Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

13 de diciembre de 2023

Martín González Prieto
105738

Santiago Langer
107912

Camila Teszkiewicz
109660

Índice

1. Introducción: análisis del problema

El problema que se abordará en el siguiente trabajo consiste en resolver el problema Hitting set. En este caso puntual, el problema consiste en confeccionar una lista final de jugadores para que Scaloni arme el equipo para un amistoso. Se cuenta con varias listas de jugadores, las cuales corresponden a las pretenciones de ciertos periodistas. Para contentar a todos, Scaloni decide armar una lista que contenga al menos un jugador de cada lista de los periodistas, y que además sea la miníma posible.

Es decir, expresando el problema en términos matemáticos, se tiene $\boldsymbol{m}$ listas de jugadores $B_1, B_2, ...B_m$ que componen un conjunto de jugadores A, tal que $B_i \subseteq A \forall i$, y se requiere una lista C de modo que $C \cap B_i \neq 0$, con $|C| \leq k$[1].

1.1. Hitting set problem: problema NP

Para comprobar que dicho problema es un problema NP, basta con comprobar que cuenta con un certificador eficiente, es decir poder verificar una solución en tiempo polinomial. Para esto, planteamos la siguiente estructura: un diccionario de peridistas donde cada uno contenga una lista de sus preferencias.

Por lo tanto, recorriendo la lista de convocados, se corrobora si pertenece a la lista de algún periodista, si es así se agrega dicho periodista a un set. Si el algoritmo logra terminar con la misma cantidad de periodistas en el set que en la lista original, quiere decir que la lista de convocados cumple con las condiciones del problema.

```python
def chequear_solucion(periodistas : dict, convocados : list):
    aux = set()
    for convocado in convocados:
        for periodista in periodistas.keys():
            if convocado in periodistas[periodista]:
                aux.add(periodista)

    return len(aux) == len(periodistas.keys())
```

En lo que respecta a la complejidad del verificador, teniendo en cuenta que la cantidad máxima de jugadores convocados es $\boldsymbol{k}$ y la cantidad de periodistas en $\boldsymbol{m}$, se puede decir que cuenta con una complejidad $O(m * k)$ y por lo tanto con un tiempo polinomial. Cabe aclarar que tanto el acceso al diccionario, como la corrobaración del jugador dentro de la lista de preferencias, como la adición del periodista al set, son operaciones de complejidad $O(1)$.

1.2. Hitting set problem: problema NP-Completo

Por otro lado, para corroborar que el problema de Hitting set, es un problema NP-Completo se requiere realizar una reducción con otro problema NP-Completo. Para esto se debe corroborar que el problema NP-Completo $\boldsymbol{Y}$ se puede reducir a nuestro problema $\boldsymbol{X}$ de Hitting set problem, es decir $Y \leqslant_p X$.

Para esto se requiere encontrar algún problema que dada una cantidad polinomial de operaciones pueda transformarse en nuestro problema de Hitting set. Dentro de los posibles problemas NP-Completos conocidos, el más acorde nos pareció que era SAT, satisfability problem.

Para comenzar planteamos un caso generico de SAT con ciertas cláusulas:

$$C_1 = x_1 \vee x_3 \vee x_4;$$
$$C_2 = \overline{x_1} \vee x_2 \vee x_5;$$
$$C_3 = x_3 \vee \overline{x_5} \vee x_6;$$
$$C_4 = x_5;$$
$$C_5 = \overline{x_7};$$

$$\mathbf{A} = (\ x_1, x_2, x_3, x_4, x_5, x_6, x_7, \overline{x_1}, \overline{x_2}, \overline{x_3}, \overline{x_4}, \overline{x_5}, \overline{x_6}, \overline{x_7})$$

[1] con k=26, incluyendo 11 titulares y 15 suplentes

Por lo tanto, para resolver dicho problema, se requiere una secuencia de las instancias $X_i, i \in [1, 7]$ de modo que se cumpla $C_1 \wedge C_2 \wedge C_3 \wedge C_4 \wedge C_5$.

Ahora bien, el desafío en el que nos encontramos es como utilizar la resolución del problema de Hitting Set para conseguir dicha secuencias de instancias. Para esto nos encontramos con dos desafíos:

1. Transformar las cláusulas en subconjuntos de Hitting Set.
2. Forzar que x_i y $\overline{x_i}$ no estén en la solución a la vez.

1.2.1. De cláusulas SAT a subconjuntos Hitting Set

Para esto, agarramos dichas cláusulas, las tranformamos directamente como subconjuntos B_i.

$$B_1 = \begin{pmatrix} x_1 \\ x_3 \\ x_4 \end{pmatrix} \; B_2 = \begin{pmatrix} \overline{x_1} \\ x_2 \\ x_5 \end{pmatrix} \; B_3 = \begin{pmatrix} x_3 \\ \overline{x_5} \\ x_6 \end{pmatrix} \; B_4 = \begin{pmatrix} x_5 \end{pmatrix} \; B_5 = \begin{pmatrix} \overline{x_7} \end{pmatrix}$$

Esta traducción directa permite que las variables que deban ser verdaderas, en el problema de SAT, sean también aquellas que queden seleccionadas en la solución de Hitting Set. Sin embargo, no es suficiente debido a los complementos, los cuales se explicarán a continuación.

1.2.2. Variables y complementos

Con el fin de que tanto x_i como $\overline{x_i}$ no se encuentren en la solución, se agregan n cláusulas de modo que: $D_i = x_i \vee \overline{x_i}$. Por lo que, en este caso puntual, los subconjuntos para el Hitting set de dichas nuevas cláusulas quedarían:

$$D_1 = \begin{pmatrix} x_1 \\ \overline{x_1} \end{pmatrix} \; D_2 = \begin{pmatrix} x_2 \\ \overline{x_2} \end{pmatrix} \; D_3 = \begin{pmatrix} x_3 \\ \overline{x_3} \end{pmatrix} \; D_4 = \begin{pmatrix} x_4 \\ \overline{x_4} \end{pmatrix} \; D_5 = \begin{pmatrix} x_5 \\ \overline{x_5} \end{pmatrix} \; D_6 = \begin{pmatrix} x_6 \\ \overline{x_6} \end{pmatrix} \; D_7 = \begin{pmatrix} x_7 \\ \overline{x_7} \end{pmatrix}$$

Lo que faltaría es asegurar que no se pueda seleccionar más de una variable del mismo subconjunto D_i. Para esto, hace falta tener en cuenta el valor K de Hitting set. Este valor es la cota máxima del tamaño de la solución, tal que $|C| \leq K$. Como la solución mínima, en este caso, es tener una variable por subconjunto y dado que hay n subconjuntos, K=n. De modo que si $|C| > K$ es porque se selecciona más de una variable por subconjunto (es decir, se seleccionó tanto la variable como su complemento).

1.2.3. Solución del primer ejemplo

$$C = \begin{pmatrix} \overline{x_1} \\ \overline{x_2} \\ x_3 \\ \overline{x_4} \\ x_5 \\ \overline{x_6} \\ \overline{x_7} \end{pmatrix}$$

Dicha solución C, tiene tamaño K (K=7) y cumple con $\overline{x_1} \in D_1; \overline{x_2} \in D_2; x_3 \in B_1, B_3, D_3; \overline{x_4} \in D_4; x_5 \in B_2, B_4, D_5; \overline{x_6} \in D_6; \overline{x_7} \in B_5, D_7$

Por lo tanto, considerando ese conjunto de solución, se pueden definir el valor de sus variables y complementos.

$(x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 0, x_5 = 1, x_6 = 0, x_7 = 0)$

$(\overline{x_1} = 1, \overline{x_2} = 1, \overline{x_3} = 0, \overline{x_4} = 1, \overline{x_5} = 0, \overline{x_6} = 1, \overline{x_7} = 1)$

Con esto valores queda resuelto el problema SAT de la siguiente manera:

$$C_1 = x_1 \vee 1 \vee x_4 = 1;$$
$$C_2 = x_1 \vee x_2 \vee 1 = 1;$$
$$C_3 = 1 \vee \overline{x_5} \vee x_6 = 1;$$
$$C_4 = 1;$$
$$C_5 = 1;$$

Así queda demostrado que $SAT \leqslant {}_pHittingSet$, y por lo tanto Hitting Set problem es NP-Completo.

1.2.4. Segundo ejemplo - SAT sin solución

Otro ejemplo más sencillo dónde un solucionador del Hitting set soluciona de manera correcta un problema SAT es:

$$C_1 = x_1;$$
$$C_2 = x_2;$$
$$C_3 = \overline{x_1} \vee \overline{x_2};$$

Por lo que, siguiendo los lineamientos anteriores, los subconjuntos quedarían:

$$B_1 = (x_1)\ B_2 = (x_2)\ D_1 = \begin{pmatrix} x_1 \\ \overline{x_1} \end{pmatrix}\ D_2 = \begin{pmatrix} x_2 \\ \overline{x_2} \end{pmatrix}$$

Y la solución miníma sería:

$$C = \begin{pmatrix} x_1 \\ x_2 \\ \overline{x_1} \end{pmatrix}$$

En este caso, queda en evidencia que SAT no debería tener solución, y que Hitting Set está seleccionando tanto x_1 como $\overline{x_1}$. Es aquí donde el valor de K toma relevancia, ya que en este caso K=2 pero el tamaño de la solución es 3, por lo que no cumple $|C| \leq K$. Por lo tanto, Hitting Set devolvería falso y SAT no tendría solución.

2. Solución óptima por Backtracking

2.1. El comienzo: fuerza bruta

Para imaginar una solución por fuerza bruta, primero pensamos cómo sería el espacio de soluciones posibles. Dado un número n, cualquier solución estaría conformada por una cantidad n de jugadores, tal que los seleccionados contenten a todos los periodistas posibles. Esto ya nos da una pauta sobre las posibilidades que tendremos que explorar: hay que probar distintas listas de jugadores hasta que alguna cumpla con lo pedido.

Luego, ¿Cómo hallaríamos el mínimo n posible tal que haya solución? Realizamos una búsqueda usando un método del estilo de divide y conquista. En rigor, una segunda aplicación podria probar linealmente desde n = 1, e ir incrementando el tamaño de la lista hasta hallar la primera solución válida - pero nuestra solución es más eficiente, y será detallada más adelante.

Empecemos hablando de la versión de decisión del hitting-set problem. Dado un conjunto A de jugadores, busco satisfacer todos los subconjuntos Bi con n jugadores. Básicamente, es una permutación sin repetición de conjuntos no-ordenados. Como bien sabemos de la teoría de probabilidades, lo que estamos buscando es el numero combinatorio. Siendo r la cantidad total de jugadores solicitados por la prensa:

$$\binom{r}{n} = \frac{r!}{(r-n)!n!}$$

Para dar una idea de la magnitud, supongamos un conjunto A de 40 jugadores, y n = 10, el algoritmo probaría 847,660,528 soluciones posibles.

$$\frac{40!}{30!10!} = 847,660,528$$

Observación: La idea de subconjuntos no ordenados podría implicar una "poda". Lo descartamos como tal por ser una caracteristica intrinseca del dominio del problema.

2.2. Razonando un poco: la solución

Un algoritmo por backtracking consiste en ir probando soluciones pero con cierto criterio, para evitar probar todas las opciones. Razonemos entonces el problema empezando por las obviedades: si un subconjunto contiene un solo elemento, ese elemento deberá incluirse en la solución.
Sabiendo esto, sabemos que nuestro mínimo de jugadores a elegir será el número de subconjuntos con solo un elemento. Así ya encontramos una **primera poda** que nos permite descartar cualquier solución que no incluya a algún elemento de un subconjunto unitario.
Una vez que incluimos un jugador en nuestro conjunto solución, podemos recorrer linealmente todo el resto de subconjuntos y descartar aquellos que tienen al jugador elegido. Es muy probable que en el recorte quitemos a más de un jugador de la lista de soluciones posibles. Esto es nuestra segunda **segunda poda**.

Profundicemos un poco en cómo funciona el algoritmo. En general, cuantos menos elementos del conjunto A formen parte del subconjunto B, más difícil será satisfacer al subconjunto B. De la misma manera que lo pensamos con subconjuntos unitarios, sucede con binarios. Si tengo un subconjunto con solo dos elementos, alguno de ellos debería formar parte de la solución. Podemos descartar entonces todas las soluciones que no contengan alguno de los dos elementos. Lo mismo sucede con tres, cuatro, etc. La diferencia, cuando se va incrementando la cantidad de elementos, es que cuanta más cantidad de elementos tenga el subconjunto B, más probable es que alguno varios de esos elementos formen parte de otro subconjunto. Por lo tanto, comenzamos probando con las combinaciones que satisfacen a los subconjuntos de menor cantidad de elementos, esperando hallar rápidamente a los elementos escenciales para nuestra solución.

2.3. Explicación y análisis del algoritmo

La solución por Backtracking utiliza tanto división y conquista como backtracking para encontrar el óptimo. Con la función $backtrackingRecursivo()$ planeamos un problema de decisión donde vemos si existe solución para cierto número de jugadores elegidos. Teniendo esta función, hacemos una busqueda binaria donde probamos distintos valores hasta llegar al mínimo de convocados que es válido.

Llamando K al número de jugadores a convocar, mientras la diferencia entre el último K que no da una solución válida y el mejor K que genere una solución válida no sea 1 [2], el algoritmo irá probando distintos valores de K jugadores convocados para saber si forman una solución. Mediante división y conquista (similar a búsqueda binaria) se irá acercando al mínimo K que forme una solución válida hasta encontrarlo.
Al utilizar DyQ nos ahorramos probar miles de combinaciones que ya detectamos que no llegarán a una solución válida, o que ya sabemos que existen mejores que son válidas.

[2] La diferencia es 1 ya que el máximo número con solución inválida será el siguiente número entero al mínimo K de solución válida.

2.4. Código de la solución por Backtracking

El código, tanto de la DyQ como la busqueda con BT bajo cierto valor de decisión es el siguiente:

División y conquista:

```
def solution_by_backtracking(data : ProblemData):
    numero_elegidos_actual = len(data.B_subsets.keys())
    numero_elegidos_iter_anterior = 0
    minimo_numero_elegidos_valido = 0
    maximo_numero_fallido = 0
    diferencia_fallido_y_minimo_valido = 2
    dicc_ordenado = setup.ordenar_diccionario(data.B_subsets)

    while diferencia_fallido_y_minimo_valido != 1:
        posibles_convocados = set()
        hay_solucion = backtracking_recursivo(dicc_ordenado, posibles_convocados, numero_elegidos_actual)
        aux_numero_elegidos_actual = numero_elegidos_actual

        if hay_solucion:
            minimo_numero_elegidos_valido = numero_elegidos_actual
            #Con el n que probo antes no habia encontrado solucion y con este si
            if numero_elegidos_iter_anterior < minimo_numero_elegidos_valido:
                numero_elegidos_actual = (numero_elegidos_iter_anterior + numero_elegidos_actual) // 2

            #Con el anterior habia solucion y con este tambien
            else:
                if maximo_numero_fallido != 0:
                    numero_elegidos_actual = maximo_numero_fallido + 1
                else:
                    numero_elegidos_actual = numero_elegidos_actual // 2

            numero_elegidos_iter_anterior = aux_numero_elegidos_actual
            convocados_definitivos = posibles_convocados

        else:
            maximo_numero_fallido = numero_elegidos_actual
            #Con el anterior no habia solucion y con este tampoco
            if minimo_numero_elegidos_valido == 0 or numero_elegidos_iter_anterior < minimo_numero_elegidos_valido:
                if minimo_numero_elegidos_valido > 0:
                    numero_elegidos_actual = minimo_numero_elegidos_valido - 1
                else:
                    numero_elegidos_actual = numero_elegidos_actual * 2
            #Con el anterior habia solucion y con este no
            else:
                numero_elegidos_actual = (numero_elegidos_iter_anterior + numero_elegidos_actual) // 2
        numero_elegidos_iter_anterior = aux_numero_elegidos_actual

        diferencia_fallido_y_minimo_valido = abs(maximo_numero_fallido - minimo_numero_elegidos_valido)

    return minimo_numero_elegidos_valido, convocados_definitivos

```

Problema de decisión con backtracking:

```
def backtracking_recursivo(periodistas:dict, convocados:set, n_minimo):
    if len(periodistas.keys()) == 0:
        return True
    if len(convocados) == n_minimo:
        return False

    eliminados = {}
    siguiente_periodista = next(iter(periodistas.items()))
    dicc_periodistas_copia = periodistas.copy()

    for jugador in siguiente_periodista[1]:
        convocados.add(jugador)
        for periodista in periodistas:
            if jugador in periodistas[periodista]:
                eliminados[periodista] = dicc_periodistas_copia.pop(periodista)

        if backtracking_recursivo(dicc_periodistas_copia, convocados, n_minimo):
            return True

        #Si no es solucion, vuelvo para atras
        convocados.remove(jugador)
        devolver_periodistas(dicc_periodistas_copia, eliminados)
    return False

```

Es interesante ver el código, porque hay un detalle de implementación a observar. Toda nuestra solución se basa en que decidir si un jugador forma parte de los seleccionados de un periodista es una operación O(1). Ya sabemos que así puede ser realizado, pero debemos hacerlo con cuidado. Es por esto que elegimos un diccionario (periodistas), donde cada una de sus claves es un subconjunto y los valores son sets de elementos.
Lo mismo sucede con el set *convocados* ya que remover un elemento de un set, en Python 3.10, es una operación O(1), no así removerlo de una lista.

2.5. Complejidad de la solución con Backtracking

Llaremamos N a la cantidad de subconjuntos B, es decir la cantidad de periodistas. Primero ordenamos por cantidad de jugadores en la preferencia de cada periodista. Esto es $O(N * logN)$ [3] Luego realizamos una accion [4] N veces - for periodista in periodistas - y luego llamamos a la función recursiva con:

$$N - periodistasSatisfechos$$

En el peor de los casos, nuestro jugador seleccionado solo satisface a un periodista. Es decir, en el peor de los casos:

$$N - periodistasSatisfechos = N - 1$$

Si esa tendencia continuara llamariamos N! veces. Luego hacemos una busqueda binaria hasta encontrar el mínimo k. Dicha busqueda arrancaría en k = N (para el cual seguro hay solución) y decrecería. Esta búsqueda binaria sería $O(logN)$.

Finalmente, nos queda la siguiente cuenta: $O(N! * logN + NlogN) = O(N! * logN)$. Quitando los términos menores, como corresponde a la notación Big Oh, nos queda que la complejidad del algoritmo por Backtracking es:

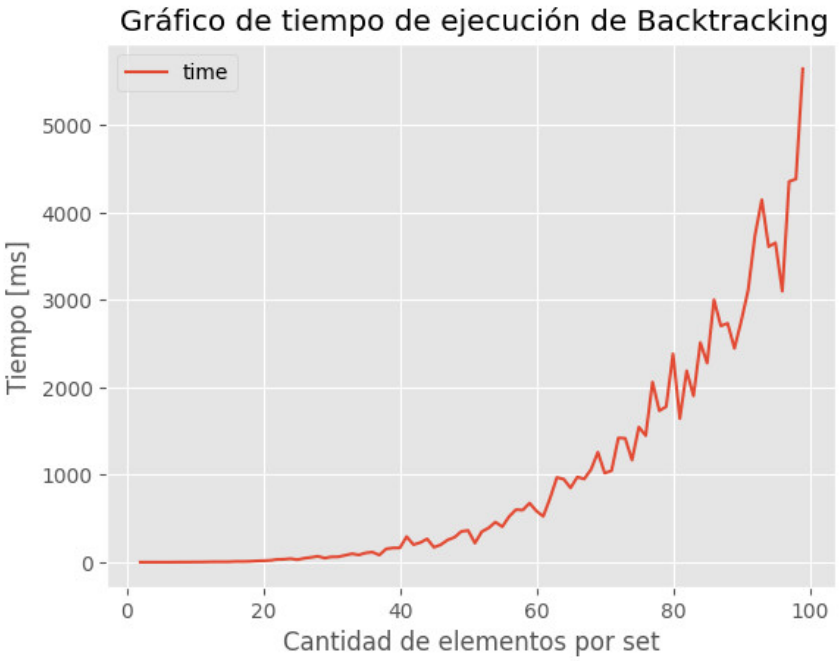
$$O(N! * logN)$$

Esta expresión de complejidad en realidad es simplemente una cota superior, ya que la poda disminuye el tiempo de ejecución considerablemente.

[3] En trabajos anteriores hemos justificado que la funcion sorted() incluída en python 3.10 garantiza dicha complejidad. También es importante tener en cuenta que la complejidad de este primer ordenamiento será despreciado por la complejidad del resto del algoritmo

[4] La acción consta en quitar a cada periodistas que ya contemple al jugador seleccionado

2.6. Análisis: mediciones de tiempos



En el gráfico se puede ver claramente como la curva descripta coincide con la complejidad previamente desarrollada:

$$O(N! * logN)$$

3. Solución óptima por Programación Lineal Entera

En lo que respecta a la solución del problema por PLE (Programación Lineal Entera) la idea es sencilla. Basicamente consiste en resolver un problema de optimización, dado un sistema de ecuaciones lineales y una ecuación objetivo a la cual minimizar o maximizar. Primeramente, dentro de este problema, se puede ver que las variables x_i que contendrán las ecuaciones son los jugadores, y lo que se busca es minimizar la cantidad de jugadores convocados que cumplan con ciertas condiciones. Por lo tanto, las variables valdrán 1 si el jugador debe ser convocado y 0 en el caso contrario, y la ecuación objetivo será:

$$\sum_{i=1}^{a} x_i$$

con a como la cantidad total de jugadores.

Por otro lado, dichas condiciones para las varibles se verán determinadas por el sistema de ecuaciones. Es por esto que se puede plantear que cada subconjunto B_j será la sumatoria de las $x_i \in B_j$ (jugadores que forman parte del subconjunto) y dicha sumatoria deberá ser mayor igual a 1 para que todos los subconjuntos tengan al menos un jugador dentro del conjunto de convocados. Es decir:

$$\begin{cases} B_1 = x_a + ... + x_n \geq 1 \\ ... \\ B_j = x_b + ... + x_m \geq 1 \end{cases} \tag{1}$$

Al menos un elemento de cada subconjunto deberá valer uno, es decir que al menos un jugador de cada subconjunto es elegido.

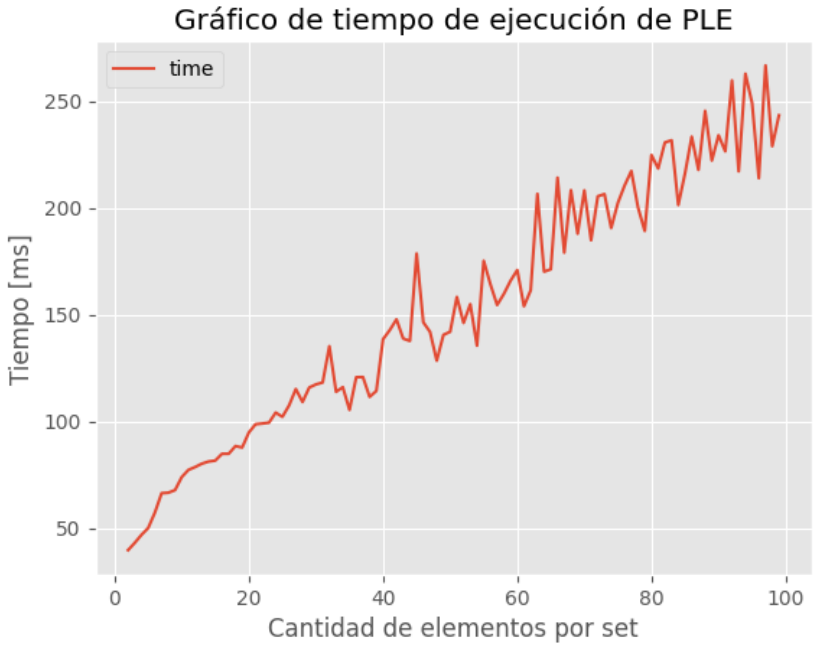
3.1. Código

Para poder realizar la optimización esperada se utiliza la libreria **PuLP** que no solo sirve para resolver el problema mediante Simplex, sino que tambien se hace uso de funciones auxiliares para determinar el sistema de ecuaciones.

```
def solution_by_lineal_programming(data : ProblemData):
    journalists = data.B_subsets
    players_list = data.players_as_list()

    problem_variables = [ pulp.LpVariable(player, cat="Binary") for player in
    players_list ]

    subsets_with_variables = prepocessing_subsets(journalists, problem_variables)

    problem = pulp.LpProblem("players_selected", pulp.LpMinimize)

    for journalist in subsets_with_variables:
        preferences_list = subsets_with_variables[journalist]
        problem += pulp.LpAffineExpression([(preferences_list[i], 1) for i in range
    (len(preferences_list))]) >= 1

    problem += pulp.LpAffineExpression([(problem_variables[i], 1) for i in range(
    len(problem_variables))])
    problem.solve()

    convocked = formating_solution(problem_variables)

    return len(convocked), convocked

```

Como se puede ver en el código, se utilizan funciones como **LpVariable** para setear binariamente a los jugadores como variables, y luego **LpAffineExpression** para inicializar el sistema de ecuaciones y tambien la ecuación objetivo que se agrega por último al problema.

3.2. Mediciones de tiempo



3.3. Análisis: comparación de solución por PL con solución por Backtracking

	5	7	$10p$	$10t$	$10v$	15	20	50	75	100	200
Esperados	2	2	3	6	10	4	5	6	8	9	9
BT	2	2	3	6	10	4	5	6	8	9	9
PL	2	2	3	6	10	4	5	6	8	9	9

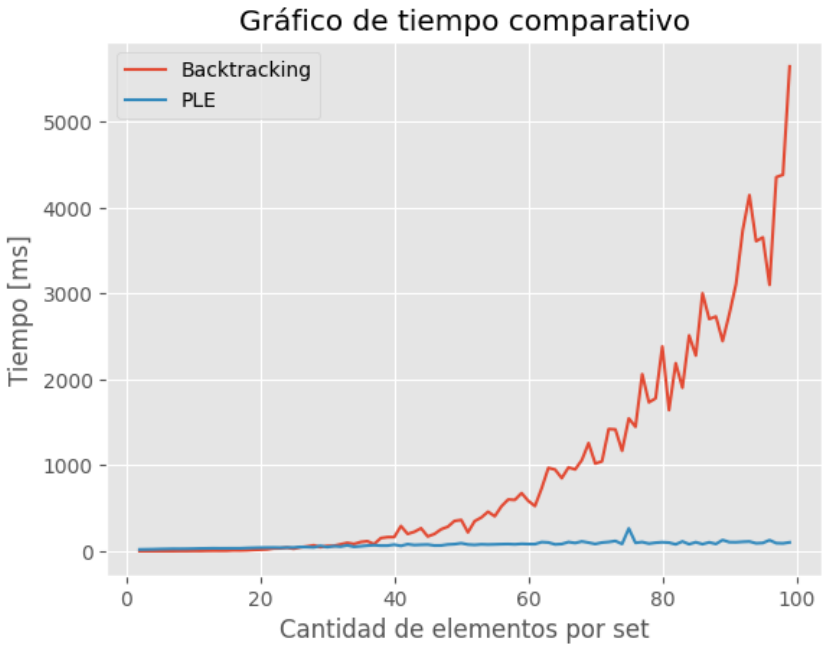
Cuadro 1: Mínimo de jugadores para sets de la cátedra y sus resultados esperados. Resuelto con Backtracking y con Programación Lineal

	20	30	40	50	70	100	130	150	200
BT	4	7	7	8	9	11	12	12	13
PL	4	7	7	8	9	11	12	12	13

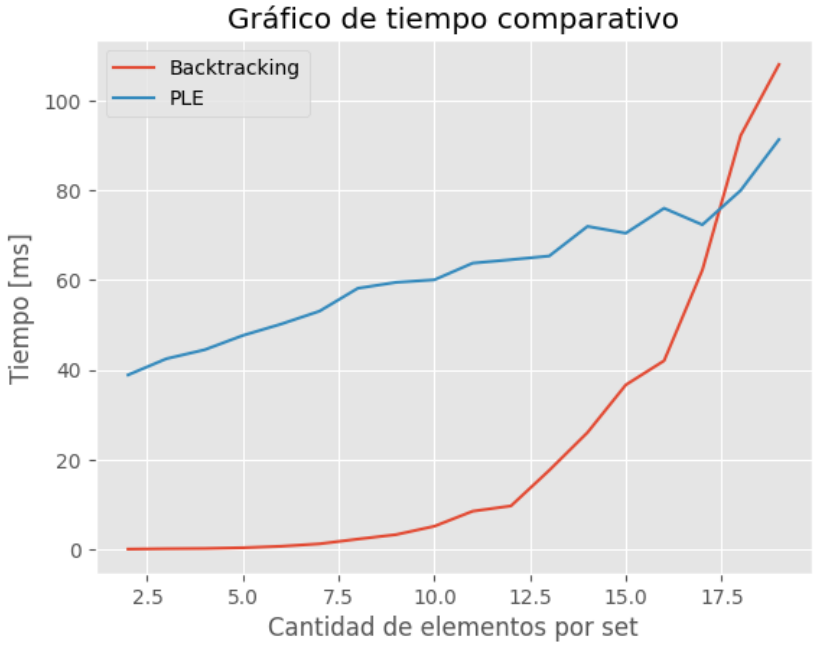
Cuadro 2: Mínimo de jugadores para sets propios, resuelto con Backtracking y con Programación Linal

Como podemos ver, ambas soluciones llegan a los mismos mínimos para los mismos sets de datos. Esto nos permite **corrobar la correctitud de los algoritmos implementados.**

3.4. Análisis: comparación de tiempos de ejecución entre Backtracking y PL



En el gráfico se puede notar como la solución por backtracking representa una gran crecimiento temporal a medida que aumenta la cantidad de sets, debido a su alta complejidad temporal. Por otro lado, la solución por programación lineal mantiene una performace con poca amplitud (entre 50 y 250 ms).



Cabe destacar que la solución por backtracking soluciona el problema más rápido que Programación Lineal entre sets de 2 a 17 elementos. Esta ventaja no se mantiene con sets más grandes.

4. Solución por aproximación con Programación Lineal

Tal como lo indica el enunciado, el modelo para Programación Lineal continua será el mismo que para PL entera, relajando la restricción sobre x_i. Esta ahora será:

$$0 \leq x_i* \leq 1$$

De la misma manera, el enunciado nos impone considerar como parte de la solución a todos los $x_i* \geq 1/b$, siendo b el tamaño del subconjunto B más grande. Llamamos T a la solución (óptima) usando PLE. Es decir:

$$T = \sum_{i=1}^{a} x_i$$

Ya sabemos que cada variable x_i suma 1, si es que suma, mientras que cada variable x_i* está acotada entre $\frac{1}{b} \leq x_i \leq 1$ por lo tanto,

$$\sum_{i=1}^{a} x_i \geq \frac{1}{b} \sum_{i=1}^{a} x_i*$$

Llamamos $\dot{T}$ a la solución aproximada con PL. Entonces:

$$T * b \geq \dot{T}$$

Hemos hallado una cota superior para la aproximación, que resulta ser una b-aproximación de un b-hitting set. Calculamos entoncres $r(A)$

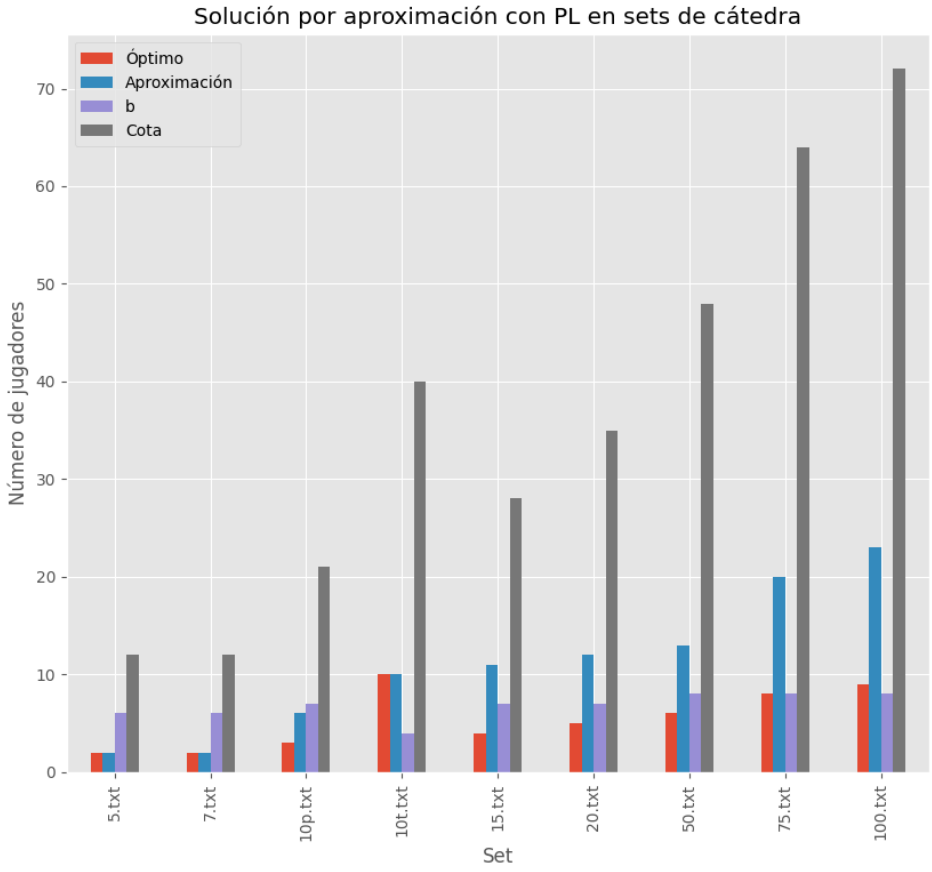
$$\frac{\dot{T}}{T} \leq r(A) \rightarrow r(A) = b$$

.

4.1. Comparación de resultados: sets de prueba provistos por la catedra

	5	7	10p	10t	15	20	50	75	100
Optima	2	2	3	10	4	5	6	8	9
Aproximacion	2	2	6	10	11	12	13	20	23
b	6	6	7	4	7	7	8	8	8
Cota	12	12	21	40	28	35	48	64	72

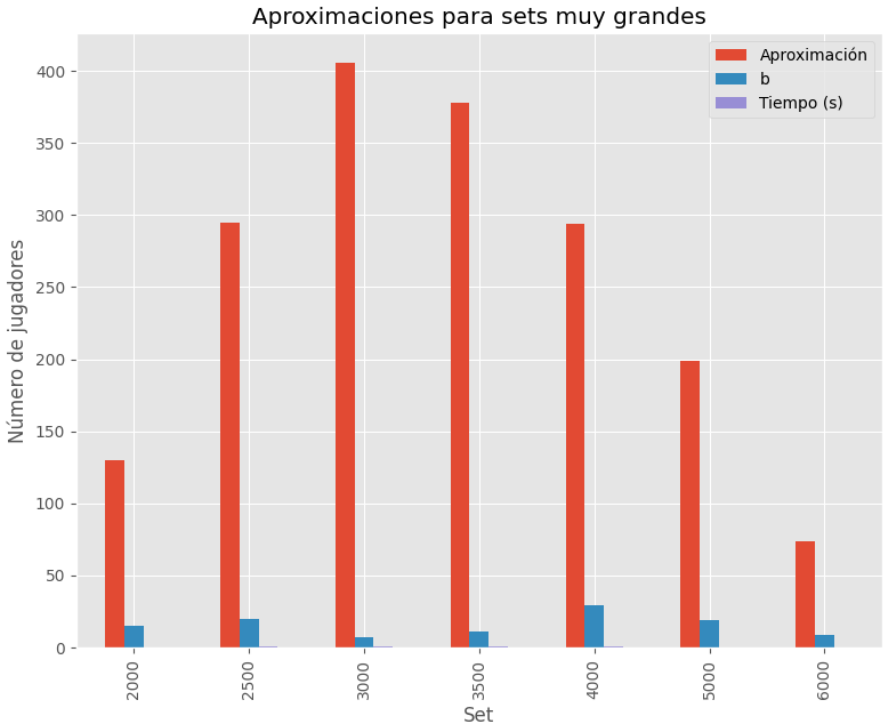
Cuadro 3: Resultado óptimo por programación lineal entera, aproximación, y cota



A continuación, usamos PL para volumenes de datos muy grandes: esta misma lista se puede corroborar en el jupyter notebook adjunto al informe.

	6000	5000	4000	3500	3000	2500	2000
Aproximacion	74	199	294	378	406	295	130
b	9	19	29	11	7	20	15
$Tiempo(s)$	0.06	0.25	1.22	0.95	0.69	0.64	0.20

Cuadro 4: Resultado aproximado usando programación lineal



Observamos que el algoritmo propuesto demora tiempos considerablemente pequeños en procesar sets considerablemente grandes.

Si bien no es da la respuesta óptima, aproximar por programación lineal continua nos permite tener algún resultado en un tiempo de respuesta corto, en particular para sets muy grandes.

5. Solución por aproximación con algoritmo Greedy

Como ya mencionamos, a partir de cierto número de subsets los métodos óptimos tardan muchisimo tiempo en responder. Si bien ya implementamos una forma de aproximar por programación lineal, también implementamos un algoritmo greedy que nos da una respuesta no óptima, pero en tiempo polinomial.

5.1. Algoritmo Greedy

Para obtener una solución pensamos el siguiente criterio Greedy:

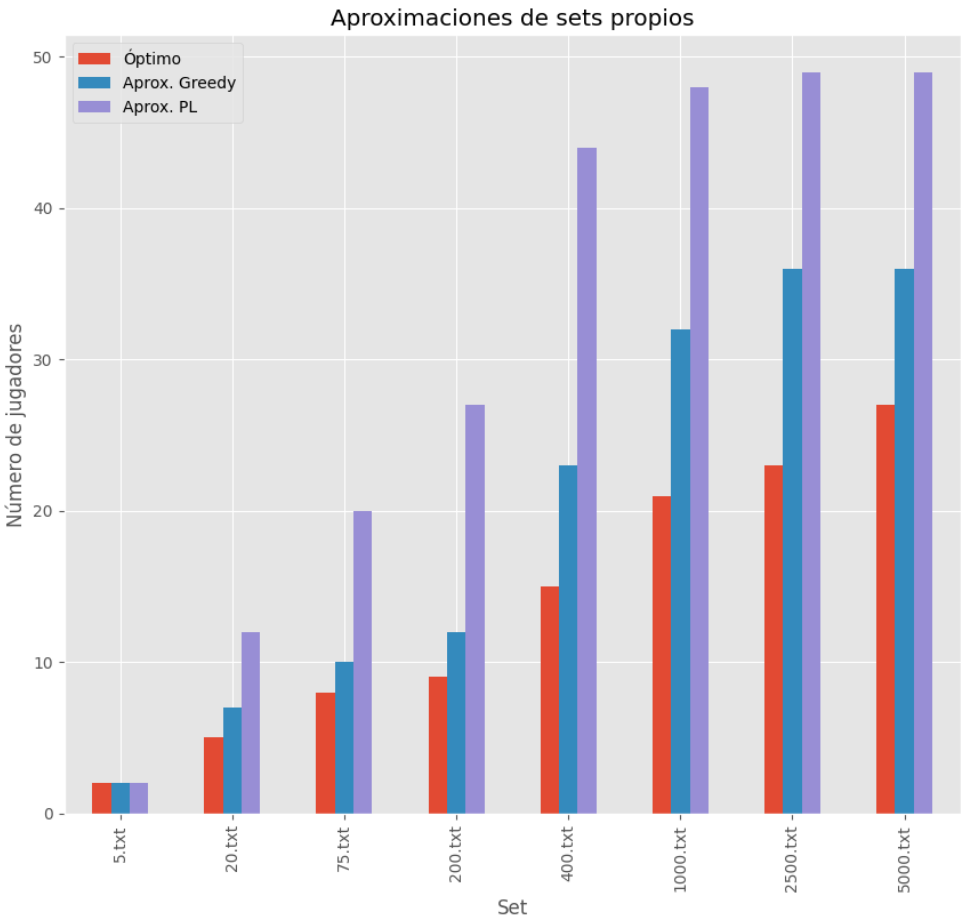
Para cada subset, elegir el jugador con más apariciones entre todos los subsets (que esté incluido en ese subset).

El objetivo será elegir a los jugadores que más prefieren los periodistas, con el objetivo de minimizar el número de jugadores elegidos. Para esto, primero contamos las apariciones de cada jugaddor, luego las ordenamos por cantidad y por último recorremos todos los subsets eligiendo a los jugadores más populares.

```
1  def aproximation_by_greedy(problem_data : ProblemData):
2      choosen_players = set()
3      appearances_by_player = count_appearances_by_player(problem_data)
4      ordered_list_by_appereance = ordered_list_by_apperance_from_dict(
       appearances_by_player)
5
6      for journalist in problem_data.B_subsets:
7          subset = problem_data.B_subsets[journalist]
8          choosen_players.add(choose_most_appeared_player_in_subset(subset,
       ordered_list_by_appereance))
9      return len(choosen_players), list(map(lambda tuple: tuple[0], choosen_players))
10
```

5.2. Resultados de la aproximación Greedy contra PL y el óptimo

A continuación se pueden ver los resultados del mínimo de jugadores necesarios para resolver el problema, según cada algoritmo. Para la solución optima se utilizó la implementación de programación lineal, que fue capaz de dar el resultado en un tiempo razonable para 2500 sets de hasta 5000 elementos.



	5	20	75	200	400	1000	2500	5000
Optimo	2	5	8	9	15	21	23	27
AproxPL	2	12	20	27	44	48	49	49
Greedy	2	7	10	12	23	32	36	36

Cuadro 5: Resultados de los distintos métodos

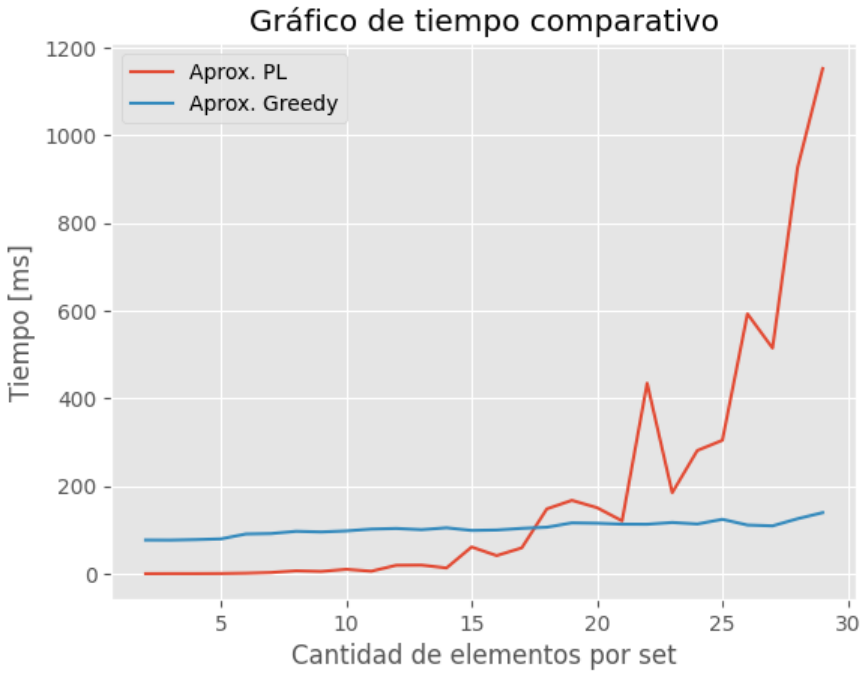
Como se puede ver, el algoritmo greedy implementado no es óptimo, pero da mejores resultados que la aproximación por programación lineal.
También es interesante notar que no es determinista. En algunos casos si se vuelve a correr bajo un mismo set de datos, da mínimos de jugadores distintos. Entendemos que es así ya que cuando un mismo jugador tiene el mismo número de apariciones, el orden que hará el ordenamiento los deja en posiciones distintas cada vez.
Con 5000 de elementos PLE tardó 47 minutos en resolverse. Decidimos no seguir probando con sets más grandes.

5.3. Complejidad de algoritmo Greedy

Siendo S el número de subsets (es decir el número de periodistas) y P el número de jugadores (el tamaño del conjunto A) estudiemos la complejidad de la aproximación Greedy:

Primero, el algoritmo cuenta las apariciones de cada jugador. Esta operación es $O(N * P)$ ya que recorre los N subsets, que cada uno podría tener como máximo a todos los jugadores P. Luego, habiendo contado las apariciones de cada jugador, las ordena de mayor a menor. Esto es $O(P * log(P))$. Por último, para cada periodista (es decir cada subset) elige al jugador con más apariciones. Esto puede tomar hasta $O(P)$ para cada subset, si en el subset está el último jugador. Así, esta última parte es $O(N * P)$. Al ser las operaciones más complejas $O(N * P)$, el algoritmo también lo es.[5]

6. Tiempos de ejecución de las dos aproximaciones



Como podemos ver comparando ambos algoritmos, cuanto más crece el volumen de los sets mayor es la ventja de greedy comparado con programación lineal. La complejidad del algoritmo de aproximación por programación lineal es peor que la del algoritmo greedy.

[5] Podría suceder que el ordenamiento sea más costoso si $N < log(P)$). Este caso excepcional tendría complejidad $O(P * log(P))$

7. Conclusión

En este trabajo práctico comprendimos cómo resolver (o aproximar) problemas mucho más complejos que los anteriores. Estudiar la complejidad de los problemas NP y NP-Completos nos permitió comprender cuando decidir resolverlos de forma óptima y cuando solamente buscar una aproximación. Frente a sets pequeños vale la pena resolverlo de forma óptima y costosa (porque en volúmenes pequeños el costo es pequeño), mientras que en sets grandes (que tarde muchísimo tiempo la forma óptima) hay que decidir si aproximar o esperar el largo tiempo de resolución optima.

En cuanto a las soluciones óptimas, fue notable como programación lineal (entera) resolvió el problema en tiempos mucho mejores que la implementación de backtracking una vez que creció el número de subsets. Ver esta diferencia nos permite entender por qué vale la pena diseñar un sistema de ecuaciones para resolver el problema con Pulp, en vez de simplemente implementar un algoritmo de backtraking. Esto último se puede evidenciar claramente en el gráfico comparativo entre tiempos de ejecución, dónde es ampliamente superior la programación lineal (entera) en los casos grandes (aquellos donde más importa mejorar el tiempo de ejecución).

En cuanto a las aproximaciones, demostrar matemáticamente la cota nos permite decidir cuando aproximar y cuando invertir el tiempo de cómputo en resolver el problema. Entre las dos opciones desarrolladas, greedy demostró hacer mejores aproximaciones en un mejor tiempo. Además, la aproximación por programación lineal no ser muy util en situaciones reales de este problema. Que un algoritmo elija al 98 % de los jugadores posibles (48 de los 49 jugadores posibles en el set de 1000 elementos, por ejemplo) no nos parece util en este caso.

Nos parece interesante recordar que la optimización de nuestros algoritmos siempre mejora los tiempos de resolución, sin necesidad de pasar a otro paradigma. Para backtracking, por ejemplo, cambiar el uso de listas por sets mejoró algunos tiempos de ejecución de 160 a 57 segundos. Esta puede ser que sea una de las razones de la ventaja de la librería de Programación Lineal. Una librería correctamente desarrollada probablemente está mejor optimizada que nuestro algoritmo de backtraking para nuestro trabajo práctico.

Sin más conclusiones que mencionar, le deseamos lo mejor a la Selección frente a Burkina Faso, esperamos que le sea útil nuestro aporte.