjunto mayor del conjunto,

Llegamos a la conclusión de que la solución continua propuesta con programación lineal cuesta 1/b o más del costo por cada variable que se incluye a la respuesta (W_{LP}, contrastando con la solución redondeada que requiere de un costo total por cada variable incluída $W(S)$. Por lo tanto

$$W_{LP} = \frac{1}{b} W(S) \Rightarrow bW_{LP} = W(S)$$

A su vez, el problema resuelto en el punto 4 se considera que es uno del tipo NP-COMPLETO. Sin embargo, al relajar las condiciones de la variables en el punto 5, donde las variables del punto previo que solamente podían valer 0 y 1 (condición fuerte) pasaron a valer un número cualquiera entre ese rango (condición un poco más relajada), logramos reducir el costo de la solución aproximada con respecto a la óptima:

$$W_{LP} \leq W(S*)$$

Finalmente, concluímos que el costo de una solución redondeada que es como mucho b veces peor que el del óptimo $S*$.

$$W(S) \leq bW(S*)$$

Una vez hallada esta última relación y sabiendo según el enunciado que:

- `I`: instancia cualquiera del Hitting-Set Problem.

- `z(I)`: solución óptima para dicha instancia.
- `A(I)`: solución aproximada para dicha instancia
- `r(A)`: ratio de la solución aproximada

$$\frac{A(I)}{z(I)} \leq r(A)$$

Con todas estas relaciones, concluímos que el ratio de aproximación es equivalente a al valor de b.

$$z(I) = W(S*), A(I) = W(S) \Rightarrow W(S) \leq bW(S*) \Rightarrow A(I) \leq bz(I) \Rightarrow \frac{A(I)}{z(I)} \leq b \Rightarrow b = r(A)$$

Como resultado final, esta aproximación es una *b-aproximación.*

Para respaldar este resultado y analizar cuán acertada es la aproximación, realizamos una serie de mediciones utilizando el siguiente algoritmo para calcular el ratio de la aproximación:

```
from lp import solution_lp
from lpa import solution_lp_approx

def calculate_approximation_ratio(optimal_sol, approx_sol):
    if optimal_sol == 0:
        return 1.0
    return approx_sol / optimal_sol

def perform_measurements(instances):
    for instance in instances:
        path = "tests/" + instance
        optimal_solution = solution_lp(path)
        approx_solution = solution_lp_approx(path)

        print(f"Optimal: {optimal_solution}")
        print(f"Approximate: {approx_solution}")
        approximation_ratio = calculate_approximation_ratio(len(optimal_solution),
    len(approx_solution))
        print(f"Approximation Ratio (r(A)): {approximation_ratio}\n")
```

Los resultados obtenidos son los siguientes:

- `N = 7`: $r(A) = 1.0$
- `N = 10`: $r(A) = 1.0$
- `N = 15`: $r(A) = 2.75$
- `N = 100`: $r(A) = 2.55$
- `N = 200`: $r(A) = 3.0$

Como se observa, el $r(A)$ vale 1.0 en los primeros casos, expresando que la aproximación es exacta. Sin embargo, al aumentar la cantidad de elementos paulatinamente, el ratio también lo hace, lo que refleja la magnitud de la diferencia entre el algoritmo aproximado con el óptimo. No obstante, este número ningún caso llega a superar la cota calculada previamente, indicando su efectividad.

2.3. Algoritmo Greedy

Para poder ir al código, primero recordemos cómo funciona un algoritmo Greedy. Los algoritmos Greedy siguen una regla sencilla que les permiten obtener un *óptimo local* según el estado actual del programa, y poder llegar a un *óptimo general* combinando los locales. A su vez presentan

desventajas, como por ejemplo que no siempre dan el resultado óptimo, o que demostrar que el resultado es óptimo es difícil. Por otro lado, son intuitivos de pensar y fácil de entender, y suelen ser eficientes.

Ahora bien, nuestra regla sencilla se basó en lo siguiente:

- Para cada uno de los elementos, en nuestro caso jugadores, se iterara el resto de subconjuntos, que en este caso serán los pedidos de los demás medios.
- Se tomará el jugador que más medios represente y pueda hacernos quedar bien con ellos. Estos medios ya se desestimaron porque ya habrá un jugador que ellos desean ver.
- En caso de que la actual iteración sea de un medio que ya está considerado ya que uno de sus jugadores fue tomado en cuenta, se continuará sin analizarlo.

Esto hace que nuestro algoritmo sea Greedy, ya que siempre buscará el mejor jugador que pueda satisfacer la mayor cantidad de medios posibles según la situación actual del programa, y de esta manera poder disminuir la cantidad de jugadores totales que se deben elegir para minimizar la solución siguiendo nuestra regla planteada.

El código es el siguiente:

```
def hittingSetGreedy(b):
    res = []
    considered = set()
    for idx, subset in enumerate(b):
        if subset in considered: continue
        maxElem = subset[0]
        countMax = 0
        consider = {}
        for elem in subset:
            consider[elem] = []
            repeated = 0
            for i in range(idx+1, len(b)):
                if b[i] in considered: continue
                for e in b[i]:
                    if e == elem:
                        repeated += 1
                        consider[elem] = consider[elem] + [i]
                        break
            if repeated > countMax:
                countMax = repeated
                maxElem = elem
        for i in consider[maxElem]:
            considered.add(b[i])
        res.append(maxElem)
    return res
```

El código funciona de la siguiente manera:

- `b`: set de subconjuntos que recibimos. Cada subconjunto representa los pedidos de los medios.
- `res`: set de resultado, aquí se añadirán todos los jugadores que deben meterse si o si en el plantel si Scaloni no quiere problemas con ningún medio.
- `considered`: set de medios que se van considerando a medida que añadimos jugadores a nuestro set de resultado.
- Para cada *subset* se itera sus elementos, que aqui seran los pedidos de los medios y los jugadores que desean ver. Para cada jugador del medio que se está iterando, se iteran los que quedan por ver, y nos guardaremos los medios que abarcara cada jugador para luego desestimarlos ya que están siendo considerados al elegir al jugador que más abarque medios.
- cuando se termina de iterar dicho medio, se agrega todos los medios del jugador que más abarca al set de medios considerados, así ahora en adelante esos no se tienen en cuenta para calcular la eficiencia de un jugador al determinar que tanto nos aportará si lo metemos o no.

La complejidad de dicho algoritmo es $\mathcal{O}\left(n^2 \times m^2\right)$ siendo n la cantidad de subsets (pedidos de los medios) y m la cantidad de elementos (jugadores) ya que por cada subset se itera cada elemento, y por cada elemento se itera $n-1$ subsets y por cada uno de esos subsets todos sus elementos.

Ahora bien, el algoritmo no es el optimo. Esto es debido a que depende mucho del orden de entrada de los subconjuntos, ya que siempre buscara al jugador que mas subconjuntos nos abarque a partir del suyo, pudiendo eliminar una posible solucion que sea seleccionar uno que abarque menos, pero que despues permita seleccionar otro jugador de otro subconjunto que este mas adelante que elimine mucho mas subconjuntos.

Por ejemplo:

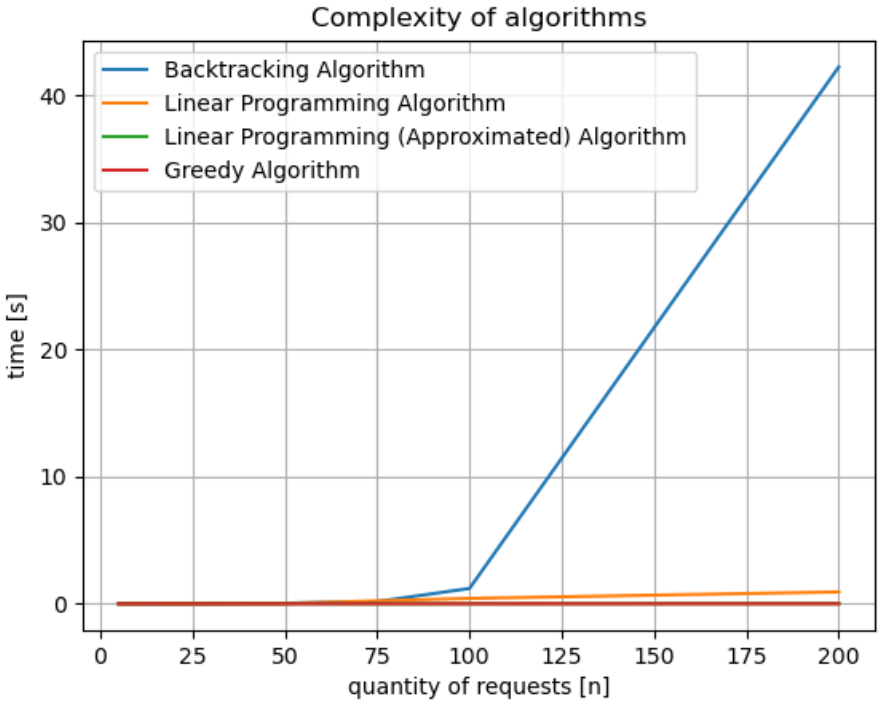
```
B = {
{1, 2, 3, 4}
{1, 3, 5}
{1, 2, 4, 5}
{1, 5}
{1, 2, 3, 4, 5}
{1, 3, 4, 5}
{2, 6, 8}
{2, 7, 8}
{5, 7}
}
```

```
Solution using greedy algorithm:
HS = {1, 2, 5}

Optimal solution:
HS = {2, 5}
```

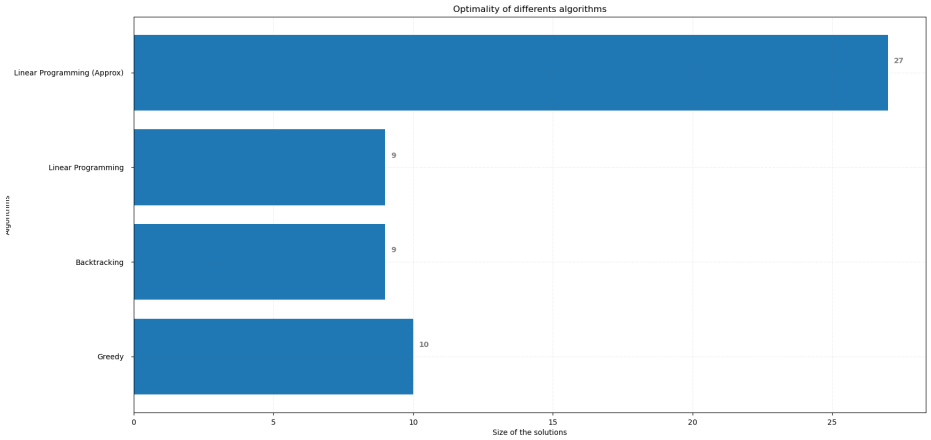
3. Mediciones

Se realizaron mediciones en base a distintos sets de pruebas que fueron desde 5 elementos hasta 200 elementos. Para cada set, se ejecutó el algoritmo por **Backtracking**, el algoritmo utilizando **Programación Lineal**, y por último el algoritmo por **Greedy**.



Como se puede apreciar, el algoritmo planteado por *Backtracking* es el algoritmo que más tarda con diferencia, haciendo honor a su complejidad respecto a los otros dos. Por detrás de él, está el algoritmo usando la técnica *Greedy*, la cual puede variar su complejidad dependiendo que tan enfocado esté en conseguir el óptimo y en que se base su regla para obtener los óptimos. Esto puede variar tanto su complejidad como qué tan lejos del resultado óptimo se encuentre. Y por último, el algoritmo que más rápido parece haber logrado con diferencia fue el usado con *Programación Lineal*. No solo ha conseguido ser el más rápido, sino que a su vez logró dar la respuesta óptima al problema. Observar que el algoritmo aproximado por *Programacion Lineal* es despreciable en cuanto a lo que tardo, al menos lo que se puede observar en el grafico. Pero los resultados en cuanto a la optimalidad de la solucion difieren cada vez mas en cuanto los subconjuntos son mayores.

Por otro lado, aunque algunos algoritmos sean mas eficientes, pueden ser menos optimos como se puede apreciar:



Se utilizo un *dataset* de 200 subconjuntos para estas mediciones.

Como podemos observar, el algoritmo por *Backtracking* da siempre la solucion optima (ya que esa es su naturaleza al ver todas las posibles soluciones con sus debidas situaciones de poda). Por otro lado, tenemos el de *Programacion Lineal Aproximada* que aunque sea muy eficiente, difiere demasiado de la solucion optima. Tambien tenemos la solucion *Greedy*, que aunque tambien es eficiente, difiere de la optima aunque con menos distancia de la que se obtuvo con la programacion lineal aproximada. Y por ultima la solucion por *Programacion Lineal* es la que sin duda la mejor opcion para este problema, ya que fue una de las mas eficientes y pudo dar la solucion optima al problema.

4. Conclusiones

Como se pudo observar, los algoritmos tuvieron sus pros y sus contras. Por un lado, el algoritmo *Greedy* fue de los que más rápido corrió. Esto hace honor también a la característica de los algoritmos Greedy y sus reglas sencillas que usan para conseguir los óptimos locales. También puede ser más personalizado, y acercarlo o alejarlo de la solución óptima, dependiendo de la regla que se siga.

Por otro lado tenemos el algoritmo planteado por *Programación Lineal*. Su velocidad no fue de esperar, y fue sorpresiva para nosotros, a la vez que entregaba la solución óptima en tan corto tiempo. Esto puede decir que usar este tipo de metodología tiene sus ventajas aunque su uso no sea uno de los más cotidianos. Aunque una pequeña observación a tener en cuenta es que a pesar de que su complejidad sea exponencial, los resultados de los tiempos parecen demostrar que no es así. Esto pudo ser debido a varios factores, como por ejemplo que el orden en el que se analizaron las variables internamente haya sido de los mejores casos, o que aunque sea una complejidad exponencial, esta metodología optimice bastante los pasos. Así como el backtracking optimiza a niveles bestiales a comparación con Fuerza Bruta, la Programación Lineal habría que ver que hace por dentro, ya que en realidad se utiliza el paquete de *PulP* para su funcionamiento. A su vez, podemos observar como el algoritmo aproximado cuanto más aumentaban los n, es decir, los subconjuntos, mas aumentaba el *ratio de aproximación*, por lo que a mayores valores, mayor será la diferencia entre la solución dada y la óptima.

Y finalmente tenemos el algoritmo por *Backtracking*. Por lejos fue el que más tardó, y demuestra que se cumple la complejidad que caracteriza a dichos algoritmos. Pero también puede tener sus ventajas: si en algún momento se solicita devolver todos las soluciones óptimas de manera que tengas distintas posibilidades de planteles y seguir alegrando a toda la prensa, bastará con modificar algunas líneas de código y nada más. A su vez, al utilizar condiciones de poda, se evita que se vuelva un algoritmo de Fuerza Bruta y aunque la complejidad computacional no cambia, a nivel práctico puede cambiar drásticamente, haciendo que no termine por un largo rato.

Finalmente, por una última vez, valoramos la última oportunidad que se nos brindó de colaborar en esta tarea crucial para que Scaloni pueda alegrar a la prensa y darle oportunidad a posibles nuevos jugadores. Estamos satisfechos por haber contribuido al éxito continuo de la selección **CAMPEONA DEL MUNDO**, y que esto nos lleve a ganar la **4ta** en 2026. Se anula todo tipo de mufa.