.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

24 de noviembre de 2023

Agostina Picchetti - 108028

Nehuén Cabibbo - 108025

Índice

1. Introducción

En el presente trabajo práctico analizaremos la complejidad del problema de hallar un *Hitting Set* mínimo, demostrando que, en su versión de decisión, es un problema $NP - Completo$ y buscaremos resolverlo de forma óptima y aproximada aplicando diferentes técnicas de diseño de algoritmos: *Backtracking* y Programación Lineal Entera para la búsqueda de la mejor solución; Programación Lineal y *Greedy* para hallar soluciones aproximadas.

2. Análisis del problema y algoritmos propuestos

El problema de *Hitting-Set* en su versión de optimización se enuncia de la siguiente manera:

> Dado un conjunto de elemento A de n elementos, m subconjuntos $B_1, B_2, ..., B_m$ de A ($B_i \subseteq A \ \forall \ i$), queremos un subconjunto $C \subseteq A$ de menor tamaño tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \emptyset$).

Procederemos a analizar la complejidad del problema y demostrar que no puede ser resuelto óptimamente en tiempo polinomial.

2.1. Análisis de la complejidad del problema

Para realizar el análisis, consideraremos la versión de decisión del problema *Hitting Set*:

> Dado un conjunto de elemento A de n elementos, m subconjuntos $B_1, B_2, ..., B_m$ de A ($B_i \subseteq A \ \forall \ i$), y un número k, ¿existe un subconjunto $C \subseteq A$ con $\mid C \mid \leq k$ tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \emptyset$)?

2.1.1. *Hitting-Set* se encuentra en NP

Para probar que el problema de $Hitting - Set$ se encuentra en NP tomamos una instancia cualquiera del problema, una solución propuesta C y un número k.

Podemos verificar en tiempo polinomial que:

- El conjunto C contiene al menos k elementos: contamos sus elementos y luego comparamos que sea menor o igual a el valor de k
- Validamos que al menos un miembro de cada subconjunto B_i pertenezca a C: por cada conjunto B_i validamos que la intersección con C no sea vacía.

El siguiente código implementa un certificador eficiente para *Hitting Set Problem*:

```
def es_hitting_set(hitting_set, subconjuntos):
    for sc in subconjuntos:
        if sc.isdisjoint(hitting_set):
            return False
    return True

def certificador_eficiente_hs(hitting_set, k, subconjuntos):
    return len(hitting_set) <= k and es_hitting_set(hitting_set, subconjuntos)
```

La función recibe como parámetros:

- `hitting_set`: un `set` con los elementos del *hitting set* solución
- `k`: del tipo `int`

- `subconjuntos`: una lista de `set` que representa los subconjuntos B_i

Si analizamos la complejidad de la función, observamos que todas las operaciones que se realizan son de tiempo constante (comparación entre dos números, consultar si un elemento está en un conjunto) o de tiempo lineal (iterar la lista de subconjuntos, calcular intersecciones entre `set`). Por cada uno de los m conjuntos se evalúa si alguno uno de sus miembros (que en el peor caso todos los conjuntos podrían contener a los n elementos) está incluído en C. Por lo tanto, la complejidad es $\mathcal{O}(m.n)$

Por lo tanto, es posible validar la solución en tiempo polinomial y el problema de *Hitting-Set* está en NP.

2.1.2. *Hitting-Set* es $NP - Completo$

Para probar que efectivamente estamos tratando con un problema $NP - Completo$ buscaremos reducir polinomialmente el problema de *Vertex Cover* a *Hitting-Set*.

Para ello, partimos de una instancia arbitraria de *Vertex Cover* y la transformamos en una válida para la caja negra que resuelve el problema de *Hitting-Set*.

- *Vertex Cover*:
  - implica encontrar la mínima cantidad de vértices que cubran todas las aristas de un grafo G
  - el problema recibe un grafo y un número natural k
- *Hitting-Set*:
  - implica encontrar la mínima cantidad de elementos que cubran todos los subconjuntos B_i
  - recibe un conjunto A de n elementos, lo m subconjuntos y un número natural k

Entonces, dado un grafo $G(V, E)$ y un número k definimos el conjunto $A = V$, es decir que cada uno de los vértices de G será uno de los n elementos contenidos en A, y por cada arista $a_i = (u, v) \in E$ creamos un conjunto $B_i = \{u, v\}$.

Esta transformación consta de los siguientes pasos:

- Obtener los vértices de G
- Crear el conjunto A y agregar cada vértice
- Obtener las aristas del grafo G
- Iterar las aristas obtenidas y por cada una crear un conjunto que contenga cada vértice terminal

Exponemos a continuación una implementación posible usando la libería de `Python Networkx`:

```python
import networkx as nx

def crear_conjuntos(aristas):
    conjuntos = []
    for arista in aristas:
        conjuntos.append(set(arista))
    return conjuntos

def transformar_entrada(grafo):
    elementos = list(grafo.nodes)
    aristas = grafo.edges
    conjuntos = crear_conjuntos(aristas)
    return elementos, conjuntos
```

Podemos observar que, indistintamente de la implementación del grafo, todas las operaciones involucradas en la transformación son realizables en tiempo polinomial.

Si asumimos una implementación de grafo con diccionarios, podemos detallar la complejidad de cada operación:

- Definimos:
  - V: cantidad de vértices
  - E: cantidad de aristas
- Obtener vértices: $\mathcal{O}(V)$
- Obtener aristas: $\mathcal{O}(V + E)$
- Iterar aristas y crear cada conjunto: $\mathcal{O}(E)$
- Insertar al final de una lista, agregar un elemento a un conjunto: $\mathcal{O}(1)$

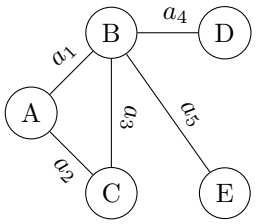
Por lo tanto, la transformación de una instancia de *Vertex Cover* a una de *Hitting-Set* puede realizarse en tiempo lineal.

Una vez adecuada la entrada, podemos preguntarle a la caja negra que resuelve *Hitting-Set* si encuentra una solución de k elementos.

Observamos que en el problema de *Vertex Cover* cada vez que se elige un vértice, se cubren todas sus aristas incidentes. Análogamente, en el problema de *Hitting-Set*, cada vez que se elige un elemento, se cubren todos los conjuntos que lo contienen.

Exponemos el siguiente ejemplo para una mejor visualización:

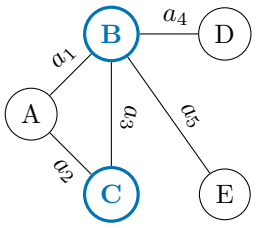
- Dado el siguiente grafo $G(V, E)$
  - $V = \{A, B, C, D, E\}$
  - $E = \{a_1, a_2, a_3, a_4, a_5\}$



- Creamos el conjunto A a partir de V y los subconjuntos B_i a partir de las aristas
  - $A = \{A, B, C, D, E\}$
  - $B_1 = a_1 = \{A, B\}$
  - $B_2 = a_2 = \{A, C\}$
  - $B_3 = a_3 = \{B, C\}$
  - $B_4 = a_4 = \{B, D\}$
  - $B_5 = a_5 = \{B, E\}$
- Si tomamos $k = 2$, podemos encontrar un *Hitting-Set* $C = \{B, C\}$ que cumple con la condición de intersecar a todos los conjuntos:
  - $B_1 = a_1 = \{A, \mathbf{B}\}$
  - $B_2 = a_2 = \{A, \mathbf{C}\}$

- $B_3 = a_3 = \{\mathbf{B}, \mathbf{C}\}$
- $B_4 = a_4 = \{\mathbf{B}, D\}$
- $B_5 = a_5 = \{\mathbf{B}, E\}$

- Observamos que el *Hitting-Set* hallado coincide con un *Vertex Cover* de $k = 2$ elementos en el grafo G



Entonces, si existe un *Hitting-Set*, C, de tamaño a lo sumo k, implicará que:

- Se pudo tomar al menos un elemento de cada subconjunto, es decir que $C \cap B_i \neq \emptyset$, y por lo tanto se cubrieron todas las aristas con a lo sumo k vértices.
- Entonces, necesariamente existe un *Vertex Cover* de tamaño al menos k.

En caso de no existir un *Hitting-Set* de tamaño k, la implicancia anterior vale negada e implicará que no existe un *Vertex Cover* de tamaño a lo sumo k.

Entonces concluimos que *Vertex Cover* $\leqslant_p$ *Hitting Set*, por lo que *Hitting Set* es al menos tan difícil como *Vertex Cover*. Dado que *Vertex Cover* es $NP - Completo$, entonces *Hitting Set* también lo es.

2.2. Algoritmos propuestos para hallar soluciones óptimas

Como ya quedó demostrado que estamos tratando con un problema difícil que no puede ser resuelto de manera eficiente, buscaremos la solución óptima por *Backtracking* y por Programación Lineal Entera, ambos algoritmos de complejidad exponencial.

2.2.1. Algoritmo por *Backtracking*

Esta será la implementación de nuestro algoritmo:

```
def _hitting_set(conjuntos, solucion_parcial, mejor_solucion, indice):
    if len(solucion_parcial) > len(conjuntos):
        return mejor_solucion

    if mejor_solucion and len(solucion_parcial) >= len(mejor_solucion):
        return mejor_solucion

    if indice >= len(conjuntos):
        if not mejor_solucion or len(solucion_parcial) < len(mejor_solucion):
            return solucion_parcial.copy()
        return mejor_solucion

    conjunto_actual = conjuntos[indice]
    if solucion_parcial.intersection(conjunto_actual):
        return _hitting_set(conjuntos, solucion_parcial, mejor_solucion, indice+1)

    for jugador in conjunto_actual:
        if jugador in solucion_parcial:
            continue
        solucion_parcial.add(jugador)
```

```
21          mejor_solucion = _hitting_set(conjuntos, solucion_parcial, mejor_solucion,
        indice+1)
22          solucion_parcial.remove(jugador)
23      return mejor_solucion
24
25  def hitting_set(conjuntos):
26      return _hitting_set(conjuntos, set(), set(), 0)
```

La función recibe como parámetros:

- `solucion_parcial`: un `set` donde se guardan los elementos que se irán incluyendo para evaluar si construyen una solución compatible
- `mejor_solucion`: un `set` para mantener referencia a la mejor solución encontrada hasta el momento, que se irá actualizando si se encuentra una combinación compatible de menor tamaño
- `indice`: del tipo `int`, el índice para avanzar en la exploración de los conjuntos
- `conjuntos`: una lista de `set` que representa los subconjuntos B_i

A pesar de que el algoritmo de *Backtracking* tiene una complejidad exponencial porque inevitablemente tienen que probar distintas permutaciones entre los elementos de los m conjuntos (y en el peor caso lo hará con todas), para evitar evaluar todo el espacio combinatorio de posiblidades y descartar opciones que no convergerán en soluciones válidas, el algoritmo propuesto realiza las siguientes podas:

- Si la solución parcial contiene más elementos que la cantidad de conjuntos, no continuamos explorando y retrocedemos. En el peor caso en que todos los conjuntos fueran disjuntos entre sí y no compartieran elementos, el *Hitting-Set* tendrá a lo sumo un elemento de cada conjunto y, por lo tanto, la cantidad de elementos incluidos no podrá ser mayor a la cantidad total de conjuntos.
- Si la solución parcial contiene más elementos que la mejor solución encontrada hasta el momento, retrocedemos y tampoco continuamos explorando esa rama de posibilidades porque ya no habría forma de mejorar el resultado por ese camino.
- Dada una lista de subconjuntos B, si existe intersección entre la `solucion_parcial` y un conjunto B_i, se descarta ese conjunto y se evalúa el siguiente B_{i+1}, ya que no tendría sentido evaluar sus elementos si el conjunto ya está cubierto por alguno contenido en la `solucion_parcial` calculada hasta el momento.
- Al recorrer los elementos del conjunto actual que estamos evaluando, si tomamos uno y ya está incluido en la `solucion_parcial`, lo salteamos porque no tendría sentido considerar repetidos.

La `solucion_parcial` será compatible con un *Hitting-Set* cuando ya no queden conjuntos por explorar, es decir hayamos cubierto todos los conjuntos B con al menos un elemento de la `solucion_parcial`. Además, si la cantidad obtenida es menor a la cantidad de elementos de la `mejor_solucion` encontrada hasta ese momento de la ejecución, actualizamos su valor por la nueva solución. Una vez agotadas todas las opciones combinatorias posibles con sus correspondientes podas, se retorna la `mejor_solucion` obtenida.

2.2.2. Algoritmo por Programación Lineal Entera

Alternativamente, proponemos el siguiente modelo de Programación Lineal Entera para resolver el problema de forma óptima:

- Por cada jugador $j \in B_i$ existe la posibilidad de incluirlo o no incluirlo en la solución C (el *Hitting Set* resultante). Definimos, entonces, una variable binaria j_k por cada jugador que tomará el valor 1 si es incluído en la solución o el valor 0 si no es incluído.
- Como buscamos un *Hitting Set* mínimo y esto implica optimizar la cantidad de jugadores seleccionados, definimos la función objetivo como la mínima sumatoria de los valores de las variables binarias.
- La restricción que aplicará a nuestro problema será que deberá haber al menos un jugador perteneciente a cada subconjunto B_i, ya que es condición que la intersección entre C y cada B_i no sea vacía.

Entonces, definimos nuestro modelo como:

- Función objetivo:

$$\text{mín} \sum_{k=1}^{n} j_k$$

- Con las restricciones:

$$j_k \in \{0, 1\}$$

$$\sum_{j_k \in B_i} j_k \geq 1 \ \forall \ B_i$$

Para programar el algoritmo que resuelva el problema modelado, elegimos la librería `Pulp` de `Python` y lo implementamos de la siguiente manera:

```python
import pulp as pl

def hitting_set(conjuntos):
    jugadores = set.union(*conjuntos)
    problema = pl.LpProblem("hitting_set", pl.LpMinimize)
    hs = {jugador:pl.LpVariable(f"{jugador}", cat="Binary") for jugador in
    jugadores}
    for conjunto in conjuntos:
        variables = []
        for jugador in conjunto:
            variables.append(hs[jugador])
        problema += pl.lpSum(variables) >= 1
    problema += pl.lpSum(hs.values())
    problema.solve(pl.PULP_CBC_CMD(msg=False))
    return [j for j in hs.keys() if pl.value(hs[j]) > 0]
```

La función recibe como parámetros:

- `conjuntos`: una lista de `set` que representa los subconjuntos B_i

Como ya mencionamos, esta solución por Programación Lineal Entera no puede ser resuelta en tiempo polinomial y su complejidad es exponencial. Esto queda demostrado porque:

- Dada una solución al problema por Programación Lineal Entera, puede validarse en tiempo polinomial con un certificador eficente (análogamente a como se muestra en la sección de análisis del problema) y por lo tanto Programación Lineal Entera está en NP.
- Transformamos la entrada de una instancia arbitraria de *Hitting-Set* en una entrada válida para el problema de Programación Lineal Entera, y esa transformación puede realizarse en tiempo polinomial.
- Entonces, $Hitting - Set \leq_p Programacion\ Lineal\ Entera$ y ya probamos en la primera sección que el problema de *Hitting-Set*, en su versión de decisión, es $NP - Completo$ por lo tanto Programación Lineal Entera también lo es.

2.3. Algoritmos propuestos para hallar soluciones aproximadas

Conocida la complejidad del problema, es evidente que la solución se vuelve intratable para procesar volúmenes de datos muy grandes. Para poder manejar estos casos, proponemos dos algoritmos de aproximación, uno por Programación Lineal y otro *Greedy*, que si bien no consiguen siempre la solución óptima, pueden ejecutarse eficientemente con una complejidad polinomial.

2.3.1. Algoritmo por Programación Lineal

Partiendo del modelo de Programación Lineal Entera detallado en la sección anterior, relajamos las restricciones y en lugar de usar variables binarias (con valores 0 y 1), utilizaremos variables contínuas que puedan tomar cualquier valor real acotado entre 0 y 1.

Definimos nuestro modelos de Programación Lineal como:

- Función objetivo:

  $$\text{mín} \sum_{k=1}^{n} j_k$$

- Con las restricciones:

  $$0 \leq j_k \leq 1$$

  $$\sum_{j_k \in B_i} j_k \geq 1 \; \forall B_i$$

Implementamos el algoritmo aproximado de la siguiente manera:

```
import pulp as pl

COTA_INFERIOR = 0
COTA_SUPERIOR = 1

def hitting_set(conjuntos):
    jugadores = set.union(*conjuntos)
    problema = pl.LpProblem("hitting_set", pl.LpMinimize)
    hs = {jugador: pl.LpVariable(f"{jugador}", COTA_INFERIOR, COTA_SUPERIOR) for
    jugador in jugadores}
    for conjunto in conjuntos:
        variables = []
        for jugador in conjunto:
            variables.append(hs[jugador])
        problema += pl.lpSum(variables) >= 1
    problema += pl.lpSum(hs.values())
    problema.solve(pl.PULP_CBC_CMD(msg=False))
    b = len(max(conjuntos, key=len))
    return [j for j in hs.keys() if pl.value(hs[j]) >= 1/b]
```

- Al trabajar con valores reales dentro de un intervalo contínuo, la sumatoria obtenida por este modelo será menor o igual a la obtenida por el modelo de Programación Lineal Entera. Si llamamos s a una solución obtenida por Programación Lineal y z a la solución óptima conseguida por Programación Lineal Entera: $s \leq z$
- Pero como los valores de las variables involucradas pueden tomar valores con coma, no definen la inclusión de un elemento dentro el *Hitting-Set* resultante.
- Redondeamos tomando un valor b según se especifica en la consigna: $b = \text{máx}\,|B|$ (la cantidad de jugadores que tenga el conjunto con máximo número de jugadores entre todos los conjuntos)y definimos que las variables de decisión tendrán un valor de 1 (se incluye al jugador en el *Hitting-Set*) si su valor en el modelo relajado es mayor o igual a $\frac{1}{b}$

- Observamos que cada B_i tendrá a lo sumo b elementos y que acorde a la restricción definida $j_1 + j_2 + \cdots + j_b \geq 1 \ \forall \ B_i$ esto implica que existe algún $j_k \geq \frac{1}{b}$ que seguro formará parte de la solución.
- Entonces, la solución aproximada para una instancia cualquiera será $A = \{ \ j_k \mid j_k \geq \frac{1}{b} \}$
- Relacionamos el valor redondeado con el valor obtenido por Programación Lineal:
  - La solución aproximada A contribuye $\frac{1}{b}$ o más por cada elemento incluído en la solución
  - La solución sin redondear aporta a lo sumo 1 por cada elemento incluído, es decir, como mucho b veces respecto al valor redondeado y por lo tanto $A \leq b.s$
- Como ya habíamos acotado la solución por Programación Lineal respecto al óptimo, la solución aproximada es a lo sumo b veces el óptimo. Es decir: $\frac{A(I)}{z(I)} \leq b \Rightarrow A(I) \leq b.z(I)$ y por lo tanto, es una $b - aproximacion$

2.3.2. Algoritmo *Greedy*

Como segunda propuesta de aproximación, planteamos un algoritmo *Greedy*. La regla sencilla que el algoritmo aplicará iterativamente será elegir el jugador que aparece con mayor frecuencia entre todos los conjuntos que aún no fueron cubiertos por las selecciones previas.

El algoritmo se ejecutará mientras quede algún conjunto sin cubrir y repetirá las siguientes operaciones:

- Calcula la frecuencia de apariciones de cada jugador entre los conjuntos restantes.
- Selecciona el de máxima frecuencia y agregarlo a la solución.
- Remueve todos los conjuntos cubiertos por ese jugador

Cuando todos los conjuntos fueron cubiertos, habrá encontrado un *Hitting-Set* que devolverá como solución.

Proponemos la siguiente implementación:

```
def calcular_frecuencias(conjuntos):
    frecuencias = {}
    for conjunto in conjuntos:
        for jugador in conjunto:
            frecuencias[jugador] = frecuencias.get(jugador, 0) + 1
    return frecuencias

def buscar_jugador_mas_pedido(conjuntos):
    frecuencias = calcular_frecuencias(conjuntos)
    return max(frecuencias, key=frecuencias.get)

def conjuntos_cubiertos(jugador, conjuntos):
    cubiertos = []
    for conjunto in conjuntos:
        if jugador not in conjunto:
            continue
        cubiertos.append(conjunto)
    return cubiertos

def hitting_set(conjuntos):
    conjuntos_restantes = set(frozenset(c) for c in conjuntos)
    hitting_set = set()
    while conjuntos_restantes:
        jugador = buscar_jugador_mas_pedido(conjuntos_restantes)
        hitting_set.add(jugador)
        cubiertos = conjuntos_cubiertos(jugador, conjuntos_restantes)
        conjuntos_restantes.difference_update(cubiertos)
    return hitting_set
```

La función recibe como parámetro una lista de `set`.

Si analizamos y detallamos la complejidad del algoritmo:

- Definimos:
  - m: cantidad de conjuntos
  - n: cantidad total de jugadores
- `calcular_frecuencias`: $\mathcal{O}(m.n)$: Por cada uno de los m conjuntos, se itera sobre todos sus elementos para contar sus frecuencias usando un diccionario que permite realizar consultas y actualizaciones en $\mathcal{O}(1)$. En el peor caso, cada conjunto contiene los n elementos.
- `buscar_jugador_mas_pedido`: $\mathcal{O}(m.n)$: Calcular la frecuencia de apariciones es $\mathcal{O}(m.n)$ y buscar el máximo entre todos los jugadores, $\mathcal{O}(n)$
- `hitting_set`: copiar los conjuntos con cada uno de sus elementos: $\mathcal{O}(m.n)$. El bloque `while` se ejecutará a lo sumo m veces si los conjuntos son todos disjuntos entre sí y por lo tanto se realizarán m operaciones $\mathcal{O}(m.n)$.

Por lo tanto, el algoritmo *Greedy* encuentra una solución de forma eficiente, pero no necesariamente la óptima.

Mostramos que el algoritmo no es óptimo con un ejemplo:

- Dado el conjunto $A = \{A, B, C, D, E\}$ y los subconjuntos:
  - $S_1 = \{A, B, C\}$
  - $S_2 = \{A, C, D\}$
  - $S_3 = \{A, D, E\}$
  - $S_4 = \{A, B, E\}$
  - $S_5 = \{B, C\}$
  - $S_6 = \{D, E\}$

Iniciamos con un conjunto solución C vacío.

Al calcular la cantidad de apariciones de cada elemento, se puede armar la siguiente tabla:

Elemento	Apariciones
A	4
B	2
C	3
D	3
E	3

Cuadro 1: Apariciones de cada elemento - 1º iteración

El algoritmo seleccionará al elemento A para incluirlo en el *Hitting-Set* y removerá a los conjuntos S_1, S_2, S_3 *y* S_4. Entonces, hasta ahora, $C = \{A\}$

En la siguiente iteración quedarán dos conjuntos sin cubrir y al recalcular la frecuencia de apariciones observamos que no hay repeticiones de los elementos restantes:

Elemento	Apariciones
B	1
C	1
D	1
E	1

Cuadro 2: Apariciones de cada elemento - 2º iteración

El algoritmo tomará alguno de ellos, ejemplificamos con B, lo agregará a la solución y removerá el conjunto S_5. La solución parcial será $C = \{A, B\}$

En la tercera iteración quedará solo un conjunto y el recálculo de las apariciones será:

Elemento	Apariciones
D	1
E	1

Cuadro 3: Apariciones de cada elemento - 3º iteración

El algoritmo seleccionará alguno de los dos elementos para cubrir el último conjunto y lo agregará a C. elegimos D. Como ya no quedan conjuntos sin cubrir, termina la ejecución y devuelve la solución encontrada: $C = \{A, B, D\}$

Observamos que el conjunto retornado cumple con la condición de *Hitting-Set*, sin embargo, es posible encontrar una mejor solución con menos elementos: $C = \{B, D\}$

- $S_1 = \{A, \mathbf{B}, C\}$
- $S_2 = \{A, C, \mathbf{D}\}$
- $S_3 = \{A, \mathbf{D}, E\}$
- $S_4 = \{A, \mathbf{B}, E\}$
- $S_5 = \{\mathbf{B}, C\}$
- $S_6 = \{\mathbf{D}, E\}$

3. Mediciones

Generamos set de datos con valores pseudo aleatorios usando el módulo `random` de `Python` que exportamos en archivos de texto con el módulo `csv`.

A partir de una lista de jugadores, armamos sets variando la cantidad de conjuntos entre cinco y doscientos conjuntos de hasta diez elementos cada uno, seleccionando pseudoaleatoriamente entre treinta jugadores. También generamos datos de mayor volumen para corroborar el comportamiento de los algoritmos aproximados. Para poder comparar los tiempos de ejecución de los diferentes algoritmos, medimos el tiempo diez veces para cada set de datos usando el módulo `time`. Estos valores se promediaron y el resultado lo usamos para graficar usando la biblioteca `Seaborn`.

3.1. Tiempo de ejecución vs cantidad de conjuntos

3.1.1. *Backtracking* vs Programación Lineal Entera

Al comparar los resultados obtenidos para ambos algoritmos, notamos que para sets de datos de hasta 100 conjuntos, el tiempo de ejecución es similar, e incluso mejor para el algoritmo de

Backtracking. Sin embargo, superados los cien conjuntos, el algoritmo de Programación Lineal Entera tiene un mejor rendimiento, que es más notorio a medida que aumenta la cantidad de conjuntos contemplados.

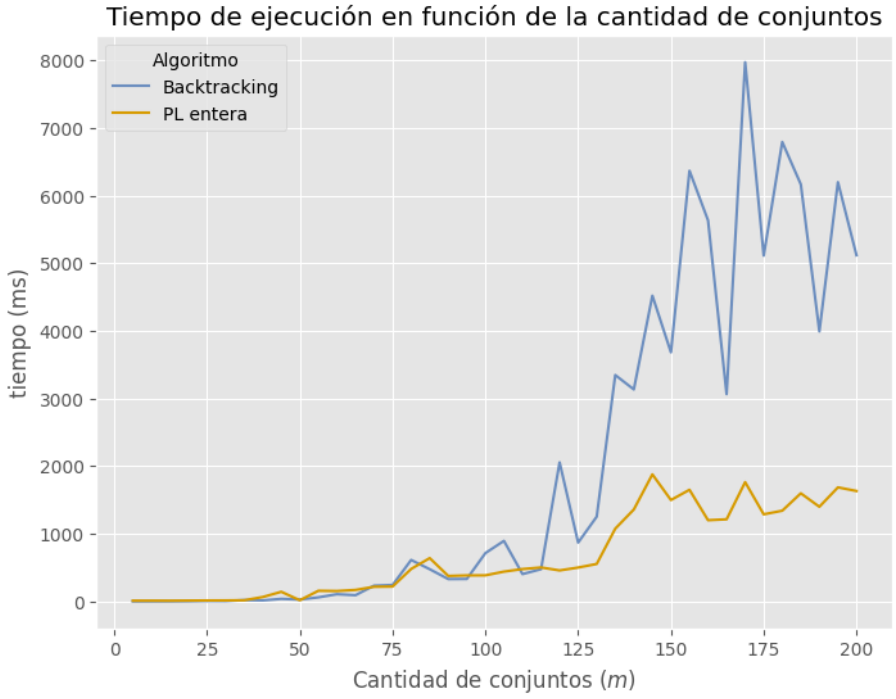


Figura 1: Tiempo de ejecución de los algoritmos de *Backtracking* y Programación Lineal Entera

También notamos que si los conjuntos son parecidos entre sí, el algoritmo de *Backtracking* ejecuta con mayor velocidad que el algoritmo de Programación Lineal Entera. Para poder hacer la comparación, variamos la selección de jugadores posibles que puedan contener los conjuntos entre diez, veinte y treinta jugadores distintos y observamos que con diez y veinte jugadores, *Backtracking* es más rápido, mientras que Programación Lineal Entera lo es para los sets de datos que varían entre treinta jugadores, sobre todo a medida que aumenta la cantidad de conjuntos.

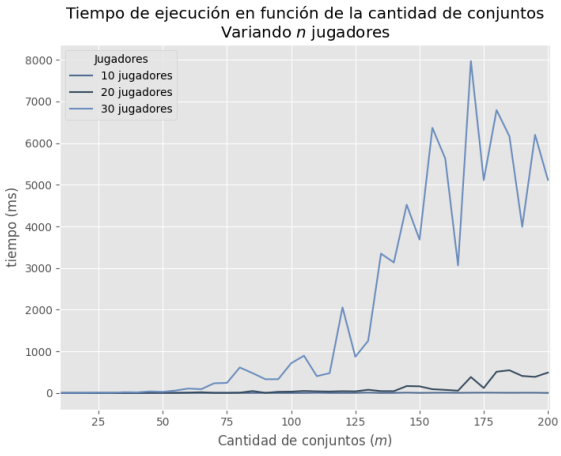


Figura 2: Tiempo de ejecución del algoritmo de *Backtracking*

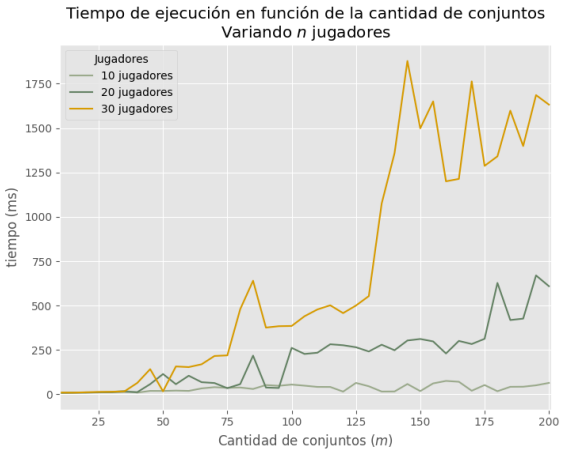


Figura 3: Tiempo de ejecución del algoritmo de Programación Lineal Entera

Esta diferencia es aún más evidente si forzamos a que todos los conjuntos compartan a un jugador, donde el algoritmo de *Backtracking* en todos los casos resuelve más rápido.

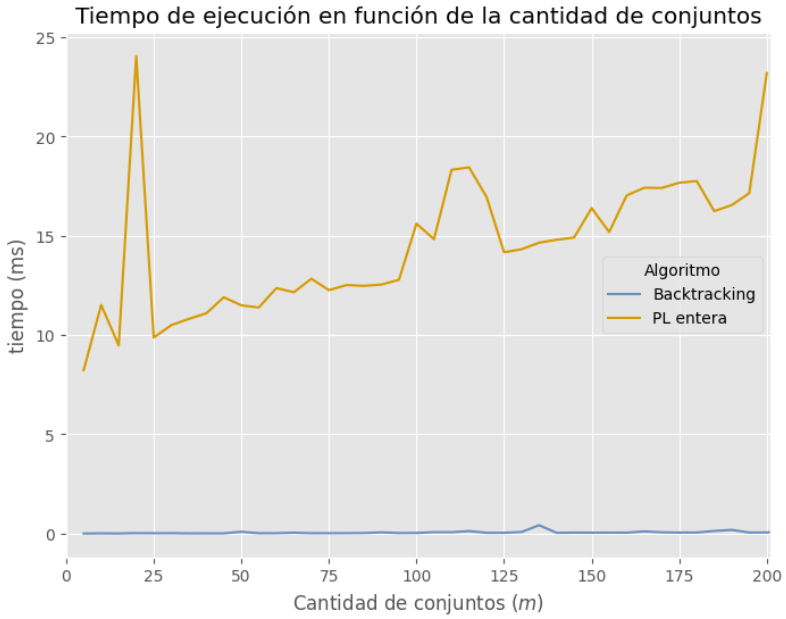


Figura 4: Tiempo de ejecución de los algoritmos de *Backtracking* y Programación Lineal Entera

3.1.2. Programación Lineal Entera vs Programación Lineal

Al comparar ambos algoritmos de Programación Lineal, observamos que los resultados son los esperados y que el algoritmo aproximado es más eficiente y que esa diferencia es evidenciable a medida que aumenta la cantidad de conjuntos.

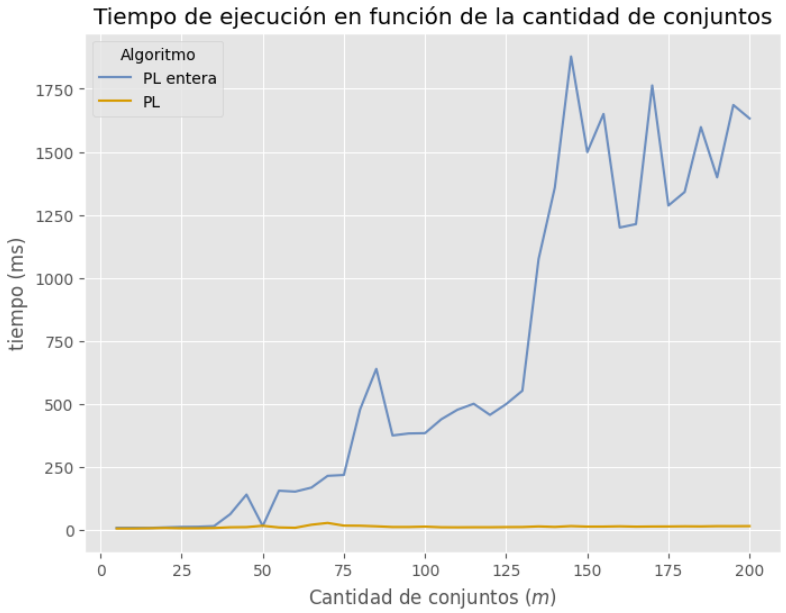


Figura 5: Tiempo de ejecución de los algoritmos de Programación Lineal Entera y Programación Lineal

3.1.3. Programación Lineal vs Greedy

En cuanto a los dos algoritmos aproximados, si bien ambos son eficientes, el algoritmo *Greedy* resuelve en un tiempo menor comparado al de Programación Lineal.

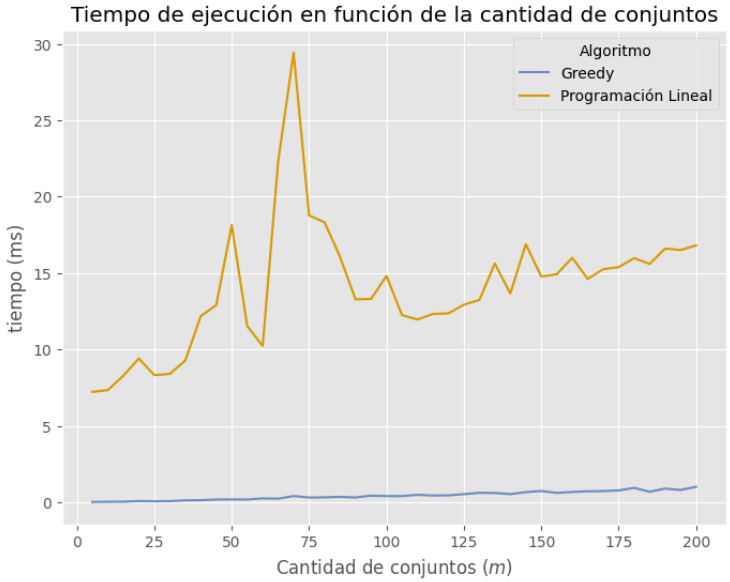


Figura 6: Tiempo de ejecución de los algoritmos de Programación Lineal y *Greedy*

3.2. Relación entre soluciones óptimas y aproximadas

Para corroborar la cota calculada en la seccion 2.3.1, calculamos la relación $\frac{A}{z}$, donde A es la solución aproximada obtenida por el algoritmo de Programación Lineal y z es la solución óptima, y verificamos que el ratio sea menor o igual al valor de b que, para los sets de datos que generamos, es diez (los conjuntos tienen como máximo diez elementos).

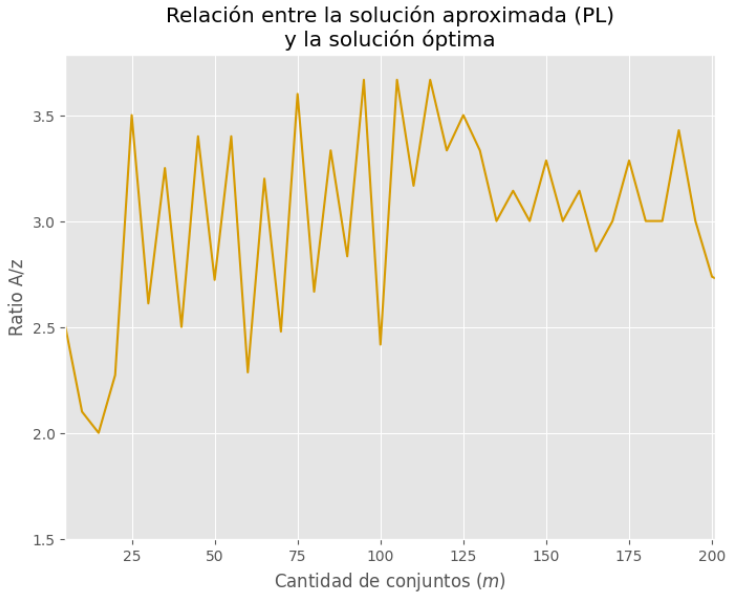


Figura 7: Relación entre la solución óptima y la solución aproximada por Programación Lineal

Observamos que el ratio obtenido no supera el valor de 3,66 $\leq$ b, que verifica el análisis realizado, donde demostramos que el algoritmo es una $b - aproximacion$.

Al repetir las mediciones para el algoritmo *Greedy*, notamos que logra una mejor aproximación ya que la relación entra la solución aproximada y la óptima no supera el valor de 1.357.

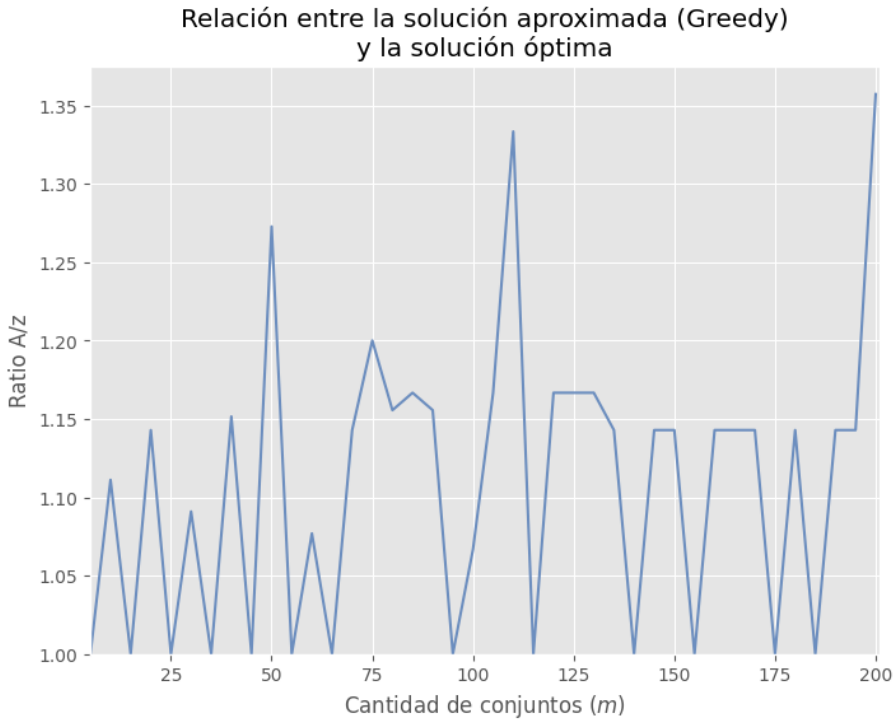


Figura 8: Relación entre la solución óptima y la solución aproximada por *Greedy*

Comparamos ambos algoritmos con volúmenes de datos más grandes, de hasta cinco mil conjuntos. Podemos corroborar por medio de los gráficos que la cota del algoritmo de Programación Lineal se cumple, y notamos que el algoritmo *Greedy* logra una mejor aproximación con un ratio, en la mayoría de los casos, muy cercano a 1.

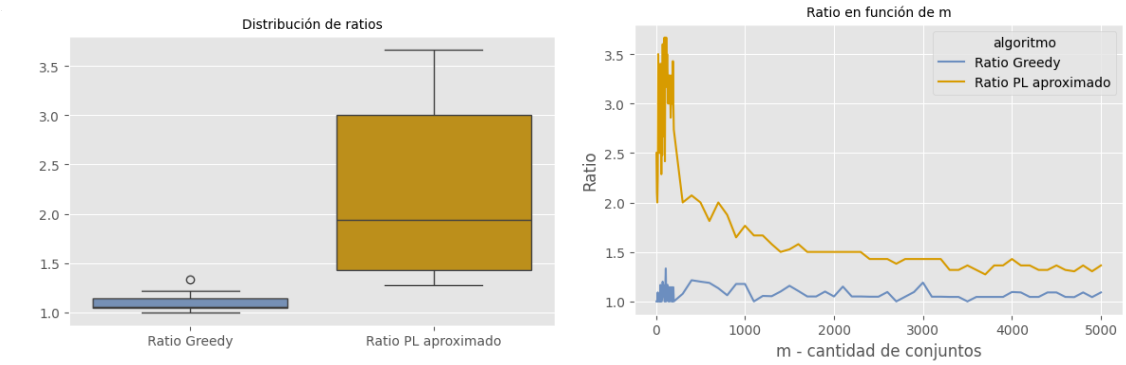


Figura 9: Comparación de cotas de aproximación de los algoritmos

4. Conclusiones

En este trabajo práctico pudimos analizar la complejidad del problema *Hitting-Set* y probar que es un problema $NP - Completo$.

Frente a la dificultad que presenta la resolución del problema, planteamos cuatro algoritmos para resolverlo tanto de forma óptima como aproximada, aplicando diversas técnicas de diseño. Al comparar los resultados obtenidos con cada algoritmo, concluimos que, si bien *Backtracking* y Programación Lineal Entera tienen una complejidad exponencial, resuelven óptimamente y, para volúmenes de datos chicos, ambos resuelven en tiempos razonables. En particular, si se trabaja con pocos conjuntos y sus elementos son similares, *Backtracking* resultó más rápido.

Para volúmenes de datos de mayor tamaño que resultan intratables para los algoritmos óptimos, el algoritmo *Greedy* consigue un resultado aproximado más cercano al óptimo que el de Programación Lineal y resuelve eficientemente.