.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

24 de noviembre 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 desición 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. Consideraciones

En la versión de decisión del *Hitting-Set Problem*, una solución C al problema es un subconjunto $C \subseteq A$, con $|C| \leq k$ y $C \cap B_i \neq \emptyset$ para todo $B_i \in B$.

3. Demostraciones

Un problema de desición 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 Bi in B:
        for b in Bi:
            if b in C:
                break
        else:
            return False
    return True
```

El algoritmo es de tiempo polinomial porque $|B| = m$, $|B_i| \leq |A|$ para todo $B_i \in B$, $B \subseteq A$ y $C \subseteq A$, por lo que la complejidad es $\mathcal{O}(N^2 m)$ con $N = |A|$.

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 decisón del problema se trata de determinar si existe un *Vertex Cover* de a lo sumo k vértices.

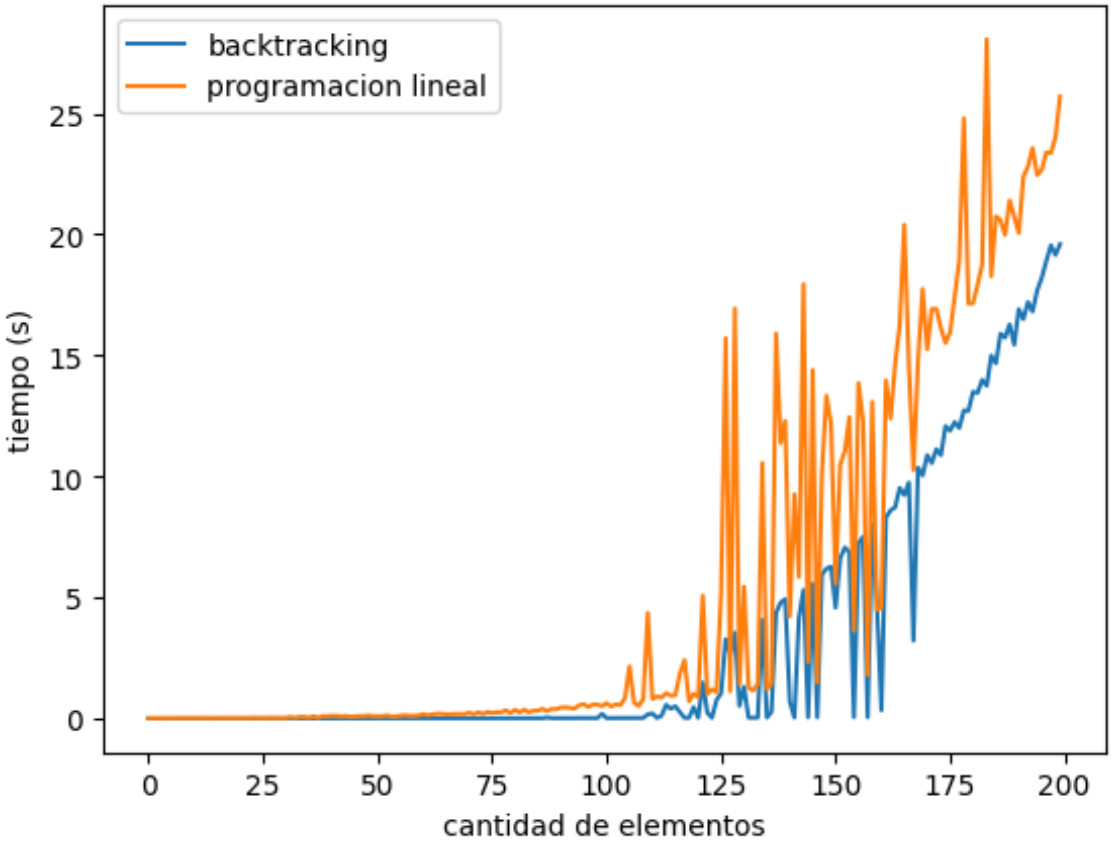
Para reducir este problema al *Hitting-Set Problem* creamos un subset $B_i = \{u, v\}$ por cada arista $(u, v) \in E$, $A = V$ y $k = k$. Luego tomamos la solución del *Hitting-Set Problem* C y con ella generamos la solución del *Vertex Cover* $V' = C$:

```
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.

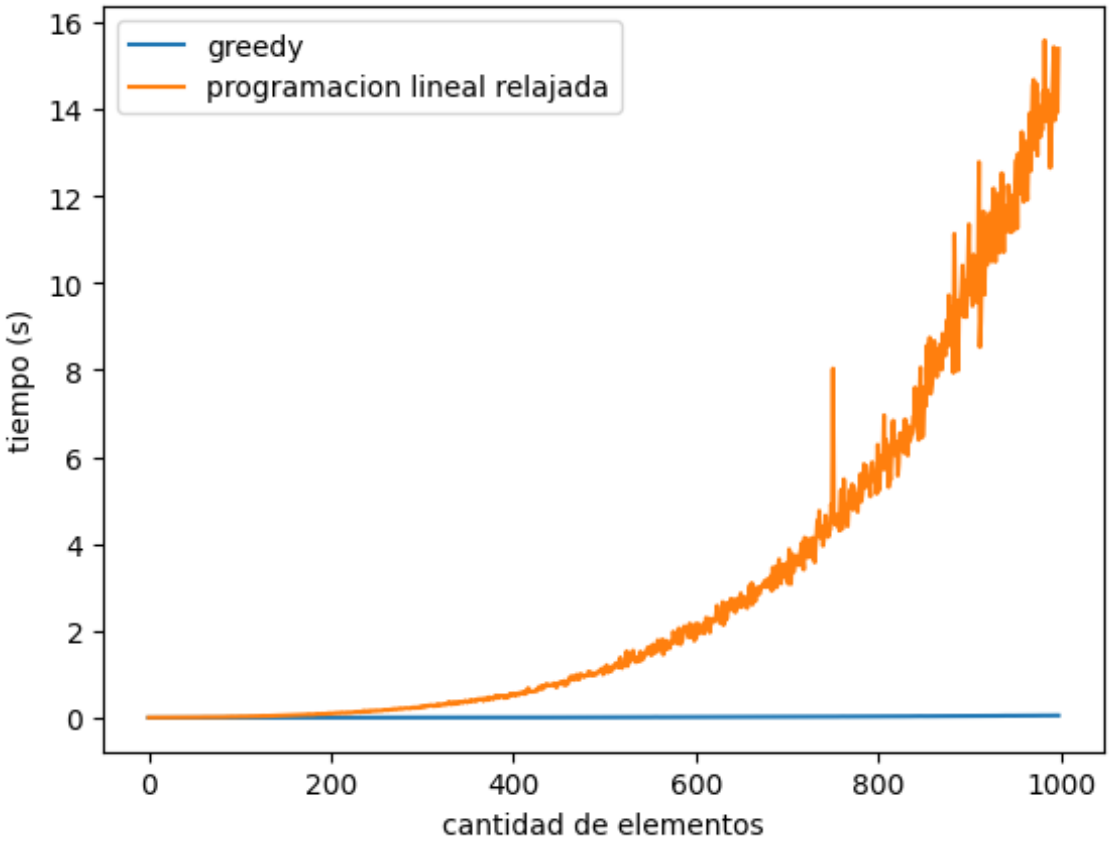
4. Mediciones

4.1. Se realizaron comparaciones entre backtracking y programación lineal



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

4.2. Se realizaron comparaciones entre greedy y programación lineal relajada



Greedy demora mucho menos tiempo que programación lineal relajada para encontrar una aproximación, pero la aproximación otorgada es peor.

5. Programación Lineal

5.1. Definiciones

5.1.1. Variables

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

5.2. Modelo

5.2.1. 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:

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

5.2.2. 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.

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

5.3. 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.

5.3.1. 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.

5.3.2. Calidad

Aprovechando la ecuación (3), obtenemos 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 $r(A) = b$.

5.4. Algoritmo

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

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

    solution = []
    missing = list(subsets)
    while missing:
        item = max(frequencies, key=frequencies.get)

        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

        del frequencies[item]
        solution.append(item)

    return solution
```

6. Conclusiones

Aca irían sus conclusiones.