Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

14 de diciembre de 2023

Juan Díaz
103488

Juan Díaz
108183

Juan Muñoz
106699

1. Introducción

En el presente trabajo práctico se demuestra que el *Hitting-Set Problem* como problema de decisión es NP-Completo, se desarrolla una solución con un algoritmo de *Backtracking* para el mismo, como problema de optimización, y se analizan posibles soluciones aproximadas.

2. Definición del *Hitting-Set Problem*

Dado un conjunto de elemento A de n elementos, m subconjuntos $B_1, B_2, \ldots, B_m$ 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, $C \cap B_i \neq \emptyset$).

2.1. Como problema de decisión

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

3. Demostraciones

Un problema de decisión P es NP-Completo si P pertenece a NP y P es NP-Difícil.

3.1. *Hitting-Set Problem* está en *NP*

Un problema pertence a NP si una solución al mismo puede ser verificada en tiempo polinomial por una máquina de Turing determinística, o alternativamente, el problema puede ser resuelto en tiempo polinomial por una máquina de Turing no determinística.

El siguiente algoritmo es un posible verificador de soluciones del *Hitting-Set Problem*:

```
def is_hitting_set(A, B, C, k):
    if len(C) > k:
        return False
    for c in C:
        if c not in A:
            return False
    for Bi in B:
        for b in Bi:
            if b in C:
                break
        else:
            return False
    return True
```

El algoritmo es de tiempo polinomial porque $|B| = m$, $B_i \subseteq A$ $\forall$ i y $C \subseteq A$, por lo que $|B_i| \leq |A|$ y $|C| \leq |A|$ y la complejidad es $\mathcal{O}(N^2 m)$ con $N = |A|$. Esto es porque la complejidad del verificador esta dada por la mayor de las siguientes:

- una comparación del tamaño de C contra k ($\mathcal{O}(1)$).
- un loop por todos los elementos de C verificando que esten en A ($\mathcal{O}(N^2)$).
- un loop por m elementos (la cantidad de subsets), con otro loop adentro de a lo sumo N elementos cada uno (la cantidad de elementos en B_i) que chequea si cada elemento esta en C (de tamaño a los sumo N), que si C es una lista es una operación de tiempo lineal ($\mathcal{O}(N^2 m)$).

3.2. *Hitting-Set Problem* es *NP*-Difícil

Para demostrar que el problema es *NP*-Difícil realizamos una reducción polinomial de un problema *NP*-Completo a nuestro problema, Vertex Cover.

Un *Vertex Cover* V' de un grafo no dirigido $G = (V, E)$, es un conjunto de vertices $V' \subseteq V$, tal que para toda arista $(u, v) \in E$, $u \in V' \ \vee \ v \in V'$, o lo que es lo mismo, todas las aristas del grafo G tienen por lo menos una esquina en V'. La versión de decisión del problema se trata de determinar si existe un *Vertex Cover* de a lo sumo k vértices.

Para reducir este problema al *Hitting-Set Problem* creamos un subset $B_i = \{u, v\}$ por cada arista $(u, v) \in E$, $A = V$ y $k = k$. Luego tomamos la solución del *Hitting-Set Problem* C y con ella generamos la solución del *Vertex Cover* $V' = C$:

```
def vertex_cover(G, k):
    A = G.V
    B = [{u, v} for (u, v) in G.E]
    V = hitting_set(A, B, k)
    return V
```

Esta reducción se puede realizar en $\mathcal{O}(V^2)$, que es el costo de crear un subset por cada arista en el grafo G.

4. Programación Lineal

4.1. Definiciones

4.1.1. Variables

$Y_i :=$ Variables binarias, indican si el elemento A_{Y_i} forma parte de la solución.
(una por cada elemento en A)

4.2. Modelo

4.2.1. Restricciones

Necesitamos que haya por lo menos un elemento en cada subset, esto los modelamos con una restricción por cada subset que toma la suma de las variables asociadas a sus elementos y fuerza a esta a valer por lo menos uno:

$$\sum_{Y \in B_i} Y \geq 1 \qquad \forall \quad B_i \in B \tag{1}$$

4.2.2. Funcional

Estamos tratando de minimizar la cantidad de elementos en el resultado, por lo que minimizamos el valor de la suma de las variables asociadas a los elementos en el conjunto A.

$$\text{mín}\{\sum_{Y \in A} Y\} \tag{2}$$

4.3. Relajación

Si dejamos que las variables Y_i tomen valores reales el nuevo problema puede ser resuelto en tiempo polinomial. Con esta solución podemos calcular una cota inferior para el k óptimo:

$$k \geq \lceil k_r \rceil \qquad \text{con } k_r := \text{óptimo del problema relajado} \tag{3}$$

Esto es porque al relajar las restricciones, la solución solo puede mejorar. Además, si tomamos las variables cuyo valor excede $\frac{1}{b}$ obtenemos una solución aproximada.

4.3.1. Complejidad

La complejidad del algoritmo con restricciones relajadas depende del algoritmo utilizado por la librería `PuLP`. Si se utilizara el método simplex, si bien es eficiente en la práctica tiene peor caso exponencial. Existen otros algoritmos para resolver problemas de programación lineal que funcionan en tiempo polinomial, como el algoritmo de Karmarkar.

4.3.2. Calidad

$$\frac{A(I)}{z(I)} \leq b \qquad \text{con } b := \max_{B_i \in B}(|B_i|)$$

Demostración. Aprovechamos la ecuación (3) para obtener una cota inferior para $z(I)$:

$$k_r \leq z(I)$$
$$k_r b \leq z(I) b$$

También sabemos que nuestra aproximación $A(I)$, define que las variables con valor mayor o igual a $\frac{1}{b}$ serán 1. En el peor caso, todas las variables valen $\frac{1}{b}$ y pasan a valer 1, lo que nos deja con una función objetivo a lo sumo b veces peor:

$$A(I) \leq k_r b \implies A(I) \leq z(I) b$$

Si definimos $r(A)$ tal que $\frac{A(I)}{z(I)} \leq r(A)$ obtenemos $r(A) = b$. □

5. Aproximación

Para encontrar una solución aproximada al *Hitting-Set Problem*, proponemos el siguiente algoritmo *Greedy*:

```python
def hitting_set(A, subsets):
    frequencies = [0] * len(A)
    for subset in subsets:
        for item in subset:
            frequencies[item] += 1

    solution = []
    missing = list(subsets)
    while missing:
        item, _ = max(enumerate(frequencies), key=lambda t: t[1])

        subset_idx = 0
        while subset_idx < len(missing):
            if item in missing[subset_idx]:
                for other in missing[subset_idx]:
                    frequencies[other] -= 1
                missing.pop(subset_idx)
            else:
                subset_idx += 1

        solution.append(item)

    return solution
```

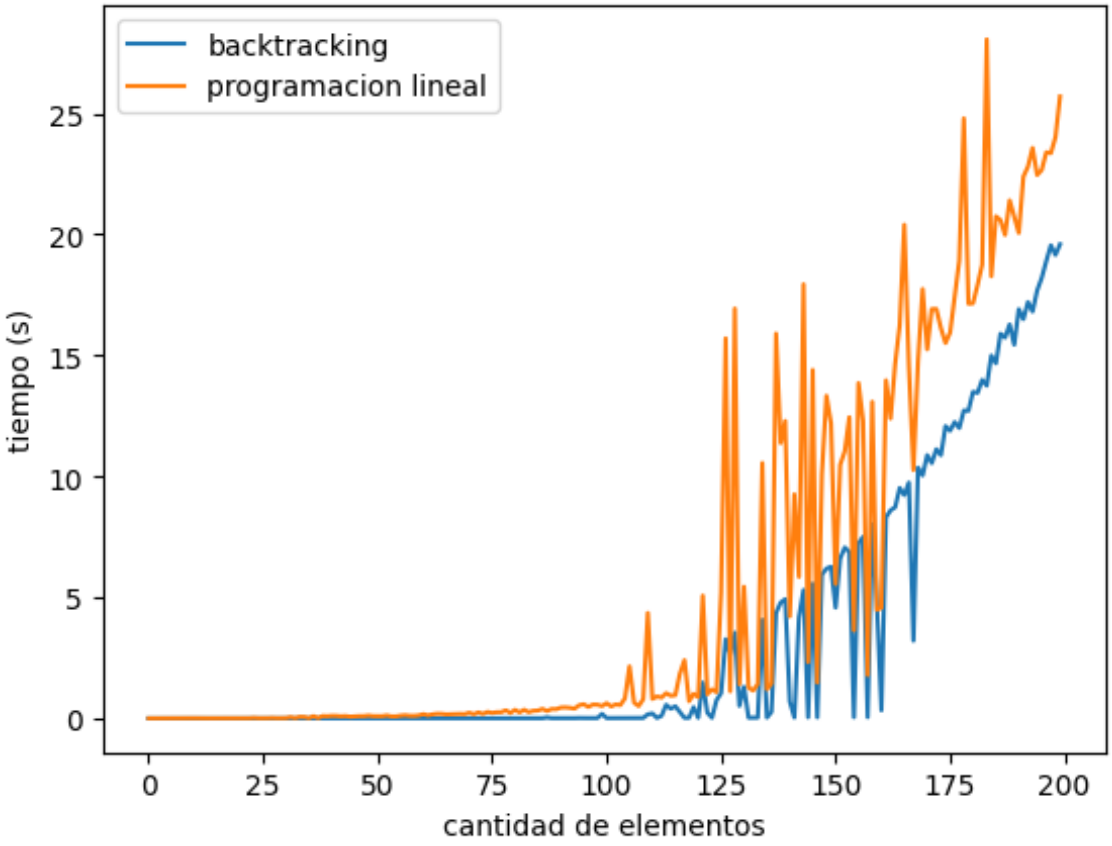
En este algoritmo se itera por todos los elementos de los subsets para calcular las frecuencias, si los subsets tienen a lo sumo N elementos cada uno, esto tiene complejidad $\mathcal{O}(Nm)$, siendo m la cantidad de subsets. Luego se itera quitando por lo menos un subset de la lista de subsets cada vez, es decir $\mathcal{O}(m)$ iteraciones, y en ese *loop* se itera por todos los subsets buscando aquellos que contengan al de mayor frecuencia realizando operaciones de costo lineal en la cantidad de elementos del subset, como lo son buscar un elemento, iterar por el mismo y remover un elemento en cualquier posición, lo que nos deja con una complejidad de $\mathcal{O}(m^2 N)$.

6. Mediciones

Se realizaron mediciones en base a crear sets de distinta cantidad de elementos, con la cantidad de subsets y la cantidad de elementos por subset siendo proporcional a la cantidad de elementos.

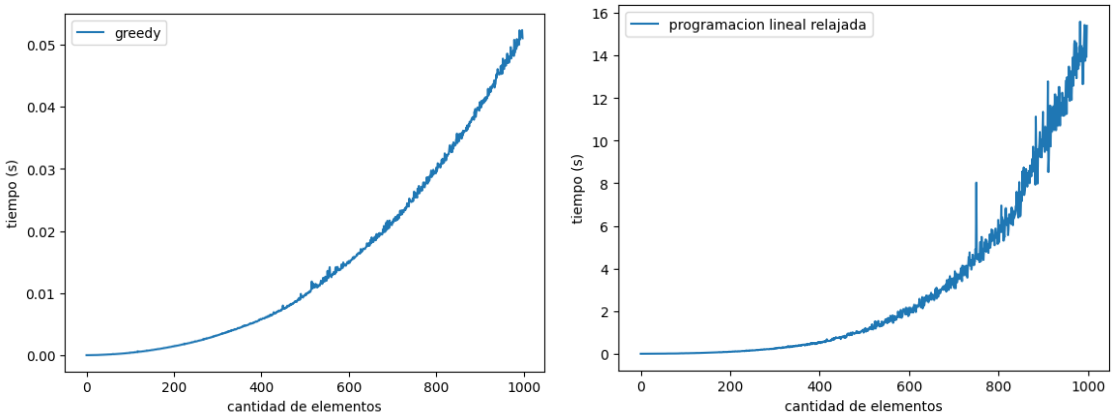
Para asegurar la validez de las comparaciones todas las mediciones fueron realizadas sobre los mismos sets de datos para los métodos que están siendo comparados. Los elementos fueron generados por los valores pseudoaleatorios del lenguaje (el módulo `random`) con la misma *seed*.

6.1. Backtracking vs. Programación Lineal



Backtracking obtiene mejores tiempo de ejecución que programación lineal para encontrar la solución óptima.

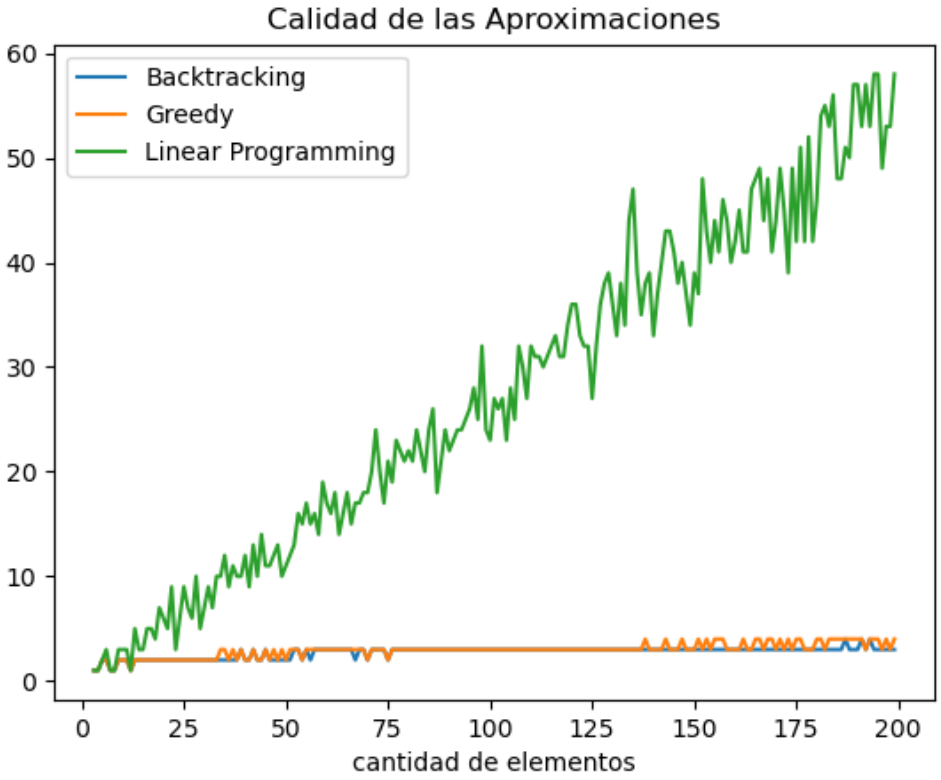
6.2. Algoritmos de aproximación



Notar la diferencia de tiempos (eje y) entre greedy y programación lineal.

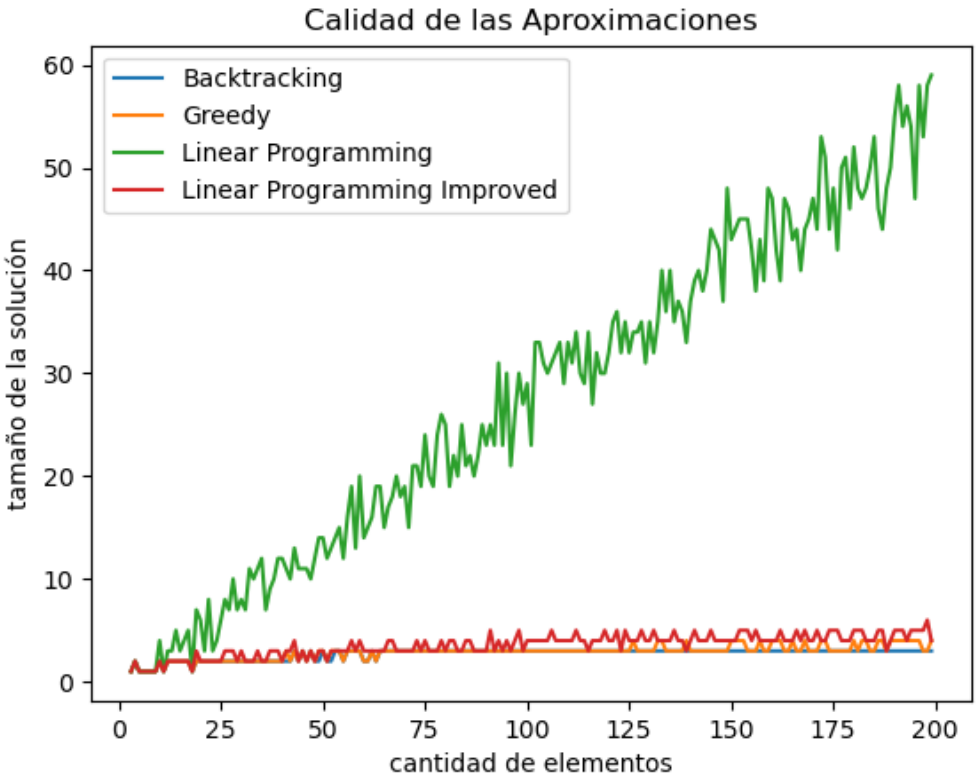
6.2.1. Calidad de las aproximaciones

Si comparamos el tamaño de la solución que obtenemos utilizando cada algoritmo, vemos que el algoritmo greedy provee mejores soluciones que el algoritmo por programación lineal relajada, y que dichas soluciones no se alejan mucho del óptimo.



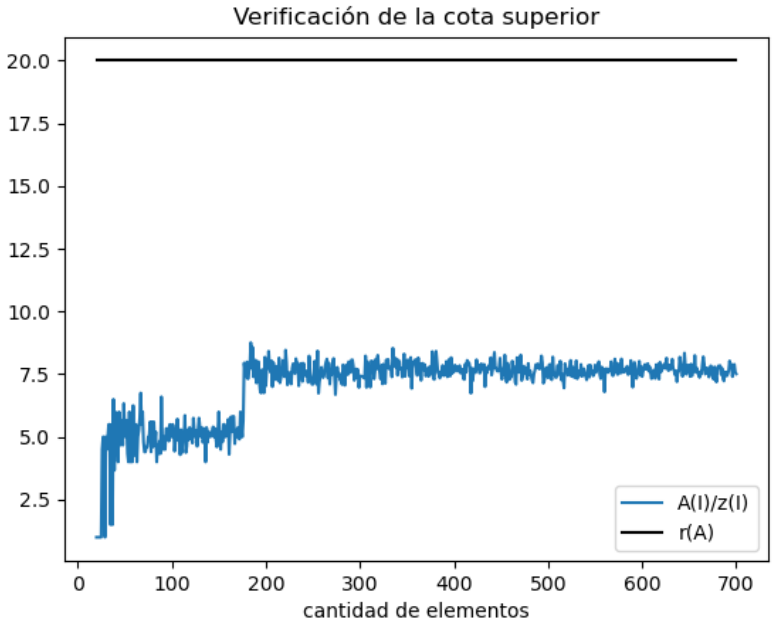
6.2.2. Mejoras al algoritmo por programación lineal relajada

Dado que las soluciones por programación lineal relajada estan tan lejos del óptimo, proponemos una mejora al algoritmo. Luego de obtener el valor en el óptimo de las variables, tomamos la variable de mayor valor de cada subset, priorizando las variables que ya forman parte de la solución, esta aproximación es por lo menos tan bueno como la aproximación utilizada anteriormente porque el valor de la variable con mayor valor de cada subset siempre es mayor o igual a $\frac{1}{b}$, por lo que también forma parte de la solución vieja.



6.3. Cota de la relajación lineal

Se realizaron mediciones para verificar la cota calculada empiricamente con un valor de $r(A) = 20$:



Vemos que $\frac{A(I)}{z(I)}$ se mantiene por debajo de la cota calculada. Cabe aclarar que a partir de los 175 elementos se utilizó una cota inferior para $z(I)$, por lo que se ve un pico en el gráfico a partir de ahí, esta cota inferior fue calculada por el mismo algoritmo de programación lineal relajada porque los volumenes de datos eran inmanejables para el algoritmo exacto.

7. Conclusiones

Algunas cosas que caben destacar son los distintos usos que dimos o podríamos dar a los algoritmos de aproximación para disminuir el costo de encontrar la solución óptima, como la utilización de la solución obtenida por el algoritmo *Greedy* como cota superior en la busqueda por *Backtracking*, y el valor del funcional al relajar las restricciones del modelo lineal puede ahorrarnos una iteración si se utilizara como cota inferior de k (aunque esto solo funcionaría si la solución es exactamente $\lceil k_r \rceil$). El primero de estos mejoró el tiempo de ejecución de la busqueda en un 10 %.