Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

26 de noviembre de 2023

Martín González Prieto
105738

Santiago Langer
107912

Camila Teszkiewicz
109660

Índice

1. Introducción: análisis del problema **3**
1.1. Hitting set problem: problema NP 3
1.2. Hitting set problem: problema NP-Completo 3

2. Solución óptima por Backtracking **4**
2.1. El comienzo: fuerza bruta 4
2.2. Razonando un poco: la solución 5
2.3. Análisis del algoritmo 5
2.4. Análisis: mediciones de tiempos 7

3. Solución óptima por Programación Lineal Entera **7**
3.1. Código 8
3.2. Análisis: mediciones de tiempos 8
3.3. Análisis: comparación con solución por Backtracking 8

4. Solución óptima por aproximación de Programación Lineal Mixta **8**

5. Conclusión **9**

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 contexto del problema se basa en confeccionar una lista final de jugadores para que Scaloni arme un equipo para un amistoso. Se cuenta con varias listas de jugadores, las cuales corresponden a las pretenciones 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 $|C| \leq k$ con k=11 ya que son la cantidad de jugadores titulares.

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 $O(mxk)$ 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 $O(1s)$.

1.2. Hitting set problem: problema NP-Completo

Por otro lado, para corrobarar 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 $\leqslant X$.
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:

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

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. Fácil, agarramos dichas cláusulas, las

tranformamos como subconjuntos B_i y buscamos cuál es el mínimo conjunto de elementos que pertenecen a dichos conjuntos.

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

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

$$C = \begin{pmatrix} x_1 \\ x_5 \\ x_7 \end{pmatrix}$$

Dicha solución C, es mínima y cumple con $x_1 \in B_1, B_2; x_5 \in B_2, B_3, B_4; x_7 \in B_5$
Por lo tanto, si consideramos como ese subconjunto de solución, son los valores que deberían ser verdaderos para nuestro caso de SAT, se cumple la condición inicial.

$$C_1 = 1 \vee x_3 \vee x_4 = 1;$$
$$C_2 = x_1 \vee x_2 \vee 1 = 1;$$
$$C_3 = x_3 \vee 1 \vee x_6 = 1;$$
$$C_4 = 1;$$
$$C_5 = 1;$$

Por lo tanto, queda demostrado así que $SAT \leq HittingSet$, y por lo tanto Hitting Set problem es NP-Completo.

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, entre 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 busqueda 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 - nuestra solución es mas robusta, 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

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

soluciones posibles.
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 se basa, básicamente, en ir probando soluciones pero con criterio. Razonemos entonces el problema empezando por las obviedades: si un subconjunto contiene un solo elemento, ese elemento deberá incluirse en la solución. Ya tenemos una cota inferior para el mínimo n (la cantidad de subconjuntos con un solo elemento). Ya encontramos una **primera poda**, descartamos cualquier solución que no incluya a algún elemento de un subconjunto unitario.
Una vez que incluimos un jugador en nuestro conjuntos solución, podemos recorrer en tiempo lineal (respecto de la cantidad de subconjuntos) y descartar las necesidades de todos los subconjuntos que contienen a dicho jugador. Es muy probable que en el recorte nos voltiemos a más de un jugador, lo que significa: menos soluciones posibles; una **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.[es] 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. Análisis del algoritmo

Primero resolvimos el problema de decisión planteado (dado un k, existe un listado de k elementos tal que todos los subconjuntos B contengan al menos un elemento del listado) y luego realizamos una busqueda del mínimo k con un algoritmo del estilo de divide y conquista.
Un código simplificado sería el siguiente:

```
ordenar(periodistas) #los ordenamos seg n mayor cantidad de elementos

def BT_recursivo(periodistas:dict, convocados:set, n_minimo = 100):
    if len(periodistas.keys()) == 0:
        return True
    if len(convocados) == n_minimo:
        return False
    eliminados = {}
    siguiente_periodista = next(iter(periodistas.items()))
    aux = periodistas.copy()

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

        #si es solucion, corto la ejecuci n y devuelvo True
        if BT_recursivo(aux, convocados, n_minimo):
            return True

        #si no es solucion, vuelvo para atras
        convocados.remove(jugador)
        devolver_periodistas(aux,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 çonvocadosza que remover un elemento de un set, en python 3.10, es

una operación O(1), no así removerlo de una lista.
Hemos planteado dos maneras de encarar esta cuestión: Podríamos buscar directamente el n mínimo, o resolver el problema de decisión y despues ir reduciendo el valor mediante una busqueda binaria, o algun algoritmo del estilo DyC (como anticipamos antes).
Respecto de la complejidad del algoritmo, no hay demasiado para comentar.

Llamamos N a la cantidad de subconjuntos B

. Primero ordenamos:

$$O(nlogn)$$

- ya hemos justificado en trabajos anteriores que la funcion sorted() incluída en python 3.10 garantiza dicha complejidad - luego realizamos una accion 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

y finalmente, resta hacer una busqueda binaria hasta encontrar el mínimo k. Dicha busqueda arrancaría en k = N (para el cual seguro hay solución) y decrecería.

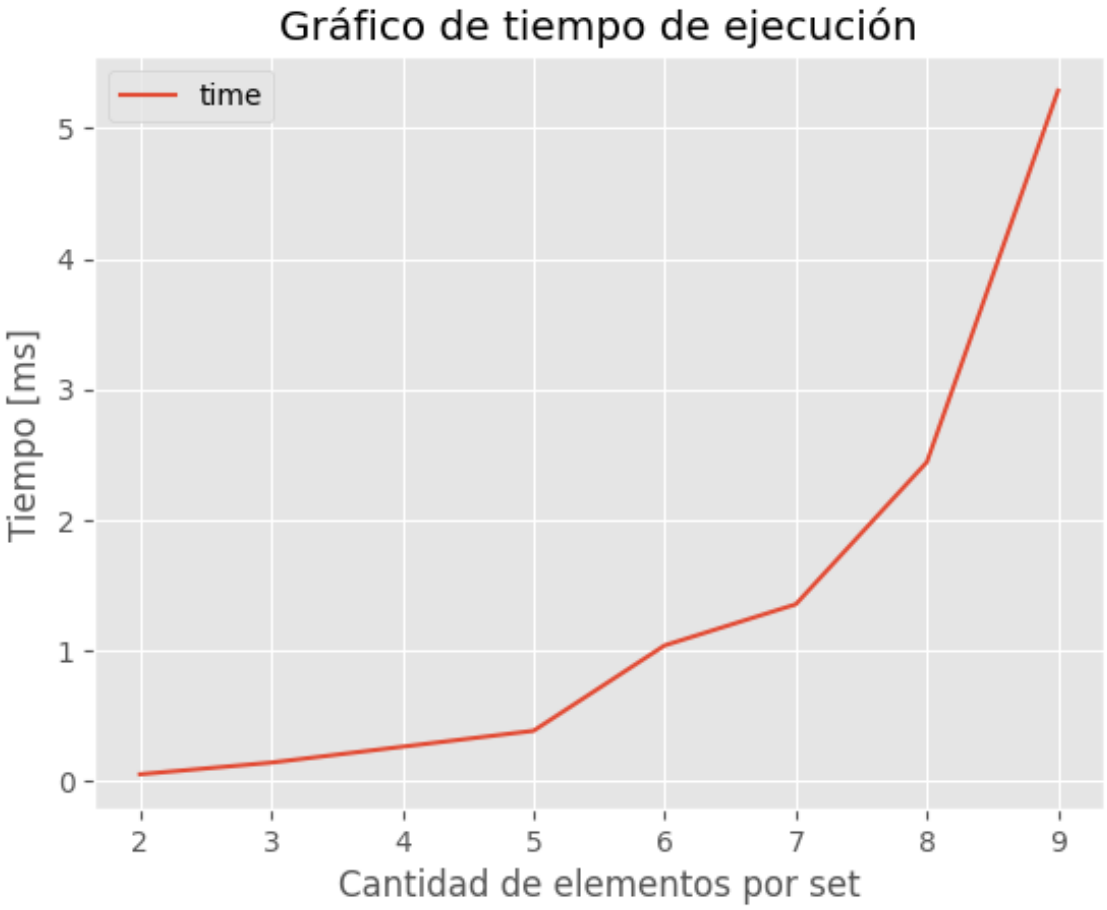
$$O(logN)$$

Finalmente, nos queda la siguiente cuenta:

$$O(N! * logN + NlogN) = O(N! * logN)$$

No desarrollamos demasiado este tema, ya que en realidad no nos dice mucho en este caso. Nos da una cota superior, eso seguro, pero con el trabajo de poda el tiempo de ejecución se reduce considerablemente.

2.4. Análisis: mediciones de tiempos



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 se indicarán como 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:

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

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. Análisis: mediciones de tiempos

3.3. Análisis: comparación con solución por Backtracking

4. Solución óptima por aproximación de Programación Lineal Mixta

Tal como lo indica el enunciado, el modelo para Programación Lineal Mixta será el mismo que para PLE, relajando la restricción sobre x_i:

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

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,

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

Lllamamos $\dot{T}$ 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.

5. Conclusión