Teoría de Algoritmos
(75.29) Curso Buchwald - Genender

Trabajo Práctico 3
Problemas NP-Completos

5 de marzo de 2024

Martin Tomas
100835

Adriana Macarena Iglesias Tripodi
103384

1. Introducción

En este trabajo se busca, mediante la modelización del problema a uno de tipo Hitting-Set, encontrar la mínima cantidad de jugadores que deben participar del próximo partido para satisfacer a los medio. Un problema Hitting-Set es aquel en el que teniendo un conjunto A y un número m de subconjuntos B_i pertenecientes a A encontrar un subconjunto de k elementos que contengan como mínimo un elemento de cada subconjunto.

1.1. Demostración que Hitting-Set es NP

Demostrar que un problema es NP es relativamente sencillo. Como $P \in NP$ solo hay que demostrar que si se tienen los resultados al problema se puede verificar su pertenencia o no a este en tiempo polinomial.

Consecuentemente, si se tiene un set X que es solución de un problema de tipo Hitting-Set, comprobar que esto es cierto implica a lo sumo comparar cada elemento del set de soluciones contra cada subconjunto B_i para asegurarse de que al menos uno de dichos elementos pertenezca a cada subconjunto B. Si la cantidad de subconjuntos B son m, su cantidad de elementos son l y la cantidad de elementos en el set X son n entonces esto lleva a lo sumo $n \times l \times m$ siendo un tiempo polinomial. Un posible algoritmo de esto es:

```
def check_if_in_subsets(B,C):
    length=0
    for Bi in B:
        if any(i in Bi for i in C):
            length+=1
    if length>=len(B):
        return True
    return False
```

Donde es fácil ver que su complejidad temporal es $O(n^2)$ siendo n la cantidad de subconjuntos B_i, ya que a lo sumo C tiene esa cantidad de elementos.

1.2. Demostración que Hitting-Set es NP-Completo

Demostrar esto es un poco más difícil que demostrar que Hitting-Set es NP esto es consecuencia de que para demostrar que un problema es NP completo se debe reducir otro problema que ya se a demostrado que es NP completo a este. Para el caso particular de un problema de tipo Hitting-Set se va a intentar demostrar que es NP completo mediante la reducción de un problema de tipo Vertex-Cover a este.

El problema de vertex-Cover es aquel en el que dado un grafo G=(V,E), un subset S de nodos ($S \in V$) cubre todos las aristas del grafo. Mientras que en el problema de Hitting-Set queremos encontrar un subconjunto de a lo sumo k elementos con el que se cubra todos los subconjuntos B_i pertenecientes a A, es decir que haya por lo menos un elemento de B_i en el subconjunto solución. Entonces, para intentar de reducir Vertex-Cover a Hitting-Set primero hay que descubrir que es lo que se va a tratar como los subconjuntos B_i sabiendo que lo que se busca es minimizar el k. Como lo que se quiere en Vertex-Cover es cubrir todas las aristas, se pasa cada una de estas (en función de los dos vértices que la componen) como un B_i y se van pasando valores de k en forma decreciente hasta encontrar el mínimo que sigue brindando un subconjunto como solución. Siendo dicho subconjunto también la solución del problema de Vertex-Cover y el k el mínimo número de vértices que lo componen.

Resolución

2. Desarrollo

Al ser un algoritmo de resolución NP-Completa, para llegar a una solución ideal se lo va a abordar de distintas formas. Ya que resolverlo de forma óptima podría conllevar un gran costo tempora.

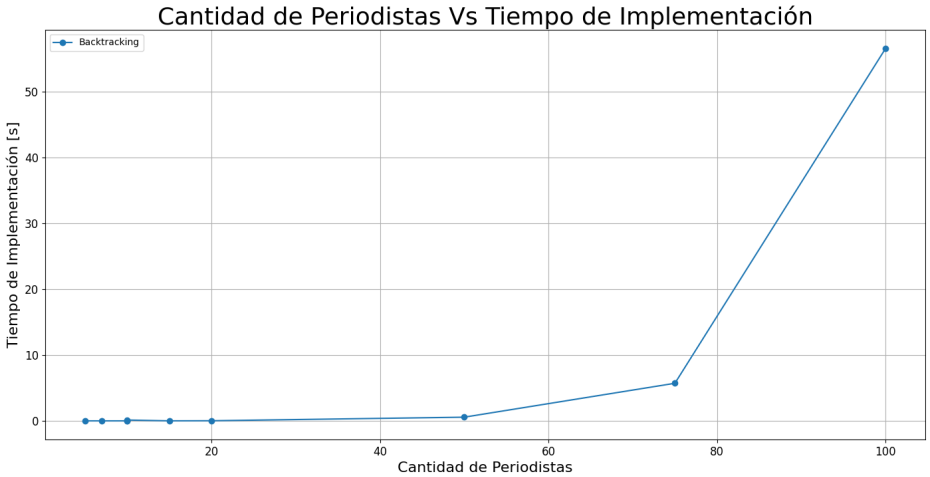
2.1. Backtracking

A continuación se muestra el código de solución del problema mediante Backtracking. En este se van escogiendo posibles integrantes del conjunto solución y en el caso de que no lleguen a una solución óptima se elimina dicho elemento de la solución y se guarda otro en su lugar.

```
def hitting_Set(B,k, actual_solution, solution):
    if k>=len(B):
        return False
    if check_if_in_subset(B[k],actual_solution):
        return hitting_Set(B,k+1,actual_solution, solution)
    for i in B[k]:
        if len(solution) != 0 and len(actual_solution) +1>= len(solution):
            return False
        actual_solution.append(i)
        if check_if_in_subsets(B[k:], actual_solution):
            solution[:] = actual_solution
        else:
            hitting_Set(B, k+1, actual_solution, solution)
        actual_solution.pop()
    return True
```

Particularmente, es algoritmo posee cinco etapas distinguibles. La primera de estas es el primer corte, donde se decide no continuar buscando elementos solución si la cantidad de elementos guardados es mayor a la cantidad de subconjuntos B (ya que esto seguro no es solución óptima). La segunda, produce una llamada recursiva si la solución actual es parte del subconjunto que se está analisando. Las otras tres etapas del algoritmo son cíclicas, al repetirse para cada elemento de cada subconjunto B a menos que algo cause que salga del ciclo. La primera de estas etapas consiste en parar de buscar soluciones si se tiene un conjunto solución de menor cantidad de elementos de los actualmente acumulados en mi nuevo conjunto solución (ya que lo que se está encontrando no es mejor a lo guardado previamente), a menos que lo guardado previamente sea el conjunto vacío.Luego de esto se guarda un elemento en el conjunto solución y comienza la cuarta etapa donde si nuestra solución actual es un conjunto solución y esta posee menos elementos que el conjunto solución previamente guardado, se reemplaza dicho conjunto por el actual y se sale del ciclo. Finalmente la última etapa hace un llamado recursivo en la que se trabaja con el próximo subconjunto B_i.

Consecuentemente, la complejidad del algoritmo propuesto no es polinómica, al, para buscar la solución, se combinan cada elemento de cada subconjunto con el resto de ellos y se van probando si esto es solución. El backtracking implementado en el algoritmo produce que las excepciones sean las situaciones de poda, donde se deja de recorrer esa rama de posibilidades. De hecho, como cada elemento tiene la posibilidad de formar parte o no del conjunto solución la complejidad del algoritmo es $\mathcal{O}(2^m)$, donde m es la cantidad de subconjuntos de B. En el siguiente gráfico, la relación directa entre el aumentó de periodistas y el aumento del tiempo de resolución es fácilmente notable, así como también su falta de correlación con una función polinómica.



2.2. Algoritmo por Programación Lineal Entera

Para poder obtener una solución óptima de un problema NP-Completo, este puede ser reducible a otro problema NP-Completo. Entonces, por ahora, se podría obtener solo una solución óptima al problema de Hitting Set si este se reduce a un problema de programación lineal entera, no continua, ya que sino $NP = P$.

Para poder crear un algoritmo de programación lineal hay que definir:

- La función objetivo y si es de maximización o minimización
- Las Restricciones
- Las forma de las variables

Y habría que buscar como adaptar las variables de nuestro problema a este.

Como se quiere encontrar la mínima cantidad de elementos que cubran todos los subconjuntos B, entonces tenemos un problema de minimización donde cada variable es un ente binario que toma valor 1 si el jugador está dentro de la solución óptima y 0 si no. En este caso el modelo quedaría:

$$Z_{min} : \sum_{B_0}^{B_m} \sum_{j=0}^{n} i_j$$

$$\sum_{j=0}^{n} i_j \geq 1 \ \forall i \in \{1, 0\}$$

Donde cada variable i es la pertenencia o no al conjunto solución del jugador j, n es la cantidad máxima de jugadores en cada subconjunto y B_m es el último subconjunto. Entonces:

```
def hitting_set_PLE(B):
    prob = LpProblem("Hitting_Set", LpMinimize)
    A=get_elements(B)
    i = LpVariable.dicts("i", A, cat='Binary')
    prob += lpSum(i[j] for j in A)
    for Bi in B:
        prob += lpSum(i[j] for j in Bi if j in A) >= 1
    prob.solve(PULP_CBC_CMD(msg=False))
    if LpStatus[prob.status] == 'Optimal':
        return [j for j in A if i[j].value() == 1]
    return None
```

Donde get_elements es una función que devuelve los elementos de los conjuntos B_i si repeticiones

```python
def get_elements(B):
    A=[]
    for Bi in B:
        for i in Bi:
            if i not in A:
                A.append(i)
    return A
```

2.3. Algoritmo por Aproximaciones

Para poder Se sabe que a lo sumo la mejor solución es que todos los subconjuntos compartan un elemento y la peor que no se comparta ninguno. Entonces:

$$1 \leq k \leq m$$

Donde m es el número de subconjuntos B_i.

2.3.1. Algoritmo Greedy

La idea que tomamos para el algoritmo greedy es primero contar para cada jugador la cantidad de periodistas que contenta, luego ordenar de mayor a menor esta cantidad. Luego comenzamos a solucionar un problema local con el fin de llegar al óptimo global que es encontrar los jugadores que contentan a todos los periodistas periodistas. Para esto, tomamos el primer jugador, lo seleccionamos y quitamos de los «periodistas a contentar» aquellos periodistas que ya contenta este jugador, reordenamos el listado y volvemos a tomar un jugador hasta completar todos los periodistas.

```python
def hitting_set_greedy(ordered_players):
    solution =[]
    visited=[]
    while True:
        player=list(ordered_players.items())[0][0]
        if len(ordered_players[player])!=0:
            solution.append(player)
        else:
            break
        visited[:]=ordered_players[player]
        ordered_players.pop(player)
        for journalist in visited:
            for other_player in ordered_players:
                if journalist in ordered_players[other_player]:
                    ordered_players[other_player].remove(journalist)
        ordered_players = sort_by_journalist(ordered_players)
    return solution
```

Con respecto a la complejidad temporal, este algoritmo contiene tres ciclos, comenzando desde el interior, el primero de estos recorre todos los jugadores restantes en la lista de jugadores totales, el segundo recorre la cantidad de periodistas que los mencionan por cada jugador y el primero trabaja siempre que quede algún periodista sin estar cubierto. Consecuentemente, la complejidad temporal de este algoritmo es $\mathcal{O}\left(n^2 \times m\right)$ donde m es la cantidad de periodistas totales y n la cantidad total de jugadores.

Respecto a un análisis empírica, se sabe que:

$$1 \leq k \leq m$$

Donde k es la solución óptima y que una ves ordenado:

$$p_n \geq p_{n+1}$$

Con cada p_j la cantidad de periodistas que quieren que el jugador j juegue el amistoso. Además:

$$\sum_{j=0}^{m} p_j \geq m$$

y si n son la cantidad de jugadores:

$$\sum_{j=0}^{m} p_j \leq n \times m$$

Adicionalmente:

$$p_n \leq m$$

Entonces:

$$\sum_{j=0}^{m-1} p_j \geq 0$$

y

$$\sum_{j=0}^{m-1} p_j \leq m \times (n-1)$$

Si se considera lo mismo para el resto de las p_j:

$$0 \leq m \times (n-m)$$

$$m \leq m \times n$$

Y se considera

$$n \times m = k*$$

de greedy:

$$k \leq k*$$

3. Comparaciones

La diferencias encontradas para cada implementación se pueden observar en las siguientes tablas:

Cantidad de Datos	Óptimo	Backtracking	PLE	Greedy
5	2	2	2	2
7	2	2	2	2
10	3	3	3	4
10	6	6	6	7
10	10	10	10	10
15	4	4	4	4
20	5	5	5	5
50	6	6	6	7
75	8	8	8	9
100	9	9	9	10
200	9	9	9	10

Cuadro 1: Resultados obtenidos según algoritmo implementado

Cantidad de Datos	Backtracking	PLE	Greedy
5	8.65e-05	5.56e-02	1.98e-05
7	8.58e-05	4.92e-02	2.87e-05
10	5.54e-04	4.62e-02	5.91e-05
10	1.01e-02	4.69e-02	7.13e-05
10	1.5e-01	4.57e-02	8.66e-05
15	2.48e-03	5.54e-02	9.49e-05
20	1.67e-02	4.48e-02	1.44e-04
50	0.56	5.36e-02	4.45e-04
75	6.70	0.262	3.32e-04
100	71.6	0.344	6.16e-04
200	1.6e+03	0.898	1.52e-03

Cuadro 2: Tiempo en segundos ocupado para obtener resultados según algoritmo implementado

Como se puede apreciar, los algoritmos de Backtracking y programación lineal entera entregan siempre la solución óptima pero, el tiempo de ambas es mucho mayor que el tiempo de implementación del algoritmo Greedy. Adicionalmente, mientras que el tiempo de implementación de PLE se mantiene relativamente constante hasta incrementar abundantemente la cantidad de periodistas, el algoritmo de Backtracking ve cambios importantes a penas se incrementa un poco dicho parámetro y el algoritmo Greedy, también tiene una correlación directa, aunque menos notable, con el aumento de este parámetro. Finalmente, es importante notar que en ningún momento el algoritmo greedy erró en más de una unidad sobre la solución óptima.

4. Conclusiones

Bueno, pudimos demostrar que el problema Hitting Set es un problema NP-Completo. Planteamos una solución utilizando backtracking y también una aproximación Greddy, la cual no siempre da resultado óptimos, pero si nos ayudo a aproximarnos al problema en tiempos polinómicos. Adicionalmente planteamos una solución que alcanza óptimos que ayuda a reducir mucho los tiempos de implementación.