

TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 3

Problemas NP-Completos

25 de noviembre 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, \dots, 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*. Según Wikipedia, la definición 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 polinomial¹.

$$P \subset NP$$

- **NP-Completo**: 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,

$$P \subset NP \subset NP - \text{Completo}$$

- **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-Completo**, 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-Completo**s, 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 $A \in \text{NP-Completo}$ s, y un problema B que desconozco su dificultad pero pertenece a NP , en caso que logre reducir A a B , B también es **NP-Completo**s. 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-Set Problem* $\in NP$ 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 $\in P \Rightarrow \text{Hitting-Set Problem} \in NP$.

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 represente el subconjunto $(9, 10)$.

Continuando con el problema del Hitting-Set, ¿Es posible que *Hitting-Set* $\in \text{NP-Completo}$? 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 SAT (prueba que es **NP-Completo**). SAT es un problema el cual propone que dadas cláusulas con variables booleanas, determinar si el problema se puede satisfacer o no probando las distintas combinaciones que puedan llegar a satisfacer todas las cláusulas. Cada cláusula está compuesta por m variables, las cuales son booleanas y conectadas por un \vee el cual hace que solo sea necesaria una variable de todas para satisfacer dicha cláusula. Ahora está la pregunta, ¿Cómo podemos hacer para transformar las cláusulas y sus variables de

tal manera que haciendo uso del Hitting-Set, devuelva si existe o no una solución al problema del SAT de tamaño como mucho k ?

En el Hitting-Set, lo que nos condiciona es que para cada conjunto, haya al menos un representante, y buscar el conjunto solución tal que sea mínimo. Por el otro lado, en el SAT lo que nos condiciona es que si o si se tienen que cumplir todas las cláusulas siendo alguna de sus variables *verdadera*, y en caso que no exista asignación posible, no habrá solución al problema. Empecemos por lo básico: transformar las cláusulas.

Ya que con lo que debemos cumplir son los subconjuntos B_i , que haya al menos un elemento de cada uno, sería acorde que las variables de cada cláusula sean los elementos de todo el conjunto A . Falta lo importante y la parte clave: cómo relacionar estas variables de manera que al pasar la transformación de nuestras cláusulas 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 cláusulas. La relación será de la siguiente manera:

- Para cada cláusula, se creará un subconjunto B_i el cual contendrá las variables de dicha cláusula, es decir, las variables pasarán a ser los elementos de cada subconjunto.
- En caso que otra cláusula contenga una misma variable de otra cláusula, se usará el mismo elemento y no uno duplicado o diferente.

De esta manera, una vez creados todos los subconjuntos B , 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 SAT, que existan como mucho k variables con valor booleana *true*.

Al ser SAT 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:

```
1 def BT2(A, jugadores, soluciones, n, utilizados, minimo):
2
3     if n == len(A):
4         copia = jugadores[:]
5         minimo[0] = len(copia)
6         soluciones.append(copia)
7         return
8
9     if len(jugadores) >= minimo[0]:
10        return
11
12    if (elementosEnComun(A[n], jugadores)):
13        BT2(A, jugadores, soluciones, n+1, utilizados, minimo)
14        return
15
16    for jugador in A[n]:
17        if jugador in utilizados:
18            if utilizados[jugador] < n: continue
19        else:
20            utilizados[jugador] = n
21            jugadores.append(jugador)
22            BT2(A, jugadores, soluciones, n+1, utilizados, minimo)
23            jugadores.remove(jugador)
24
25    return soluciones
```

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.
- *soluciones*: 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
- *mínimo*: El mínimo es un número que representa la longitud del subconjunto más pequeño que se encuentra en soluciones. (Se utiliza dentro de una lista dado que las listas actúan como punteros)

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

²Siendo *n* el menor valor encontrado hasta el momento

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:

- 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.

2.1.2. Obtener la solución óptima de todas las ya encontradas

Para obtener la solución óptima se implementó el siguiente algoritmo:

```
1 import input
2 import backtracking
3
4 def solucionOptima(ruta):
5     A = input.ReadInputs(ruta)
6     soluciones = backtracking.BT2(A, [], [], 0, {}, [float("inf")])
7     return min(soluciones, key=lambda item: len(item))
```

El código lo único que hace una vez invocada a la función es buscar la mínima solución en base a la longitud.

2.2. Programación Lineal

2.2.1. Variables Binarias

Se detalla a continuación el código y su funcionamiento del algoritmo planteado con *Programación Lineal*:

```
1 from pulp.apis import PULP_CBC_CMD
2 from pulp import LpProblem, LpMinimize, LpVariable, lpSum
3
4 def hitting_set_lp(sets):
5     # 1) We create binary variables for each player
6     total_players = {player for s in sets.values() for player in s}
7     players_vars = {player: LpVariable(f"{player}", cat='Binary') for player in total_players}
8
9     # 2) We create a linear programming problem
```

```
10 problem = LpProblem("HittingSet", LpMinimize)
11
12 # 3) Constraints: each subset must have at least one player
13 for _, subset_players in sets.items():
14     problem += lpSum(players_vars[player] for player in subset_players) >= 1
15
16 # 4) We minimize the total number of selected elements
17 problem += lpSum(players_vars.values())
18 problem.solve(PULP_CBC_CMD(msg=False))
19
20 # 5) We extract the solution
21 solution = []
22 for _, player in players_vars.items():
23     if player.varValue > 0:
24         solution.append(f"{player}")
25
26 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

```
1 from pulp import LpVariable, LpProblem, LpMinimize, lpSum
2 from pulp.apis import PULP_CBC_CMD
3
4 def lp_approx(sets):
5     # 1) We create continuous variables for each player
6     total_players = {player for s in sets.values() for player in s}
```

```

7  players_vars = {player: LpVariable(f"{player}", lowBound=0, upBound=1) for
   player in total_players}
8
9  # 2) We create a linear programming problem
10 problem = LpProblem("Selection", LpMinimize)
11
12 # 3) Constraints: each subset must have at least one player
13 for _, subset_players in sets.items():
14     problem += lpSum(players_vars[player] for player in subset_players) >= 1
15
16 # 4) We minimize the total number of selected elements
17 problem += lpSum(players_vars.values())
18 problem.solve(PULP_CBC_CMD(msg=False))
19
20 # 5) We extract the continuous solution
21 cont_solution = [player.name for player in players_vars.values() if player.
   varValue > 0]
22
23 # 6) We calculate the rounding threshold
24 b = max(len(subset) for subset in sets.values())
25
26 # 7) Rounding the solution
27 rounded_solution = [player for player in total_players if players_vars[player].
   varValue >= 1/b]
28
29 return rounded_solution, cont_solution

```

Para utilizar el algoritmo propuesto por Bilardo, reutilizamos el código anterior haciendo algunas modificaciones:

- En lugar de variables de decisión binarias, donde es un simplemente sí o no para cada jugador, utilizamos variables de decisión continuas, siendo estas un número entre 0 y 1, que representa la probabilidad de seleccionar aquél jugador.
- Luego de resolver el problema y de obtener las probabilidades de selección para cada jugador, procedimos a redondear aquellas probabilidades: Para decidir a quién seleccionar definitivamente, comparamos las probabilidades redondeadas con un umbral calculado, un número b en función de la cantidad de jugadores que tiene el subconjunto más grande. Si la probabilidad de un jugador es mayor o igual a $1/b$, se lo agrega a la solución redondeada.

Con respecto a la complejidad, la complejidad total del algoritmo es de $\mathcal{O}(N + M)$, donde N es la cantidad de conjuntos y M es la cantidad total de jugadores en esos conjuntos.

Algoritmo para calcular el ratio de aproximación:

```

1  from lp import solution_lp
2  from lpa import solution_lp_approx
3
4  def calculate_approximation_ratio(optimal_sol, approx_sol):
5      if optimal_sol == 0:
6          return 1.0
7      return approx_sol / optimal_sol
8
9  def perform_measurements(instances):
10     for instance in instances:
11         path = "tests/" + instance
12         optimal_solution = solution_lp(path)
13         approx_solution = solution_lp_approx(path)
14
15         print(f"Optimal: {optimal_solution}")
16         print(f"Approximate: {approx_solution}")
17         approximation_ratio = calculate_approximation_ratio(len(optimal_solution),
18             len(approx_solution))
19         print(f"Approximation Ratio (r(A)): {approximation_ratio}\n")

```

En cuanto a la medición de cuán buena es la aproximación, calculamos el ratio de aproximación, siendo este un valor menor o igual a la división de la solución aproximada sobre la solución óptima. Los resultados obtenidos en cada caso fueron los siguientes ($r(A)$: ratio de aproximación):

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

Como se puede observar, el ratio $r(A)$ vale 1.0 en los primeros casos, lo que expresa que la aproximación es exacta, por lo tanto la cota utilizada está bien calculada. Sin embargo, al aumentar la cantidad de elementos considerablemente, el $r(A)$ aumenta, lo que refleja la magnitud de la diferencia entre el algoritmo aproximado con el óptimo.

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 hará lo mejor posible teniendo en cuenta la información actual del programa ya que siempre buscará el mejor jugador que pueda satisfacer la mayor cantidad de medios posibles.

El código es el siguiente:

```
1 def hittingSetGreedy(b):
2     res = []
3     considered = set()
4     for idx, subset in enumerate(b):
5         if subset in considered: continue
6         maxElem = subset[0]
7         countMax = 0
8         consider = {}
9         for elem in subset:
10             consider[elem] = []
11             repeated = 0
12             for i in range(idx+1, len(b)):
13                 if b[i] in considered: continue
14                 for e in b[i]:
15                     if e == elem:
16                         repeated += 1
17                         consider[elem] = consider[elem] + [i]
18                         break
19             if repeated > countMax:
20                 countMax = repeated
21                 maxElem = elem
22         for i in consider[maxElem]:
23             considered.add(b[i])
24         res.append(maxElem)
25     return res
```

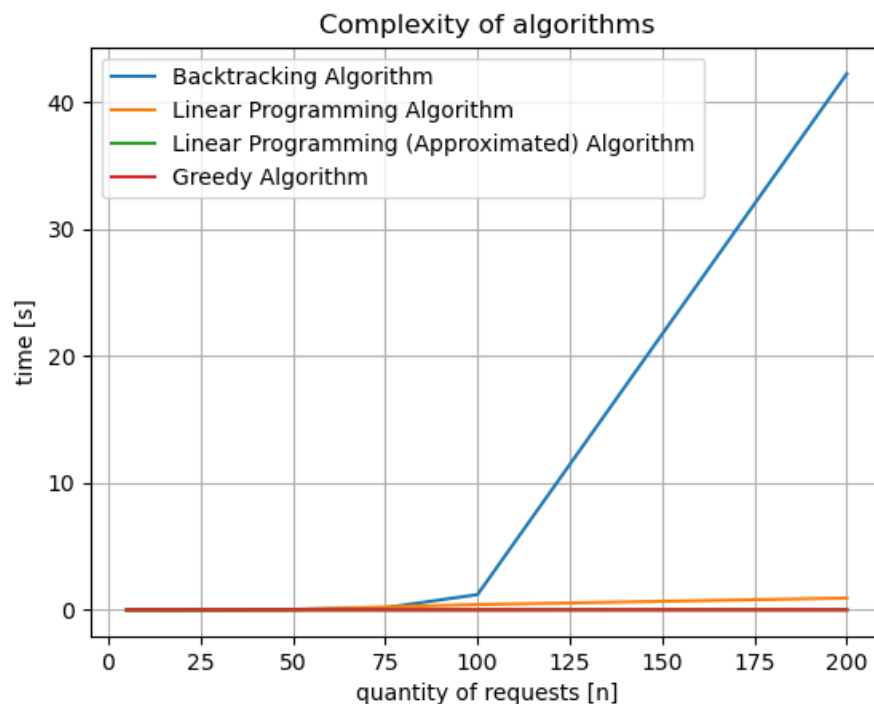
El código funciona de la siguiente manera:

- **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 aquí serán 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 $\mathcal{O}(n^2 \times m^2)$ 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.

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 *Programación 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 solución difieren cada vez mas en cuanto los subconjuntos son mayores.

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 todas 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.