.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3: Backtracking

27 de noviembre de 2023

Castro Martínez José Ignacio
106957

Makkos Juan Sebastian
106229

"*A la Selección Argentina, porque trajeron la tercera*"
"*A Lionel Scaloni, esta va con toda la intención de ayudarte*"

"*En un alba sin pájaros el mago vio cernirse contra los muros el incendio concéntrico.*"
Jorge Luis Borges

1. El problema

Scaloni ya está armando la lista de 43 jugadores que van a ir al mundial 2026. Hay mucha presión por parte de la prensa para bajar línea de cuál debería ser el 11 inicial. Lo de siempre. Algunos medios quieren que juegue Roncaglia, otros quieren que juegue Mateo Messi, y así. Cada medio tiene un subconjunto de jugadores que quiere que jueguen. A Scaloni esto no le importa, no va a dejar que la prensa lo condicione, pero tiene jugadores jóvenes a los que esto puede afectarles.

Justo hay un partido amistoso contra Burkina Faso la semana que viene. Oportunidad ideal para poner un equipo que contente a todos, baje la presión y poder aislar al equipo.

El problema es, ¿cómo elegir el conjunto de jugadores que jueguen ese partido (entre titulares y suplentes que vayan a entrar)? Además, Scaloni quiere poder usar ese partido para probar cosas aparte. No puede gastar el amistoso para contentar a un periodista mufa que habla mal de Messi, por ejemplo. Quiere definir el conjunto más pequeño de jugadores necesarios para contentarlos y poder seguir con la suya. Con elegir un jugador que contente a cada periodista/medio, le es suficiente.

Ante este problema, Bilardo se sentó con Scaloni para explicarle que en realidad este es un problema conocido (viejo zorro como es, ya se comió todas las operetas de prensa así que se conoce este problema de memoria). Se sirvió una copa de *Gatorei* y le comentó:

Esto no es más que un caso particular del Hitting-Set Problem. El cual es: Dado un conjunto A de n elementos y m subconjuntos $B_1, B_2, ..., B_m$ 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, $C \cap B_i \neq \emptyset$). En nuestro caso, A son los jugadores convocados, los B_i son los deseos de la prensa, y C es el conjunto de jugadores que deberían jugar contra Burkina Faso si o si".

Bueno, ahora con un poco más claridad en el tema, Scaloni necesita de nuestra ayuda para ver si obtener este subconjunto se puede hacer de forma eficiente (polinomial) o, si no queda otra, con qué alternativas contamos.

2. Estudiando Hitting Set Problem

2.1. Demostrando su pertenencia a NP

Para demostrar que Hitting Set se encuentra en NP bastaría con tener una solución del problema: un conjunto H y y el set de datos de entrada al mismo, un conjunto de subconjuntos $B = B_1, B_2, ...B_m$ ($B_i \subseteq A \forall i$) de manera que, simplemente, se debe iterar sobre el conjunto de subconjuntos y sobre cada subconjunto validando si alguno de los elementos $b_j \in B_i$ ciertamente pertenece a H. En caso de haber probado con todos los elementos de un B_i y ninguno de sus elementos b_j pertenezca a H entonces, la solución brindada no es un Hitting Set, en caso de haber llegado al final entonces dicho conjunto si es un Hitting Set

2.1.1. Implementación del validador

```
def is_solution(hitting_set, subsets):
    for subset in subsets:
        flag = False
        for element in subset:
            if element in hitting_set:
                flag = True
                break
        if not flag: return False
    return True
```

Listing 1: validador

2.1.2. Complejidad temporal del validador

Para verificar finalmente que dicho problema falta con probar que este validador posee tiempo polinomial

Veamos que:

- Recorrer la lista de subconjuntos es lineal en la cantidad de subconjuntos, dicha cantidad se llamará m.
- Recorrer los elementos del subconjunto B_i es lineal en la cantidad de elementos del subconjunto, dicha cantidad se llamará k.

Ahora si decimos generalizamos k como el máximo cardinal existente de algún conjunto B_i se puede establecer la siguiente cota:

$$T(n) = O(m \times k)$$

Como dicha complejidad es polinomial a m y k entonces se cumple que Hitting Set esta en NP

2.2. Demostrando pertenece a NP completo

Para probar este segmento del informe se debe realizar una reduccion de algun problema NP-Completo conocido a Hitting-Set, en este caso, dicho problema sera Vertex Cover.

En un principio se establecen ambos problemas de decisión:

> **El problema de decisión del vertex cover implica, dado un grafo G y un número k, determinar si existe un vertex cover de tamaño a lo sumo k**

Por otro lado:

> **El problema de decisión del hitting set implica, sea A un conjunto, B un conjunto de subconjuntos pertenecientes a A, determinar si existe un conjunto H de tamaño al menos k tal que todo subconjunto de B la intersección entre B_i y H no es vacía**

Entonces veamos la siguiente reducción:

$$\text{Vertex Cover} \leq_p \text{Hitting Set}$$

Sea G un grafo, V su conjunto de vertices y E su conjunto de aristas, entonces se toma A como el conjunto de vertices. Ahora, que sera B, en este caso se podria definir como el conjunto de subconjunto de los pares (v_i, v_j) siempre y cuando exista una arista entre dichos vertices, aunque, esto seria simplemente el conjunto de las aristas de G

Habiendo establecido lo anterior, se puede observar que si existe un Hitting Set de tamaño k para los conjuntos $A = V(G)$ y $B = E(G)$ entonces necesariamente existe un vertex cover de tamaño al menos k, por lo tanto, sabiendo que vertex cover es un problema NP-Completo se puede concluir finalmente que Hitting set también es NP-Completo

3. Implementando una solución

Para implementar alguna solución exacta del problema dado lo probado en la sección anterior, es decir, la pertenencia del problema a conjunto NP-completo habrá que diseñar algún algoritmo mediante la técnica de backtracking que pruebe todas las combinaciones existentes usando los conjuntos de subconjuntos. También se puede plantear alguna solución por programación lineal entera que resuelva el problema

3.1. Backtracking

Para construir una solución con una estrategia backtracking habrá que generar todas las posibles combinaciones entre los elementos pertenecientes a los conjuntos de subconjuntos.

Para que sea una solución por backtracking y no por fuerza bruta (probar absolutamente todas las combinaciones) habrá que diseñar casos de poda, en otras palabras, realizar eliminaciones de posibles soluciones invalidas

3.1.1. Casos de poda

Los casos de poda son esenciales para disminuir la duración en los tiempos de ejecución a la hora de buscar una solución al problema. En el caso particular de este problema, se habla sobre:

- **solución candidata**: que puede ser solución o no
- **solución encontrada**: un candidato a solución que esta validada como solución, pero puede no ser la mínima
- **solución mínima**: la solución más pequeña

Por lo tanto los casos de poda se determina de la siguiente manera:

- No se considera la secuencia si el candidato a solución es más grande que la solución anteriormente encontrada
- Si un subconjunto i se encuentra incluido no se lo considera
- No se repite el uso de elementos entre subconjuntos en la solución encontrada

3.1.2. Implementación

```python
def has_a_player(subset, act_sol):
    for player in subset:
        if player in act_sol:
            return True
    return False

def search_for_min_hitting_set(subsets, a):
    return  _search_for_min_hitting_set(subsets, set(), set(), 0, set())

def _search_for_min_hitting_set(subsets: list, best_sol: set, act_sol: set, act_sub
    : int, used_players: set):

    if len(best_sol) > 0 and len(act_sol) > len(best_sol):
        return best_sol

    if act_sub == len(subsets) - 1 and (len(best_sol) == 0 or len(act_sol) < len(
    best_sol)) and is_solution(act_sol, subsets):
        best_sol = act_sol.copy()
        return best_sol

```

```
19      if act_sub > len(subsets) - 1:
20          return best_sol
21
22      if has_a_player(subsets[act_sub], act_sol):
23          return _search_for_min_hitting_set(subsets, best_sol, act_sol, act_sub + 1,
         used_players)
24
25      selected_players = set()
26
27      for player in subsets[act_sub]:
28          if player in used_players:
29              continue
30          act_sol.add(player)
31          selected_players.add(player)
32          used_players.add(player)
33          best_sol = _search_for_min_hitting_set(subsets, best_sol, act_sol, act_sub
        + 1, used_players)
34          act_sol.remove(player)
35
36      used_players.difference_update(selected_players)
37      return best_sol
```

Listing 2: solución por backtracking

3.1.3. Análisis de la complejidad

Si bien es cierto que los casos de poda reducen el tiempo de ejecución del algoritmo no determinan una reducción teórica sobre el tiempo, es simplemente una manera de acelerar la apreciación experimental de conseguir una respuesta al problema. Por lo tanto la complejidad algorítmica del problema viene determinada por todas las combinaciones existentes en el problema, esto es entonces considerar todas las posibilidades sobre la pertenencia de un elemento del conjunto b_{i_n}, lo cual también puede ser visto como todas las combinaciones posibles sobre la pertenencia de los elementos del conjunto C

Por lo tanto:

$$T(n) = 2^n \text{ Siendo n} = |C|$$

Es facíl, observar que una solución trivial al problema Hitting Set esta dada por todos los elementos del conjunto C

3.2. Programación lineal entera

3.2.1. Diseñando la solución por programación lineal

Para generar una solución al Hitting Set problem mediante programación lineal entera se deben construir ecuaciones que modelen matemáticamente el problema. En un principio, el objetivo del problema es encontrar el mínimo de elementos de C los cuales estén presentes en todos los conjuntos, por lo que, surge la primera ecuación:

$$\text{mín} \left(\sum_{i \in C} C_i \right)$$

Con lo cual se construye la función objetivo del problema, evidentemente esta ecuación por si sola no es suficiente puesto que no considera el conjunto de subconjuntos $B = B_1, B_2, ...B_m$ $(B_i \subseteq A \forall i)$, faltaría entonces, exigir la pertenencia de al menos un elemento de cada subconjunto. Por lo que, surge la segunda y ultima ecuación necesaria para plantear el modelo:

$$\sum_{b_j \in B_i} b_j \geq 1 \quad \forall B_i \in B$$

También cabe aclarar que

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

3.2.2. Implementación

Para construir una implementación con programación lineal entera se recurre a la librería **pulp** de python y se plantean las ecuaciones tal cual han sido descritas:

```
def search_hs_linealp(subsets, set):
    dict_variables = {elem: pulp.LpVariable(f"{elem}", cat="Binary") for elem in
    set}

    problem = pulp.LpProblem("hitting_set_problem", pulp.LpMinimize)
    problem += pulp.lpSum(dict_variables[elem] for elem in set)
    for subset in subsets:
        problem += pulp.lpSum(dict_variables[elem] for elem in subset) >= 1

    pulp.LpSolverDefault.msg = 0
    problem.solve()

    hitting_set_solution = {var.name for var in dict_variables.values() if pulp.
    value(var) == 1}
    return hitting_set_solution
```

Listing 3: solución por programación líneal

Cabe destacar que esta solución requiere el paso previo de la reconstrucción del conjunto C, lo cual se realiza fácilmente usando diccionarios

3.2.3. Análisis de la complejidad

La programación lineal entera esta clasificada también como un problema NP-Completo (véase la referencia) y analizando un poco más las ecuaciones planteadas llegamos a que simplemente es una formulación de las posibles soluciones también consideradas en la resolución del problema mediante backtracking, por lo que tenemos que la complejidad temporal del algoritmo es también:

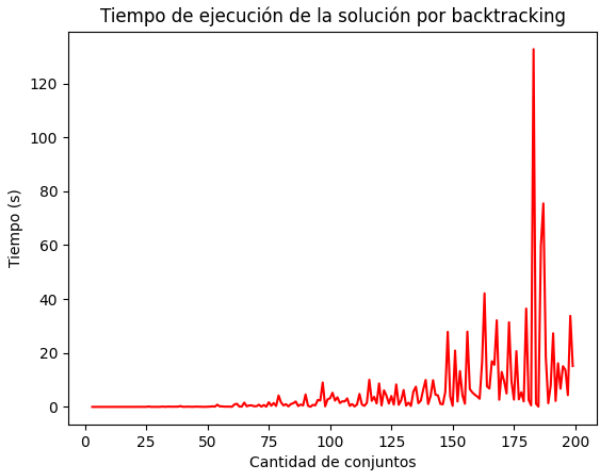
$$T(n) = 2^n \quad \text{Siendo } n = |C|$$

4. Gráficas

A continuación se introducen una serie de gráficas con la intención de observar el comportamiento de las dos implementaciones que consiguen el mínimo global al Hitting Set Problem

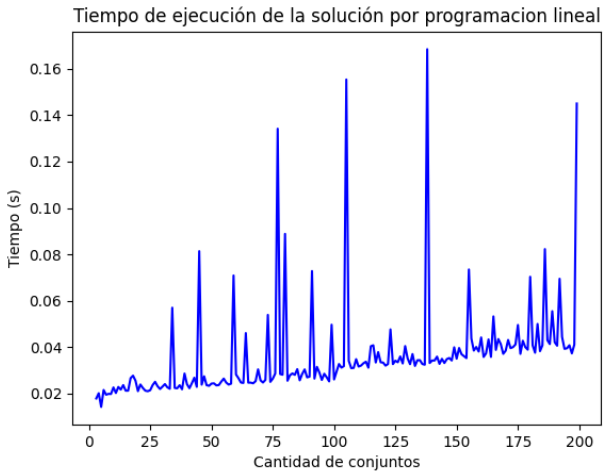
Se generaron datos aleatorios mediante la biblioteca Faker que permite la generación de nombres. Se generaron doscientos nombres aleatorios y a partir de allí se generan conjuntos de subconjuntos para todos los m $\in$ $(1, 200)$ de manera que se puedan apreciar bien las complejidades del algoritmo

4.1. Tiempos de la resolución por backtracking



En la gráfica anterior se observan los tiempos de ejecución para la implementación usando backtracking, es importante destacar la gran variabilidad entre los tiempos de ejecución existentes entre los conjuntos. Esto se debe a la variabilidad que existe entre los conjuntos con los cuales se va probando en cada paso el algoritmo

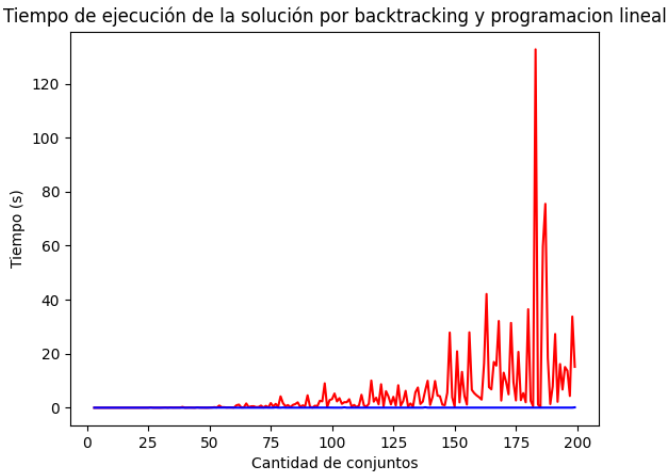
4.2. Tiempos de ejecución por Programación Entera



Por otro lado los tiempos de ejecución para la solución por programación lineal entera son más pequeños en comparación a la solución por backtracking. Debido al uso de la biblioteca **pulp** se estima que dicha resolución cuenta con optimizaciones ocultas al desarrollador y por lo tanto pareciera que la tendencia temporal tiende a ser algo lineal

4.3. Comparación entre ambos métodos

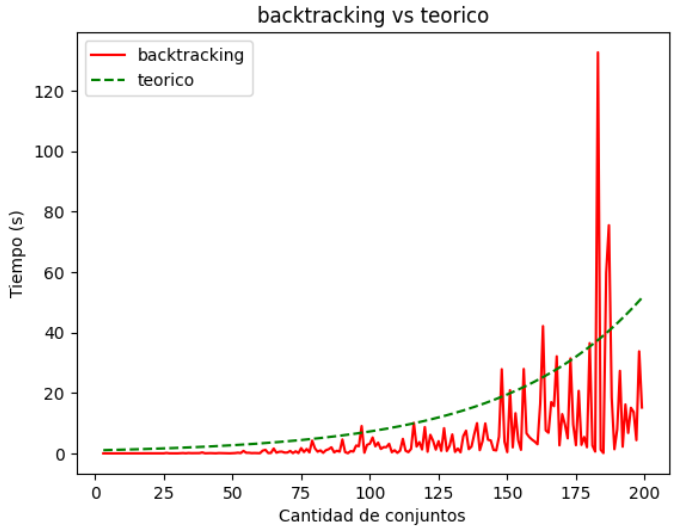
Finalmente se grafican sobre una misma escala ambos métodos y se aprecia la amplia diferencia entre ambos



5. Estimación contra cota teórica

5.1. Backtracking acotado por curva teórica

Se genero una gráfica con una curva teórica exponencial, con lo cual se puede apreciar que la implementación por backtracking se comporta efectivamente de manera exponencial al conjunto de entrada



6. Construyendo aproximaciones

6.1. Solución Greedy

Como es sabido el Hitting Set Problem pertenece a NP-completo y por lo tanto no existe ninguna solución que de en tiempo polinomial de una respuesta exacta en todo caso posible, pero, es posible construir una aproximación a la solución utilizando una estrategia greedy

6.1.1. La estrategia

Para construir una aproximación greedy se deben cumplir las siguientes reglas:

- Aplicar una regla sencilla
- Debe conseguir un óptimo local al problema actual
- Se deben repetir estas reglas hasta conseguir una respuesta

Por lo tanto se construye la siguiente estrategia greedy para resolver el problema:

- Conseguir el jugador más representativos cuya frecuencia de aparición es la más alta entre todos los jugadores
- Eliminar todos los subconjuntos en donde este jugador aparezca
- Repetir hasta que no quede ningún subconjunto de jugadores

6.1.2. La implementación

Siguiendo los pasos anteriormente mencionados se construye la implementación en python del algoritmo de aproximación greedy

```python
def aprox_hs_by_greedy(subsets: set, a: set):
    aprox_sol = set()
    missing_sets = set(range(len(subsets)))
    subsets = list(subsets)
    return _aprox_greedy(subsets, aprox_sol, missing_sets)

def _aprox_greedy(subsets: list, aprox_sol: set, missing_sets: set):
    while(not is_solution(aprox_sol, subsets)):
        diference = set()
        player = find_most_frequent_player(missing_sets, subsets)
        for i in missing_sets:
            if has_a_player(subsets[i], player):
                diference.add(i)
        aprox_sol.add(player)
        missing_sets.difference_update(diference)
    return aprox_sol

def find_most_frequent_player(missing_sets: set, subsets: list):
    frequency = dict()
    for subset_index in missing_sets:
        for player in subsets[subset_index]:
            frequency[player] = frequency.get(player, 0) + 1
    most_frequent_player = max(frequency, key=frequency.get)
    return most_frequent_player
```

Listing 4: aproximación greedy

6.1.3. Complejidad temporal

Para analizar la complejidad temporal del algoritmo vemos que se hacen las siguientes iteraciones sobre el algoritmo:

1. Se busca al jugador más frecuente, esto es recorre m subconjuntos y por cada uno de ellos sus k jugadores
2. Se ejecuta un ciclo while, que en el peor de los casos realiza m iteraciones y en cada una de esas realiza la llamada a buscar el jugador más frecuente

Por lo tanto se puede concluir que el algoritmo tiene la siguiente complejidad:

$$T(n) = O(m^2 \cdot k)$$

Donde k se puede estimar como el cardinal del mayor subconjunto perteneciente a m y m como el cardinal de m. Importante destacar que esta cota corresponde al peor caso posible y el algoritmo podría comportarse mejor de lo esperado dependiendo de la cantidad de jugadores que necesite k

6.2. Solución por programación lineal

Para construir esta aproximación se relajan las reglas implementadas en el caso de programación lineal entera y se permite que los valores de los jugadores varíen entre 0 y 1. Por lo que las ecuaciones en este caso serían:

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

$$\sum_{b_j \in B_i} b_j \geq 1 \quad \forall B_i \in B \tag{2}$$

$$b_j \in [0, 1] \tag{3}$$

6.2.1. Clasificación de la aproximación

TO-DO

6.3. Medición de las aproximaciones

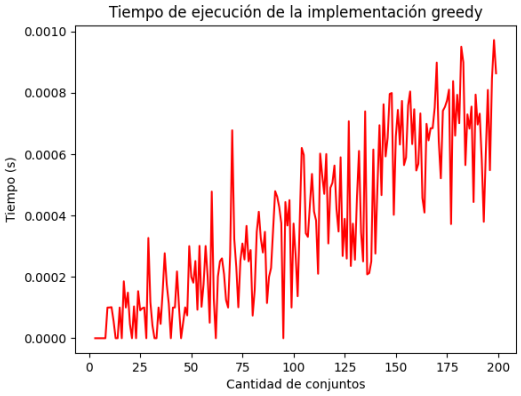


Figura 1: Aproximación greedy

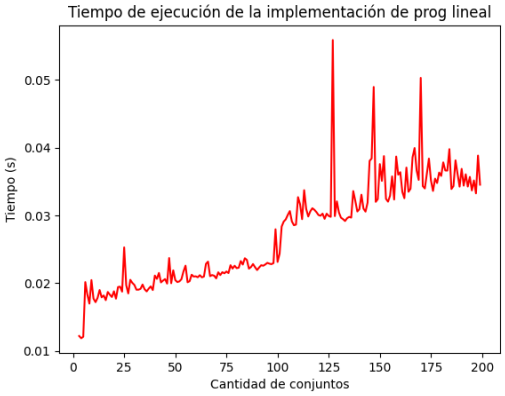
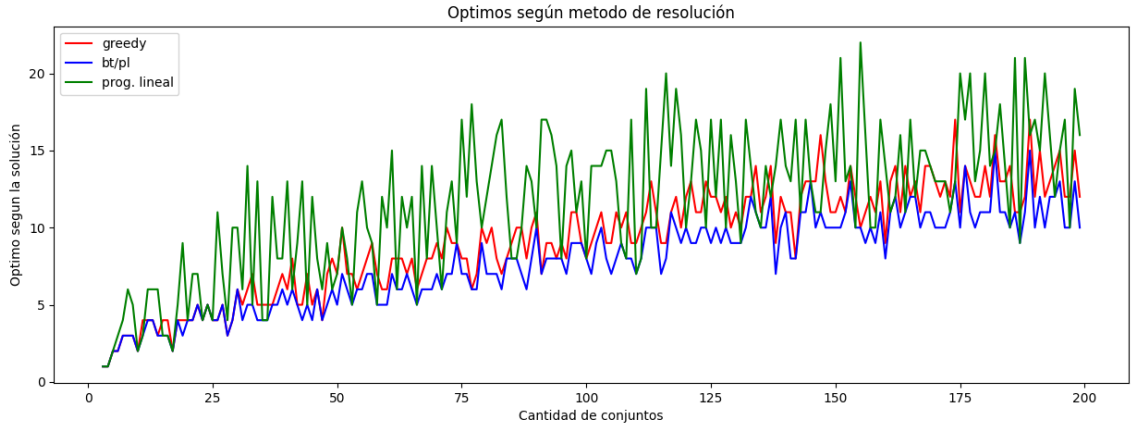


Figura 2: Aproximación por prog. lineal

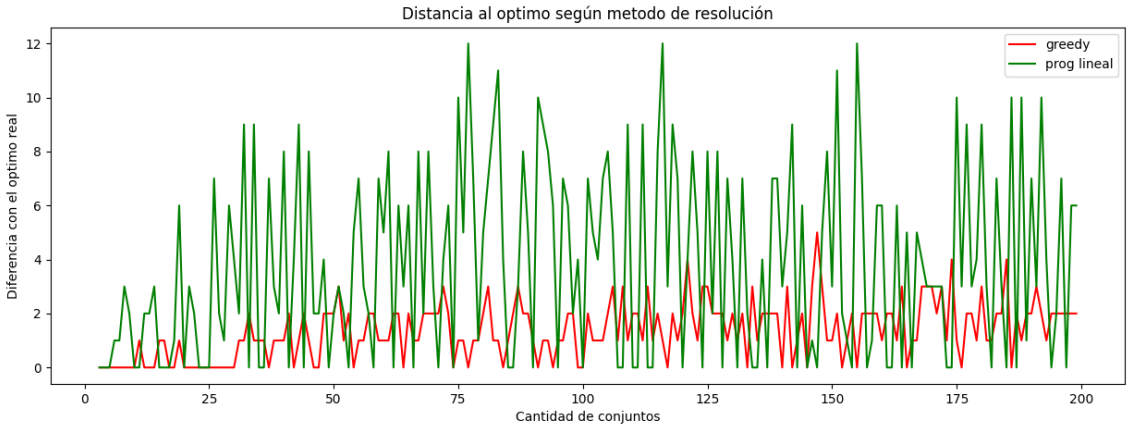
En las gráficas anteriores se puede observar la duración en microsegundos para el mismo set de datos utilizado a la hora de tomar mediciones sobre las soluciones exactas al problema. Se observa que la solución greedy posee una variabilidad mayor entre valores de m, a comparación de la solución por programación lineal que se mantiene mas estable entre los diferentes tamaños de m

6.4. Comparación entre los tamaños de los óptimos

En el siguiente gráfico se podrán observar los valores de los óptimos resultantes de ambas aproximaciones y el óptimo arrojado por una de las soluciones exactas, es fácilmente apreciable que, la aproximación por un algoritmo greedy es la cual posee el menor error a la hora de aproximar. Al mismo tiempo, se aprecia que la aproximación realizada por programación lineal no se perfila como una buena aproximación a utilizar



En este gráfico se analizan las distancias con el óptimo real de ambas aproximaciones, confirmando una vez más que la aproximación por el algoritmo greedy tiende a distar menos del óptimo real, siendo la máxima distancia al óptimo de 5 elementos



6.5. Mediciones en volumen de las aproximaciones

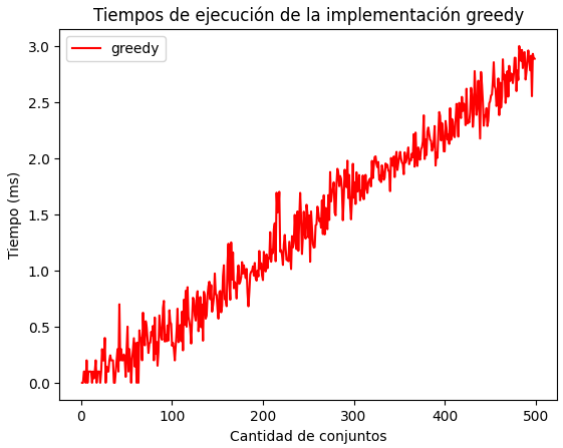


Figura 3: Tiempo en vol. por greedy

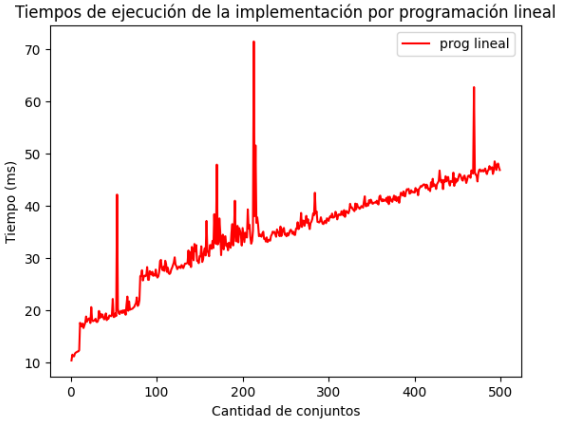


Figura 4: Tiempo en vol. por prog. Lineal

7. Conclusión