Teoría de Algoritmos

(75.29) Curso Buchwald - Genender

Trabajo Práctico 3

25 de noviembre de 2023

Joaquin Lonardi, 105970

Trabajo Práctico 3 - Hitting Set Problem

1. Introducción

El siguiente informe pretende abordar el problema que tiene Scaloni para contentar al periodismo en su afán interminable de operar a favor de ciertos jugadores, valgan estos la pena o no. Scaloni debe encontrar la cantidad mínima de jugadores que satisfaga a todos los periodistas, siendo imperativo para satisfacerlos convocar en la próxima lista al menos un jugador que haya pedido cada periodista.

Para esto, se va a extrapolar su problema al Hitting-Set Problem, por recomendación del Dr. Carlos Salvador Bilardo. De esta manera, vamos a poder realizar tres análisis que le van a permitir a Scaloni resolver su problema: vamos a analizar la complejidad del problema (así, Scaloni va a saber qué tanto debe preocuparse a medida que aumente la cantidad de periodistas que operan y de jugadores operados), vamos a implementar un algoritmo que utilize *backtracking* para resolver su problema de manera óptima, y vamos a hacer lo mismo utilizando programación lineal, tanto generando un modelo de programación lineal como utilizando un algoritmo aproximado por recomendación del Dr. Bilardo.

2. Complejidad del Hitting-Set Problem

Para analizar la complejidad del Hitting-Set problem, vamos a intentar probar que este problema pertenece a los problemas NP-Completos. Primero, repasemos los pasos a seguir para probar que un problema X es NP-Completo:

1. Probar que $X \in \mathcal{NP}$.
2. Escoger un problema Y conocido que sea NP-Completo.
3. Probar que se puede reducir a X. Es decir $Y \leq_p X$.

Comencemos, entonces, por el primer paso.

2.1. Probando que Hitting-Set $\in \mathcal{NP}$

La demostración de que un problema pertenece a NP no debería ser muy complicada. Por definición, un problema es NP si existe una manera de validar una posible solución en tiempo polinomial. Esto se comprueba simplemente creando un certificador y asegurándose de que corra en tiempo polinomial. Se propone el siguiente certificador:

```
fun hitting_set_certifier(solucion, sets) {
    para cada set en sets {
        si interseccion entre set y solucion es nula:
            return False
    }
    return True
}
```

Analizando rápidamente la complejidad de este algoritmo, vemos que por cada set en el conjunto de sets ($\mathcal{O}(|sets|)$) se debe calcular la interesección entre el set y el conjunto solución (esta operación es $\mathcal{O}(|solucion|)$, asumiendo para simplificar que siempre la solución va a ser mas pequeña que el set). Por lo tanto, la complejidad de este certificador se mantendría en un tiempo polinomial, siendo este particularmente $\mathcal{O}(|sets| * |solucion|)$.

Siendo esta condición necesaria y suficiente para que un problema pertenezca al conjunto de problemas NP, podemos concluir que el Hitting Set Problem se encuentra en NP.

2.2. Probando que Hitting-Set $\in \mathcal{NP}$-Completo

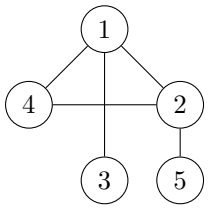
Habiendo probado que el problema que estamos analizando es efectivamente NP, ahora queda probar que el problema también pertenece a los NP-Completos. El siguiente paso para esto es elegir un problema que sepamos pertenezca al conjunto de los NP-Completos que podamos utilizar para reducir a nuestro problema actual. En este caso, vamos a elegir el problema del Vertex Cover. Este problema plantea lo siguiente:

> Dado un grafo G y un número k, ¿existe una cobertura de vértices –es decir, un conjunto de vértices tales que cada arista del grafo es incidente a al menos un vértice del conjunto– en G de como mucho k elementos?

Podemos ver cómo el problema del Vertex Cover puede ser reducido al Hitting-Set Problem, probando así que este último es NP-Completo:

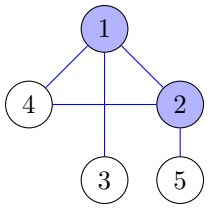
Imaginemos que tenemos una caja negra que resuelve el Hitting-Set Problem. Para resolver el Vertex Cover Problem, podemos transformar el grafo $G = (V, E)$ en un conjunto de sets S de la siguiente manera: los vértices pasan a ser el universo de elementos U que pueden ser incluídos en los subsets y los subsets S del universo U los podemos armar viendo los adyacentes de cada vértice y chequeando si un par de adyacentes son también adyacentes entre sí. Esta transformación es polinomial, ya que recorrer un grafo y sus adyacentes lo es, así que es una transformación válida para nuestro objetivo. Finalmente, teniendo la transformación de grafo a conjunto de sets, podemos aplicar la caja negra del Hitting-Set Problem y nos devolverá el resultado adecuado.

Veamos esto con un ejemplo. Supongamos que tenemos el siguiente grafo:



Para adaptarlo a un Hitting Set Problem, primero definiríamos al universo $U = V$, es decir $U = \{1, 2, 3, 4, 5\}$. Luego, recorremos los vértices del grafo. Por cada vértice recorremos sus adyacentes y, si un adyacente a del vértice v es a su vez adyacente a otro adyacente a' de v, estos vértices se agregan a un set. En este caso particular, el conjunto S de subsets quedaría de la siguiente manera: $S = \{\{1, 2, 4\}, \{1, 3\}, \{2, 5\}\}$ (notar que, por ejemplo, 1 es adyacente de 2 y de 4, y 4 y 2 son adyacentes entre sí).

Sin haber implementado una solución aún, podemos ver a simple vista que un hitting set para el conjunto S es $HS = \{1, 2\}$. También a simple vista podemos validar que, aplicado al grafo G, el conjunto HS también es un Vertex Cover del grafo:



Por lo tanto, habiendo demostrado que Vertex Cover puede reducirse al problema del Hitting Set, podemos afirmar que Hitting Set Problem $\in$ NP-Completo.

3. Soluciones del Hitting-Set Problem

Habiendo probado que el Hitting-Set Problem es NP-Completo, ahora podemos comenzar a hallar soluciones al problema. Vamos a encontrar métodos para resolver este problema utilizando dos técnicas distintas: *backtracking* y programación lineal.

3.1. Solución por *Backtracking*

El algoritmo implementado busca de manera recursiva una solución óptima, explorando diversas combinaciones al agregar y quitar elementos del conjunto universo U. Una solución potencial solo se valida si se determina que es mejor que la solución actual (en este caso, solo se valida si es de tamaño menor a la solución actual). El código del algoritmo en Python es el siguiente:

```python
def solucion_es_valida(S, solucion):
    sets_alcanzados = 0
    for conjunto in S:
        if conjunto.intersection(solucion):
            sets_alcanzados += 1

    return sets_alcanzados == len(S)
def _backtrack(S, U, solucion_actual, solucion, idx):
    if len(solucion_actual) >= len(solucion):
        return

    if idx == len(U):
        if solucion_es_valida(S, solucion_actual):
            if len(solucion_actual) < len(solucion):
                solucion.clear()
                solucion.extend(solucion_actual)
        return

    solucion_actual.append(U[idx])
    _backtrack(S, U, solucion_actual, solucion, idx + 1)
    solucion_actual.pop()
    _backtrack(S, U, solucion_actual, solucion, idx + 1)

def hitting_set(S, U):
    solucion = list(U)
    solucion_actual = []
    _backtrack(S, list(U), solucion_actual, solucion, 0)
    return len(solucion)
```

3.1.1. Mediciones

Para probar este algoritmo, se generaron 10 archivos de prueba aumentando paulatinamente la cantidad de sets, generados con una cantidad de elementos aleatoria y manteniendo un universo de elementos fijo de 20 elementos. Luego, se corrió el algoritmo para cada archivo de prueba y se calculó el tiempo de ejecución. Los tiempos de esta primera corrida fueron los siguientes:

Cantidad de Subconjuntos	Cantidad mínima	Tiempo de ejecución (s)
5	2	$3{,}36 \times 10^{-6}$
7	3	$2{,}01 \times 10^{-3}$
10	3	$9{,}51 \times 10^{-3}$
15	6	$3{,}92 \times 10^{-1}$
25	5	$1{,}55 \times 10^{-1}$
40	7	2.96
50	7	3.70
75	8	21.1
90	9	100
100	10	194

Como se ve a simple vista en la tabla, vemos que a medida que aumenta la cantidad de subconjuntos que se deben analizar, aumenta el tiempo que le toma al algoritmo hallar una solución.

También se generó otro set diferente de datos, achicando el conjunto universal, pasando de 20 elementos a solo 7. Manteniendo la misma cantidad de subconjuntos que en el análisis anterior, estos fueron los tiempos que se recibieron:

Cantidad de Subconjuntos	Cantidad mínima	Tiempo de ejecución (s)
5	1	0,00
7	1	0,00
10	2	0,00
15	2	0,00
25	3	0,00
40	3	0,00
50	3	$1{,}00 \times 10^{-3}$
75	4	$9{,}98 \times 10^{-4}$
90	3	$2{,}00 \times 10^{-3}$
100	4	$1{,}51 \times 10^{-3}$

Como se puede observar, los tiempos de ejecución se redujeron drásticamente, incluso siendo las primeras 6 medidas demasiado pequeñas como para arrojar un valor mayor a cero.

3.2. Solución por Programación Lineal

El algoritmo que se utilizó para implementar una solución mediante programación lineal se escribió en Python utilizando la biblioteca PuLP y es el siguiente:

```
def hitting_set(S, U):
    variables_elementos = pulp.LpVariable.dicts("elemento", U, cat=pulp.LpBinary)

    hitting_set = pulp.LpProblem("HittingSetProblem", pulp.LpMinimize)
    hitting_set += pulp.lpSum(variables_elementos)

    for conjunto in S:
        hitting_set += pulp.lpSum(
            variables_elementos[elemento] for elemento in conjunto
        ) >= 1

    hitting_set.solve()

    solucion = {elemento for elemento in hitting_set.variables() if
            elemento.value() == 1
        }

    return len(solucion)
```

El modelo de programación lineal tiene tres definiciones importantes. En principio, la linea:

```
variables_elementos = pulp.LpVariable.dicts("elemento", U, cat=pulp.LpBinary)
```

crea una variable binaria por cada elemento en el universo.

Luego, las lineas:

```
    hitting_set = pulp.LpProblem("Hitting Set Problem", pulp.LpMinimize)
    hitting_set += pulp.lpSum(variables_elementos)
```

definen la función a optimizar. En este caso, se va a buscar minimizar la suma de todas las variables definidas anteriormente.

Finalmente, las líneas

```
for conjunto in S:
        hitting_set += pulp.lpSum(
            variables_elementos[elemento] for elemento in conjunto
        ) >= 1
```

definen la condición que hace que este modelo solucione el Hitting Set Problem. Esta restricción hace que para todos los conjuntos, la sumatoria de las variables que pertenecen a ese conjunto sea mayor a 1 (es decir, cada conjunto debe tener al menos un elemento).

3.2.1. Mediciones

Para realizar las mediciones del algoritmo de programación lineal se utilizaron los mismos sets de datos que se utilizaron con la solución por *backtracking*. Los resultados que arrojó el algoritmo por programación lineal para el primer set de datos fueron los siguientes:

Cantidad de Subconjuntos	Cantidad mínima	Tiempo de ejecución (s)
5	2	$2{,}51 \times 10^{-2}$
7	3	$2{,}26 \times 10^{-2}$
10	3	$2{,}45 \times 10^{-2}$
15	6	$2{,}30 \times 10^{-2}$
25	5	$2{,}55 \times 10^{-2}$
40	7	$2{,}96 \times 10^{-2}$
50	7	$2{,}77 \times 10^{-2}$
75	8	$6{,}52 \times 10^{-2}$
90	9	$2{,}96 \times 10^{-2}$
100	10	$5{,}16 \times 10^{-2}$

Como podemos observar, en cantidades de subconjuntos pequeñas, los tiempos de ejecución son algo más grandes que los obtenidos con el algoritmo de *backtracking*. Pero, a medida que aumentamos la complejidad de los datos de entrada, observamos que no hay un aumento significativo en los tiempos de ejecución, dejándonos con tiempos de ejecución muchísimo mas pequeños que los conseguidos con el algoritmo de *backtracking*.

También corrimos el modelo de programación lineal para el segundo set de datos, que contaba con un conjunto universo reducido. Los tiempos de ejecución fueron los siguientes:

Cantidad de Subconjuntos	Cantidad mínima	Tiempo de ejecución (s)
5	1	$4{,}71 \times 10^{-2}$
7	1	$2{,}15 \times 10^{-2}$
10	2	$2{,}26 \times 10^{-2}$
15	2	$2{,}21 \times 10^{-2}$
25	3	$2{,}50 \times 10^{-2}$
40	3	$2{,}36 \times 10^{-2}$
50	3	$2{,}62 \times 10^{-2}$
75	4	$2{,}86 \times 10^{-2}$
90	3	$2{,}76 \times 10^{-2}$
100	4	$3{,}16 \times 10^{-2}$

Vemos que los tiempos, si bien son menores, no presentan una gran diferencia con el set anterior. Comparando con el algoritmo de *backtracking*, en este caso los tiempos sí son mayores por algunas órdenes de magnitud más. Sin embargo, esto se deba muy probablemente a inicializaciones y procesos propios de la biblioteca que se utilizó para modelar este problema.

3.3. Solución Aproximada por Programación Lineal

El tercer y último método que vamos a analizar para solucionar el Hitting Set Problem es el de utilizar programación lineal pero, a diferencia del algoritmo anterior, utilizando variables reales y redondeando nuestra solución. El código para realizar esto es idéntico al de la sección 3.2, pero cambiando el tipo de variables y agregando la lógica de redondeo:

```
def hitting_set_aprox(S, U):
    variables_elementos = pulp.LpVariable.dicts("elemento", U, lowBound=0, upBound
    =1)
    hitting_set = pulp.LpProblem("HittingSetProblem", pulp.LpMinimize)

    hitting_set += pulp.lpSum(variables_elementos)

    max_cardinal_conjunto = max([len(conjunto) for conjunto in S])

    for conjunto in S:
        hitting_set += pulp.lpSum(
            variables_elementos[elemento] for elemento in conjunto
        ) >= 1

    hitting_set.solve()

    solucion = 0
    for elemento in hitting_set.variables():
        # Redondeo
        if elemento.value() >= 1/max_cardinal_conjunto:
            solucion += 1

    return solucion
```

3.3.1. Complejidad

Hasta ahora, hallamos soluciones óptimas del Hitting Set Problem. Al ser este un problema NP-Completo, cualquier solución óptima al problema no va a poder correrse en tiempo polinomial –al menos por lo que se sabe hasta el momento. Sin embargo, el método que estamos analizando en esta sección no es óptimo, sino aproximado. La programación lineal entera –como la utilizada en la sección 3.2– también es un problema NP-Completo. No obstante, podemos resolver un modelo en tiempo polinomial cambiando las variables binarias por variables reales continuas. El método *simplex*, que se utiliza para resolver modelos de programación lineal, puede resolver modelos con variables reales en tiempo polinomial, solo tardando tiempos exponenciales en casos muy puntuales, rara vez encontrados (Kleinberg y Tardos 2005).

Por lo tanto, analizando el código tenemos las siguientes complejidades:

- **Creación de variables**: en principio se crean $n = |U|$ variables. Esto tiene una complejidad de $\mathcal{O}(n)$.
- **Hallar el máximo del conjunto S**: para hallar conjunto más grande, se deben recorrer todos los conjuntos. Entonces, la complejidad es $\mathcal{O}(m)$ donde $m = |S|$.
- **Crear las restricciones**: por cada conjunto, debemos crear una restricción. Esto también es $\mathcal{O}(m)$.
- **Solución del problema**: como ya vimos, la programación lineal continua es en general polinomial con respecto a las variables. Por lo tanto, la complejidad de esta sección es $\mathcal{O}(n)$[1].
- **Redondeo**: para redondear el resultado y obtener una solución, se deben recorrer todas las variables para decidir si se agrega o no a la solución. El recorrido toma $\mathcal{O}(n)$, y el chequear si se agrega o no es $\mathcal{O}(1)$. Por lo tanto esta sección tiene una complejidad igual a $\mathcal{O}(n)$.

[1] En la documentación de PuLP no se encontró referencia a qué método utiliza por detrás la libreria. Para simplificar el análisis, se asume que utiliza *simplex*.

Como se puede observar, ninguna sección de código toma más que $\mathcal{O}(n)$ o $\mathcal{O}(m)$. Por consiguiente, la complejidad de este algoritmo va a ser igual a $\mathcal{O}(max(n, m))$.

3.3.2. Efectividad de la aproximación

Para analizar qué tan efectiva es la aproximación por programación lineal continua, analicemos el modelo:

Tenemos como entrada un conjunto $U = \{u_1, u_2, \ldots, u_n\}$ de n elementos y un conjunto $S = \{S_1, S_2, \ldots, S_m\}$ de m conjuntos tal que $S_j \subseteq U$ para cualquier $j = 1, 2, \ldots, m$. El modelo de programación lineal es el siguiente:

$$\text{mín} \sum_{i=1}^{n} x_i$$

$$\text{sujeto a: } 0 \leq x_i \leq 1, i = 1, \ldots, n; \sum_{i:e_i \in S_j} x_i \geq 1, j = 1, \ldots, m,$$

$$x_{e_i} \geq 0$$

Definimos a I como una instancia cualquiera del Hitting Set Problem, definimos $z(I)$ como una solución óptima del mismo y a como una solución del modelo de programación lineal. Definimos a b como el máximo cardinal entre los elementos del conjunto S, y también definimos el conjunto C como $C = \{u_i \mid x_i \geq 1/b\}$ (es decir, el conjunto de los valores que "pasan.[el] redondeo). Definimos $A(I) = |C|$ como la solución aproximada a nuestro problema.

Sabemos que el valor de a puede ser menor que el valor de $z(I)$. Definimos la relación

$$a \leq z(I) \tag{1}$$

Luego, tenemos que

$$a = \sum_{i \in U} x_i$$

Si solo nos quedamos con los valores en C, tenemos

$$a \geq \sum_{i \in C} x_i$$

Sabemos que cualquier elemento en C va a ser como mínimo $1/b$. Por lo tanto podemos reemplazar en la inecuación anterior y nos queda

$$a \geq \sum_{i \in C} \frac{1}{b} = \frac{1}{b} \sum_{i \in C} 1 = \frac{1}{b}|C| = \frac{1}{b}A(I)$$

Combinando con la ecuación 1, nos queda

$$A(I) \leq ba \leq bz(I)$$

Por lo tanto, vemos que la solución aproximada $A(I)$ puede ser, como mucho, b veces la solución óptima.

m	$z(I)$	$A(I)$	b	$r(I)$
10	4	8	48	2.00
20	7	16	49	2.29
30	9	24	50	2.67
40	11	29	49	2.63
50	13	33	49	2.54
60	14	41	50	2.93
70	14	45	50	3.21
80	17	49	49	2.88
90	21	52	50	2.48
100	20	54	50	2.70
200	28	87	50	3.11
500	46	131	50	2.85

3.3.3. Mediciones

Para medir el algoritmo, se generaron sets de datos variando la cantidad de subconjuntos, el tamaño de los subconjuntos y con un universo de 500 elementos. Se corrió tanto con el algoritmo aproximado como con el algoritmo exacto para poder comparar sus mediciones y hallar el valor de $r(A)$. Estos fueron los resultados:

Como se puede observar, en las mediciones que se realizaron la cota de error no llega a ser tan grande como el peor caso posible. La máxima cota que se dio es tan solo 3,21, cuando según lo visto en la demostración este valor podía ascender al 50. Podemos entender que si bien la demostración mostró que el valor máximo de la cota de error es b, esto es solo en teoría. En la práctica, como lo demuestran estas mediciones, es complicado llegar a ese número.

4. Conclusiones

A lo largo de este trabajo se vieron diferentes técnicas para resolver un problema NP-Completo, en este caso el Hitting-Set Problem. En principio, se expuso cómo demostrar que un problema es NP-Completo y luego se procedió a formular diferentes técnicas para resolverlo, mostrando sus virtudes y sus falencias en el camino, en pos de encontrar la mejor manera de solucionar el problema.

De acuerdo a las mediciones y resultados obtenidos, podemos afirmar que, si nos encontráramos en la situación de ser parte del cuerpo técnico de Scaloni y tuviéramos que ayudarlo a armar una lista que contente al periodismo, nuestra mejor herramienta sería un modelo de programación lineal entera, de esta forma nos aseguramos no convocar a jugadores de más. Si se diera la situación en la que son demasiados los pedidos de los periodistas y Lionel nos apura un poco, podemos recurrir al método por programación lineal aproximada. Así, daríamos un buen estimado de la cantidad óptima de jugadores, sin necesitar tanto tiempo.

Bibliografía

Kleinberg, Jon y Éva Tardos (2005). *Algorithm Design*. Pearson.
Mitchell, Stuart et al. (2009). *Optimization with PuLP*. URL: `https://coin-or.github.io/pulp/`.