



TEORÍA DE ALGORITMOS
(75.29) CURSO BUCHWALD - GENENDER

Trabajo Práctico 3

Problemas NP-Completo

[illegible]

28 de febrero de 2024

Ariel Aguirre
100199
Víctor Bravo
98.882

Lucia Magdalena Berard
101213
Maximiliano Prystupiuik
94.853

1. Algoritmo

1.1. 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 Rondaglia, 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 (siendo $B_i \subseteq A$), 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 sí o sí.

Bueno, ahora con un poco más de 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.

Consigna

Para los primeros dos puntos, considerar la versión de decisión del Hitting-Set Problem:

Dado un conjunto de elemento A de n elementos, m subconjuntos B_1, B_2, \dots, B_m de A (siendo $B_i \subseteq A$), y un número k , ¿existe un subconjunto $C \subseteq A$ con $|C| \leq k$ tal que C tenga al menos un elemento de cada B_i (es decir, $C \cap B_i \neq \emptyset$)?

1.2. Demostrar que el Hitting-Set Problem se encuentra en NP

Para demostrar que HSP es NP tenemos que demostrar dado un certificado podemos verificar que es solución en tiempo polinomial.

```
1 funcion verificador
2   Sea C el conjunto solucion
3   Sean Bi los subconjuntos de A
4
5   Devolver Falso si |C| > k
6
7   Para cada subconjunto Bi
8     Si la interseccion de C y Bi es vacia
9       Devolver Falso
10  Devolver Verdadero
```

Asumimos $O(1)$ para el tamaño del subconjunto. Luego, en los ciclos for anidados, tenemos que en el primero recorre los m subconjuntos y el segundo calcula la intersección de dos sets de tamaños m y n que es $O(nm)$, lo cual juntos es $O(nm^2)$. Finalmente, la verificación es $O(nm^2)$ lo cual es polinomial, y por lo tanto HSP es NP.

1.3. Demostrar que el Hitting-Set Problem es, en efecto, un problema NP-Completo

Para probar que un problema es NP-Completo es necesario que

- Sea NP
- Se pueda reducir un problema NP-Completo a nuestro problema

La primera parte ya fue demostrada en el punto anterior y para la segunda parte proponemos reducir Vertex Cover, que es NP-Completo a HSP.

Reducción de Vertex Cover a Hitting-Set Problem

Sea $G = (V, E)$ un grafo en un problema de Vertex Cover y k un entero positivo. Convertiremos el problema a HSP de la siguiente manera.

- Sea A el conjunto V
- Para cada arista $e = (u, v)$ de E , conjunto de aristas, formamos un subconjunto $B_e = \{u, v\}$

Podemos notar que si existe un conjunto C con $|C| < k$ solución de HSP, entonces existe solución para Vertex-Cover con el conjunto de nodos igual a C y mismo k .

En conclusión, como HSP es NP y se puede reducir un problema NP-Completo a HSP, entonces HSP es NP-Completo.

1.4. Algoritmo backtracking

El siguiente algoritmo de backtracking consiste en empezar con un set solución vacío y en cada paso probar con un elemento que no esté en él hasta llegar a un conjunto que cumpla con ser solución o pasarse del límite k pedido; en dicho caso, devuelve un set vacío.

La función $backtracking(A, B, C, k)$ en si no encuentra el set con menor cantidad de elementos, pero si uno que cumple con ser $|C| < k$, por ello la función $hsp(A, B, k)$ itera desde $0..k$ y devuelve el primer conjunto solución encontrado y por lo tanto el menor.

```
1 import sys, csv, json, time
2
3 def is_solution(B, C):
4     for b in B:
5         if len(C.intersection(b)) == 0:
6             return False
7     return True
8
9 def backtracking(A, B, C, k):
10     if len(C) > k:
11         return set()
12
13     if is_solution(B, C):
14         return C
15
16     resto = A - C
17     for elem in resto:
18         copia = C.copy()
19         copia.add(elem)
20         resultado = backtracking(A, B, copia, k)
21         if len(resultado) > 0:
22             return resultado
23     return set()
```

```
24
25 def hsp(A, B, k):
26     for i in range(k+1):
27         r = backtracking(A, B, set(), i)
28         if r:
29             return r
30     return None
```

Análisis

El código proporcionado está intentando resolver el problema del conjunto golpeador (Hitting Set Problem), que es conocido por ser NP-completo. Esto significa que se espera que la cantidad de tiempo requerida para encontrar una solución aumente exponencialmente con el tamaño del problema.

En el JSON armado, cada entrada incrementa el tamaño del conjunto A y el conjunto de subconjuntos B, y para cada caso, se busca un conjunto golpeador de tamaño menor o igual a k. Dado que el backtracking prueba todas las posibles combinaciones de elementos para encontrar una solución válida, el tiempo requerido para encontrar una solución (o determinar que no existe) puede aumentar muy rápidamente a medida que el tamaño del problema aumenta.

Tiempo en función N

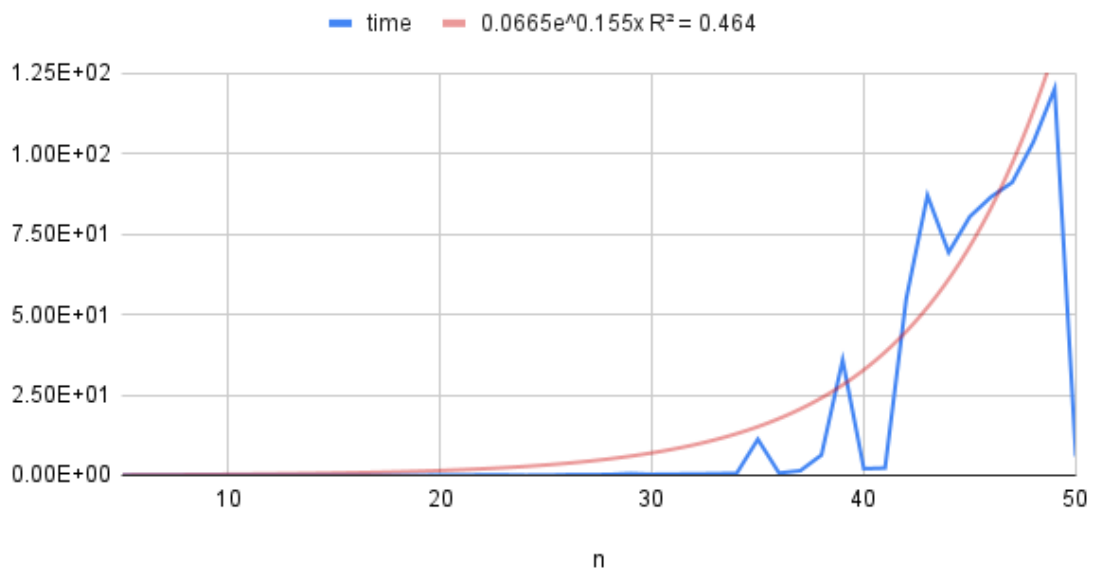


Figura 1: Tiempo ejecución backtracking

1.5. Algoritmo Greedy

El siguiente algoritmo greedy consiste en para cada iteración, seleccionar al elemento que más veces se repita en los sets de B y agregarlo al conjunto solución. Luego, se descartan los sets donde el elemento aparece y el mismo elemento, para empezar nuevamente con este conjunto más acotado. Eventualmente, se llega a un conjunto solución aunque no necesariamente sea el mínimo.

```
1 import sys, csv, json, time
2
3 def next_candidate(non_hit, candidates):
4     """
```

```

5  Entrada
6  non_hit: Set de sets disponibles que no tienen elementos en el hitting-set.
7  candidates: Elementos candidatos a estar en el hitting-set
8
9  Salida: Elemento elegido que aparece la mayor cantidad de veces en non_hit
10 """
11 max = 0
12 candidate = None
13 d = {}
14 for c in candidates: # O(n)
15     for b in non_hit: # O(m)
16         if c in b: # O(n)
17             d[c] = d.get(c, 0) + 1 # O(1)
18             if d[c] > max:
19                 max = d[c]
20                 candidate = c
21 candidates.remove(candidate) # O(n)
22 return candidate # ==> O(mn^2)
23
24 def hit_sets(non_hit, candidate):
25     """
26     Entrada
27     non_hit: Set de sets disponibles que no tienen elementos en el hitting-set.
28     candidato: Elemento elegido para entrar en el hitting-set
29
30     Salida: Se actualiza non_hit quitando los sets donde aparezca el candidato
31     """
32     for s in non_hit.copy(): # O(m)
33         if candidate in s: # O(n)
34             non_hit.remove(s) # O(n)
35     return non_hit # ==> O(mn^2)
36
37
38 def is_solution(B, C):
39     """
40     Indica si C es hitting-set de B
41     """
42     for b in B: # O(m)
43         if len(C.intersection(b)) == 0: # O(n)
44             return False
45     return True # ==> O(mn)
46
47 def hsp(A, B, k):
48     C = set()
49     non_hit = B.copy()
50     candidates = A.copy()
51
52     while len(C) < k and not is_solution(B, C): # O(n)
53         candidate = next_candidate(non_hit, candidates) # O(mn^2)
54         non_hit = hit_sets(non_hit, candidate) # O(mn^2)
55         C.add(candidate)
56     return C # ==> O(mn^3)

```

Análisis de Complejidad

Empezamos por seleccionar el elemento candidato. La cantidad de candidatos posibles el $|A| = n$, así que el ciclo más externo es $O(n)$. Al interior, se itera sobre los conjuntos B_i disponibles los cuales son m . Un bloque más adentro tenemos si el candidato pertenece al conjunto b lo cual es $O(n)$ dado que en el peor de los casos el conjunto tiene todos los elementos. Un paso más adentro tenemos la comparación del máximo que es $O(1)$.

En conclusión, seleccionar el candidato es $O(mn^2)$

```

1 def next_candidate(non_hit, candidates):
2     max = 0
3     candidate = None
4     d = {}

```

```
5     for c in candidates: # O(n)
6         for b in non_hit: # O(m)
7             if c in b: # O(n)
8                 d[c] = d.get(c, 0) + 1 # O(1)
9                 if d[c] > max:
10                     max = d[c]
11                     candidate = c
12     candidates.remove(candidate) # O(n)
13     return candidate # ==> O(mn^2)
```

Luego, tenemos el descarte de los conjuntos donde el candidato hace hit.

En el ciclo externo se itera sobre cada subconjunto b lo cual es $O(m)$. Luego, ver si el candidato pertenece es $O(n)$ visto anteriormente. Quitar el subconjunto es $O(n)$. Finalmente, la función corre en $O(mn^2)$

```
1 def hit_sets(non_hit, candidate):
2     for s in non_hit.copy(): # O(m)
3         if candidate in s: # O(n)
4             non_hit.remove(s) # O(n)
5     return non_hit # ==> O(mn^2)
```

Por otro lado, ver si el conjunto C es solución para B comprende iterar para cada b unas m veces, y ver si la intersección no es vacía es $O(n)$. Por lo tanto, es $O(mn)$.

```
1 def is_solution(B, C):
2     for b in B: # O(m)
3         if len(C.intersection(b)) == 0: # O(n)
4             return False
5     return True # ==> O(mn)
```

Por último, la función *hsp* realiza un ciclo que en el peor de los casos agrega todos los elementos de A a la solución, lo cual es $O(n)$. Visto anteriormente, elegir un candidato es $O(mn^2)$ y lo mismo para descartar los sets. Agregar un elemento a un set es $O(1)$.

Se concluye entonces que el algoritmo es $O(mn^3)$.

```
1 def hsp(A, B, k):
2     C = set()
3     non_hit = B.copy()
4     candidates = A.copy()
5
6     while len(C) < k and not is_solution(B, C): # O(n)
7         candidate = next_candidate(non_hit, candidates) # O(mn^2)
8         non_hit = hit_sets(non_hit, candidate) # O(mn^2)
9         C.add(candidate)
10    return C # ==> O(mn^3)
```

Mediciones

Utilizando el mismo set usado anteriormente se obtiene un ajuste similar a n^4 , lo cual tiene sentido si pensamos en que el tamaño del set B fue idéntico a n , es decir $m = n$, y por lo tanto teóricamente debería dar $O(n^4)$.

Greedy: Tiempo en funcion de N

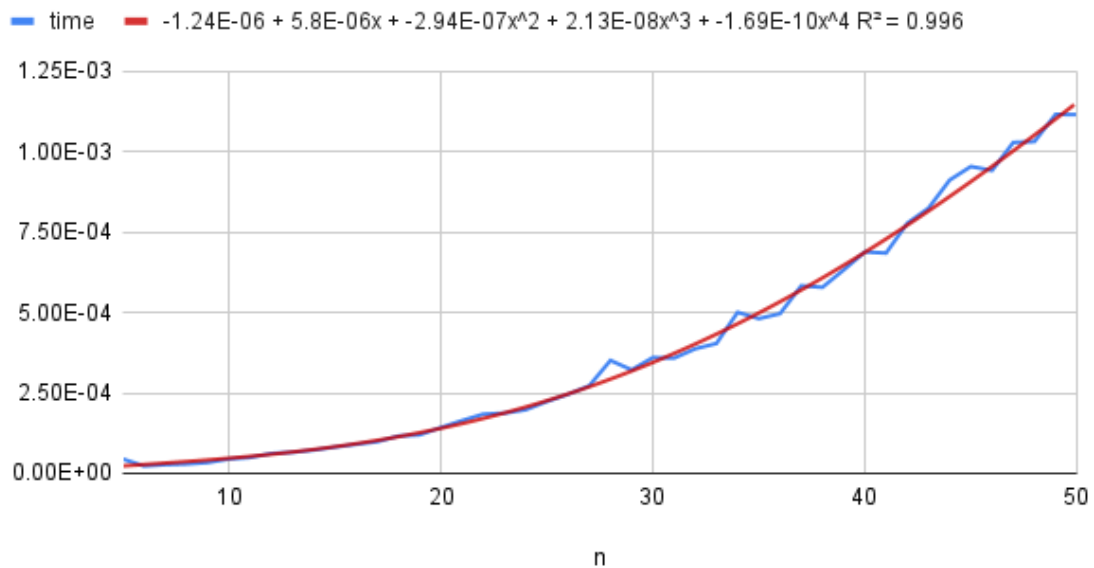


Figura 2: Tiempo ejecución Greedy

Cota empírica de aproximación

Comparando el largo del set solución entre los algoritmos de backtracking y greedy se obtuvo que el algoritmo greedy como mucho agrega un 50% más de elementos que backtracking. (ver Anexo)

1.6. Conclusiones

Los dos algoritmos resuelven el mismo problema (Hitting Set Problem), pero utilizan estrategias diferentes para encontrar una solución.

Diferencias clave

1. Estrategia

- **Greedy:** El algoritmo greedy selecciona en cada paso el elemento que golpea la mayor cantidad de subconjuntos no golpeados aún. Este enfoque intenta reducir rápidamente el número de subconjuntos no golpeados pero no garantiza encontrar la solución óptima.
- **Backtracking:** El algoritmo de backtracking explora todas las posibles combinaciones de elementos para encontrar una solución óptima. Retrocede (backtracks) si una combinación parcial no puede llevar a una solución válida, asegurando así que se explore el espacio de soluciones de manera exhaustiva.

2. Complejidad

- **Greedy:** La complejidad principal viene de buscar el próximo candidato (*next_candidate*) y actualizar los subconjuntos no golpeados (*hit_sets*). Ambas operaciones tienen una complejidad de $O(mn^2)$ donde m es el número de subconjuntos y n el tamaño del conjunto universal. La complejidad total es $O(mn^3)$ debido al bucle mientras se busca una solución.

- **Backtracking:** La complejidad es más difícil de determinar porque depende del tamaño del conjunto universal A , el número de subconjuntos B , y la profundidad de la recursión. En el peor caso, podría explorar todas las posibles combinaciones de elementos en A lo que lleva a una complejidad exponencial.

3. Aplicabilidad

- **Greedy:** Es más eficiente para conjuntos grandes donde encontrar una solución óptima no es práctico debido al tiempo de ejecución. Proporciona soluciones rápidas que son buenas en muchos casos, aunque no siempre óptimas.
- **Backtracking:** Es preferible cuando el conjunto de elementos es relativamente pequeño, o se necesita encontrar la solución óptima sin importar el tiempo de ejecución.

Cuando usar cada uno

- **Greedy:**
 - Se necesita una solución rápida y razonablemente buena, pero no necesariamente la óptima.
 - El conjunto de datos es grande, haciendo inviable la búsqueda exhaustiva debido al tiempo de ejecución.
- **Backtracking:**
 - Se requiere encontrar la solución óptima, sin importar el tiempo de ejecución.
 - El conjunto de datos es de tamaño manejable, permitiendo una búsqueda exhaustiva en un tiempo razonable.

En resumen, la elección entre greedy y backtracking depende principalmente de la necesidad de optimización versus el tiempo de ejecución y el tamaño de los datos con los que se trabaja. El enfoque greedy es adecuado para obtener soluciones rápidas en grandes conjuntos de datos, mientras que el backtracking es la mejor opción cuando se busca la solución óptima y el tamaño del problema lo permite.

1.7. Anexo

Largo del set solución Backtracking vs Greedy

Donde

- N : Cantidad de elementos
- diff : Diferencia entre largos
- error : Error relativo porcentual

N	len-back	len-greedy	diff	error
5	2	2	0	0
6	2	2	0	0
7	2	2	0	0
8	2	2	0	0
9	2	2	0	0
10	3	3	0	0
11	3	3	0	0
12	3	3	0	0
13	3	3	0	0
14	2	3	1	50
15	3	3	0	0
16	3	3	0	0
17	3	3	0	0
18	3	3	0	0
19	3	3	0	0
20	3	3	0	0
21	4	4	0	0
22	4	4	0	0
23	4	4	0	0
24	3	3	0	0
25	3	4	1	33.33
26	4	4	0	0
27	4	4	0	0
28	4	5	1	25
29	4	4	0	0
30	4	4	0	0
31	4	4	0	0
32	4	4	0	0
33	4	4	0	0
34	4	5	1	25
35	4	5	1	25
36	4	4	0	0
37	4	5	1	25
38	4	4	0	0
39	5	5	0	0
40	4	5	1	25
41	4	4	0	0
42	5	5	0	0
43	5	5	0	0
44	5	5	0	0
45	5	5	0	0
46	5	5	0	0
47	5	5	0	0
48	5	5	0	0
49	5	5	0	0
50	4	5	1	25