Set Cover Problem

Leonardo Javier Ramirez Calatayud

September 2024

Contents

1 Definición del problema:

El Set Cover Problem se define formalmente como sigue:

- **Entrada**: Un conjunto universal $U = \{u_1, u_2, \ldots, u_n\}$, y una colección de conjuntos $S = \{S_1, S_2, \ldots, S_m\}$ donde $S_i \subseteq U$.
- **Objetivo**: Encontrar un subconjunto $C \subseteq S$ tal que:

$$\bigcup_{S_i \in C} S_i = U$$

  minimizando el tamaño de C.
- **Problema de decisión**: ¿Existe un conjunto C tal que $|C| \leq k$ y cubra U?

2 Reducción del Vertex Cover al Set Cover

Para demostrar que el SCP es NP-completo, haremos una reducción desde el *Vertex Cover Problem* (VC), problema NP-completo estudiado en clase.

2.1 Descripción del Vertex Cover Problem

El *Vertex Cover Problem* se define como sigue:

- **Entrada**: Un grafo no dirigido $G = (V, E)$, donde V es el conjunto de vértices y E es el conjunto de aristas.
- **Objetivo**: Encontrar un subconjunto de vértices $C \subseteq V$ tal que cada arista $e \in E$ esté cubierta por al menos un vértice en C, minimizando $|C|$.
- **Problema de decisión**: ¿Existe un conjunto de vértices C tal que $|C| \leq k$ y cubra todas las aristas de G?

2.2 Reducción de VC a SCP

Para reducir el problema de *Vertex Cover* a *Set Cover*, seguimos los siguientes pasos:

1. Dado un grafo $G = (V, E)$ para el problema de VC, construimos una instancia del SCP.
2. Definimos el conjunto universal U como el conjunto de aristas de G, es decir, $U = E$.
3. Para cada vértice $v_i \in V$, definimos un conjunto S_i en el SCP, tal que S_i contiene todas las aristas incidentes en v_i. Formalmente, $S_i = \{e \in E \mid v_i \text{ es un extremo de } e\}$.

4. El objetivo en el SCP será encontrar el conjunto más pequeño de subconjuntos S_i cuya unión cubra todas las aristas en U. Esto corresponde directamente a encontrar un *vertex cover* en G, ya que elegir un conjunto S_i es equivalente a seleccionar el vértice v_i en la solución de VC.

De este modo, una solución al problema de *Set Cover* proporciona una solución al problema de *Vertex Cover*. Si podemos resolver el SCP en tiempo polinomial, entonces también podemos resolver el VC en tiempo polinomial.

2.3 Ejemplo

Consideremos un grafo simple $G = (V, E)$ con $V = \{v_1, v_2, v_3\}$ y $E = \{e_1 = (v_1, v_2), e_2 = (v_2, v_3)\}$. La instancia correspondiente del SCP sería:

- Conjunto universal: $U = \{e_1, e_2\}$.
- Colección de subconjuntos:

$$S_1 = \{e_1\}, \quad S_2 = \{e_1, e_2\}, \quad S_3 = \{e_2\}.$$

La solución óptima sería elegir los subconjuntos S_1 y S_3, lo que corresponde a elegir los vértices v_1 y v_3, que es una *vertex cover* válida.

3 SCP es NP-Completo

Para demostrar que el SCP es NP-completo, necesitamos cumplir dos condiciones:

1. SCP está en NP: Dada una solución propuesta (un conjunto C), podemos verificar en tiempo polinomial si cubre U y si $|C| \leq k$.
2. SCP es NP-duro: Esto se cumple ya que hemos mostrado una reducción polinomial desde el *Vertex Cover Problem*, que es NP-completo, al SCP.

Dado que ambos criterios se cumplen, podemos concluir que el SCP es NP-completo.

4 Solución Básica con Backtracking

El enfoque básico para resolver el Set Cover Problem es mediante el uso de un algoritmo de **backtracking**, que prueba todas las combinaciones posibles de subconjuntos para cubrir el conjunto universal U. Este enfoque garantiza encontrar la solución óptima, pero su complejidad es exponencial debido a la exploración exhaustiva de todas las posibles combinaciones de subconjuntos.

4.1 Descripción del Algoritmo

Dado un conjunto U y una colección de subconjuntos $S = \{S_1, S_2, ..., S_m\}$, el algoritmo intenta cubrir U probando todas las combinaciones de subconjuntos. En cada paso, el algoritmo selecciona o no un subconjunto S_i, y continúa explorando el espacio de soluciones recursivamente.

- **Entrada**: Un conjunto universal U y una colección de subconjuntos S.
- **Objetivo**: Encontrar el subconjunto de S cuya unión cubra U minimizando el número de subconjuntos seleccionados.

El algoritmo se basa en:

- Elegir un subconjunto S_i y marcar los elementos cubiertos.
- Llamar recursivamente para cubrir los elementos restantes.
- Si una solución cubre U, se guarda como posible solución.
- Volver atrás para explorar otras combinaciones.

4.2 Complejidad Temporal

El algoritmo tiene una complejidad temporal de $O(2^m)$, donde m es el número de subconjuntos en S. Esto se debe a que el algoritmo explora todas las combinaciones posibles de subconjuntos, lo cual es poco práctico para entradas grandes.

5 Atacando el problema de manera inteligente: Algoritmo Greedy

Para atacar el problema vamos a tener en cuenta una generalización del mismo donde se agregan pesos a los subconjuntos. En la versión previa dichos pesos son uniforme para todos los subconjuntos dados.
El algoritmo greedy sigue una estrategia iterativa en la que en cada paso se selecciona el subconjunto que cubre la mayor cantidad de elementos descubiertos al menor costo por elemento. Este proceso se repite hasta que todos los elementos del conjunto universal U están cubiertos.

5.1 Pasos del Algoritmo

El algoritmo greedy para la cobertura de conjuntos con costo mínimo funciona de la siguiente manera:

1. Inicialmente, todos los elementos de U están descubiertos.

2. En cada iteración, para cada conjunto S_i que no ha sido seleccionado, se calcula el **costo por elemento cubierto**:

$$\text{Costo por elemento} = \frac{c(S_i)}{|S_i \cap U_{\text{descubiertos}}|}$$

c(S_i) : *Costo del subconjunto* S_i

3. Seleccionamos el conjunto S_i que minimiza este costo por elemento.

4. Marcamos los elementos cubiertos por S_i como cubiertos y actualizamos el conjunto de elementos descubiertos.

5. Repetimos los pasos anteriores hasta que todos los elementos de U estén cubiertos.

6 Factor de Aproximación

El algoritmo greedy para el problema de cobertura de conjuntos logra un factor de aproximación acotado por el número armónico H_k, que se define como:

$$H_k = \sum_{i=1}^{k} \frac{1}{i} \to 0.5 + \log k$$

Donde k es el valor del subconjunto de mayor cardinalidad.

Para demostrar lo planteado anteriormente primero es necesario plantear el siguiente lema:

Lema: Sea $S_i \in S$, y sea $u_1, ..., u_l$ los elementos de S_i en el orden en el que fueron escogidos por el algoritmo. El precio de $u_j \leq c(S_i)/(l - j + 1)$

Demostración: En la iteración donde el algoritmo escoge a u_j se cumple:

- $\leq j - 1$ elementos de S_i ya fueron escogidos
- $\geq l - j + 1$ elementos de S_i no han sido escogidos
- El costo por elemento de S_i es $\leq c(S_i)/l - j + 1$
- El precio no puede ser mayor debido a la elección Greedy

De donde tenemos que:
$\forall S_i \in S, precio(S_i) = \sum^{l}_{j=1} precio(u_j) \leq c(S) * H_l$

Luego, sea $\{S_1, ..., S_m\}$ la solución óptima con $OPT = \sum^{m}_{i=1} c(S_i)$

El costo de la solución devuelta por el algoritmo greedy será:

$$precio(U) = \sum_{u \in U} precio(u) \leq \sum_{i=1}^{m} precio(S_i)$$

$$\leq \sum_{i=1}^{m} c(S_i) * H_k = OPT * H_k$$

Este factor garantiza que la solución obtenida por el algoritmo greedy no será peor que H_k veces el costo de la solución óptima. En particular, si el costo óptimo es OPT, entonces el costo del algoritmo greedy estará acotado por:

$$\text{Costo greedy} \leq H_k \cdot OPT$$

El número armónico crece logarítmicamente, por lo que la aproximación tiene un rendimiento aceptable en la práctica.

7 Pseudocódigo:

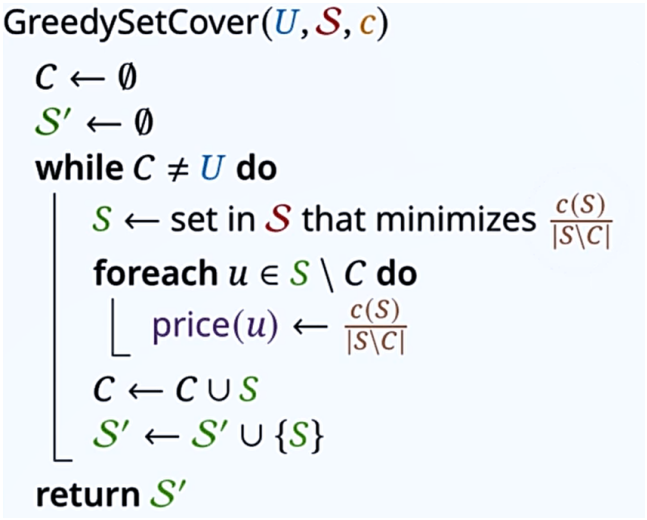

Figure 1: Pseudocódigo

Donde U es el conjunto universo, S es la lista de los subconjuntos y c son los costos de los subconjuntos.

8 Código del algoritmo Greedy

A continuación, se presenta una implementación en Python del algoritmo Greedy:

8.1 Descripción del Algoritmo

El algoritmo greedy sigue los siguientes pasos:

1. Inicialmente, todos los elementos del conjunto universal U están descubiertos.
2. En cada iteración, selecciona el subconjunto $S_i \in S$ que cubra la mayor cantidad de elementos no cubiertos al menor costo por elemento.
3. Marca como cubiertos los elementos de U cubiertos por S_i.
4. Repite el proceso hasta que todos los elementos de U estén cubiertos.

8.2 Complejidad Temporal

La complejidad temporal del algoritmo `greedy_set_cover` es $O(m \cdot n^2)$, donde m es el número de subconjuntos en S y n es el tamaño del conjunto universal U. Esto se debe a que en cada iteración se evalúan todos los subconjuntos (m), y para cada subconjunto se realiza una intersección con el conjunto de elementos no cubiertos ($O(n)$). El ciclo principal se repite como máximo $O(n)$ veces, lo que lleva a la complejidad total de $O(m \cdot n^2)$.

9 Anexos

9.1 Código backtrack

```python
from itertools import combinations

# Funciones del problema (cubre_U y set_cover)

def cubre_U(U, subconjuntos_seleccionados):
    union_subconjuntos = set()
    for subconjunto in subconjuntos_seleccionados:
        union_subconjuntos.update(subconjunto)
    return union_subconjuntos == U

def set_cover(U, S):
    n = len(S)
    mejor_cobertura = None

    for r in range(1, n + 1):
        for combinacion in combinations(S, r):
            if cubre_U(U, combinacion):
                if mejor_cobertura is None or len(combinacion) <
                    len(mejor_cobertura):
                    mejor_cobertura = combinacion
        if mejor_cobertura is not None:
            break

    return mejor_cobertura
```

Listing 1: Código backtrack

9.2 Código greedy

```python
def greedy_set_cover(U, S, costos):
    # Inicialmente, todos los elementos de U est n descubiertos
    descubiertos = set(U)
    cobertura = []

    while descubiertos:
        mejor_subconjunto = None
        mejor_costo_por_elemento = float('inf')

        # Iterar sobre los subconjuntos que a n no han sido seleccionados
        for i, subconjunto in enumerate(S):
            # Calcular los elementos descubiertos cubiertos por el subconjunto actual
            elementos_cubiertos = descubiertos.intersection(subconjunto)

            if elementos_cubiertos:
                # Calcular el costo por elemento cubierto
                costo_por_elemento = costos[i] / len(elementos_cubiertos)

                # Elegir el subconjunto que minimiza el costo por elemento cubierto
                if costo_por_elemento < mejor_costo_por_elemento:
                    mejor_costo_por_elemento = costo_por_elemento
                    mejor_subconjunto = i

        # A adir el subconjunto elegido a la cobertura
        if mejor_subconjunto is not None:
            cobertura.append(S[mejor_subconjunto])
            # Actualizar los elementos descubiertos
            descubiertos -= S[mejor_subconjunto]
        else:
            break

    # Devolver la cobertura si todos los elementos fueron cubiertos
    if not descubiertos:
        return cobertura
    else:
        return None
```

Listing 2: Código greedy

References

[1] Richard Karp. *Reducibility Among Combinatorial Problems*. In Complexity of Computer Computations, 1972.