.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

25 de noviembre de 2023

Nombre	Padrón
Buono, Fernando	103523
Katta, Gabriel	105935
Jarmolinski, Arian	94727

1. Análisis del Problema

La problemática a ser estudiada a lo largo de este trabajo practico es la de un Hitting-Set-Problem, específicamente, el director técnico de la selección, necesita escoger un conjunto de jugadores que satisfaga las necesidades de la prensa, Scaloni cuenta con un conjunto de jugadores de donde puede elegir, llamémoslo A, y la prensa tiene m subconjuntos B_1 , B_2, ... , B_3 que modelan los jugadores que ellos quisieran ver. Lo que Scaloni necesita resolver es la obtención del conjunto mínimo C de jugadores del cual tenga al menos un jugador de cada subconjunto B, de esta manera se puede satisfacer a todos los periodistas y evitar sus comentarios innecesarios.

Antes de empezar a proponer soluciones y algoritmos puede aportar valor la caracterización previa de este problema, para poder darnos una idea de que tipo de soluciones serian posibles.

Para esto se demostrara si este problema se encuentra en NP.

1.1. Demostración: ¿Hitting-Set esta en NP?

Se dice que un problema está en NP si existe un algoritmo polinómico que verifica soluciones propuestas. En el caso del Hitting-Set Problem, una solución propuesta es un conjunto C, y queremos verificar si C es efectivamente una solución del problema, es decir, interseca cada conjunto B_i.

La verificación se puede hacer de la siguiente manera:

- Para cada conjunto B_i: $\rightarrow\rightarrow$
  - Verificar si la intersección entre C y B_i es no vacía.
  - Esto se puede hacer en tiempo lineal, ya que el tamaño de C y B_i es finito y conocido.
- Si la intersección es no vacía para todos los conjuntos B_i, entonces C es una solución del Hitting-Set Problem.

El proceso de verificación se puede realizar en timepo polinómico, ya que estamos realizando una operación constante (verificar la intersección de C) para cada uno de los conjuntos B_i.

1.2. Demostración: ¿Es NP Completo?

Se dice que un problema es NP-Completo si pertenece a NP y cualquier problema en NP se puede reducir polinómicamente a él. Resolver eficientemente un problema NP-Completo implicaría resolver eficientemente todos los problemas en NP.

Para probar la NP-Completitud vamos a demostrar que el Vertex Cover Problem se puede reducir a el Hitting-Set Problem.

Pero antes, recordemos de que trata el Vertex Cover Problem:

- Dado un grafo no dirigido $G = (V, E)$, un Vertex Cover es un conjunto C de vértices tal que cada arista en E tiene al menos un extremo en C.
- El problema es decidir si existe un Vertex Cover de tamaño k para un k dado.

Procedemos entonces con la demostración de la reducción del Vertex Cover Problem al Hitting-Set Problem:

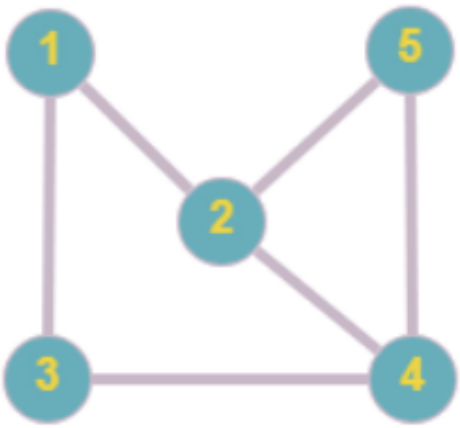
Dado un grafo $G = (V, E)$ en el problema Vertex Cover, construimos una instancia equivalente del Hitting-Set Problem:

- Conjunto de elementos A: será el conjunto de vértices V del grafo.

- Subconjuntos B_i: por cada arista (u, v) en el grafo, creamos un conjunto B_i en el Hitting-Set Problem que contiene los vertices u y v.

Verificación de la Reducción:

- Si existe un Vertex Cover C en G de tamaño k, entonces para una arista (u, v), u o v pertenece a C. Por lo tanto, C conforma el Hitting Set ya que cada subconjunto B_i tiene al menos un elemento en común con C.
- Si existe un Hitting-Set C' en A, entonces C' interseca todos los subconjuntos de A. Por lo tanto, como C' cubre al menos una arista de cada vértice de G, C' conforma un Vertex Cover en G.



Si G tiene vértices 1, 2, 3, 4, 5 y aristas (1, 2), (1, 3), (2, 4), (2, 5), (3, 4), la construcción equivalente en el Hitting-Set Problem sería:

- A = 1, 2, 3, 4, 5
- B_1 = 1, 2
- B_2 = 1, 3
- B_3 = 2, 4
- B_4 = 2, 5
- B_5 = 3, 4

Un Vertex Cover C de tamaño k = 3 en G es C = 1, 2, 4 ya que al menos un extremo de cada arista está contenido en C. Como estos vértices cubren todas las aristas y los subconjuntos B_i de A están conformados por los vértices de las aristas, entonces C es un Hitting Set en A.

A su vez, un Hitting-Set C' de tamaño k = 3 en A es C' = 1, 2, 4 ya que al menos un elemento de cada subconjunto B_i está contenido en C'. Como los subconjuntos B_i están formados por las aristas (u, v), y u o v pertenecen a C', entonces C' es un Vertex Cover en G.

Análisis de la Solución

2. Análisis del Algoritmo: Backtracking

El algoritmo de backtracking propuesto para resolver la problemática fue el siguiente:

```
def seleccion_backtracking(jugadores, indice_jugador, pedidos, seleccion_actual, seleccion_final):
    if not pedidos:
        if len(seleccion_actual) < len(seleccion_final):
            seleccion_final = seleccion_actual.copy()
        return seleccion_final

    for i in range(indice_jugador, len(jugadores)):

        if len(seleccion_actual) + 1 >= len(seleccion_final):
            break

        jugador_actual = jugadores[i]
        if jugador_actual not in seleccion_actual:
            seleccion_actual.add(jugador_actual)
            nuevos_pedidos = [conjunto for conjunto in pedidos if jugador_actual not in conjunto]
            seleccion_final = seleccion_backtracking(jugadores, i + 1 , nuevos_pedidos, seleccion_actual, seleccion_final)
            seleccion_actual.remove(jugador_actual)

    return seleccion_final
```

En el loop principal del algoritmo, se decide iterar por cada uno de los jugadores que se tiene como posibilidad para satisfacer a la prensa.

Si observamos que el jugador actual que se esta analizando no se encuentra en la lista de seleccionados actualmente, iniciamos a explorar el espacio de soluciones mediante el agregado de dicho jugador a la selección actual. Acto seguido se procede a recopilar los pedidos que aun no satisface Scaloni llevando al jugador actual, para luego entrar en el siguiente nivel de recursión. Luego de volver de la recursión, se retira al jugador de la selección actual, para seguir con el próximo.

En la llamada recursiva se envían el mismo conjunto de jugadores, sin embargo, se aumenta en 1 el índice de jugadores explorados, de esta manera cada nivel de recursión va dejando atrás a jugadores ya explorados, además, se pasan los pedidos que actualmente no han sido satisfechos.

Como podas del espacio de soluciones tenemos primero que todo preguntar si ya no quedan pedidos por satisfacer, esto nos indica que con los jugadores en la selección actual cumplimos con los pedidos de la prensa, sin embargo nada nos asegura que esa es la mejor solución, por ende se consulta si es mas pequeña que la selección final, si lo es, se actualiza, si no, devolvemos la selección final en ambos casos.

Luego dentro de el loop de jugadores, nos adelantamos preguntando que pasaría si agregamos 1 jugador mas, si esto ya hace que la selección actual sea mas grande que la final, podemos ignorar ese camino de soluciones ya que no nos llevara a un mejor resultado.

Esta solución cumplió con todos los casos de optimalidad proporcionados por la cátedra con muy buenos tiempos de ejecución.

```
Test con conjuntos_5
EXPECTED: 2 --- RESULT: 2 --- STATUS: PASSED --- TIME: 0.000052 seg
Test con conjuntos_7
EXPECTED: 2 --- RESULT: 2 --- STATUS: PASSED --- TIME: 0.000216 seg
Test con conjuntos_10_pocos
EXPECTED: 3 --- RESULT: 3 --- STATUS: PASSED --- TIME: 0.001189 seg
Test con conjuntos_10_varios
EXPECTED: 6 --- RESULT: 6 --- STATUS: PASSED --- TIME: 0.429568 seg
Test con conjuntos_10_todos
EXPECTED: 10 --- RESULT: 10 --- STATUS: PASSED --- TIME: 99.480591 seg
Test con conjuntos_15
EXPECTED: 4 --- RESULT: 4 --- STATUS: PASSED --- TIME: 0.019585 seg
Test con conjuntos_20
EXPECTED: 5 --- RESULT: 5 --- STATUS: PASSED --- TIME: 0.361728 seg
Test con conjuntos_50
EXPECTED: 6 --- RESULT: 6 --- STATUS: PASSED --- TIME: 1.987822 seg
Test con conjuntos_75
EXPECTED: 8 --- RESULT: 8 --- STATUS: PASSED --- TIME: 20.402882 seg
Test con conjuntos_100
EXPECTED: 9 --- RESULT: 9 --- STATUS: PASSED --- TIME: 53.558264 seg
Test con conjuntos_200
EXPECTED: 9 --- RESULT: 9 --- STATUS: PASSED --- TIME: 74.929752 seg
------------------------------------------------------------
```

2.1. Complejidad

La complejidad de el algoritmo es 2^n, siendo N la cantidad de jugadores. En cada nivel de recursión iteramos por los jugadores y se revisa si agregándolo al seleccionado llegamos a una mejor solución, en caso de que no, se vuelve a subir en un nivel de recursión y se continua la iteración con el resto de los jugadores.

2.2. Variabilidad de Datos

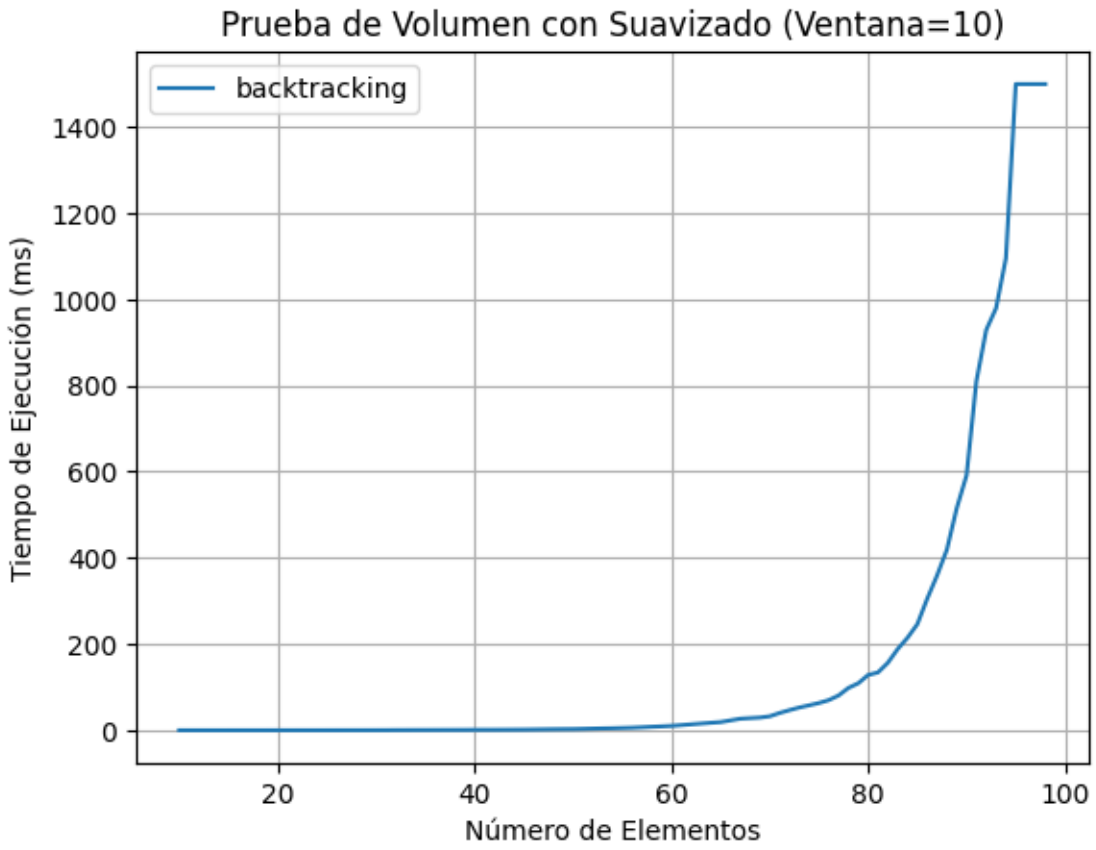
Al momento de ejecutar los casos de prueba de la cátedra, notamos que un caso en particular causaba que el algoritmo tardara mucho mas de lo debido para el volumen de dicho caso.

Este caso fue el de 10 Pedidos de prensa en el cual había una particularidad de que ningún jugador se repetía en otro pedido de la prensa.

Observando la lógica del algoritmo, esta configuración de datos directamente lo llevaba a comportarse como un algoritmo de fuerza bruta, ya que ninguna de las podas que existen logran achicar el espacio de soluciones, resultando en que se exploren en su totalidad y el árbol de recursión crezca a medida que se avanza por cada jugador.

Además de ese caso en particular, claramente al ser dependiente de la cantidad de jugadores, un N elevado causa que el tiempo de ejecución se eleve exponencialmente.

2.3. Tiempos de Ejecución - Volumen



Acá podemos evidenciar como al momento de aumentar el numero de jugadores (El cual, por como armábamos nuestros datos de ejecución, crecía a medida que se solicitaban mas pedidos) causan un aumento exponencial de los tiempos de respuesta, corroborando así la complejidad algorítmica deducida previamente.

3. Análisis del Algoritmo: Programación Lineal

El algoritmo de programación lineal propuesto para resolver esta problemática es el siguiente:

```python
def seleccionar_pl(jugadores, pedidos_prensa):
    prob = LpProblem("SeleccionJugadores", LpMinimize)

    x = {jugador: LpVariable(name=f"x_{jugador}", cat="Binary") for jugador in
    jugadores}

    prob += lpSum(x[jugador] for jugador in jugadores)

    for conjunto in pedidos_prensa:
        prob += lpSum(x[jugador] for jugador in conjunto) >= 1

    prob.solve()

    seleccion_final = {jugador: int(x[jugador].value()) for jugador in jugadores}
    return [player for player in seleccion_final.keys() if seleccion_final[player]
    == 1]
```

Variables de decisión:

Sea x_i igual a 1 si el jugador i es seleccionado y 0 de lo contrario.

$$x_i \in 0, 1$$

Función Objetivo:

Minimizar la cantidad total de jugadores seleccionados. Esto se puede expresar como:

$$Min \sum x_i$$

Restricciones:

Cada conjunto debe estar cubierto. Si un conjunto está cubierto, al menos un jugador de ese conjunto B_j debe ser seleccionado. Esto se puede expresar como:

$$\sum_{i \in \mathrm{B}_j} x_i \geq 1, \quad \forall j$$

3.1. Algoritmo Programación Lineal Continua

Como algoritmo de aproximación, empleamos el mismo enfoque que el de Programación Lineal, pero *relajamos* las variables. En otras palabras, las variables x_i, que representan a los jugadores, ya no son binarias, sino que pueden tomar valores en el rango de 0 a 1:

```python
def seleccionar_aproximado(jugadores, pedidos_prensa):
    prob = LpProblem("SeleccionJugadores", LpMinimize)

    x = {jugador: LpVariable(name=f"x_{jugador}", lowBound=0, upBound=1) for
    jugador in jugadores}

    prob += lpSum(x[jugador] for jugador in jugadores)

    for conjunto in pedidos_prensa:
        prob += lpSum(x[jugador] for jugador in conjunto) >= 1

    prob.solve()

    b = max(len(conjunto) for conjunto in pedidos_prensa)
    for jugador in jugadores:
        valor_relajado = x[jugador].value()
        x[jugador].value = 1 if valor_relajado >= 1 / b else 0

    seleccion_final = {jugador: int(x[jugador].value) for jugador in jugadores}
    return [player for player in seleccion_final.keys() if seleccion_final[player]
    == 1]
```

Variables de decisión:

Sea $x_i \in [0, 1]$

Función Objetivo:

Minimizar la cantidad total de jugadores seleccionados. Esto se puede expresar como:

$$Min \sum x_i$$

Restricciones:

Cada conjunto debe estar cubierto. Si un conjunto está cubierto, al menos un jugador de ese conjunto B_j debe ser seleccionado. Esto se puede expresar como:

$$\sum_{i \in B_j} x_i \geq 1, \quad \forall j$$

Redondeo:

Para redondear, obtenemos el valor b como la longitud del pedido de prensa con mayor cantidad de jugadores, y definimos que la variables de decisión de cada jugador serán 1 si su valor en el modelo relajado es mayor o igual a $1/b$.

3.2. Aproximación Greedy

Además, se decidió agregar como aproximación extra, un algoritmo greedy para resolver el problema:

```python
# Ordenamiento Greedy por Frecuencia de aparicion en los pedidos
def sort_jugadores_por_frecuencia(pedidos):
    freq = {}
    for conjunto in pedidos:
        for jugador in conjunto:
            freq[jugador] = freq.get(jugador, 0) + 1
    return sorted(freq.keys(), key=lambda jugador: freq[jugador], reverse=True)


# Algoritmo Greedy
def seleccion_greedy(prensa):

    seleccion_final = set()
    jugadores_ordenados = sort_jugadores_por_frecuencia(prensa)

    for jugador in jugadores_ordenados:

      if not prensa:
        break

      pedidos_pendientes = [pedido for pedido in prensa if jugador in pedido]
      if pedidos_pendientes:
        seleccion_final.add(jugador)
        prensa = [pedido for pedido in prensa if jugador not in pedido]

    return seleccion_final
```

Como ordenamiento para el algoritmo, decidimos ordenar los jugadores de mayor a menor en términos de la frecuencia con la cual aparecían en los pedidos.

Esto nos permitía agarrar al jugador mas frecuente, consultar si tiene pedidos que aun necesiten que este elegido, si es así, lo agregamos a la solución final y hacemos que los pedidos restantes sean solo aquellos que no incluyan al jugador actual.

De esta manera cuando nos quedemos sin pedidos restantes podemos cortar el ciclo y salir con la ejecución final.

La complejidad de este algoritmo puede ser analiza por partes, primero que todo la parte de ordenamiento itera por los n pedidos y dentro de cada pedido itera sobre los m jugadores y los empieza a sumar por frecuencia, también se puede entender que la m es la longitud de cada pedido.

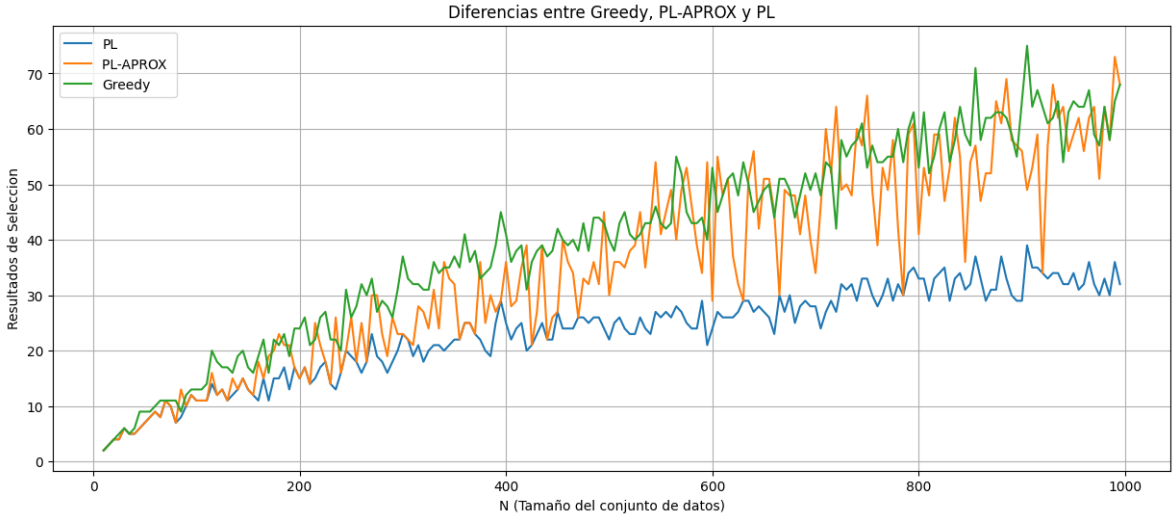
Luego la función sort de python tiene una complejidad logarítmica, es decir, solamente el sorting por parte de frecuencia tiene una complejidad de $O(n * m * log(m))$

Luego el algoritmo en si itera por cada jugador m y luego por los pedidos restantes por el evaluar, es decir tiene una complejidad de $O(n * m)$ en el peor de los casos.

Con esto podemos concluir que la complejidad final es: $O(n * m * log(m))$

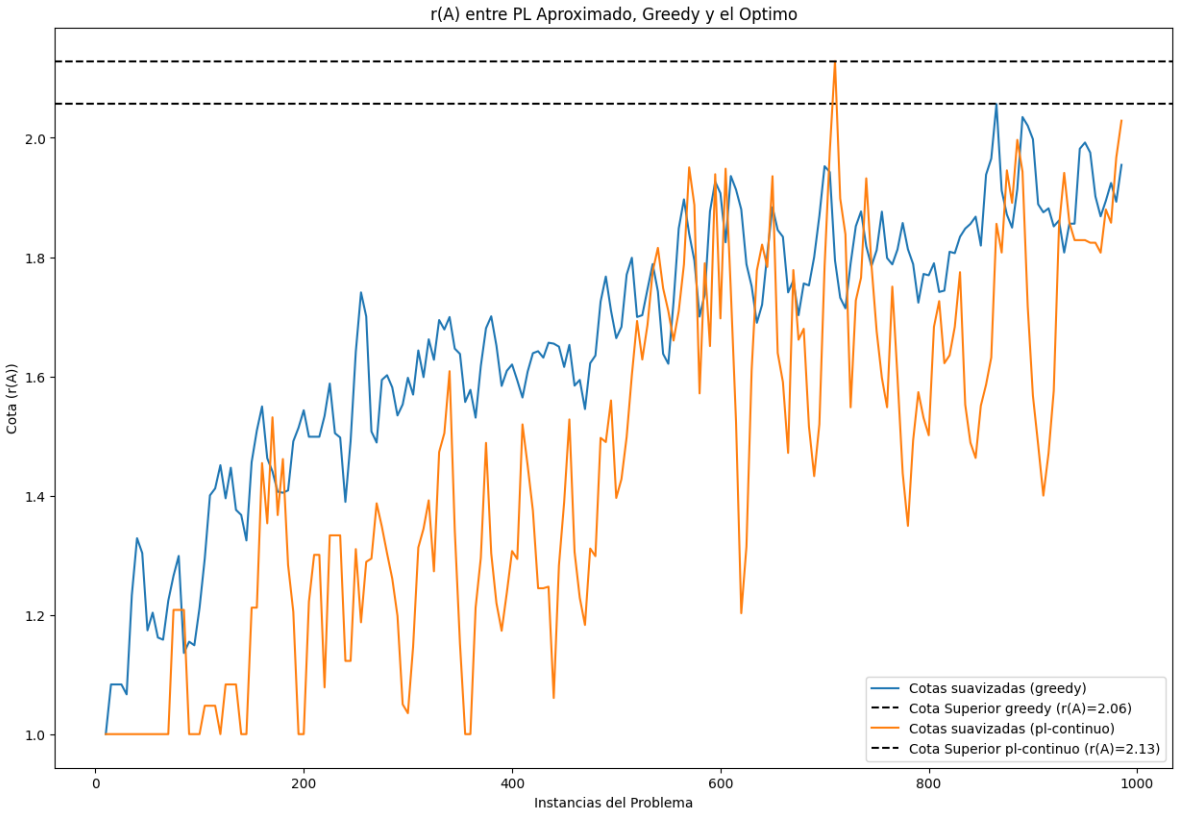
3.3. Análisis de Cotas

Para analizar las cotas de cada aproximación, primero observemos como se comportan las distintas respuestas de los métodos operando sobre los mismos set de datos:



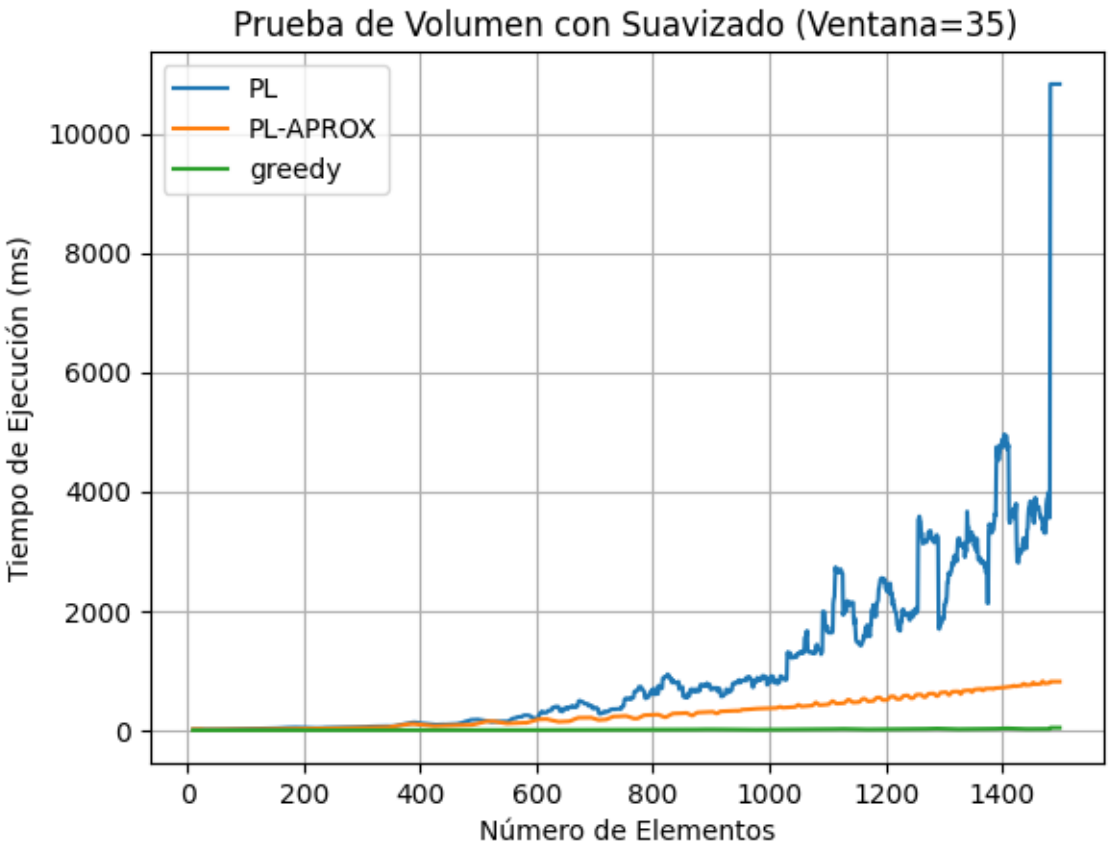
Observamos que claramente, el PL optimo gana al resto de las aproximaciones como era de esperarse. A simple vista observamos que greedy y PL-Continuo están bastante cerca sin embargo PL-Continuo es ligeramente mas acertado en sus aproximaciones.

Ahora, usando el calculo $A(I)/z(I)$ siendo $A(I)$ una aproximación de una instancia del problema y $z(I)$ la respuesta óptima del problema, veamos como evolucionan para cada método la evolución de este $r(A)$



Vemos que aunque la cota superior de greedy sea menor que la de PL-Continuo, este ultimo sigue siendo una mejor aproximacion a lo largo de la ejecucucion, estando siempre por debajo del algoritmo greedy.

3.4. Tiempos de Ejecución



Como podemos observar, la aproximación de PL es mucho mas rápida que el modelo de programación lineal optimo, esto claro, porque relaja las restricciones para que dejen de ser enteras y procede a redondear para llegar a un resultado similar en menos tiempo.

Luego el greedy al recortar mucho mas el espacio de soluciones que quiere observar, tiene un tiempo de respuesta muy rápido.

4. Conclusiones

Una conclusión relevante es que se observó que el algoritmo de backtracking es muy propenso a convertirse en uno de fuerza bruta, dependiendo de la naturaleza de los datos que necesita analizar. Esto fue evidente durante las pruebas de optimalidad del algoritmo, donde si la frecuencia de los jugadores era mínima entre pedidos, el espacio de soluciones a explorar crecía de manera considerable, ya que ningún jugador contribuía a reducirlo.

Además de esto es importante destacar viendo la performance del óptimo en comparación con las aproximaciones, que existe un trade-off bastante aparente entre la exactitud de la respuesta deseada y el tiempo en que necesitemos dicha respuesta.