
Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

# Trabajo Práctico 3
# Problemas NP-Completos

8 de enero de 2024

Milena Marchese
100962

Gianni Boccazzi
109304

# 1. Introducción

El propósito de este trabajo consiste en analizar un algoritmo de Backtracking y un modelo de programación lineal diseñado para abordar un Problema NP-Completo, así como analizar posibles aproximaciones.

En esta ocasión, Scaloni se enfrenta al desafío de elegir a los futbolistas que jugarán en el próximo partido amistoso contra Burkina Faso. La presión de los medios es considerable, lo que podría afectar a jugadores más jóvenes. Por lo tanto, Scaloni busca determinar el conjunto más reducido de jugadores necesario para satisfacer a la prensa y así poder seleccionar a los jugadores que él prefiera. Basta con elegir un jugador que satisfaga a cada periodista o medio.

El problema a analizar es el "Hitting Set Problem". Dado un conjunto $A$ de $n$ elementos y $m$ subconjuntos $B_1, B_2, \ldots, B_m$ de $A$ ($B_i \subseteq A \forall i$), queremos encontrar 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 este informe, se demostrará que el problema es efectivamente NP-Completo. Además, se llevará a cabo un análisis detallado de implementaciones para distintos escenarios. Finalmente, se implementarán y analizarán diversas aproximaciones para abordar eficientemente este problema. La resolución se encuentra en este repositorio [1] de GitHub.

# 2. Interpretación del problema

Con el fin de interpretar el problema, se plantea un ejemplo donde Scaloni, de sus **6 jugadores convocados**, debe lidiar con **3 medios distintos**, donde 2 de esos medios esperan que jueguen 3 jugadores específicos, y el otro medio espera ver jugar a 4 jugadores específicos:


Figura 1: Esquema, problema de Scaloni

Entonces, se debe encontrar el menor número de jugadores que jueguen sí o sí, incluyendo por lo menos un jugador de cada medio, a fin de que Scaloni pueda elegir la mayor cantidad de jugadores que él desea. Entonces, la solución en este caso, sería 2 jugadores:


Figura 2: Solución óptima, problema de Scaloni

Siendo que se necesita por lo menos un jugador que pide cada medio, la solución óptima podría ser elegir a los jugadores Nahuel "Perrito"Barrios y Agustín Braida, debido a que el conjunto del medio 1 concuerda con el medio 2 en que Nahuel debe jugar, y es necesario un jugador del medio 3. También, podría ser elegir a Agustin Braida y cualquier jugador del Medio 2.

Este sería un Hitting Set Problem donde el conjunto $A$ sería los jugadores convocados ($n = 6$), y los 3 medios serían 3 subconjuntos, $B_1, B_2, B_3$, entonces la solución sería un conjunto $C$ de tamaño 2, incluyendo a los 2 jugadores mencionados anteriormente, que satisfacen que $C \cap B_i \neq \emptyset$

Si el medio 3 hubiese elegido a Nahuel Barrios tal como lo hicieron el medio 1 y el medio 2, entonces la solución óptima dejaría de ser 2 jugadores y sería solamente 1, ya que se busca obtener la solución mínima, y dicho jugador estaría en los tres subconjuntos.

# 3. Análisis de complejidad computacional

## 3.1. Hitting Set Problem se encuentra en NP

Para demostrar que el problema propuesto es NP-Completo, es necesario comenzar demostrando que el problema está en NP. En otras palabras, se debe establecer la existencia de un certificador eficiente para este problema, lo que implica que **la validez de una solución se puede verificar en tiempo polinómico**.

Entonces, dado un conjunto $A$, un subconjunto solución $C$ de tamaño $k$ y subconjuntos $B_i$ de $A$, se debe verificar, en tiempo polinomial, que $C \cap B_i \neq \emptyset$. Un verificador eficiente para este problema es, en Python:

```python
def verificador_hitting_set(A, C, subconjuntos, k):
    subconjuntos_intersecados = []
    if len(C) > k or len(C) == 0:
        return False
    for jugador in C:
        if jugador not in A:
            return False
        if len(subconjuntos_intersecados) == len(subconjuntos):
            continue
        for subconjunto in subconjuntos:
            if jugador in subconjunto and subconjunto not in
    subconjuntos_intersecados:
                subconjuntos_intersecados.append(subconjunto)
    return len(subconjuntos_intersecados) == len(subconjuntos)
```

De esta forma, se verifica si el subconjunto $C$ es, efectivamente, una solución al problema. Además, se verifica que $C \subseteq A$ y $|C| \leq k$. La complejidad temporal del verificador resulta $O(n \cdot i)$ siendo $n$ la cantidad de jugadores del conjunto $C$ e $i$ la cantidad de subconjuntos "prensa"($B_i$). Esto se

debe a que, en el peor de los casos, por cada jugador de la solución, se debe recorrer cada prensa y verificar si dicha prensa ha elegido a ese jugador.

En conclusión, se puede verificar la solución en tiempo polinomial, y, por lo tanto, Hitting Set Problem está en NP.

## 3.2. Hitting Set Problem es un problema NP-Completo

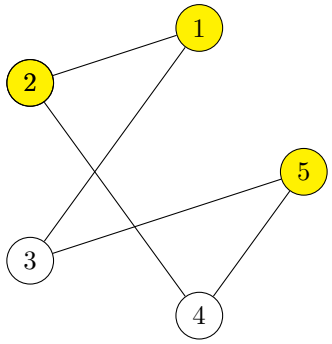
Siendo que el problema está en NP, para que sea NP-Completo se debe cumplir que $\forall Y \in$ NP, $Y \leq_p X$ siendo $X$ el problema a analizar. Por propiedad de transitividad, basta con reducir un problema NP-Completo a Hitting Set problem para demostrar su complejidad computacional, ya que, como cualquier problema se puede reducir a un problema NP-Completo, si se reduce dicho problema a Hitting Set Problem, entonces cualquier problema se puede reducir a Hitting Set.

Una manera de lograr esto es reduciendo el problema Vertex Cover. Este problema consiste en: Dado $G = (V, E)$ un grafo, donde $V$ es el conjunto de vértices y $E$ es el conjunto de aristas. Un conjunto $C \subseteq V$ es un Vertex Cover si, para cada arista $(u, v) \in E$, al menos uno de los vértices $u$ o $v$ pertenece a $C$.

Entonces, se debe adaptar convertir el grafo en distintos conjuntos de tal manera de que, si hay un Hitting Set de tamaño k, entonces hay un Vertex Cover de dicho tamaño. Esto se puede lograr considerando que:

- Se crea un conjunto $A$ que contenga a cada uno de los vértices del grafo de Vertex Cover
- Para cada arista $(u, v) \in E$, se crea un subconjunto $B_i$ que contenga a los vertices que se conectan con dicha arista.

De esta forma, Hitting Set Problem, en su versión de decisión, verificará si existe un conjunto $C \leq k$, y, como cada subconjunto $B_i$ representa una arista, entonces verificará si hay un conjunto $C \in V, C \leq k$ que cubra todas las aristas.




$$A = \{1, 2, 3, 4, 5\}$$
$$B_1 = \{1, 2\},\ B_2 = \{1, 3\}$$
$$B_3 = \{2, 4\},\ B_4 = \{3, 5\},\ B_5 = \{4, 5\}$$
$$\text{Solución } C = \{1, 2, 5\}$$
(en su versión de decisión, sería True para un $k \geq 3$)

Entonces, como se pudo reducir un problema NP Completo a Hitting Set, Hitting Set es NP Completo.

# 4. Algoritmo por Backtracking para obtener la solución óptima

Como Hitting Set Problem se encuentra en NP y $P \neq NP$, no se puede encontrar la solución óptima en tiempo polinomial. Por ello, sería eficiente implementar un algoritmo por Backtracking ya que es una mejora sobre un enfoque de fuerza bruta en situaciones en las que es posible evitar la exploración completa del espacio de búsqueda, y esto mejoraría su tiempo de ejecución, aunque siga siendo de complejidad temporal exponencial.

El algoritmo consistiría en recorrer recursivamente cada uno de los elementos del conjunto $A$, generando una solución parcial y guardando la solución óptima, siguiendo los siguientes pasos:

```
1. Si ya se recorrió por todos los elementos del conjunto A, o la solución parcial
es mayor que la solución mínima conseguida hasta esa iteración, se vuelve en
la recursividad.
2. Se visita el siguiente elemento del conjunto
3. Si el elemento actual está en alguno de los subconjuntos B:
    a. Agregar el elemento a la solución parcial.
    b. Si la solución parcial cubre todos los subconjuntos B y es la mínima
    solución actual:
         - Establecerla como la mejor solución y retornar.
    c. Si no cumple la condición anterior, llamar recursivamente con el nuevo
    elemento agregado.
4. Eliminar el elemento actual agregado en 3) y llamar recursivamente con el
siguiente elemento, sin haber agregado el elemento actual.
```

De esta forma, nos aseguramos de podar en las situaciones ideales. Si la solución parcial es mayor o igual que la solución mínima actual, no tiene sentido seguir iterando por esa rama de la recursividad, ya que lo único que se haría sería encontrar soluciones mayores a la solución mínima encontrada hasta el momento. A su vez, si se acaba de encontrar la mejor solución hasta esa iteración, se deja de iterar por esa rama de la recursividad, ya que ocurriría lo mismo que en el caso anterior, se seguiría iterando encontrando soluciones iguales o peores a la mínima actual.

## 4.1. Implementación en Python

El código resulta:

```python
def hitting_set_problem_BT(A, subconjuntos):
    set_subconjuntos = get_set_subconjuntos(subconjuntos)
    solucion_minima, solucion_parcial, actual  = [], [], 0
    hitting_set_problem_BT_rec(A,subconjuntos, actual, solucion_minima,
    solucion_parcial, set_subconjuntos)
    return solucion_minima

def hitting_set_problem_BT_rec(A, subconjuntos, actual, solucion_minima,
    solucion_parcial, set_subconjuntos):
    if actual == len(A) or (len(solucion_parcial) >= len(solucion_minima) and len(
    solucion_minima) != 0):
        return
    if A[actual] in set_subconjuntos:
        solucion_parcial.append(A[actual])
        if cubre_todos_los_subconjuntos(solucion_parcial, subconjuntos):
            if len(solucion_minima) == 0 or len(solucion_parcial) < len(
    solucion_minima):
                solucion_minima[:] = solucion_parcial[:]
                solucion_parcial.pop()
                return
        hitting_set_problem_BT_rec(A, subconjuntos, actual+1, solucion_minima,
    solucion_parcial, set_subconjuntos)
        solucion_parcial.pop()
```

```
19      hitting_set_problem_BT_rec(A, subconjuntos, actual+1, solucion_minima,
        solucion_parcial, set_subconjuntos)
20
21  def get_set_subconjuntos(subconjuntos:list):
22      set_subconjuntos = set()
23      for subconjunto in subconjuntos:
24          set_subconjuntos.update(subconjunto)
25      return set_subconjuntos
26
27  def cubre_todos_los_subconjuntos(C, subconjuntos):
28      for subconjunto in subconjuntos:
29          if not subconjunto.intersection(C):
30              return False
31      return True
```

La función `get_set_subconjuntos(subconjuntos)` tiene como finalidad crear un subconjunto que incluya a todos los elementos de cada $B_i$, en los casos en que haya jugadores que ningún medio de la prensa haya elegido.

## 4.2. Seguimiento

Con los siguientes datos:

A: [Messi, Barrios, John Kennedy, Quintero, Romagnoli]

$B_1$:[Barrios, Messi]

$B_2$:[Messi, John Kennedy]

$B_3$:[Barrios, Quintero]

Se puede realizar el siguiente seguimiento:

En el primer llamado a la funcion recursiva, Messi es el actual. Se agrega en la solución parcial ya que pertenece a algún subconjunto, y se llama recursivamente para el siguiente, que sería Barrios.

Solución Parcial: [Messi]

Actual: Barrios

Luego, se agrega a Barrios porque pertenece en algún subconjunto B, y se confirma que cubre todos los subconjuntos B, por lo tanto, es la mejor solución, y se vuelve en la recursividad, eliminando a Barrios.

Solución Parcial: [Messi]

Solución Mínima: [Messi, Barrios]

Actual: Messi

Se elimina a Messi de la solución parcial y se llama recursivamente. Siendo Barrios el actual, se agrega a la solución parcial y se verifica si es solución. Como no lo es, se llama recursivamente y el actual pasa a ser Kennedy, (quien dejó a Boca sin la séptima con un golazo).

Solución Parcial: [Barrios]

Solución Mínima: [Messi, Barrios]

Actual: Kennedy

Se agrega a Kennedy en la solución ya que pertenece a algún subconjunto, y además es solución pero no es mejor que la solución mínima actual, por lo tanto se va a podar ya que no tiene sentido seguir agregando elementos.

Siguiendo el algoritmo, la solución resulta:

Solución Mínima: [Messi, Barrios]

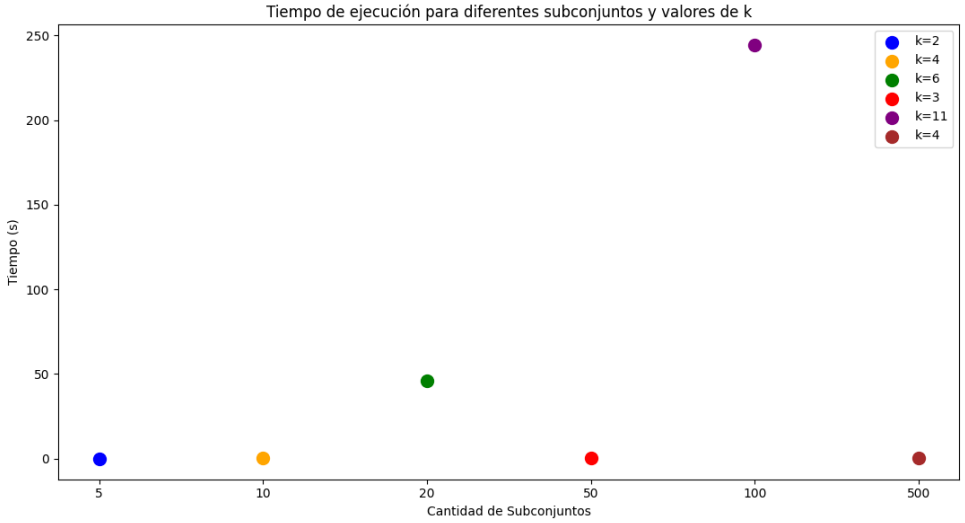
### 4.3. Ejemplos de ejecución y mediciones

Para los ejemplos de ejecución, se implementó la siguiente función para generar sets de datos:

```
def set_generator(m, k, nombre_archivo):
    jugadores = [f"Jugador{i}" for i in range(1, JUGADORES_CONVOCADOS)]
    jugadores_optimos = sample(jugadores, k)
    with open(os.path.join(DIRECTORIO, nombre_archivo), 'w') as archivo:
        for _ in range(m):
            subconjunto = sample(jugadores, randint(1, MAX_POR_CONJUNTO))
            subconjunto.insert(randint(0, k - 1), jugadores_optimos[randint(0, k - 1)])
            archivo.write(','.join(subconjunto) + '\n')
```

De esta manera, se crean sets en los que la solución será de valor k o menor. Para los ejemplos a utilizar en el informe, se asegura que **la solución mínima es del tamaño k específicado**. Se utilizarán los mismos sets para los distintos algoritmos a fin de comparar los resultados y tiempos de ejecución.

Para todos los sets, el algoritmo de backtracking encuentra la solución óptima. Las soluciones óptimas se encuentran en el repositorio. Sin embargo, como era de esperar, los tiempos de ejecución para algunos casos no son muy buenos.




Se puede notar que el tiempo de ejecución aumenta significativamente dependiendo del tamaño que tendrá la solución. Esto demuestra la eficacia del algoritmo de Backtracking, siendo que nunca va a buscar soluciones mayores a la solución mínima encontrada. Para volúmenes más grandes, el tiempo de ejecución se vuelve inmanejable.

## 5. Modelo de programación lineal para obtener la solución óptima

Siendo la programación lineal una técnica de diseño que permite resolver problemas de optimización a través de la formulación de un sistema de ecuaciones lineales en varias variables, se puede obtener la solución óptima del Hitting Set problem mediante la definición de un conjunto de variables binarias $Y_i$ para cada jugador en el conjunto $A$. En este contexto, si un jugador forma parte de la solución, entonces $Y_i = 1$; de lo contrario, el jugador no estará en la solución.

La formulación para encontrar el conjunto mínimo se traduce en la ecuación de minimización mín $\sum_i Y_i$ donde $i$ es la cantidad de elementos del conjunto $A$. Además, se requiere establecer la

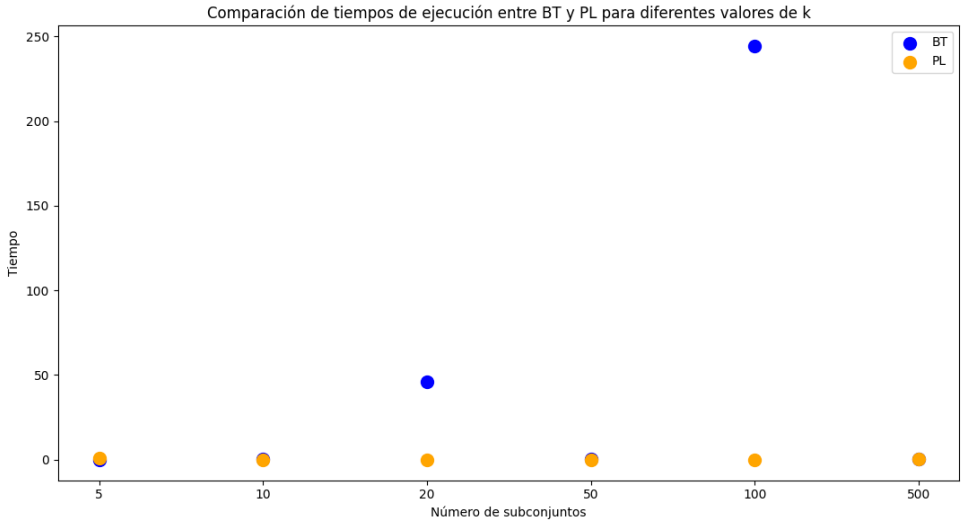
condición de que en la solución debe haber al menos un jugador de cada subconjunto, expresada como $\forall B_i \in A : \sum_{j \in B_i} Y_j \geq 1$. Con dichas ecuaciones lineales, se puede obtener la solución óptima.

El código, en Python, de dicho modelo resulta:

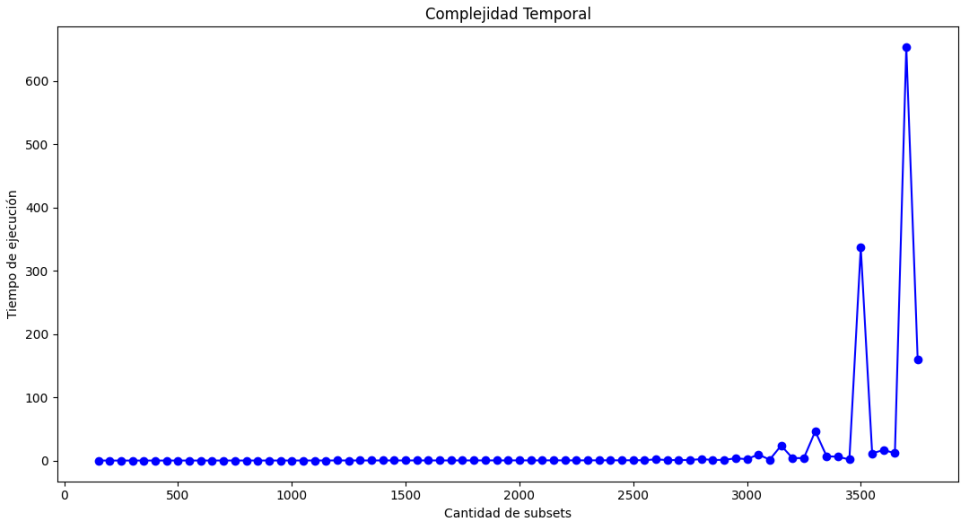
```
def hitting_set_problem_pl(A, subconjuntos):
    prob = LpProblem("Hitting Set", LpMinimize)
    variables_binarias = LpVariable.dicts("Variable", A, 0, 1, LpBinary)
    prob += lpSum(variables_binarias[i] for i in A)
    for subconjunto in subconjuntos:
        prob += lpSum(variables_binarias[j] for j in subconjunto) >= 1
    prob.solve()
    return [jugador for jugador in A if variables_binarias[jugador].value() > 0]
```

## 5.1. Ejemplos de ejecución y mediciones

Dicho modelo devuelve, en cada caso, el resultado óptimo. Además, se puede ver una diferencia notable en tiempos de ejecución, comparado al algoritmo de Backtracking:




Se puede notar que el modelo no se ve afectado en tiempos de ejecución por tamaño de la solución mínima, ni por la cantidad de subconjuntos. Si se realiza un gráfico de tiempos de ejecución para volumenes mucho más grandes:




A partir de 3000 subsets, el tiempo de ejecución empieza a aumentar exponencialmente, llegando a volverse inmanejable luego de 3600 subsets. Esto no es siempre así, ya que depende también del tamaño de la solución, como se mencionó anteriormente.

# 6. Una aproximación a Hitting Set problem: Modelo relajado de programación lineal

Se presenta ahora una aproximación del problema Hitting Set mediante programación lineal. Este modelo presenta una variación con respecto al enfoque anterior. Ahora, las variables $Y_i$ no son binarias, si no que son reales, es decir, **podrán ser cualquier número real entre 0 y 1**. Esto lleva a cambiar la regla de decisión planteada anteriormente (1 si el jugador es parte del conjunto solución, 0 si no lo es). Ahora, se plantea que el jugador $Y_i$ pertenece a la solución de la siguiente forma: Con las mismas ecuaciones para cada subconjunto $B_i$ del modelo anterior, un jugador pertenecerá a la solución si $Y_i$ es mayor a $\frac{1}{b}$, donde $b$ será la cantidad de aquel conjunto $B_i$ que tenga la mayor cantidad de jugadores.

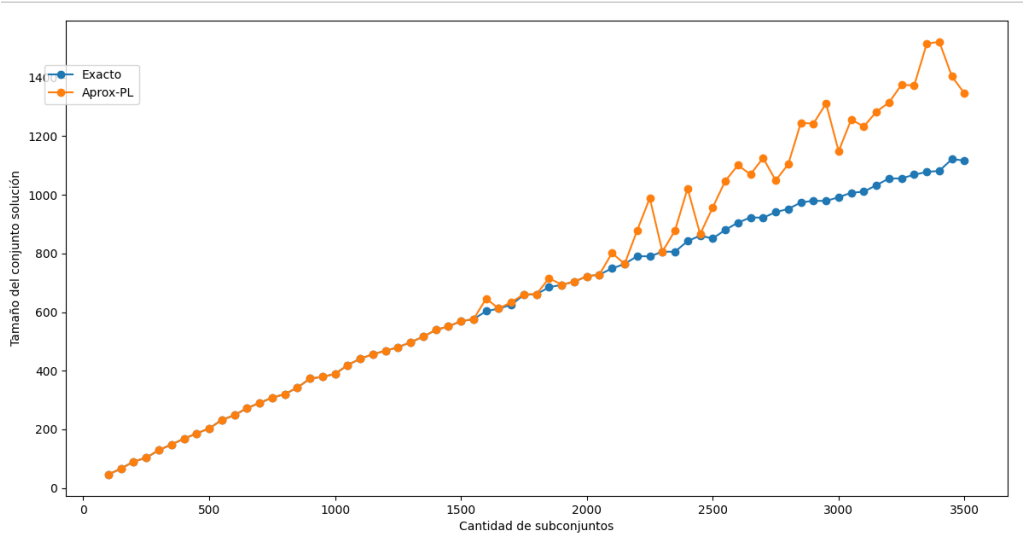
El código, en Python:

```python
def hitting_set_problem_pl_relajado(A, subconjuntos):
    prob = LpProblem("Hitting Set", LpMinimize)
    variables_continuas = LpVariable.dicts("Variable", A, 0, 1, LpContinuous)
    prob += lpSum(variables_continuas[i] for i in A)
    for subconjunto in subconjuntos:
        prob += lpSum(variables_continuas[j] for j in subconjunto) >= 1
    prob.solve()
    b = max([len(subconjunto) for subconjunto in subconjuntos])
    return [jugador for jugador in A if variables_continuas[jugador].value() >= 1/b
    ]
```

Siendo que, con esta aproximación, se han relajado las condiciones, entonces la función objetivo mín $\sum_i Y_i$ va a encontrar mejores soluciones, ya que serán más exactas. Por lo tanto $C_{\text{lp}} \leq C_{S^*}$, siendo $C_{\text{lp}}$ el resultado por la aproximación, y $C_{S^*}$ el conjunto solución óptimo. Esto significa que se ha acotado el óptimo. Como vamos a tomar los valores mayores o iguales a $\frac{1}{b}$, se puede definir que $Y_i^* \geq \frac{1}{b}$.

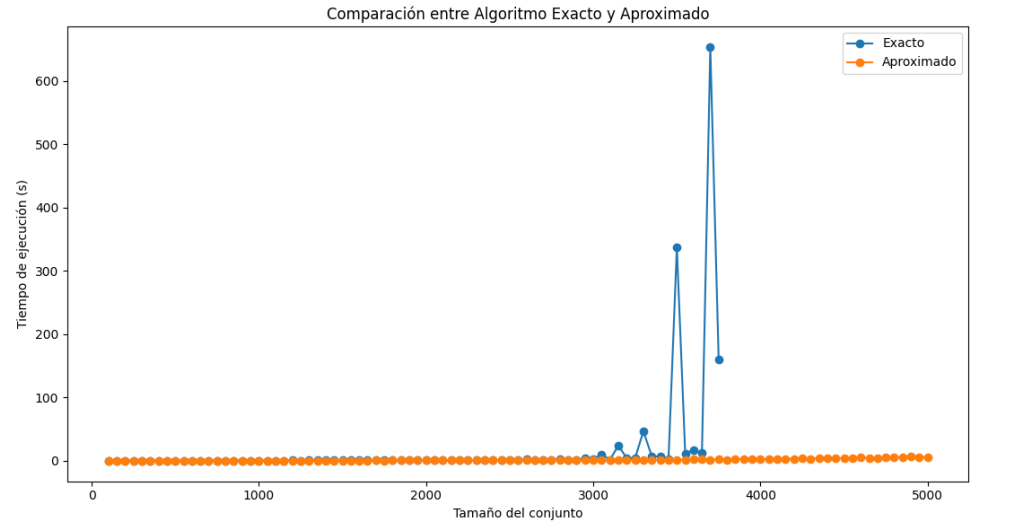
Con el redondeo determinado, podemos decir que la aproximación es $\frac{1}{b}$ mayor que $C_{S^*}$. Definiendo $z(I)$ una solución óptima para dicha instancia, y sea $A(I)$ la solución aproximada, se define $\frac{A(I)}{z(I)} \leq r(A)$. Entonces, reemplazando $C_{S^*}$ por $z(I)$ y $C_{\text{lp}}$ por $A(I)$, despejando resulta $\frac{A(I)}{z(I)} \leq b$. Se puede concluir que es una $b$ aproximación.

Si comparamos los tamaños de la solución por medio del algoritmo exacto y la aproximación:




Para los sets utilizados, $b = 6$, y se puede notar claramente que en ningún caso llega a ser 6 veces mayor al resultado óptimo, lo cual demuestra que la cota es correcta.

Además, esta aproximación llega a volúmenes mucho mayores que el algoritmo exacto:



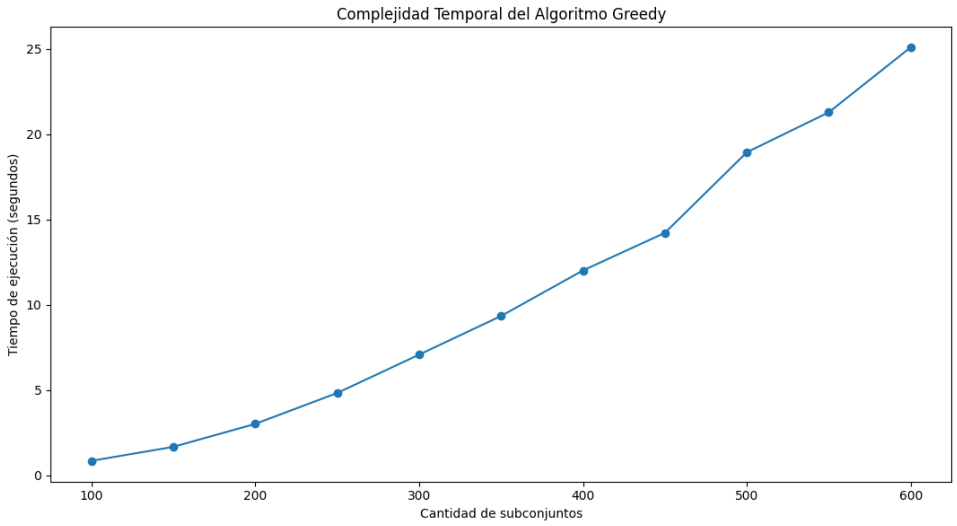

# 7. Otra aproximación a Hitting Set problem: Algoritmo Greedy

El principal objetivo de los algoritmos de aproximación es encontrar soluciones que garantizan estar cerca del resultado óptimo, en tiempo polinomial. Una de las principales metodologías de algoritmos de aproximación son los algoritmos Greedy, los cuales se caracterizan por aplicar una regla iterativamente para encontrar un óptimo local, con el fin de llegar al óptimo general. Esta aproximación nos servirá para encontrar, en tiempo polinomial, una solución aproximada a Hitting Set problem. El algoritmo a plantear consiste en la siguiente regla: buscar el próximo elemento que tenga la mayor cantidad de apariciones en los subconjuntos $B_i$, y luego descartar ese subconjunto $B_i$, para luego volver a buscar el próximo jugador con mayor cantidad de apariciones en los $B_i$ restantes, hasta que todos los subconjuntos estén intersecados. El código, en Python, resulta:

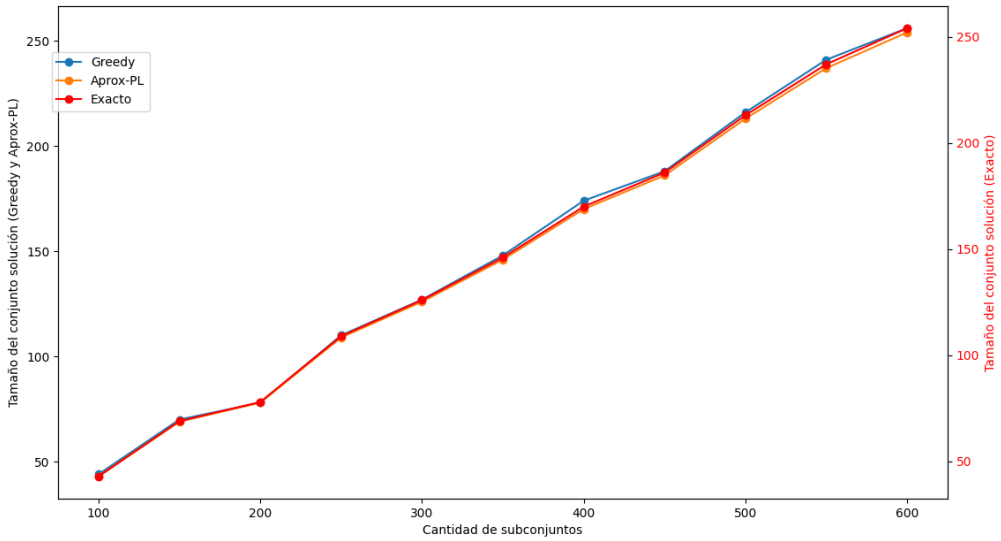
```python
def hitting_set_problem_greedy(A, subconjuntos):
    solucion = set()
    subconjuntos_intersecados = []
    subconjuntos_copia = subconjuntos[:]
    while len(subconjuntos_intersecados) < len(subconjuntos):
        diccionario_cantidad_apariciones = cantidad_apariciones(A,
    subconjuntos_copia)
        jugador = max(diccionario_cantidad_apariciones, key=
    diccionario_cantidad_apariciones.get)
        solucion.add(jugador)
        for subconjunto in subconjuntos_copia:
            if jugador in subconjunto and subconjunto not in
    subconjuntos_intersecados:
                subconjuntos_intersecados.append(subconjunto)
                subconjuntos_copia.remove(subconjunto)
    return solucion


def cantidad_apariciones(A, subconjuntos):
    diccionario_cantidad_apariciones = {}
    for jugador in A:
        for subconjunto in subconjuntos:
            if jugador in subconjunto:
                diccionario_cantidad_apariciones[jugador] =
    diccionario_cantidad_apariciones.get(jugador, 0) + 1
    return diccionario_cantidad_apariciones
```

Si denotamos la cantidad de jugadores como $m$ y la cantidad de subconjuntos como $n$, entonces la complejidad de `cantidad_apariciones` es $O(m * n)$.

El bucle principal en `hitting_set_problem_greedy` itera hasta que todos los subconjuntos estén cubiertos, y en cada iteración, se ejecuta la función `cantidad_apariciones`. Dado que se puede llamar a `cantidad_apariciones` hasta n veces, la complejidad total del bucle principal es $O(n^2 * m)$. Debido a su complejidad, a pesar de no ser exponencial, al no tener ninguna estrategia de poda, su tiempo ejecución es siempre alto.




Si comparamos resultados entre la solución exacta, el algoritmo aproximado por programación lineal y el algoritmo aproximado greedy:




A pesar de que para volúmenes grandes el algoritmo greedy se vuelve inmanejable, se puede notar que la aproximación por programación lineal se acerca más a la solución exacta.

# 8. Conclusiones

Se pueden obtener distintas conclusiones en base a las cuatro implementaciones de Hitting Set problem.

El algoritmo de Backtracking, a pesar de ser capaz de proporcionar soluciones óptimas, no es tan eficiente al enfrentarse a conjuntos de datos más grandes debido a su complejidad exponencial. Especialmente, cuando la solución tenga un tamaño grande.

Por otro lado, la Programación Lineal demostró ser eficiente, obteniendo la solución óptima y además, logrando obtener la solución más rápido que por Backtracking.

La Aproximación por Programación Lineal destacó al proporcionar soluciones rápidas y eficientes para conjuntos de datos considerables, sin embargo, en volúmenes grandes la aproximación se aleja a la solución óptima, cumpliéndose la cota $b$.

Finalmente, la Aproximación Greedy no resultó ser eficaz en tiempos de ejecución, y su aproximación se aleja bastante a la solución óptima.

# 9. Anexo: Nueva implementación de algoritmo de Backtracking

El código de la nueva implementación del algoritmo de Backtracking resulta:

```
1  def hitting_set_problem_BT_mejorado(A, subconjuntos):
2      time_start = time.time()
3      subconjuntos.sort(key=lambda x: len(x))
4      solucion_minima = hitting_set_problem_BT_rec(subconjuntos, 0, set(), set())
5      time_end = time.time()
6      return solucion_minima, time_end - time_start
7
8  def hitting_set_problem_BT_rec(subconjuntos, actual, solucion_minima,
       solucion_parcial):
9      if actual >= len(subconjuntos) or (len(solucion_parcial) >= len(solucion_minima
       ) and len(solucion_minima) != 0):
10         return solucion_minima
```

```
    for jugador in subconjuntos[actual]:
        if jugador in solucion_parcial or not hittea_nuevo_set(jugador, actual, subconjuntos):
            continue
        solucion_parcial.add(jugador)
        if cubre_todos_los_subconjuntos(solucion_parcial, subconjuntos):
            if len(solucion_minima) == 0 or len(solucion_parcial) < len(solucion_minima):
                solucion_minima = solucion_parcial.copy()
                solucion_parcial.remove(jugador)
                return solucion_minima
        subconjuntos_restantes = obtener_subconjuntos_restantes(jugador, actual, subconjuntos)
        solucion_minima = hitting_set_problem_BT_rec(subconjuntos_restantes, actual, solucion_minima, solucion_parcial)
        solucion_parcial.remove(jugador)
    return solucion_minima

def cubre_todos_los_subconjuntos(C, subconjuntos):
    for subconjunto in subconjuntos:
        if not subconjunto.intersection(C):
            return False
    return True

def hittea_nuevo_set(jugador, actual, subconjuntos):
    for i in range(actual, len(subconjuntos)):
        if jugador in subconjuntos[i]:
            return True
    return False

def obtener_subconjuntos_restantes(jugador, actual, subconjuntos):
    subconjuntos_restantes = subconjuntos.copy()
    for i in range(len(subconjuntos)-1, actual-1, -1):
        if jugador in subconjuntos_restantes[i]:
            subconjuntos_restantes.pop(i)
    return subconjuntos_restantes
```

A diferencia de la primera implementación, este algoritmo no se basa en la exploración de los jugadores, evaluando los sets, si no que se exploran los subconjuntos uno a uno con el fin de hittear cada uno de ellos con, por lo menos, un jugador de cada set. Además, antes de comenzar a recorrer cada subconjunto recursivamente, se ordena el arreglo de subconjuntos por menor cantidad de jugadores, ya que de esta manera se logra satisfacer primero a los periodistas que tienen menos jugadores requeridos, y se llega a la solución mínima con menos cantidad de llamadas recursivas. Por ejemplo, si un pedido tiene solamente 2 jugadores, necesariamente uno de ellos va a tener que estar en la solución mínima, y es más eficiente que el algoritmo comience a buscar la dicha solución con alguno de estos jugadores.

A su vez, este nuevo algoritmo se caracteriza por verificar si cada jugador que se puede estar agregando a la solución hittea algún nuevo set, ya que en caso de no hacerlo no tiene sentido explorar dicha rama.

También, al agregar un nuevo jugador a la solución, se filtran los nuevos sets hitteados, de manera que el próximo llamado recursivo siempre sea para un set que no está hitteado.

# Referencias

[1] *Repositorio*. URL: `https://github.com/milemarchese/TDA-buchwald`.