DAA

Proyecto Final

Ejercicio 3- k-Hitting Set

Autor:

Karen Dianelis Cantero López (C411) karen.canterolopez@gmail.com

2024

Contents

1 Definición formal del problema

Given a ground set X of elements and also a grouping collection C of subsets available in X and an integer k, the task is to find the smallest subset of X, such that the smallest subset, H hits every set comprised in C. This implies that the intersection of H and S is null for every set S belonging to C, with size k.

2 Demostrando que es un problema NP-Completo

Para demostar que un problema pertenece a NP-Completo debemos demostrar que pertenece a NP, y a NP-hard

2.1 Demostrando que está en NP

Si algún problema está en NP, entonces dado un 'certificado', que es una solución al problema y una instancia del problema (un conjunto base X, una colección C de subconjuntos S y k), podremos verificar el certificado (si la solución es correcta o no) en tiempo polinomial. Esto se puede hacer de la siguiente manera:

Dada una solución candidata H, necesitamos verificar que H sea un conjunto válido para todos los subconjuntos $S \in C$ tales que $|H| \leq k$.

1. - Para cada subconjunto $S \in C$, verificamos si $S \cap H \neq \emptyset$. Esto requiere:
   (a) Iterar a través de cada subconjunto $S \in C$.
   (b) Para cada S, verificar si H se interseca con S.
2. Verificar $|H| \leq k$.

La cantidad de subconjuntos en C es finita y se basa en el tamaño de entrada, por lo que verificar que H interseca a cada uno de ellos se puede realizar en tiempo polinomial.

Por lo tanto, el problema está en NP.

2.2 Demostrando que esta en NP-Hard

Para demostrar que el problema del k Hitting Set es NP-hard, podemos realizar una reducción en tiempo polinomial a partir del problema de Vertex Cover, que se sabe que es NP-duro.

2.2.1 Definiendo el problema de Vertex Cover

- **Entrada:** Un grafo $G = (V, E)$, donde V es el conjunto de vértices y E es el conjunto de aristas, y un entero k.
- **Objetivo:** Encontrar un subconjunto $V' \subseteq V$ de vértices tal que:
  1. V' tiene tamaño $\leq k$
  2. Cada arista $e = (u, v) \in E$ tiene al menos uno de sus extremos en V'.

2.2.2 Reducción

Necesitamos reducir el problema de Vertex Cover al problema de k-Hitting Set en tiempo polinomial.

- Definimos el conjunto base X como el conjunto de vértices en V. Por lo tanto, $X = V$.
- Para cada arista $e = (u, v) \in E$, creamos un subconjunto $S_e = \{u, v\}$. Por lo tanto, cada subconjunto S_e representa una arista, y la colección C de subconjuntos corresponde al conjunto de aristas en G. Cada arista es un conjunto de dos vértices (puntos finales de la arista), por lo que cada subconjunto en C tiene tamaño 2.
- El entero k para el problema de k Hitting Set corresponde al número de vértices que queremos en el Vertex Cover.

El objetivo del problema de Vertex Cover es encontrar un subconjunto $V' \subseteq V$ (de tamaño $\leq k$) que cubra todas las aristas. Es decir, para cada arista $(u, v) \in E$, al menos un vértice de V' debe ser uno de los puntos finales u o v.

En el problema de k Hitting Set, el objetivo es encontrar un subconjunto $H \subseteq X$ que "cubra" cada subconjunto en C, lo que significa que $H \cap S_e \neq \emptyset$ para cada subconjunto $S_e \in C$. Dado que cada subconjunto S_e corresponde a una arista y es de tamaño 2, esto es equivalente a requerir que al menos uno de los vértices de cada arista (u, v) esté incluido en el hitting set.

Si hay un vertex cover de tamaño k en G, entonces hay un hitting set de tamaño k , y viceversa.

Conclusión:

Hemos demostrado que cualquier instancia del problema de vertex cover se puede transformar en una instancia equivalente del problema de k Hitting Set en tiempo polinomial.

Dado que Vertex Cover es NP-hard, y podemos reducirla al problema de hitting set en tiempo polinomial, el problema de hitting set es al menos tan difícil como el vertex cover.

Por lo tanto, el problema es NP-hard.

3 Solucion fuerza bruta

3.1 Algoritmo

Una solución de fuerza bruta para el problema k Hitting Set sigue estos pasos:

1. Generar todos los subconjuntos posibles de X: el número total de subconjuntos de un conjunto X de tamaño $|X|$ es $2^{|X|}$. Por lo tanto, el enfoque de fuerza bruta tiene que examinar cada uno de estos subconjuntos.
2. Verificar cada subconjunto: para cada subconjunto de X, el algoritmo verifica si "alcanza" cada subconjunto en C (si se interseca con cada subconjunto en C).
3. Devolver el subconjunto más pequeño que alcanza cada subconjunto en C y su tamaño es menor que k.

3.2 Complejidad Temporal

Como se ha dicho, hay $2^{|X|}$ subconjuntos posibles de X.

Para cada subconjunto de X, debemos comprobar si interseca a todos los subconjuntos de C. El tamaño de C es n, por lo que para cada subconjunto de X, debemos comprobar todos los n subconjuntos de C. Comprobar si un subconjunto interseca a otro subconjunto puede llevar hasta $O(|X|)$ tiempo (ya que en el peor de los casos, es posible que deba comparar todos los elementos).

Por lo tanto, la complejidad temporal general es:

$$O(2^{|X|} \times |C| \times |X|)$$

4 Solución Greedy

Una solución greedy consiste en seleccionar iterativamente el elemento que cubre el número máximo de subconjuntos no cubiertos.

Seguría los siguientes pasos:

1. Inicializar un conjunto de impacto vacío.
2. Mientras haya subconjuntos no cubiertos:
   (a) Seleccionar el elemento que interseca la mayor cantidad de subconjuntos no cubiertos.
   (b) Agregar este elemento al hitting set.
   (c) Eliminar los subconjuntos cubiertos de la colección.
   (d) Detenerse cuando el tamaño del hitting set alcance k o no haya más elementos que puedan mejorar la cobertura.

4.1 Aproximacion

En cada paso, el algoritmo selecciona el elemento de X que cubre el mayor número de subconjuntos no cubiertos, lo agrega al hitting set y elimina todos los subconjuntos que están cubiertos por este elemento de la lista de subconjuntos descubiertos. El algoritmo termina una vez que el tamaño del hitting set alcanza k o todos los subconjuntos están cubiertos.

Sea T_j el número de subconjuntos descubiertos después de que se haya agregado el $j - ésimo$ elemento al conjunto de impacto. Inicialmente, $T_0 = n$ (donde n es el número total de subconjuntos). Después de agregar el $j - ésimo$ elemento, el número de subconjuntos descubiertos se reduce a $T_j = T_{j-1} - c(u_j)$, donde $c(u_j)$ es el número de subconjuntos cubiertos por el elemento u_j. En el paso j, los subconjuntos no cubiertos debe ser al menos T_{j-1}, y el algoritmo selecciona el elemento u_j que cubre la mayor cantidad de subconjuntos. Por lo tanto, el número de subconjuntos descubiertos después de agregar el $j - ésimo$ elemento está acotado por:

$$T_j \leq T_{j-1} - \frac{T_{j-1} * c(u_j)}{|X|} = T_{j-1}(1 - \frac{c(u_j)}{|X|})$$

Dado que el algoritmo elige el elemento que maximiza la cobertura, $c(u_j)$ es al menos el número promedio de subconjuntos descubiertos cubiertos por cualquier elemento, es decir,

$$c(u_j) \geq \frac{T_{j-1}}{|X|}$$

Por lo tanto, podemos acotar T_j como:

$$T_j \leq T_{j-1}(1 - \frac{1}{|X|})$$

Esta desigualdad se cumple en cada iteración, y después de j iteraciones, el número de subconjuntos descubiertos es:

$$T_j \leq n(1 - \frac{1}{|X|})^j$$

Podemos aproximar $(1 - \frac{1}{|X|})^j$ usando la desigualdad $1 - x \leq e^{-x}$. Esto da:

$$T_j \leq ne^{\frac{-j}{|X|}}$$

Para cubrir todos los subconjuntos, $T_j = 0$, por lo que necesitamos $T_j < 1$, lo que implica:

$$ne^{\frac{-j}{|X|}} < 1$$

Esto se debe a que $T_j \leq ne^{\frac{-j}{|X|}}$ proporciona un límite superior para el número de subconjuntos descubiertos después de j pasos, y si este límite superior es menor que 1, entonces el número real de subconjuntos descubiertos T_j también debe ser menor que 1.

Dividimos por n:

$$e^{\frac{-j}{|X|}} < \frac{1}{n}$$

Aplicamos ln en ambos lados:

$$-\frac{j}{|X|} < ln(\frac{1}{n})$$

Simplificando $ln(\frac{1}{n}) = ln(1) - ln(n) = 0 - ln(n) = ln(n)$:

$$-\frac{j}{|X|} < -ln(n)$$

$$\frac{j}{|X|} > ln(n)$$

Y finalmente:

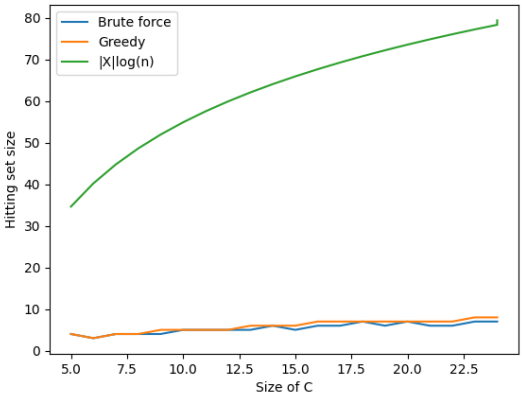
$$j > |X|ln(n)$$

Por lo tanto, para cubrir todos los subconjuntos, el algoritmo necesita como máximo $|X|ln(n)$ iteraciones. Dado que en cada iteración, el algoritmo agrega un nuevo conjunto al conjunto de impacto, el tamaño del conjunto de impacto está limitado por $|X|ln(n)$.

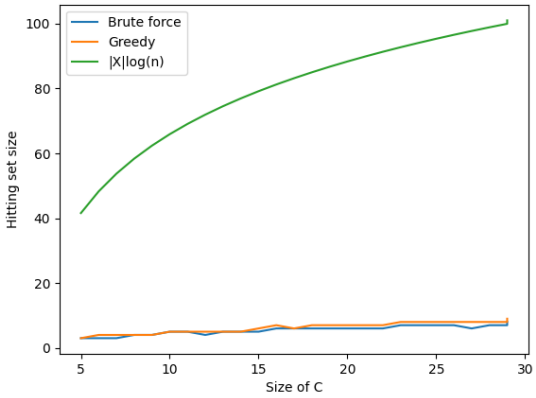
Hemos demostrado que la solución está limitada por $|X|ln(n)$

4.2 Visualización de la acotación

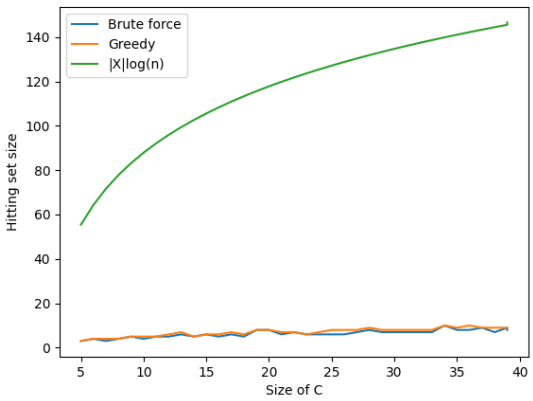
Se probaron 10000 instancias del problema para cada tamaño de X (ground set). Se agruparon los resultados por el tamaño de C (los subconjuntos que deber ser cubiertos) y se tomó el valor máximo devuelto por el algoritmo greedy. A continuación se puede observar la acotación para varios valores de $|X|$.



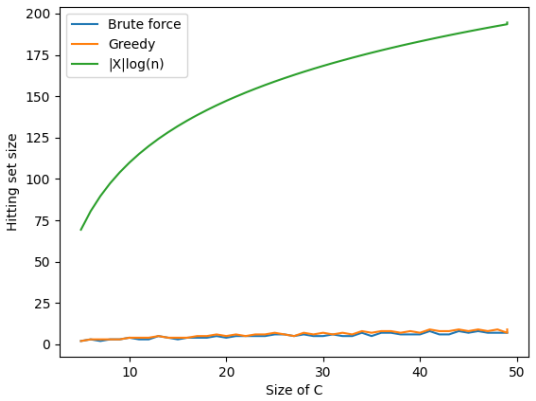
(a) $|X| = 25$
Solución Correcta: 91.66%
Proporción Real: 1.08 veces
Proporción teórica: 24.19 veces



(b) $|X| = 30$
Solución Correcta: 90.41%
Proporción Real: 1.09 veces
Proporción teórica: 28.73 veces



(a) $|X| = 40$
Solución Correcta: 88.26%
Proporción Real: 1.11 veces
Proporción teórica: 37.45 veces



(b) $|X| = 50$
Solución Correcta: 88.14%
Proporción Real: 1.13 veces
Proporción teórica: 42.74 veces

Figure 2: Probando los resultados para varias instancias del problema

4.3 Acotación caso especial

Como un caso especial, tenemos cuando $|S| = |C|$, en otras palabras, hay la misma cantidad de elementos en el conjunto S, que subconjuntos en el conjunto C.

Veamos que ocurre en este caso: Sea R el tamaño mínimo de los subconjuntos de C ($|S_i| \geq R \quad \forall S_i \in C$). De igual manera que en la acotación anterior, Denotamos por T_j el número de subconjuntos que aún no están cubiertos después de que se hayan agregado j elementos a S. Inicialmente, $T_0 = n$, y queremos encontrar el j más pequeño tal que $T_j = 0$, lo que significa que todos los subconjuntos están cubiertos.

Sea u_j el $j-ésimo$ elemento agregado a S. El número de subconjuntos cubiertos después de agregar u_j está determinado por el valor de $c(u_j)$, el número de subconjuntos que contienen a u_j en ese punto.

En el paso j, el número de subconjuntos restantes T_j se actualiza como:

$$T_j = T_{j-1} - c(u_j)$$

Dado que el número total de subconjuntos descubiertos al comienzo del paso j es T_{j-1}, y cada subconjunto contiene al menos R elementos, el recuento total de subconjuntos restantes sumado a todos los elementos restantes (es decir, elementos que aún no se han agregado a H es:

$$\sum_{v \in V/\{u_1,u_2,\ldots,u_{j-1}\}} c(v) = T_{j-1}R$$

Hay $n-j+1$ elementos que aún no se han agregado a S, por lo que el recuento promedio para un elemento es:

$$\frac{T_{j-1}R}{n-j+1}$$

Por la elección codiciosa, el elemento u_j elegido en este paso tiene el conteo máximo $c(u_j)$, por lo que debe satisfacer:

$$c(u_j) \geq \frac{T_{j-1}R}{n-j+1}$$

Usando esto, podemos limitar la reducción en T_j de la siguiente manera:

$$T_j \leq T_{j-1} - \frac{T_{j-1}R}{n-j+1} = T_{j-1}(1 - \frac{R}{n-j+1})$$

Podemos desenrollar esta recurrencia para expresar T_j en términos de T_0 (que es n):

$$T_j \leq T_0 \prod_{l=0}^{j-1}(1 - \frac{R}{n-l}) = n \prod_{l=0}^{j-1}(1 - \frac{R}{n-l})$$

Para simplificar este producto, utilizamos la desigualdad $1 - x \leq e^{-x}$. Aplicando esto a cada término del producto, obtenemos:

$$n \prod_{l=0}^{j-1}(1 - \frac{R}{n-l})\ leqn \prod_{l=0}^{j-1} e^{-\frac{R}{n-l}} = ne^{-\sum_{l=0}^{j-1}\frac{R}{n-l}}$$

La suma $\sum_{l=0}^{j-1}\frac{R}{n-l}$ se puede aproximar como:

$$\sum_{l=0}^{j-1}\frac{R}{n-l} \approx \frac{Rj}{n}$$

Por lo tanto, tenemos:

$$T_j \leq ne^{-\frac{Rj}{n}}$$

Para asegurarnos de que todos los subconjuntos estén cubiertos, queremos que $T_j \leq 1$:

$$ne^{-\frac{Rj}{n}} < 1$$

Dividiendo por n y aplicando logaritmo, obtenemos:

$$-\frac{Rj}{n} < ln(\frac{1}{n}) = -ln(n)$$

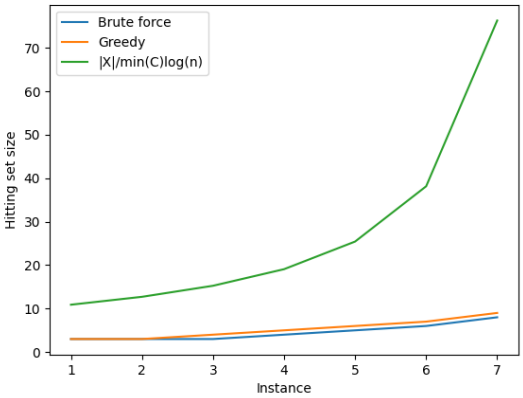
$$\frac{Rj}{n} > ln(n)$$

$$j > \frac{n}{R}ln(n)$$

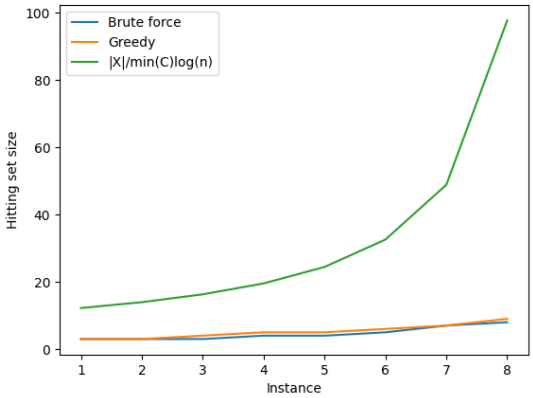
Por lo tanto, el número de pasos j necesarios para cubrir todos los subconjuntos es como máximo $\frac{n}{R}ln(n)$, lo que significa que el tamaño del conjunto impactante S es como máximo $\frac{n}{R}ln(n)$

4.4 Visualización de la acotación

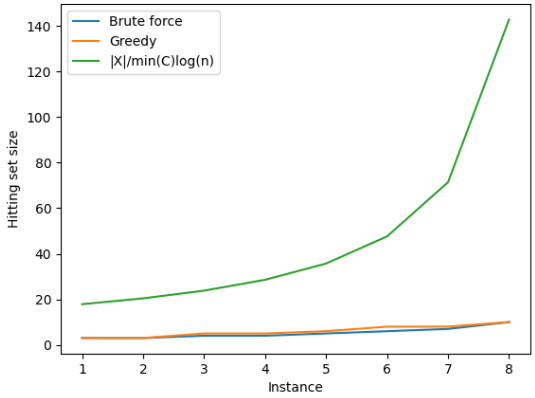
De igual manera que anteriormente, se probaron 10000 instancias del problema para cada tamaño asegurandonos de que el tamaño de C fuera el mismo que el tamaño de S. Obtuvimos los siguientes resultados:



(a) $|X| = 25$
Solución Correcta: 89.24%
Proporción Real: 1.11 veces
Proporción teórica: 15.69 veces



(b) $|X| = 30$
Solución Correcta: 87.39%
Proporción Real: 1.13 veces
Proporción teórica: 18.52 veces



(a) $|X| = 40$
Solución Correcta: 84.73%
Proporción Real: 1.15 veces
Proporción teórica: 23.16 veces

Figure 4: Probando los resultados para varias instancias del problema

Como se puede observar, esta acotación para casos especiales acota mucho más la respuesta que la vista anteriormente. La diferencia puede verse claramente en "Proporción Teórica"

4.5 Complejidad Temporal

La complejidad temporal del algoritmo greedy puede analizarse de la siguiente manera:

- Inicialización de subconjuntos no cubiertos: tiene una complejidad de $O(|C|$

- Bucle Principal: El bucle se ejecuta como máximo k veces, ya que el tamaño del conjunto de golpeo ('hitting_set') puede crecer hasta k elementos.
  - Selección del mejor elemento('for element in X'): Para cada elemento en X (lo que ocurre $|X|$ veces), se cuenta cuántos subconjuntos no cubiertos contiene ese elemento.
    * Este conteo involucra un segundo bucle sobre 'uncovered_subsets' (hasta $|C|$ subconjuntos).
    * Para cada subconjunto no cubierto, se verifica si el elemento está en ese subconjunto. Esto tiene una complejidad de $O(1)$ porque se trata de una verificación en un conjunto.

    Entonces, la complejidad del bucle interno es $O(|X| \cdot |C|)$ por iteración del bucle externo.
  - La actualización de 'uncovered_subsets' implica construir un nuevo conjunto filtrando aquellos subconjuntos que ya están cubiertos por el elemento recién añadido. Esto tiene una complejidad de $O(|C|)$ por iteración del bucle externo.

El bucle principal puede ejecutarse como máximo k veces, por lo que la complejidad total es:

$$O(k \cdot |X| \cdot |C|)$$

5 Solución plasmada en código

5.1 Resolviendo NPC fuerza bruta

```
from itertools import combinations

def is_valid_hitting_set(hitting_set, collection):
    """
    Check if hitting_set intersects with every set in the collection.
    """
    for subset in collection:
        if not (hitting_set & subset):  # Check if hitting_set intersects with subset
            return False
    return True

def find_k_hitting_set(X, C, k):
    """
    Find the smallest hitting set of size  k that hits every subset in C.

    :param X: The ground set (list or set)
    :param C: The collection of subsets (list of sets)
    :param k: The maximum size of the hitting set
    :return: The smallest hitting set of size  k or None if no such set exists
    """
    X = set(X)  # Ensure X is a set

    # Generate all combinations of elements from X with size  k
    for size in range(1, k + 1):
        for combination in combinations(X, size):
            hitting_set = set(combination)
            if is_valid_hitting_set(hitting_set, C):
                return hitting_set  # Return the first valid hitting set found

    return None  # No valid hitting set found

# Example usage
X = {1, 2, 3, 4, 5}
```

```
C = [{1, 2}, {2, 3}, {3, 4}, {4, 5}]
k = 3

hitting_set = find_k_hitting_set(X, C, k)
if hitting_set:
    print(f"Found hitting set: {hitting_set}")
else:
    print("No hitting set found.")
```

5.2 Solución Greedy

```
def greedy_k_hitting_set(X, C, k):
    """
    Find a k-sized hitting set using a greedy approach.

    :param X: The ground set (list or set)
    :param C: The collection of subsets (list of sets)
    :param k: The maximum size of the hitting set
    :return: A k-sized hitting set or None if no such set exists
    """
    X = set(X)  # Ensure X is a set
    uncovered_subsets = set(range(len(C)))  # Track uncovered subsets by index
    hitting_set = set()

    while len(hitting_set) < k and uncovered_subsets:
        best_element = None
        best_cover = 0

        # Find the element that covers the most uncovered subsets
        for element in X:
            cover_count = 0
            for idx in uncovered_subsets:
                if element in C[idx]:
                    cover_count += 1

            if cover_count > best_cover:
                best_cover = cover_count
                best_element = element

        if best_element is None:  # No element can cover any remaining uncovered subset
            break

        # Add the best element to the hitting set
        hitting_set.add(best_element)

        # Update the list of uncovered subsets
        uncovered_subsets = {idx for idx in uncovered_subsets if best_element not in C[idx]}

    # If the hitting set size is less than k but all subsets are covered, return the hitting set
    if len(hitting_set) <= k and not uncovered_subsets:
        return hitting_set
    else:
        return None  # No valid hitting set of size  k found
```

```
# Example usage
X = {1, 2, 3, 4, 5}
C = [{1, 2}, {2, 3}, {3, 4}, {4, 5}]
k = 3

hitting_set = greedy_k_hitting_set(X, C, k)
if hitting_set:
    print(f"Found k-sized hitting set: {hitting_set}")
else:
    print("No valid k-sized hitting set found.")
```