.UBAfiuba
FACULTAD DE INGENIERÍA

Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

24 de noviembre de 2023

Guillermo Stancanelli	Simón Stein	Santiago Tissoni
104244	106127	103856

1. Consideraciones

En el trabajo práctico se nos propone realizar un análisis al siguiente problema:

Se plantea la situación en la cual Scaloni tiene que preparar la convocatoria de jugadores para un partido amistoso. Los diferentes medios de comunicacion mantienen una lista de los jugadores que Scaloni deberia poner para el partido. Scaloni para poder satisfacer la demanda de los diferentes medios, quiere poner al menos algun jugador de los que propone cada medio.

Este problema se asemeja al Hitting-Set problem, en donde cada jugador es un elemento de un conjunto A y los conjuntos $B_1, B_2, ..., B_m$ son de jugadores que propone cada medio y definimos un k como el tamaño maximo de un conjunto $C \in A$ con $|C| \leq A$, que cubre un elemento de cada subconjunto $C \in B_i \neq \emptyset$)

2. Hitting-Set Problem

Realizaremos a continuación el análisis del Hitting-Set Problem, demostrando que es NP-Completo.

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

2.1. Demostración de pertenencia a NP

Para demostrar que el problema se encuentra en NP, debemos hallar un validador del problema que pueda verificar una posible solucion al mismo en tiempo polinomial. En este caso, es sencillo comprobar una posible solución C, ya que basta con recorrer todos los subconjuntos B_i y por cada uno chequear si al menos un elemento pertenece a C. Como cota maxima de complijad, este verificador es $O(n * m)$ con $n = |C|$ y $m = max|B|$.

2.2. Demostración de NP-Completo

Un problema NP-Completo es aquel al que pueden reducirse todos los problemas que se encuentran en NP. Para demostrar que Hitting-Set Problem es NP-Completo, tenemos que reducir un problema NP-Completo al mismo.

Tomaremos en este caso, Vertex Cover:

Dado un grafo G, encontrar un conjunto de vertices C pertenciente a G tal que los vertices de C cubran todas las aristas de G en no mas de k vertices.

En este grafo de ejemplo, podemos ver que $C = [1, 4, 6]$

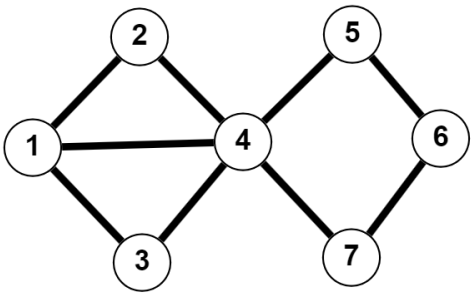

Figura 1: Grafo de ejemplo

Debemos transformar este grafo a una entrada que el Hitting-Set Problem pueda solucionar. Es decir un conjunto de elementos A, varios conjutos de elementos B_i y un numero k.

Vamos a crear un conjunto de elementos A que contenga a todos los vertices de G. En el caso del grafo de ejemplo este conjunto quedaria como: $[1, 2, 3, 4, 5, 6, 7]$. Luego por cada arista de G crearemos un conjunto B que la represente, donde los elementos de B seran los vertices que componen a la arista representada. En el ejemplo nos queda:

- $B_1 = [1, 2]$
- $B_2 = [1, 3]$
- $B_3 = [1, 4]$
- $B_4 = [2, 4]$

- $B_5 = [3, 4]$
- $B_6 = [5, 4]$
- $B_7 = [7, 4]$
- $B_8 = [5, 6]$
- $B_9 = [7, 6]$

Si le pasamos los conjuntos creados al Hitting-Set Problem junto a nuestro numero k en el problema Vertex Cover y el mismo resuelve que existe un Hitting-Set de tamaño k, podremos decir que existe un Vertex Cover de tamaño k en nuestro problema original. Esto se debe a que estaremos buscando un subconjunto C de vertices que pueda cubrir todas las aristas. Es decir, cada arista de nuestro grafo tiene que coincidir en al menos un vertice con el subconjunto C.

3. Algoritmos Propuestos

3.1. Backtracking

Un algoritmo diseñado con la técnica de Backtracking buscará la solución del problema recorriendo todo el conjunto de soluciones posibles con alguna poda del dominio de posibilidades al reconocer que la solución no puede ser encontrada por un camino en particular. Teniendo esto claro, definimos que nuestro algoritmo tendrá una constante k como número de elementos máximo para la solución (la poda) e irá recorriendo todos los elementos del conjunto A y los irá guardando en un arreglo con posible solución C. Por cada paso asume al elemento como parte de la solución, luego verifica si es una solución, o es posible seguir construyendo una solución desde ese punto y hará un llamado recursivo con el siguiente elemento del conjunto. En caso de no haber posible solución (por ejemplo que nos hayamos excedido del tamaño k), volveremos para atrás, quitando el elemento del arreglo de posible solución y probando con otro elemento.

```
def _hitting_set_rec(A, B, k, C, i):
    if len(C) > k:
        return None
    if is_feasible(B, C):
        return C
    if i >= len(A):
        return None

    C.add(A[i])
    res = _hitting_set_rec(A, B, k, C, i + 1)
    if res:
        return res

    C.remove(A[i])
    return _hitting_set_rec(A, B, k, C, i + 1)
```

Teniendo definido esto, nuestro algoritmo se ejecutará un cierto número de veces, probando diferentes k para tratar de minimizarlo. Se ejecutará una búsqueda binaria del k mínimo. Se prueba con un k especifico. Si hay solución, se vuelve a probar con un k menor, caso contrario, con uno mayor. De esta forma iremos probando polinomialmente varias soluciones hasta encontrar el k menor.

```
def hitting_set(A, B):
    low, high = 0, len(A)
    C = None
    while low < high:
        mid = (low + high) // 2
        res = _hitting_set_rec(A, B, mid, set(), 0)
        if res:
            if not C:
                C = res.copy()
            elif len(res) < len(C):
                C = res.copy()
            high = mid
        else:
            low = mid + 1
    return C
```

A continuación podemos ver cómo el algoritmo de backtracking tiene una tendencia exponencial en el tiempo a medida que aumenta el número de periodistas. Esto es de esperarse en un algoritmo de backtracking donde el peor caso equivale a la fuerza bruta.

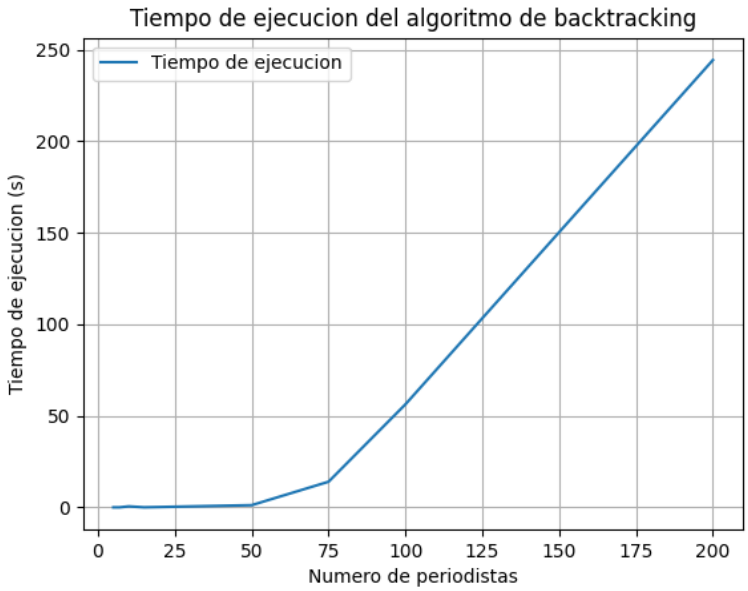

Figura 2: mediciones del algoritmo de backtracking

3.2. Programación lineal

Para formular el Hitting Set Problem como un modelo de programación lineal, vamos a definir el problema de manera más formal y adaptarlo a variables y restricciones.

- Tenemos un conjunto de elementos $A = a_1, a_2, ..., a_n$.
- Tenemos un conjunto de subconjuntos $B = B_1, B_2, ..., B_m$ donde cada $B_i \in A$.
- Nuestro objetivo es encontrar un Hitting Set $C \in A$ tal que C sea lo más chico posible y contenga al menos un elemento de cada B_i.

Para pasar esto a un problema de programación lineal, creamos variables binarias y restricciones de la siguiente manera:

- Variables Binarias: Para cada a_j, definimos una variable binaria x_j donde:
  - $x_j = 1$ si a_j está en el conjunto C.
  - $x_j = 0$ de lo contrario.
- Función Objetivo: Como nuestro objetivo es minimizar el tamaño de C, nuestra función objetivo es minimizar: $\sum_{j=1}^{n} x_j$.
- Restricciones: Por cada subconjunto B_i tenemos que asegurarnos de que al menos un elemento del mismo esté presente en C. Por ende, para cada B_i agregamos la restricción: $\sum_{x_j \epsilon B_i} x_j \geq 1$.

De esta manera, tenemos definido nuestro problema de programación lineal con variables binarias y podemos pasarlo a código:

```
def lineal_hitting_set(A, B):
    prob = pulp.LpProblem('Hitting_Set', pulp.LpMinimize)
    x = pulp.LpVariable.dicts('x', A, cat=pulp.LpBinary)
    prob += pulp.lpSum([x[a] for a in A])
    for b in B:
        prob += pulp.lpSum([x[a] for a in b]) >= 1
    prob.solve(pulp.PULP_CBC_CMD(msg=False))
    return set(a for a in A if x[a].value() == 1)
```

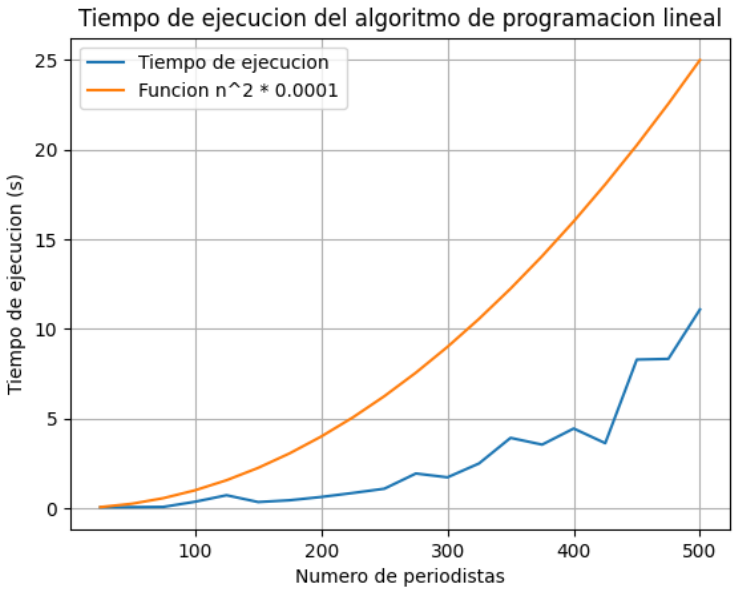

Figura 3: mediciones para el algoritmo de programación lineal

Podemos ver que el algoritmo de programación lineal binaria, performó mucho mejor que el algoritmo de backtracking. Sin embargo, los algoritmos de programación lineal entera, en el peor caso, tienen complejidad exponencial, aunque en la práctica muchos problemas pueden ser resueltos en tiempo mucho más rápido debido a la variabilidad en la complejidad dependiendo del caso específico. En este caso, puede que el tamaño de las muestras haya afectado a la medición.

3.3. Aproximación por programación lineal

Realizaremos una aproximación del Hitting Set problem por programación lineal. Para esto, haremos una versión relajada del algoritmo de programación lineal binaria y permitiremos que las variables puedan tomar valores reales. Una vez que se haya encontrado la solución óptima del problema relajado, redondearemos utilizando el siguiente criterio:

Para cada variable x_j, si su valor en la solución relajada es mayor o igual a $1/b$, siendo b el tamaño del subconjunto de B con mayor tamaño, redondearemos el valor de x_j a 1. De lo contrario, se redondeará a 0.

Modelamos esta solución en código:

```
def aproximated_hitting_set(A, B):
    prob = pulp.LpProblem("HittingSetProblem_Relaxed", pulp.LpMinimize)

    x = pulp.LpVariable.dicts("x", A, 0, 1)
    for b in B:
        prob += sum(x[a] for a in b if a in A) >= 1

    prob += sum(x[a] for a in A)
    prob.solve(pulp.PULP_CBC_CMD(msg=False))

    b = max(len(b) for b in B)

    return set(a for a in A if x[a].value() >= 1 / b)
```

La complejidad de este algoritmo depende principalmente de la resolución del problema de programación lineal, que puede variar según el solucionador utilizado. Si se utiliza el método Simplex, en términos generales, la complejidad suele ser polinómica en la práctica para instancias de tamaño moderado.

Haremos un análisis de la aproximación para ver que tan buena es:

Siendo $A(I)$ la solución aproximada y $z(I)$ la solución óptima para cualquier instancia del Hitting Set Problem. Buscaremos un $r(A)$ tal que $A(I)/z(I) \leq r(A)$ para todas las instancias posibles. Para esto, analizaremos la restricción de nuestro modelo: $\sum_{x_j \epsilon B_i} x_j \geq 1$.

Podemos notar que esta sumatoria como mucho, tendrá b términos. Esto implica que en el peor de los casos, cada elemento de un subconjunto B_i podría contribuir con un valor de exactamente $1/b$ a la suma, cumpliendo así la restricción. Esto quiere decir que podríamos llegar a seleccionar todos los elementos del subconjunto B_i con mayor tamaño en el peor de los casos, es decir, b elementos. Por otro lado, la solución óptima binaria, $z(I)$, es siempre ≥ 1, ya que al menos un elemento es necesario para obtener un Hitting Set de A.

De esta forma, si en el peor de los casos nuestra solución aproximada $A(I)$ tiene tamaño b y como nuestra solución aproximada, $z(I) \geq 1$, podemos decir que, en el peor de los casos, $A(I)/z(I) \leq b/1 = b$. Indicando así que la solución aproximada es una b-aproximación de la solución óptima.

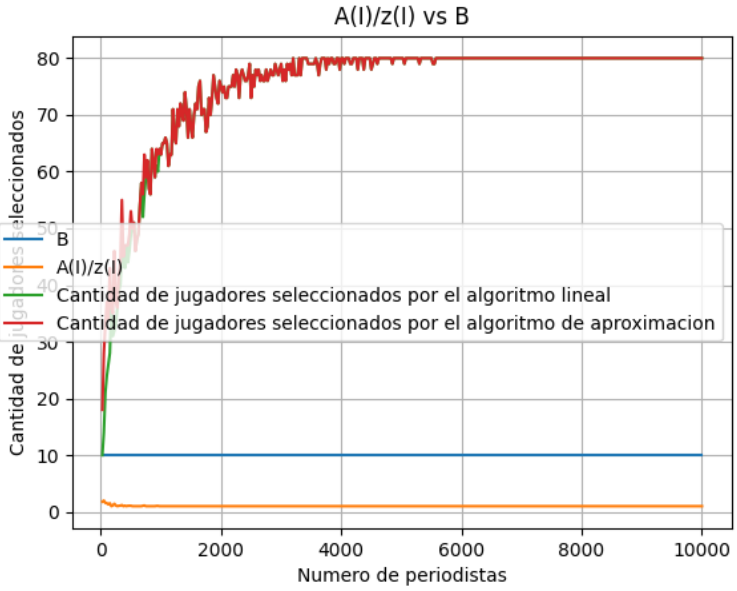

Figura 4: mediciones de $A(I)/z(I)$

Podemos observar en el gráfico, que $A(I)/z(I)$ nunca supera b para ninguna instancia probada. Sin embargo, es posible que exista una cota más baja ya que la relación no se acercó mucho a b para ninguna instancia del problema.

3.4. Aproximación Greedy

Para nuestra solución greedy del problema, comenzamos aplicando un ordenamiento como es natural en muchos problemas greedy. Ordenamos los elementos de A_i según cuántos B_is contienen dichos elementos. Luego agregamos la solución C el A_i que queda en el primer lugar del array ordenando, extrayéndolo en el proceso, y quitamos de B a todos los B_is que contenían a ese A_i. Repetiremos este proceso hasta que B este vacío, habiendo así solucionado el hitting set problem. No podemos garantizar que el resultado sea minimal, pero sí es una resolución intuitiva que se completa en tiempo polinomial.

```
1 def matching_bs(a, B):
2   matching_bs =  []
3   for b in B:
```

```
    if a in b:
      matching_bs.append(b)

  return matching_bs


def sort_most_matched_elements(A, B):
  a_with_matching_bs = []
  for a in A:
    mbs = matching_bs(a, B)
    a_with_matching_bs.append((a, mbs))

  return sorted(a_with_matching_bs, key=lambda a_s : len(a_s[1]), reverse=True)

def greedy_hitting_set(A, B):
  C = set()
  while not len(B) == 0:
    sorted_as_with_matching_bs = sort_most_matched_elements(A, B)
    (most_matched_a, matching_bs) = sorted_as_with_matching_bs[0]
    B = [b for b in B if b not in matching_bs]
    C.add(most_matched_a)
    A.remove(most_matched_a)

  return C
```

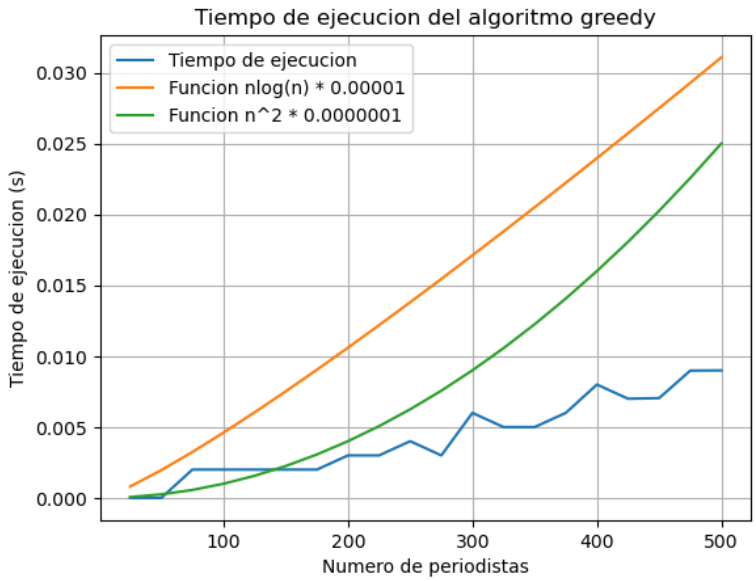

Figura 5: medición del algoritmo greedy

Sea m el largo de B y n el largo de A. Recorrer todos los B_is en la función matching_bs() tiene complejidad $O(m)$, y chequear si esos conjuntos contienen determinado elemento es $O(1)$. Luego, repetimos esto para cada A_i en la función sort_most_matched_elements(), para luego ordenarlos, por lo que quedamos con una complejidad $O(n * m + nlog(n))$. Si tenemos en cuenta de que todo esto se repite en el peor caso m veces en greedy_hitting_set(), entonces tenemos una complejidad $O(m^2 * n + m * nlog(n))$, que se reduce a $O(m^2 * n)$ si nos quedamos con el término más pesado.

4. Conclusiones

Al resolver el trabajo practico, pudimos entender que si bien el Hitting Set Problem es complejo y desafiante (al ser NP completo), existen varios métodos para abordarlo. La elección del método depende del tamaño de la instancia y de si se requiere una solución exacta o una aproximación razonable en un tiempo práctico.

Para nuestro caso descubrimos que una solución por programación lineal fue lo más rápido, y podemos concluir que aquellos problemas que se puedan equivaler a Hitting-Set mediante un proceso de reducción, podrán ser resueltos con los mismos algoritmos que hemos probado