

TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 3: Backtracking

25 de noviembre de 2023

Castro Martínez José Ignacio
106957

Makkos Juan Sebastian
106229

*"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, \dots, 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 el set de datos de entrada al mismo, un conjunto de subconjuntos $B = B_1, B_2, \dots, B_m$ ($B_i \subseteq A \forall i$) 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

```
1 def is_solution(hitting_set, subsets):
2     for subset in subsets:
3         flag = False
4         for element in subset:
5             if element in hitting_set:
6                 flag = True
7                 break
8         if not flag: return False
9     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 inválidas

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

```
1 def has_a_player(subset, act_sol):
2     for player in subset:
3         if player in act_sol:
4             return True
5     return False
6
7 def search_for_min_hitting_set(subsets, a):
8     return _search_for_min_hitting_set(subsets, set(), set(), 0, set())
9
10 def _search_for_min_hitting_set(subsets: list, best_sol: set, act_sol: set, act_sub
    : int, used_players: set):
11
12     if len(best_sol) > 0 and len(act_sol) > len(best_sol):
13         return best_sol
14
15     if act_sub == len(subsets) - 1 and (len(best_sol) == 0 or len(act_sol) < len(
        best_sol)) and is_solution(act_sol, subsets):
16         best_sol = act_sol.copy()
17         return best_sol
18
```

```
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,
24                                             used_players)
25
26     selected_players = set()
27
28     for player in subsets[act_sub]:
29         if player in used_players:
30             continue
31         act_sol.add(player)
32         selected_players.add(player)
33         used_players.add(player)
34         best_sol = _search_for_min_hitting_set(subsets, best_sol, act_sol, act_sub
35         + 1, used_players)
36         act_sol.remove(player)
37
38     used_players.difference_update(selected_players)
39     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 fácil, 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:

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

Con lo cual se contruye la función objetivo del problema, evidentemente esta ecuación por si sola no es suficiente puesto que no considera el conjunto de subconjuntos $B = B_1, B_2, \dots, B_m$ ($B_i \subseteq A \forall i$), 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:

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

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:

```
1 def search_hs_linealp(subsets, set):
2     dict_variables = {elem: pulp.LpVariable(f"{elem}", cat="Binary") for elem in
3         set}
4     problem = pulp.LpProblem("hitting_set_problem", pulp.LpMinimize)
5     problem += pulp.lpSum(dict_variables[elem] for elem in set)
6     for subset in subsets:
7         problem += pulp.lpSum(dict_variables[elem] for elem in subset) >= 1
8
9     pulp.LpSolverDefault.msg = 0
10    problem.solve()
11
12    hitting_set_solution = {var.name for var in dict_variables.values() if pulp.
13        value(var) == 1}
14    return hitting_set_solution
```

Listing 3: solución por programación lineal

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:

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

4. Gráficas

5. Construyendo aproximaciones