Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

29 de febrero de 2024

Emanuel Pelozo 99444

Agustina Fraccaro 103199

Agustina Schmidt 103409

Matias Gonzalez 104913

1. Análisis del problema: Hitting-Set Problem

El problema que necesitamos resolver para ayudar a Scaloni es una instancia de el Hitting-Set Problem, el cual está definido como:

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.

A continuación, demostraremos que este problema pertenece a NP-Completo, para esto, tomaremos la versión de decisión de este problema: Dado un conjunto de elementos A de n elementos, m subconjutnos $B_1, B_2, ..., B_m$ de A ($B_i \subseteq A \forall i$) y un número k, ¿Existe un subconjunto $C \subseteq A$ con $\mid C \mid \leq k$ tal que C tenga al menos un elementos de cada B_i?

1.1. Demostrar que Hitting-Set Problem se encuentra en NP

Para demostrar que el problema está en NP debemos demostrar que, dada una resolución de este problema, o sea dado un subconjunto C de elementos y los B_i, se puede verificar en tiempo polinomial que esta resolución es correcta.

Debemos asegurarnos que cada B_i tenga al menos un elemento en C.

```
def verificador(B, C, k):
    if len(C) > k:
        return False
    contiene = [False] * len(B)
    for c in C:
        for i in range(len(B)):
            if c in B[i]:
                contiene[i] = True
    return all(contiene)
```

Podemos ver que el algoritmo anterior es polinomial ya que el primer bucle se ejecutará como mucho k veces (es el tamaño del subconjunto solución C) y el segundo for se ejecutará m veces. En el peor de los casos, habrá un solo subconjunto B_i de tamaño k. Por lo tanto en el peor de los casos este algoritmo es $O(k^2)$.

Dado que hallamos un verificador polinomial del problema, podemos decir que Hitting-Set Problem está en NP.

1.2. Demostrar que Hitting-Set Problem es un problema NP-Completo

Para poder demostrar que Hitting-Set es un problema NP-Completo, debemos lograr reducir un problema ya conocido como NP-Completo hacia nuestro problema. De esta manera el problema elegido, será a lo sumo tan díficil como resolver Hitting-Set.

El problema elegido para reducir es Vertex Cover: dado un grafo $G = (V, E)$, existe un subconjunto de vértices de tamaño k tal que por lo menos uno de los dos extremos de cada arista está en el conjunto.

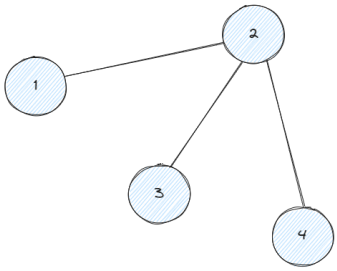
Reducción:

- Los vértices del grafo serán el conjunto A de Hitting Set.
- Cada subconjunto B_i será un par de vértices u, v unidos por una arista en G.
- El tamaño k será la misma para ambos problemas.

Entonces, si existe una solución para Hitting Set, también existe para Vertex Cover ya que en el último caso cada arista debe tener al menos un vértice en el subconjunto, y si los subconjuntos representan a las aristas, Hitting Set asegura que al menos uno de los elementos de cada subconjunto

estará en mi solución, así es como me aseguro que al menos un extremo de cada arista estará en el subconjunto.

Pondremos un ejemplo, tenemos el siguiente grafo:



El conjunto A serían los vértices: $\{1, 2, 3, 4, 5\}$. Los subconjuntos B_i serían las aristas: $\{1, 2\}, \{3, 2\}, \{4, 2\}$. Queremos encontrar un Vertex Cover de $k = 1$.

Un Hitting Set de tamaño 1 es: 2.

Por lo tanto, 2 es un Vertex Cover de tamaño $k = 1$.

Podemos concluir entonces que el problema Hitting Set es NP-Completo.

2. Algoritmo de Backtracking

A continuación se detalla la implementación del algoritmo de backtracking para resolver el problema de Scaloni de poner la mínima cantidad posible de jugadores pedidos por la prensa, cumpliendo la condición de que cada periodista tendrá al menos uno de los jugadores que pide en la solución final.

La firma de la función de nuestro algoritmo backtracking es la siguiente:

```
def jugadores_prensa(pedidos_prensa, sol_parcial, i, mejor_solucion):
```

Donde:

- pedidos_prensa: es una lista de listas, donde cada lista es el pedido de jugadores de un periodista.
- sol_parcial: una lista de jugadores la cual es la solución parcial al momento.
- i: el índice de la lista de pedidos de la prensa, es el pedido que se está analizando actualmente.
- mejor_solucion: una lista de jugadores seleccionados como mejor solución.

Comenzando a analizar el algoritmo, lo primero que nos encontramos es con una condición de corte:

```
if len(mejor_solucion) > 0 and len(sol_parcial) == len(mejor_solucion):
    return False
```

La misma chequea que la lista de jugadores actual tenga el mismo largo que la mejor solución, de ser así, no podríamos agregar otro jugador dado que tendríamos una solución de mayor tamaño a la obtenida como la mejor, y estamos buscando el mínimo conjunto.

Luego, antes de agregar un elemento, chequeamos si ese periodista tiene o no otro jugador de su lista en la solución actual. Si lo tiene, se avanza al siguiente periodista a través de una llamada recursiva:

```
if esta_en_solucion(sol_parcial, pedido_actual):
    return jugadores_prensa(pedidos_prensa, sol_parcial, i + 1, mejor_solucion)
```

La función está_en_solucion es la encargada de hacer el chequeo de la siguiente forma:

```
def esta_en_solucion(solucion, pedido):
    return any(jugador in pedido for jugador in solucion)
```

Si el pedido del periodista actual no está satisfecho, intenta agregar jugadores a la solución parcial y verifica si esta nueva solución parcial es válida (es_eleccion_valida). Si es válida, compara con la mejor solución encontrada hasta ahora y actualiza mejor_solucion si es más pequeña.

Luego, continúa la exploración llamándose a sí mismo de manera recursiva con la solución parcial actualizada. Si una de las opciones no es válida, se retrocede (sol_parcial.pop()) para probar otras combinaciones.

```
for jugador in pedido_actual:
    if cant_solucion > 0 and cant_jugadores + 1 >= cant_solucion:
        return True

    sol_parcial.append(jugador)
    cant_jugadores += 1

    if es_eleccion_valida(sol_parcial, pedidos_prensa[i:]):
        mejor_solucion[:] = sol_parcial
        cant_solucion = cant_jugadores
    else:
        valido = jugadores_prensa(pedidos_prensa, sol_parcial, i + 1,
    mejor_solucion)
        if not valido:
            sol_parcial.pop()
            break
    sol_parcial.pop()
    cant_jugadores -= 1
```

2.1. Complejidad

Vamos a analizar la complejidad de las funciones mencionadas anteriormente:

- es_eleccion_valida: para cada pedido de la prensa, se realiza una búsqueda en la solución, lo que lleva a una complejidad de $O(n*m)$ donde n es la cantidad de pedidos y m es el tamaño medio de los pedidos.
- esta_en_solucion: esta función tiene una complejidad de $O(m)$ ya que, en el peor de los casos, tiene que verificar todos los elementos del pedido.
- jugadores_prensa: la función utiliza la recursión y realiza un bucle sobre los jugadores del pedido actual. En el peor de los casos, jugadores_prensa deberá explorar todos los casos que son $O(2^n)$. Cada exploración le cuesta $O(m) + O(n*m) = O(n*m)$. Por lo tanto la complejidad queda $O(n*m*2^n)$.

3. Algoritmo Greedy

En esta sección, resolveremos el mismo problema utilizando un algoritmo greedy.

En este caso, la firma de la función es la siguiente:

```
def jugadores_prensa_greedy(pedidos_prensa)
```

Recibe unicamente pedidos_prensa que nuevamente representa para cada periodista, la lista de jugadores. Si no hay ningun pedido de prensa, la función retornará un array vacío.

Comenzamos preparando los datos:

```
apariciones_por_jugador = {}

for pedido in pedidos_prensa:
```

```
    for jugador in pedido:
        if jugador not in apariciones_por_jugador:
            apariciones_por_jugador[jugador] = 1
        else:
            apariciones_por_jugador[jugador] += 1
```

Se recorren todos los jugadores de todos los pedidos de prensa para poder obtener la cantidad de veces que figura cada uno.

Esto es necesario para poder priorizar a aquellos que figuren una mayor cantidad de veces. De esta forma, en cada paso, podemos minimizar lo máximo posible los pedidos que faltan por cubrir.

Por otro lado, inicializamos un listado para llevar registro sobre cuales son los pedidos que ya fueron cubiertos y cuales no:

```
pedidos_mapeados = []
for pedido in pedidos_prensa:
    pedidos_mapeados.append([False, pedido])
```

Además, creamos dos nuevas variables donde guardamos la solución que se va creando y la cantidad de pedidos que restan por cubrir:

```
pedidos_sin_cubrir = len(pedidos_prensa)
solucion = []
```

Una vez hecho esto, comenzamos con la resolución:

```
for _ in range(len(apariciones_por_jugador.values())):

    if pedidos_sin_cubrir == 0:
        break

    jugador_con_mas_apariciones = max(apariciones_por_jugador, key=
    apariciones_por_jugador.get)
    pedidos_cubiertos_con_nuevo_jugador = actualizar_pedidos_cubiertos_por_jugador(
        pedidos_cubiertos, jugador_con_mas_apariciones
    )

    pedidos_sin_cubrir -= pedidos_cubiertos_con_nuevo_jugador
    solucion.append(jugador_con_mas_apariciones)
    apariciones_por_jugador.pop(jugador_con_mas_apariciones)
```

Iteramos utilizando el vector de apariciones previamente creado. En cada paso, buscamos aquel jugador que tenga más apariciones. Con este dato, hacemos una actualización de los pedidos cubiertos:

```
def actualizar_pedidos_cubiertos_por_jugador(pedidos_cubiertos, jugador):
    nuevos_pedidos_cubiertos = 0

    for idx in range(len(pedidos_cubiertos)):
        if pedidos_cubiertos[idx][0]:
            continue

        if jugador in pedidos_cubiertos[idx][1]:
            pedidos_cubiertos[idx][0] = True
            nuevos_pedidos_cubiertos += 1

    return nuevos_pedidos_cubiertos
```

Por cada pedido se revisa si ya fue cubierto o no. Si el pedido fue cubierto, continúa con el próximo. Caso contrario, si el jugador que habiamos seleccionado forma parte del mismo, se actualizan los valores: el registro indicará que cubrimos el pedido. A medida que avanzamos, son cada vez menos los pedidos por cubrir.

Luego, agregamos el jugador a la solución y lo retiramos del vector de apariciones.

3.1. Complejidad

Vamos a analizar la complejidad de las funciones mencionadas anteriormente:

- apariciones_por_jugador: Para crear este vector se recorren todos los pedidos y por cada uno de ellos, se recorren todos los jugadores para poder obtener las cantidades. Siendo m la cantidad de pedidos y n la de jugadores, resulta en una complejidad $O(m * n)$
- pedidos_mapeados: Para inicializar cada pedido como no cubierto, se recorre cada uno de ellos por lo que la complejidad es $O(m)$
- solución: el bucle principal se repetirá como máximo tantas veces como cantidad de apariciones por jugador haya, la complejidad resulta en $O(n)$. Dentro de este bucle, para calcular el valor máximo entre las apariciones, también se tiene una complejidad de $O(n)$. En el caso de actualizar_pedidos_cubiertos_por_jugador, la complejidad es $O(m*n)$ ya que en el peor de los casos se recorren todos los pedidos y todos los jugadores de ese pedido.

La complejidad final resulta ser: $O(n) * (O(n) + O(m * n)) = O(n^2) + O(n^2 * m) = O(n^2 * m)$

3.2. Cota de aproximación

Para calcular la cota de aproximación del algoritmo Greedy, definiremos dos variables:

- n: cantidad de subconjuntos (o sea de listas de periodistas).
- m: cantidad de jugadores diferentes que hay en los subconjuntos.

Al elegir el primer jugador, estamos priorizando al que aparece en la mayor cantidad de subconjuntos, ya que esa es la condición por la cual decidimos ordenarlos.

Para continuar con el análisis, vamos a definir k como la cantidad de veces que aparece ese jugador, es decir, la cantidad de subconjuntos en los que se encuentra. Dado k, en el peor de los casos, al elegir al primer jugador nos quedarán $n - k$ subconjuntos sin cubrir y $m - 1$ jugadores disponibles. Finalmente vamos a definir j como la cantidad de jugadores disponibles en esos $n - k$ subconjuntos restantes.

Las definiciones quedan de la siguiente manera:

- k: Cantidad de subconjuntos en los que aparece el jugador más frecuente.
- $n - k$: Cantidad de subconjuntos que quedan por cubrir luego de agregar el primer jugador.
- j: Cantidad de jugadores disponibles en los $n - k$ conjuntos restantes luego de agregar el primer jugador.

Por otro lado, nuestra solución óptima en el mejor de los casos y para $n-k > 0$ necesita agregar al menos un (1) jugador más a la solución. Teniendo en cuenta esto, en nuestra aproximación, el peor caso se da si nos encontramos en un escenario de solapamiento donde todos los jugadores con más apariciones solo agreguen un subconjunto más dentro la solución final, más adelante veremos un ejemplo de este caso.

Por esto, podríamos decir que nuestra cota de error es $min(j - 1, (n - k) - 1)$, en ambos casos restamos el valor uno (1) porque suponemos que ese valor se encuentra en la solución ótima. De esta manera nos queda que la cota es la cantidad mínima entre:

- La cantidad restante de subconjuntos que no fueron cubiertos con el primer jugador.
- La cantidad de jugadores disponibles dentro de los subconjuntos que no fueron cubiertos por el primer jugador.

Pondremos dos ejemplos donde se ven las dos situaciones planteadas anteriormente. Presentaremos dos escenarios luego de haber elegido el elemento que aparece la mayor cantidad de veces (los subconjuntos los planteamos con números para que sea más sencilla la comprensión):

Tipo de caso	Caso más jugadores restantes que subconjuntos restantes	Caso más subconjuntos restantes que jugadores restantes
Subconjuntos input	1,8,5,4, 1,2,20,22 1,2,11 1,2,7 1,2,5,45 34,2,5,78,99 10,5,11,89	1,8,5,4, 1,2,20,22 1,2,11 1,2,7 1,2,5,45 2, 5, 5 2,5
n-k	2	4
j	8	2
min(j-1, (n-k)-1)	1	1
Solución óptima	2, 5	2,5
Solución greedy	1,2,5 (me alejo 1 de la real)	1,2,5 (me alejo 1 de la real)

Ejemplo 1: caso donde hay más jugadores restantes que subconjuntos restantes.

El primer elemento que se elige es el 1 porque es el que aparece en más subconjuntos, por lo tanto el 1 es parte de la solución. Luego, nos quedan los últimos dos subconjuntos sin cubrir: $\{34, 2, 5, 78, 99\}$ y $\{10, 5, 11, 89\}$ (en total son: $n - k = 2$) y nos quedan $j = 8$ jugadores restantes.

Ejemplo 2: caso donde hay más subconjuntos que jugadores restantes.

El primer elemento que se elige es el 1 porque es el que aparece en más subconjuntos, por lo tanto el 1 es parte de la solución. Luego, nos quedan $n - k = 3$ subconjuntos sin cubrir y $j = 2$ jugadores restantes.

Con esto, podríamos decir que nuestra aproximación puede alejarse de la solución óptima como máximo en $min(j - 1, (n - k) - 1)$, siendo j, n y k los valores previamente mencionados.

4. Mediciones empíricas

A continuación se presentan los resultados de las diferentes mediciones empíricas realizadas con datos generados aleatoriamente, en ellas compararemos tiempos de respuesta y tamaños de soluciones obtenidas.

Por cada tamaño de set de datos, se generaron sets con distinta distribución de jugadores dentro de los subconjuntos. Para esto se utilizó un parametro p que indica la probabilidad de que un jugador este o no dentro de un subconjunto.

Por cada combinacion tamaño-de-set-de-datos y valor-p se realizaron 200 mediciones y se tomó el promedio para evitar ruido.

Tamaños de set de datos: $1 < n < 30$

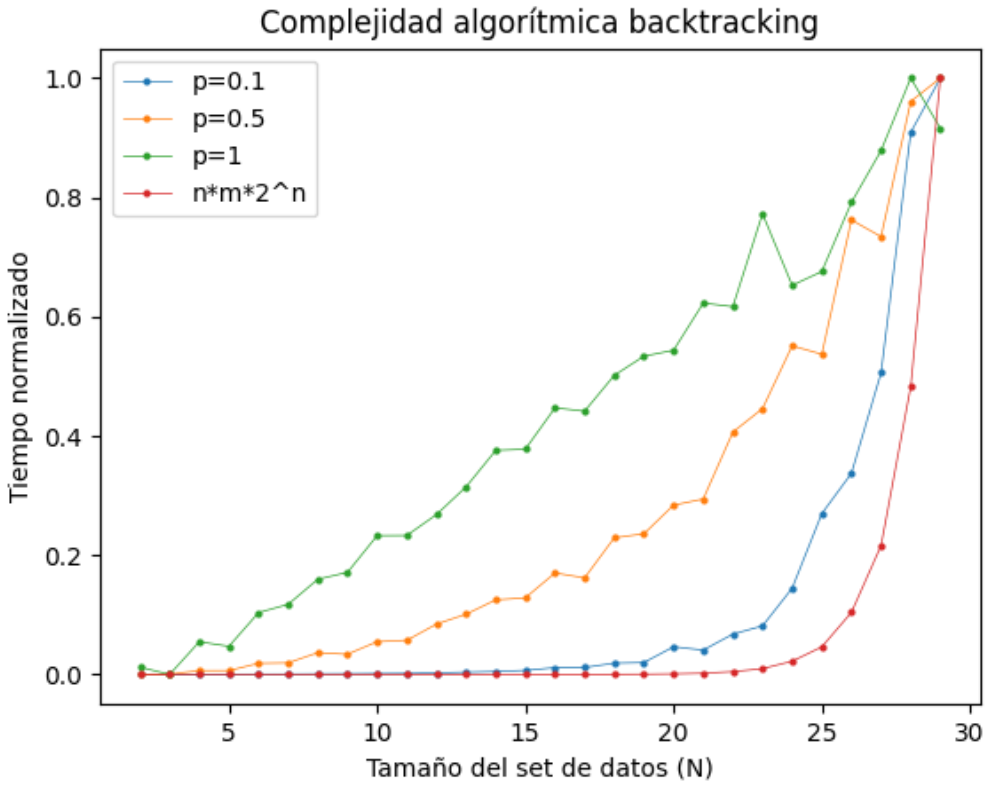
Cantidad de puntos: 28

Probabilidades = $\{0{,}1, 0{,}5, 1\}$

Mediciones por punto: 200

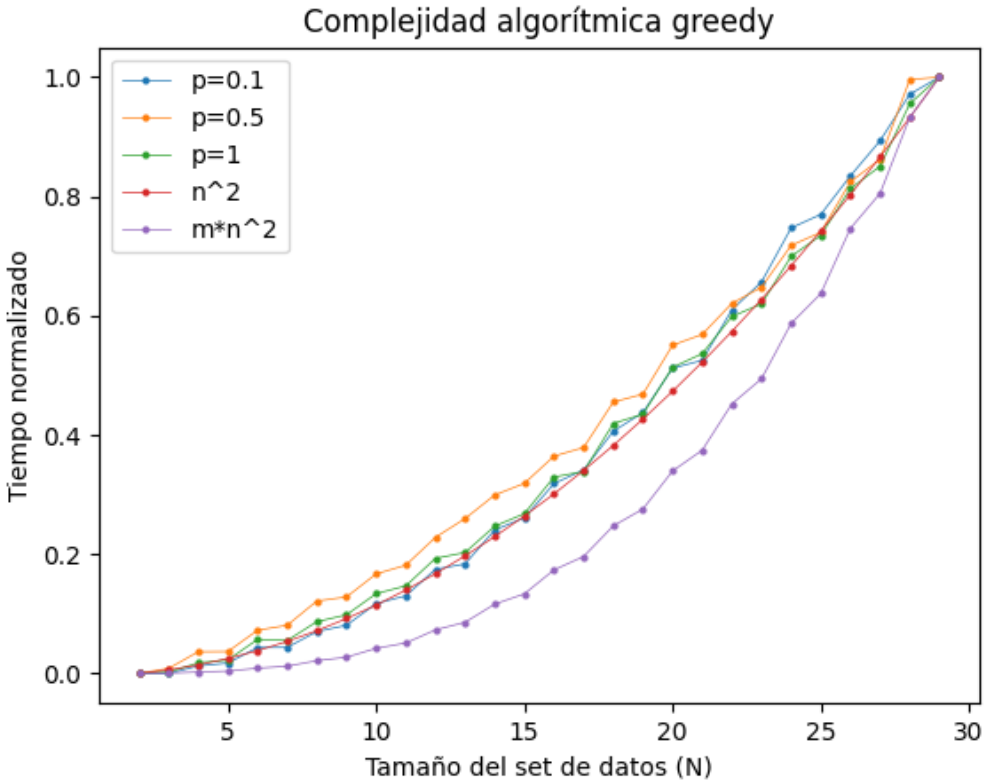
Mediciones totales: 16800

4.1. Medición de complejidad para la solución Backtracking



Se puede observar que cuando p disminuye, el algoritmo tarda más en devolver la solución. En el peor de los casos, el algoritmo tiene una complejidad de $O(n * m * 2^n)$.

4.2. Medición de complejidad para la solución Greedy



Teoricamente se calculó una complejidad de $O(m * n^2)$ pero empiricamente se observa una complejidad de $O(n^2)$. Esto puede deberse a la estrategia de generación de set de datos, donde $m = \frac{n}{2}$.

En el gráfico se puede apreciar que la variabilidad de p también afecta al algoritmo greedy pero de distinta manera. Si todos los jugadores aparecen en todos los subconjuntos o si la probabilidad es baja, la complejidad aumenta y se asemeja a $O(n^2)$. Con una probabilidad de 0.5, la complejidad aparenta ser apenas menor.

5. Comparación de la aproximación con el óptimo

Esta comparación consiste en resolver el mismo problema utilizando el algoritmo de backtracking y el greedy y asi comparar el óptimo con su aproximación.

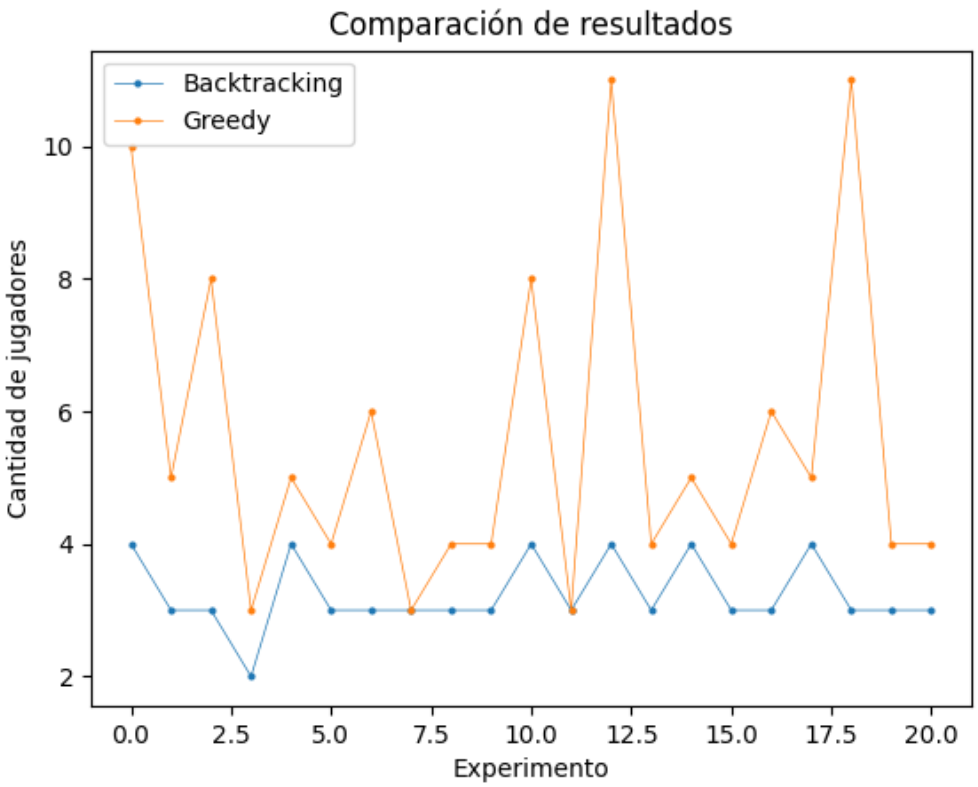
Se utilizaron los siguientes datos:

n = 30

m = 15

p = 0.3

Se generaron 21 casos diferentes con los datos anteriores y se compara el valor k del óptimo (la longitud del conjunto mínimo).



Se puede observar en el gráfico como la aproximación puede llegar a obtener el óptimo. Se observa que a veces aproxima bien y otras devuelve un valor demasiado alto comparado al óptimo. En todos los casos, la aproximación es mayor o igual al óptimo.

6. Conclusiones

A través del desarrollo del informe, pudimos deducir como el problema al que nos enfrentamos fue un problema NP-Completo para el cual pudimos generar la solución óptima a través de la utilización de un algoritmo del tipo Backtracking. Esta resolución se ve afectada en sus tiempos al aumentarse la cantidad de datos que debe manejar, por esto también se presentó una aproximación Greedy que nos permite brindar una solución aproximada de resolución que en la mayoría de casos no es la óptima, frente a esta aproximación no pudimos generar un set de datos acorde al peor caso para demostrar su complejidad teórica.