

TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 3

Problemas NP-Completos

Borja Garibotti
106124

Mauro Rizzi
104761

Julieta Taras
104728

1. Problemas NP-Completo

El problema consiste en encontrar el conjunto mínimo de jugadores que satisface los deseos de la prensa. Como datos de entrada tenemos los deseos de la prensa.

1.1. Demostración que Hitting Set es un problema NP

1.1.1. Definición de problemas NP

Los problemas NP son aquellos para los que existe un certificador eficiente, es decir, aquellos que se pueden validar en tiempo polinomial.

1.1.2. Demostración

Introducción

Este problema consiste en determinar si, dado un conjunto A de n elementos, m subconjuntos B_1, B_2, \dots, B_m de A , y un número entero k , existe un subconjunto C de A con $|C| \leq k$ tal que $C \cap B_i \neq \emptyset$ para cada i en el rango de 1 a m , es decir, que el subconjunto C contenga al menos un elemento de cada subconjunto B_i .

La pertenencia a NP se demuestra que, dada una solución propuesta C , es posible verificar en tiempo polinómico si cumple con las condiciones del problema. Esto implica verificar el tamaño de C para asegurarse de que $|C| \leq k$, lo cual es un proceso constante, y verificar la intersección de C con cada B_i , garantizando que $C \cap B_i \neq \emptyset$ para cada i , en tiempo polinómico.

Verificación

Tenemos dos condiciones que corroborar:

- *Tamaño del conjunto:* Verificar que $|C| \leq k$.
- *Intersección con cada B_i :* Verificar que $C \cap B_i \neq \emptyset$ para cada i en el rango de 1 a m .

Para la verificación del tamaño del conjunto, la operación de comparación $|C| \leq k$ toma $O(k)$ pasos, ya que se itera sobre C un máximo de k veces y se rechaza la solución si al llegar a k no ha finalizado la iteración, indicando que el conjunto supera los k elementos. Como k es constante, $O(k)$ puede expresarse como $O(1)$.

Para la verificación de la intersección con cada B_i , se realiza una búsqueda o comparación de cada elemento de C con los elementos de B_i . Dado que $|C| \leq k$ y $|B_i|$ está acotado por el tamaño de A , la búsqueda toma $O(k \cdot n)$ pasos y, como se tienen m subconjuntos B_i , la verificación de cada intersección es $O(k \cdot n \cdot m)$. Como k es constante, puede expresarse como $O(n \cdot m)$.

En total, el tiempo de verificación es de $O(1 + n \cdot m)$, equivalente a $O(n \cdot m)$, lo cual es polinómico. Por lo tanto, la verificación del Hitting-Set Problem ocurre en tiempo polinómico y, en consecuencia, el problema está en NP.

1.2. Demostración que Hitting Set es un problema NP-Completo

1.2.1. Definición de problemas NP-completo

$$X \in \text{NP-Completo} \Leftrightarrow X \in \text{NP} \wedge \forall Y \in \text{NP} (Y \leq_p X).$$

Esto quiere decir que: Sabemos que un problema es NP-Completo si y solo si es un problema que está en NP y también sucede que para cualquier problema que está en NP se puede reducir polinomialmente a dicho problema NP-Completo.

1.2.2. Demostración

Utilizaremos el problema de satisfacibilidad con cláusulas de 3 variables (3-SAT) como ejemplo de un problema NP-completo. Este problema plantea la pregunta de si, dada una expresión donde cada cláusula contiene tres variables booleanas conectadas por operadores OR, es posible encontrar una asignación de valores a las variables que satisfaga todas las cláusulas. Para demostrar que Hitting Set es un problema NP-Completo, podemos reducir polinómicamente 3-SAT a este problema.

Dada una instancia de 3-SAT, construimos una fórmula booleana F para Hitting-Set Problem de la siguiente manera:

- Cada variable en F se convierte en un conjunto en el conjunto universal X .
- Cada cláusula en F se convierte en un subconjunto en B_i .

Entonces, para resolver SAT, necesitamos encontrar un conjunto C tal que $|C| \leq k$ y $C \cap B_i \neq \emptyset$ para cada cláusula B_i en F .

Supongamos que tenemos la siguiente instancia de 3-SAT:

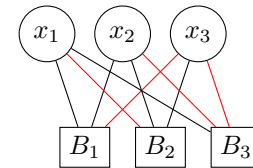
$$F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3)$$

Ahora, construimos la instancia correspondiente del *Hitting-Set Problem* utilizando la reducción mencionada:

Conjunto universal: $X = \{x_1, \neg x_1, x_2, \neg x_2, x_3, \neg x_3\}$.

Subconjuntos B_i :

$$\begin{aligned} B_1 &= \{x_1, x_2, \neg x_3\} \\ B_2 &= \{\neg x_1, x_2, x_3\} \\ B_3 &= \{x_1, \neg x_2, \neg x_3\} \end{aligned}$$



Entonces, para resolver el *Hitting-Set Problem*, necesitamos encontrar un conjunto de elementos seleccionados C tal que $|C| \leq k$ y $C \cap B_i \neq \emptyset$ para cada B_i en F .

Como F está compuesto por tres términos, suponemos que $k = 3$, ya que una solución no puede contener una variable que es positiva y negativa al mismo tiempo.

Un conjunto de elementos seleccionados posible podría ser $C = \{x_1, x_2\}$:

$$x_1 = \text{True}, \quad x_2 = \text{True}, \quad x_3 = \text{False}$$

Verificamos si esta asignación satisface todas las cláusulas de 3-SAT:

$$\begin{aligned} B_1 : (x_1 \vee x_2 \vee \neg x_3) &= (\text{True} \vee \text{True} \vee \text{True}) = \text{True} \\ B_2 : (\neg x_1 \vee x_2 \vee x_3) &= (\text{False} \vee \text{True} \vee \text{False}) = \text{True} \\ B_3 : (x_1 \vee \neg x_2 \vee \neg x_3) &= (\text{True} \vee \text{False} \vee \text{False}) = \text{True} \end{aligned}$$

La asignación satisface todas las cláusulas de 3-SAT ✓.

A su vez, vemos que:

- **Tamaño del conjunto:** $|C| = 2 \leq 3$ (cumple con $k = 3$).
- **Intersección con cada B_i :**

$$C \cap B_1 = \{x_1\} \neq \emptyset$$

$$C \cap B_2 = \{x_2\} \neq \emptyset$$

$$C \cap B_3 = \{x_1\} \neq \emptyset$$

Por lo tanto, $C = \{x_1, x_2\}$ es un conjunto de elementos seleccionados válido para la instancia del *Hitting-Set Problem*, pero dado que 3-SAT es un problema de decisión nos basta con saber que hay solución válida posible.

En resumen, hemos demostrado que el *Hitting-Set Problem* está en NP y que es NP-completo al mostrar una reducción polinómica desde 3-SAT.

2. Algoritmo de Backtracking

2.1. Introducción

El algoritmo consiste en explorar un espacio exponencial de soluciones al problema probando si agregando un jugador a la lista de jugadores convocados se logra satisfacer el Hitting Set Problem. Al ser esto una solución de fuerza bruta buscamos optimizar el algoritmo a través de varias condiciones de corte para evitar explorar soluciones que no resuelven el problema o son menos óptimas que una solución ya encontrada.

Para asistir con estas optimizaciones decidimos representar el problema con una clase de manera que nos facilite guardar en el estado interno de la misma la mejor solución encontrada.

Se aplicaron las siguientes optimizaciones:

- Como el algoritmo va probando las soluciones desde la menor cantidad de jugadores a la mayor pudimos hacer una optimización muy grande cortando la rama recursiva una vez que encontramos una solución menor a la almacenada para el problema (o cualquier solución válida en el caso de todavía no tener una).
- Reducimos en cada llamado recursivo la lista de deseos con aquellos deseos que todavía quedan por satisfacer (excluyendo los deseos que ya tienen al menos un jugador que es parte de la solución actual)
- Agregamos una lista con los jugadores descartados de manera que podamos detectar cuando algún deseo ya no se puede cumplir con los jugadores que quedan sin descartar.

2.2. Complejidad algorítmica

La complejidad algorítmica es $O(2^n)$ ya que en el peor de los casos este algoritmo termina corriendo como su base de fuerza bruta, donde se prueban todas las combinaciones posibles de jugadores hasta encontrar una solución válida.

2.3. El algoritmo

```
1 def hitting_set_backtracking(deseos):
2     return HittingSetBacktracking(deseos).buscar_solucion()
3
4 class HittingSetBacktracking():
5     def __init__(self, deseos_prensa):
6         self._solucion = None
7         self._deseos_prensa = deseos_prensa
8
9     def buscar_solucion(self):
10        self._buscar_solucion(self._deseos_prensa, [], [])
11        return list(self._solucion)
12
13    def _buscar_solucion(self, deseos_pendientes, convocados, descartados):
14        if not self._puede_ser_solucion_optima(convocados):
15            return
16
17        deseos_pendientes = _deseos_pendientes(deseos_pendientes, convocados)
18
19        if not any(deseos_pendientes):
20            self._marcar_mejor_solucion(convocados)
21            return
22
23        jugador = _jugador_siguiete(deseos_pendientes, descartados)
24
25        if jugador == None:
26            return
27
28        if not _todos_los_deseos_se_pueden_cumplir(deseos_pendientes, descartados):
29            return
30
31        self._buscar_solucion(deseos_pendientes, convocados, [*descartados, jugador
32        ])
33        self._buscar_solucion(deseos_pendientes, [*convocados, jugador],
34        descartados)
35
36    def _marcar_mejor_solucion(self, nueva_solucion):
37        if self._puede_ser_solucion_optima(nueva_solucion):
38            self._solucion = nueva_solucion
39
40    def _puede_ser_solucion_optima(self, convocados):
41        return not self._solucion or len(convocados) < len(self._solucion)
42
43    def _deseos_pendientes(deseos_prensa, convocados):
44        deseos_restantes = []
45
46        for deseo in deseos_prensa:
47            cumplido = any(filter(lambda jugador: jugador in deseo, convocados))
48
49            if not cumplido:
50                deseos_restantes.append(deseo)
51
52        return deseos_restantes
53
54    def _jugador_siguiete(deseos_pendientes, descartados):
55        for deseo in deseos_pendientes:
56            for jugador in deseo:
57                if jugador in descartados:
58                    continue
59                return jugador
60        return None
61
62    def _todos_los_deseos_se_pueden_cumplir(deseos_prensa, descartados):
63        for deseo in deseos_prensa:
64            if any(filter(lambda jugador: not jugador in descartados, deseo)):
65                continue
66        return False
67    return True
```

3. Algoritmo de programación lineal entera

3.1. Introducción

Este algoritmo hace uso de la biblioteca PuLP de Python para resolver el problema mediante Programación Lineal. Consiste en definir qué problema debe resolver el modelo y cuáles son sus restricciones a través de ecuaciones matemáticas.

Para comenzar, se definen las variables de decisión creando un diccionario x con los jugadores como variables binarias: se utiliza la lógica binaria con 1s y 0s para representar si un jugador $x[\text{jugador}]$ es convocado o no.

Después, se define el problema de programación lineal entera cuya función objetivo busca minimizar la suma de las variables x para así minimizar el tamaño del conjunto de los jugadores convocados. Además, por cada deseo se agrega una restricción donde la suma de las variables x correspondientes a los jugadores del deseo debe ser mayor o igual a 1 para asegurar que al menos un jugador de cada conjunto sea convocado.

Una vez terminado de definir el modelo, se utiliza el metodo `solve()` para encontrar la solución óptima, y se obtienen la lista de jugadores convocados revisando las variables x que tienen un valor de 1 en la solución óptima.

3.2. El algoritmo

```
1 from pulp import LpMinimize, LpProblem, LpVariable, lpSum
2
3 def hitting_set_programacion_lineal(deseos):
4     deseos = [set(deseo) for deseo in deseos]
5
6     jugadores = set.union(*deseos)
7     x = LpVariable.dicts('x', jugadores, cat='Binary')
8
9     hitting_set = LpProblem("HittingSet", LpMinimize)
10    hitting_set += lpSum(x[jugador] for jugador in jugadores)
11    for deseo in deseos:
12        hitting_set += lpSum(x[jugador] for jugador in deseo) >= 1
13
14    hitting_set.solve()
15    convocados = [jugador for jugador in jugadores if x[jugador].value() == 1]
16    return convocados
```

4. Algoritmo de aproximación Bilardo

4.1. Introducción

Este algoritmo es una modificación del algoritmo anterior buscando obtener una aproximación en vez de una solución óptima utilizando valores reales en vez de enteros, permitiendo que se calcule la probabilidad de que el jugador participe en una solución.

Finalmente, decide si el jugador se elige o no en base a si su valor de participación se encuentra entre $1/b$ y 1 (donde b = La cantidad de jugadores del deseo más grande).

4.2. Análisis de la aproximación

Sea I una instancia cualquiera del Hitting-Set Problem, y $z(I)$ una solución óptima S para dicha instancia, y sea $A(I)$ la solución aproximada S^* , se define $\frac{A(I)}{z(I)} \leq r(A)$ para todas las instancias posibles.

Supongamos que la elección de cada jugador x_i tiene un costo w_i . Necesitamos minimizar $w(S) = \sum_{i \in S} w_i$.

Por cada deseo de prensa, tenemos

$$\sum w_j x_j \geq 1 \forall x_j \in B_i$$

$$\text{y } 0 \leq x_i \leq 1, i \in S$$

Si tomamos como parte de la solución todo $x_i \geq \frac{1}{b}$, entonces

$$w(S^*) = \sum_i w_i x_i^* \geq \sum_{i \in S} w_i x_i^* \geq \frac{1}{b} \sum_{i \in S} w_i = \frac{1}{b} w(S)$$

$$b \cdot w(S^*) \geq w(S)$$

$$b \cdot w(A(I)) \geq w(z(I))$$

La solución aproximada $A(I)$ es como mucho b veces más grande que la solución óptima $z(I)$.

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

4.3. El algoritmo

```
1 from pulp import LpMinimize, LpProblem, LpVariable, lpSum
2
3 def hitting_set_aproximacion_bilardo(deseos):
4     deseos = [set(deseo) for deseo in deseos]
5
6     b = max(len(deseo) for deseo in deseos)
7     jugadores = set.union(*deseos)
8     x = LpVariable.dict("x", jugadores, lowBound=0, upBound=1, cat='Continuous')
9
10    hitting_set_aproximado = LpProblem(name="HittingSetAproximado", sense=
11    LpMinimize)
12    hitting_set_aproximado += lpSum(x[jugador] for jugador in jugadores)
13    for deseo in deseos:
14        hitting_set_aproximado += lpSum(x[jugador] for jugador in deseo) >= 1
15
16    hitting_set_aproximado.solve()
17    convocados = [jugador for jugador in jugadores if x[jugador].value() >= 1/b]
18    return convocados
```

5. Algoritmo Greedy

5.1. Introducción

Este algoritmo aproxima el resultado eligiendo en cada iteración al jugador más deseado (el que más se repite entre todos los deseos aún sin cumplir), de manera que se logra satisfacer siempre la mayor cantidad de deseos posibles en cada iteración.

5.2. Análisis de la aproximación

En un principio podemos suponer que todos los jugadores tienen la misma probabilidad de ser elegidos, donde $p(x_i) \geq \frac{1}{n} \forall x_i \in A$ y $\sum_i p(x_i) = 1$.

Cada vez que se busca al jugador más deseado entre los deseos restantes, se está eligiendo al jugador que tiene más probabilidades de pertenecer a la solución. Sin embargo, pueden haber pequeños deseos que tengan jugadores pocos frecuentes por lo que son más difíciles de satisfacer.

Por ejemplo, si nuestros deseos son: $\{\{1,2\}, \{1,2\}, \{1,3\}, \{1,3\}, \{2,4\}, \{3,5\}\}$ Podemos ver que el 1 es el que más se repite (4 veces) y será el primero en ser seleccionado. Esto nos dejará los deseos 2,4, 3,5 sin cumplir, por lo que necesitaremos tomar todavía 2 números más, por lo que el Greedy nos quedará de tamaño 3. Podemos comprobar fácilmente que una solución óptima habría sido 2,3.

Lo mismo sucedería en el caso que contemos con deseos con la siguiente estructura:

```
{  
{1,2}, {1,2}, {1,3}, {1,3}, {2,4}, {3,5},  
{6,7}, {6,7}, {6,8}, {6,8}, {7,9}, {8,10},  
{11,12}, {11,12}, {11,13}, {11,13}, {12,14}, {13,15},  
}
```

La solución Greedy sería: $\{1,6,11,2,3,7,8,12,13\}$ (9), mientras una solución óptima sería $\{2,3,7,8,12,13\}$ (6).

Sea T el tamaño de una solución óptima, y T^* el tamaño de una solución aproximada, podemos concluir que siempre $T \leq T^* \leq n$.

5.3. Complejidad algorítmica

La complejidad algorítmica es $O(m^2 * n)$ porque en el peor de los casos se itera dos veces sobre los deseos de prensa, es decir, en el caso de que en cada iteración solo se cumpla un deseo se van a realizar m iteraciones. Dentro de las cuales se itera por todos los deseos restantes y por cada deseo se itera hasta por n jugadores buscando el que mas deseos cumpla sumando $O(n)$ a la complejidad.

5.4. El algoritmo

```
1 def hitting_set_greedy(deseos_prensa):  
2     convocados = []  
3  
4     while len(deseos_prensa) > 0:  
5         cant_jugadores = {}  
6         max = None  
7  
8         for deseo in deseos_prensa:  
9             for jugador in deseo:  
10                cant_jugadores[jugador] = cant_jugadores.get(jugador, 0) + 1  
11                if not max or cant_jugadores[jugador] > cant_jugadores[max]:  
12                    max = jugador  
13  
14                convocados.append(max)  
15                deseos_prensa = _deseos_restantes(deseos_prensa, convocados)  
16     return convocados
```

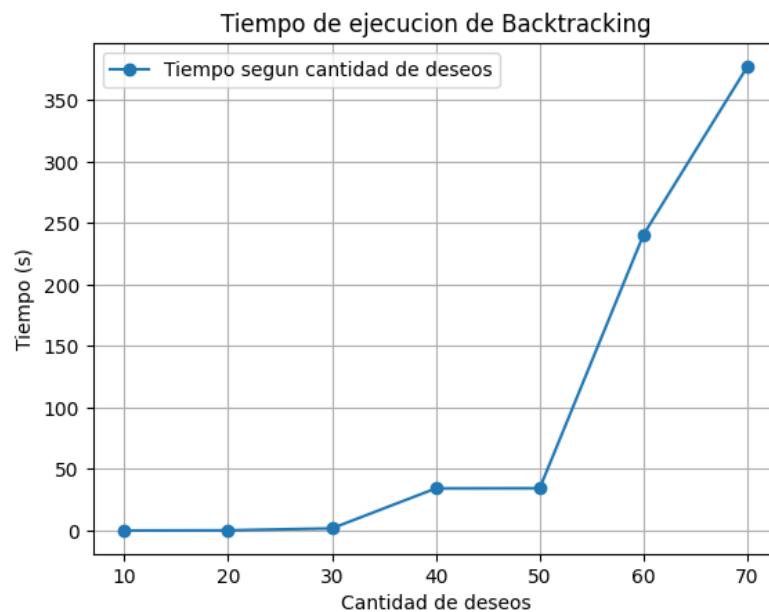

6. Mediciones

6.1. Metodología

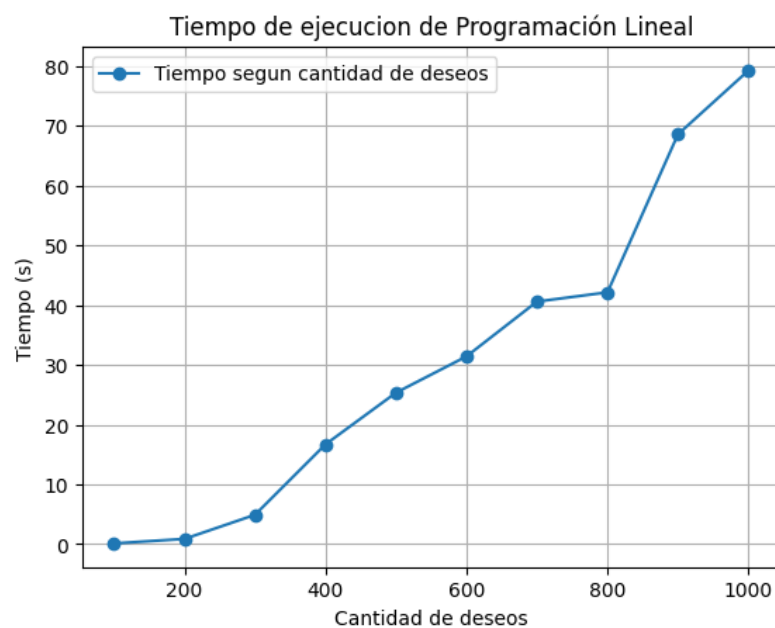
Creamos diferentes sets de datos utilizando el notebook `notebooks/Generador de archivos.ipynb`. Los datos generados se encuentran en la misma carpeta como archivos de texto `.txt`.

Estos sets de datos nos permitieron realizar las mediciones de los tiempos de ejecución, con su análisis y grafos. Esto se puede encontrar en el archivo `notebooks/Algoritmos.ipynb`.

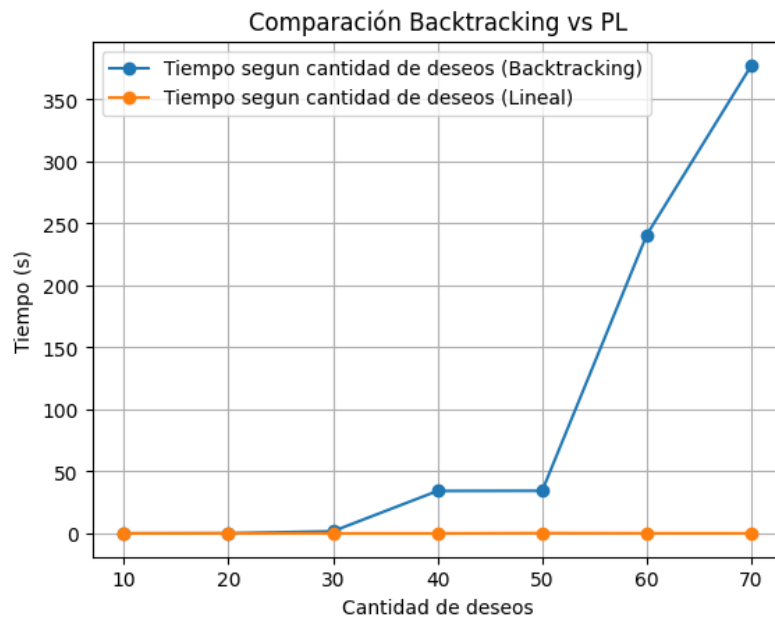
6.2. Resultados



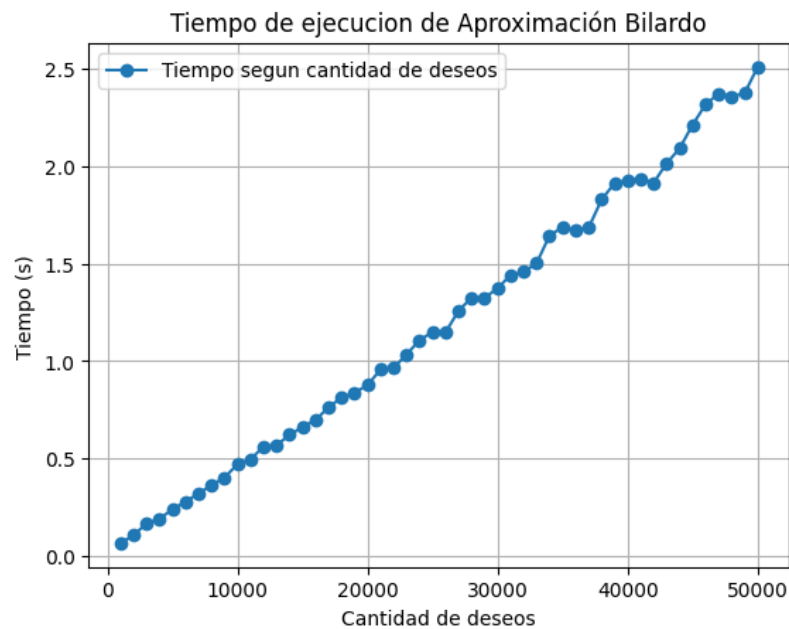
Podemos observar que, a medida que se va aumentando la cantidad de deseos, el tiempo de ejecución del algoritmo de Backtracking aumenta exponencialmente.



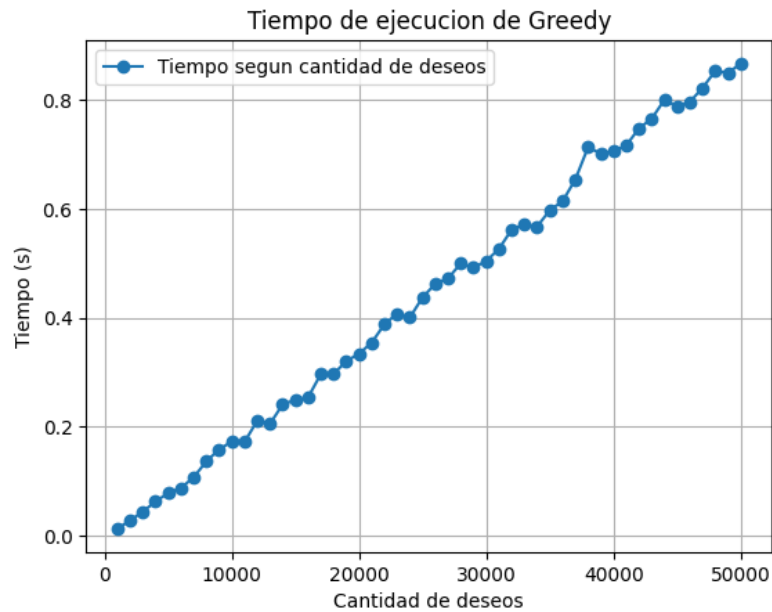
Podemos observar que, la curva con el algoritmo con Programación Lineal es más gradual que en comparación con la curva del algoritmo de Backtracking.



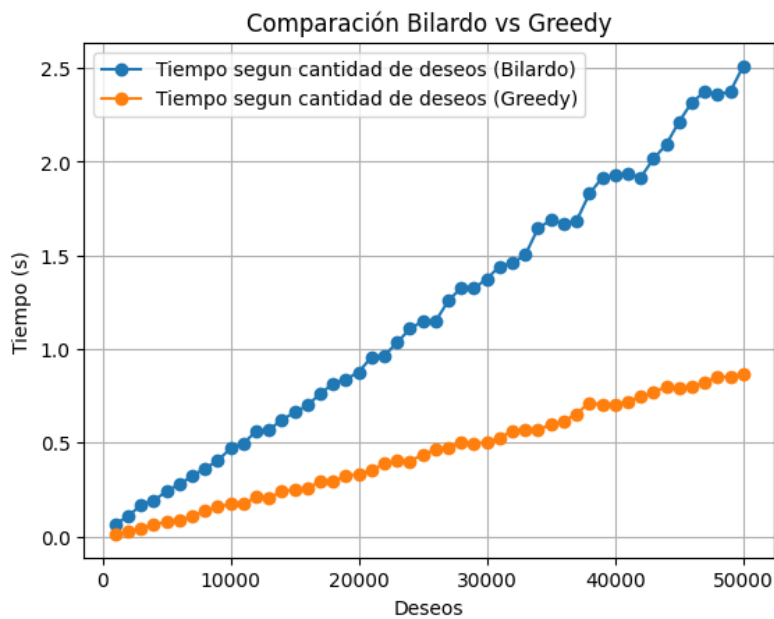
Si comparamos los resultados de evaluar ambos algoritmos con los mismos sets de datos, observamos que el algoritmo de Programación Lineal se desempeña mejor que el algoritmo de Backtracking, el cual con pocos sets de datos aumenta exponencialmente sus tiempos de ejecución.



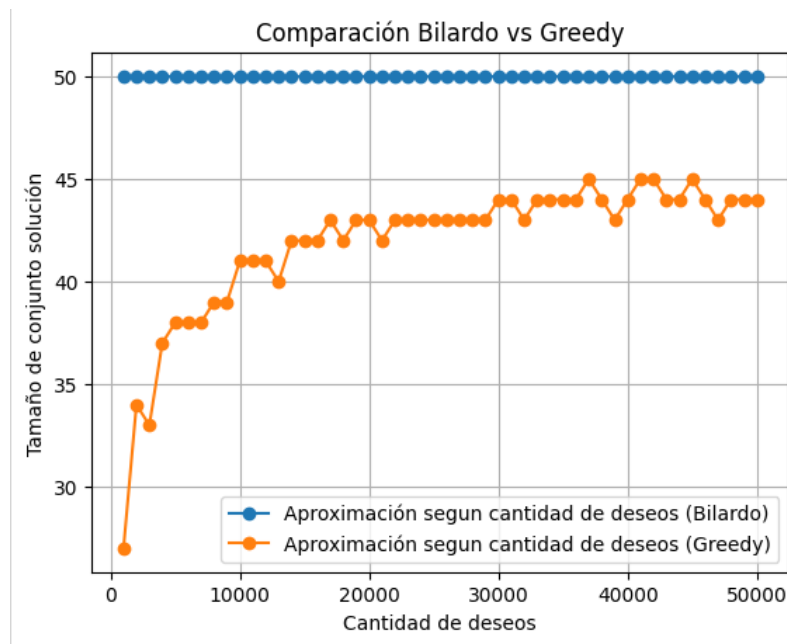
Se puede observar que el algoritmo de aproximación Bilardo reduce significativamente los tiempos de ejecución, a cambio de obtener una solución que puede diferir bastante con la solución óptima.



También con este algoritmo de aproximación sucede que, se reduce significativamente el tiempo de ejecución a cambio de obtener una solución que puede diferir de la solución óptima.



Podemos observar como los diferentes algoritmos de aproximación difieren en su complejidad, mostrando que el algoritmo Greedy tiene un tiempo de ejecución menor al algoritmo de Bilardo según la cantidad de deseos.



Vemos que el algoritmo de Bilardo siempre devuelve más jugadores según cantidad de deseos a comparación que el algoritmo Greedy, por lo tanto, pareciera ser mas óptimo este último.

7. Conclusiones

Podemos concluir que el algoritmo de Programación Lineal nos permitió encontrar una solución óptima evitando explorar un espacio exponencial de soluciones como se hace con el algoritmo de Backtracking, por lo cuál, gracias a esta metodología pudimos reducir significativamente los tiempos de ejecución del algoritmo.

También encontramos que el algoritmo de aproximación Greedy que implementamos, en promedio, tiene mejor rendimiento que el algoritmo por aproximación Bilardo. Tanto en los tiempos de ejecución como hallando soluciones más próximas a la óptima.

Con grandes sets de datos, obtener la solución óptima requiere aumentar los tiempos de ejecución, por lo cuál, puede ser preferible utilizar un algoritmo de aproximación sabiendo que puede no obtener una solución óptima.