Algoritmos aproximados

Alex y Javi

25 de diciembre de 2021

Índice

1. Introducción

Los problemas $\mathcal{NP}$-difíciles aparecen a menudo en la realidad. Por ello, no debemos renunciar a resolverlos, aunque aun no se hayan descubierto algoritmos polinómicos. Se puede emplear algunas de las siguientes estrategias:

1. Resolver instancias pequeñas. Un algoritmo exponencial puede ser útil si la entrada es pequeña.
2. Aplicar potentes algoritmos de ramificación y poda, así como de vuelta atrás. Para algunos problemas como el $SAT - FNC$ se conocen algoritmos que pueden tratar entradas de tamaño considerable.
3. Restringir el problema general. Para el problema $2-SAT$ existen algoritmos polinómicos que lo resuelven.
4. Utilizar algoritmos de aproximación. Se basan en obtener soluciones próximas a la óptima. Sin embargo, no para todos los problemas $\mathcal{NP}$-difíciles podemos obtener un algoritmo aproximado.

Los algoritmos aproximados se ejecutan en tiempo polinómico, por lo que son eficientes. Además es posible demostrar que su solución dista de la óptima como mucho en una constante o factor que depende de la entrada

Se definen dos tipos de algoritmos aproximados según lo que diste su solución C de la óptima del problema C^* :

- **Algoritmo aproximado absoluto**: Para cualquier entrada E del problema se tiene que

  $$|C - C^*| \leq K$$

  para algún K constante.

- **Algoritmo $\rho(n)$ - aproximado**: Para cualquier entrada E de tamaño n, la solución C dista de la óptima en proporción a $\rho(n)$.

  $$\text{máx}(\frac{C}{C^*}, \frac{C^*}{C}) \leq \rho(n)$$

  La diferencia con los algoritmos absolutos es que en este caso la distancia de la solución aproximada depende del tamaño de la entrada.

Se desprende de ambas definiciones que estos algoritmos son mejores cuanto menor sea el radio de aproximación. Por lo tanto, podemos establecer una clasificación de los problemas $\mathcal{NP}$ - difíciles:

1. **APX**: Tienen algoritmos ρ-aproximados. Para algunos de ellos se han encontrado límites inferiores de ρ.
2. **PTAS**: algoritmos aproximados cuya solución está arbitrariamente cerca de la óptima. Es decir, dado $\epsilon > 0$, devuelve una solución que dista de la óptima $1 + \epsilon$. Sin embargo, su coste puede estar en el orden exponencial de $(\frac{1}{\epsilon})$. Dentro de este grupo se encuentran los **FPTAS** (FULLY POLINOMIAL TIME APROXIMATION SCHEMES), cuyo coste es polinómico en n y $\frac{1}{\epsilon}$, por lo que son mas eficientes.
3. Elaborar un algoritmo aproximado es igual de complicado que el propio problema.

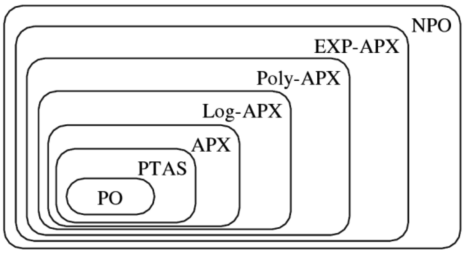


Figura 1: Relación de problemas según sus algoritmos aproximados [5]

2. Problemas con algoritmos ρ - aproximados (APX)

En esta sección vamos a ver varios ejemplos de algoritmos cuya aproximación es eficaz. Además, se ha probado que el radio de aproximación en estos casos tiene una cota inferior. De hecho, existen muy pocos problemas $\mathcal{NP}$ - difíciles para los que se conoce un algoritmo aproximado absoluto.

2.1. Problema del viajante euclídeo

Dado un grafo $\mathcal{G}$ no dirigido,completo y valorado, el problema consiste en encontrar un ciclo hamiltoniano de coste mínimo. Recordamos que si $\mathcal{G}$ verificaba la desigualdad triangular

$$\mathcal{G}[u,w] \leq \mathcal{G}[u,v] + \mathcal{G}[v,w]$$

ya habíamos estudiado un algoritmo polinómico 2- aproximado. Este transformaba un ARM mediante su recorrido en preorden en un ciclo hamiltoniano. Además aprovechaba la desigualdad triangular para obtener radio de aproximación 2. [3][4]
De hecho, existen algoritmos aun más precisos, como el que vamos a mostrar a continuación, de radio 1.5
Pasos:

1. Hallar ARM, cuyo coste será una cota inferior de la solución óptima (si eliminamos cualquier arista del ciclo hamiltoniano tenemos un AR).

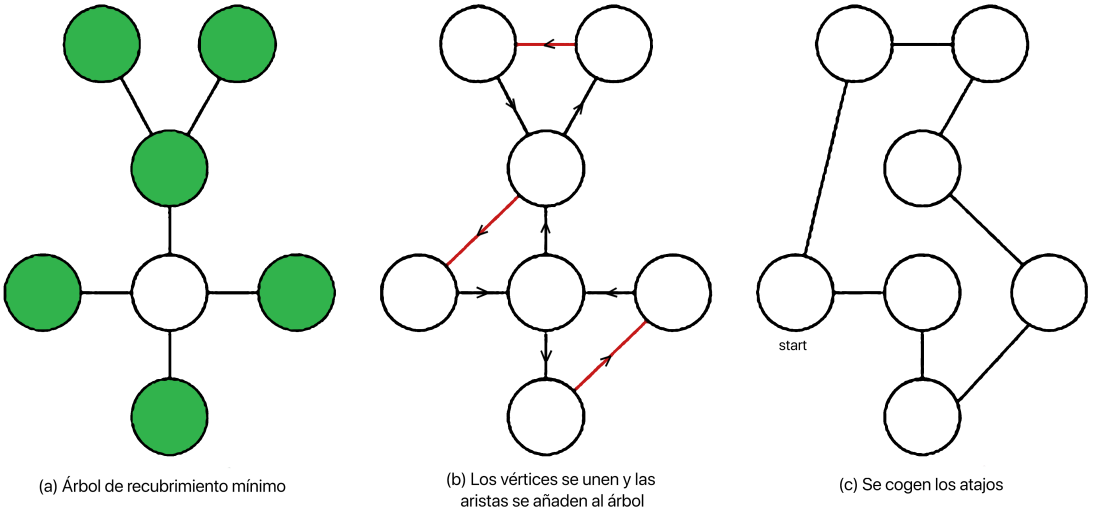
2. Se denota V' al conjunto de vértices del ARM de los que salen un número impar de aristas. Siempre van a verificarlo un numero par de ellos.[1] Se emparejan los vértices de V', de forma que se minimiza la suma de los pesos de las aristas que los unen.

3. Añadimos las aristas que unen los emparejamientos recientemente creados (puede ser que esa arista ya estuviera en el ARM). Para este nuevo subgrafo, como todos los vértices tienen grado par, existe un ciclo euleriano (recorre todas las aristas una única vez).

4. Para convertir el ciclo euleriano en otro hamiltoniano, operamos igual que en algoritmo 2-aproximado: utilizamos la desigualdad triangular para evitar pasar dos veces por el mismo vértice.

Hallar el ARM mediante el algoritmo de Prim es del $O(E \cdot \ln |V|)$ y calcular el emparejamiento de V' se puede hacer en $O(|V|^2 \cdot E)$ (Edmonds' blossom algorithm). Además, hallar el ciclo euleriano y recorrerlo se puede conseguir en $O(|V|^2)$ Por lo tanto este algoritmo se puede hacer en tiempo polinómico. En cuanto a la calidad de la aproximación se tiene lo siguiente:

[1] Se puede comprobar por inducción viendo que sucede al añadir una arista.



(a) Árbol de recubrimiento mínimo

(b) Los vértices se unen y las aristas se añaden al árbol

(c) Se cogen los atajos

Figura 2: Pasos algoritmo 1.5-aproximado con V'

Teorema. *Sea C^* el coste de la solución óptima y sea C el de la obtenida mediante este algoritmo. Entonces se tiene que*

$$C \leq 1.5 \cdot C^*$$

Demostración. Sea C_{E^*} la suma de los costes de las aristas añadidas en el paso 3, tenemos que comprobar que $C_{E^*} \leq \frac{1}{2} \cdot C^*$.

Para ello, recorremos el ciclo hamiltoniano óptimo, de forma que cada vez que aparece $v \notin V'$, se elimina gracias a la desigualdad triangular. El resultado será un ciclo que solo recorre los vértices de V', cuyo coste es C_E. De este ciclo podemos extraer dos emparejamientos diferentes E_1 Y E_2, cuyos costes son C_{E_1} y C_{E_2}. Como el número de elementos de V' es par, empezamos en un vértice arbitrario y ambos emparejamientos vienen definidos según si lo unimos con el de la derecha o el de la izquierda.

Como en el algoritmo hallamos el emparejamiento mínimo de V',de coste C_{E^*}, se tiene que

$$C_{E^*} \leq \text{mín}(C_{E_1}, C_{E_2}) \leq \frac{1}{2} \cdot C^*$$

El ciclo euleriano tiene coste $C_M + C_{E^*}$ donde C_M es el coste del ARM. Para obtener el ciclo hamiltoniano, eliminamos algunas aristas de este ciclo. Gracias a la desigualdad triangular y a que $C_M \leq C^*$, se tiene que

$$C \leq C_M + C_{E^*} \leq 1.5 \cdot C^*$$

□

En cuanto a la cota inferior del radio de aproximación, se ha demostrado que el problema del viajante euclídeo no se puede 1,013-aproximar (si $\mathcal{P} \neq \mathcal{NP}$)

2.2. Problema de los envases

Dados n objetos, cada uno con un volumen, se pretende empaquetarlos en el menor número posibles de envases, teniendo todos los envases la misma capacidad E.

Vamos a ver que existe un algoritmo voraz aproximado con un ratio 1,5 de aproximación, pero para tratar este problema, vamos a considerar que la capacidad de los envases $E = 1$ y los tamaños de los objetos como fracciones de la unidad. También necesitamos que los objetos estén ordenados de forma decreciente.

Algoritmo 1: Algoritmo envases [3]

```
   Entrada: s[1...n] : real
   Salida: num_of_bins: nat
   Datos: ∀ i, j : 1 ≤ i ≤ j ≤ n : s[i] ≥ s[j]
 1 bin = [ ];
 2 i = 0;
 3 mientras i ≤ n hacer
 4 |   packed = false;
 5 |   b = 0;
 6 |   mientras b < bin.size ∧ !packed hacer
 7 |   |   size = bin[b] + s[i];
 8 |   |   si size ≤ 1.0 entonces
 9 |   |   |   bin[b] := size;
10 |   |   |   packed = true;
11 |   |   sinó
12 |   |   |   b = b + 1;
13 |   |   fin
14 |   |   si !packed entonces
15 |   |   |   bin[b] = s[i];
16 |   |   fin
17 |   fin
18 |   i = 4 + 1;
19 fin
20 devolver bin.size
```

El coste peor de este algoritmo está en $O(n^2)$. La variable bin lleva los envases ocupados hasta el momento y su tasa de ocupación.

Sean A y O el algoritmo aproximado y el número de objetos que obtiene, y sean A^* y O^* el algoritmo y el número de envases óptimo. Y a cada envase utilizado de más en O lo llamaremos envase adicional.

Lema. *Si j es el primer objeto envasado por A en un envase adicional, entonces $s_j \leq \frac{1}{3}$*

Demostración. Supongamos por reducción al absurdo que $s_j > \frac{1}{3}$

1. Por estar ordenados, $s_1, \ldots, s_{j-1} > \frac{1}{3}$ luego los envases de A numerados del 1 al O^* tendrán como mucho 2 objetos.

2. Si todos los envases de A del 1 al O^* tuvieran 2 objetos de $\{s_1, \ldots, s_{j-1}\}$ en A^* también lo tendrían (quizás junto con otros objetos más pequeños) ya que 3 no caben. Entonces no habría espacio para el objeto j, lo que contradice que A^* utilice solo O^* envases.

3. En A entonces habrá al menos un envase con 1 objeto. Si todos los envases en A tuvieran 1 solo objeto de $\{1, \ldots, j-1\}$, entonces no sería posible envasar 2 objetos juntos de ese conjunto (voraz). Además, el objeto j no cabe en ninguno de ellos ya que A lo envasa en $O^* + 1$. Y en A^* tampoco cabría en los envases del 1 al O^*, lo que contradice que A^* utilice solo O^* envases.

4. Por lo tanto en A habrá envases con 2 y con 1 objetos. Vamos a ver que A deja a la izquierda todos los envases con un objeto y con 2 a continuación. (Obs: no es posible que haya envase con 1 objeto después de uno con 2 -dibujo-).

   $$s_a \geq s_b \geq s_c \geq s_j \Rightarrow s_a + s_b \geq s_c + s_j$$

   Así que j habría cabido junto con s_c en el mismo envase, luego no habría hecho falta un nuevo envase para él.

5. Tenemos entonces: del envase 1 al k hay 1 objeto por envase, y no cabe ninguno de los siguientes; del envase $k+1$ al $j-1$ los objetos se envasan de 2 en 2 ocupando los envases $k+1$ hasta O^*; el objeto j no cabe en ninguno de ellos. Entonces cualquier solución al problema necesitaría más de O^* envases, lo que contradice que A^* utilice solo O^* envases.

Por tanto, $s_j \leq \frac{1}{3}$ □

Teorema. *El algoritmo es 1,5-aproximado.*

Demostración.

1. Probemos que el número de objetos en envases adicionales de A es como mucho $O^* - 1$.

   Supongamos lo contrario, y observemos los O^* primeros objetos envasados de ese modo. Sean $s'_1, \ldots, s'_{O^*}$ y sean las ocupaciones de los O^* primeros $o_1, \ldots, o_{O^*}$. En esos envases A no puede ubicar los O^* objetos que ha ubicado en los envases adicionales. Entonces,

   $$\sum_{i=1}^{n} s_i \geq \sum_{i=1}^{O^*} o_i + \sum_{i=1}^{O^*} s'_i = \sum_{i=1}^{O^*} (o_i + s'_i) > \sum_{i=1}^{O^*} 1 = O^*$$

   lo que contradice que $\sum_{i=1}^{n} s_i \leq O^*$, ya que A^* ha conseguido envasar todos los objetos en O^* envases l. Por tanto A envasa a lo sumo $O^* - 1$ objetos en envases adicionales.

2. Esto junto al lema anterior, A envasa a lo sumo $O^* - 1$ objetos de tamaño a lo sumo $\frac{1}{3}$ en envases adicionales. Por lo tanto

   $$O = O^* + \left\lceil \frac{O^* - 1}{3} \right\rceil \leq O^* + \frac{O^* - 1 + 2}{3} = \frac{4}{3}O^* + \frac{1}{3} \quad \Rightarrow \quad \frac{O}{O^*} \leq \frac{4}{3} + \frac{1}{3O^*}.$$

   El valor más bajo posible de O^* es 2 ya que $O^* = 1$ implicaría que todos los objetos cabrían en 1 envase y esa solución la encuentra A. Entonces, con $O^* = 2$ tenemos:

   $$\frac{O}{O^*} \leq \frac{4}{3} + \frac{1}{3 \cdot 2} = \frac{3}{2} = 1,5 \quad \Rightarrow \quad O \leq 1,5 \cdot O^*.$$

□

2.3. Problema de 3-SAT-MAX

En este problema lo que queremos conseguir es maximizar el número de cláusulas posibles del problema SAT, es decir, calcular cuán cerca nos hemos quedado de la satisfactibilidad, y consideramos cláusulas formadas por 3 literales (distintos y no uno y su negado).

Vamos a ver cómo podemos conseguir un algoritmo aproximado de ratio $\frac{8}{7}$ utilizando un algoritmo probabilista, asignando a cada variable 1 con probabilidad 1/2 y 0 con probabilidad 1/2.

Teorema. *Dada una instancia del problema 3-SAT-MAX con n variables $x_1, \ldots, x_n$ y m cláusulas, el algoritmo que asigna a cada variable 1 con probabilidad 1/2 y 0 con probabilidad 1/2 es un algoritmo probabilista $\frac{8}{7}$- aproximado. [1]*

Demostración.

1. Supongamos asignadas independientemente a cada variable 1 o 0 con probabilidad 1/2. Para $i = 1, \ldots, n$ tenemos la variable

$$Y_i = \begin{cases} 1 & \text{si la cláusula } i \text{ se satisface} \\ 0 & \text{en caso contrario} \end{cases}.$$

2. Como un literal no aparece más de una vez en una misma cláusula, ni tampoco su negado, las configuraciones de cada literal en las cláusulas son independientes entre sí.

3. Por lo tanto, tenemos que una cláusula no es satisfactible si y solo si sus tres variables son 0, es decir, $\Pr\{\text{cláusula } i \text{ no es satisfactible}\} = (\frac{1}{2}^3) = \frac{1}{8}$, luego $\Pr\{\text{cláusula } i \text{ es satisfactible}\} = 1 - \frac{1}{8} = \frac{7}{8}$, y por cómo hemos definido Y_i, tenemos que $E[Y_i] = \frac{7}{8}$.

4. Sea Y el número total de cláusulas que son satisfactibles, $Y = Y_1 + Y_2 + \cdots + Y_m$, por lo tanto tenemos que:

$$E[Y] = E\left[\sum_{i=1}^{m} Y_i\right] = \sum_{i=1}^{m} E[Y_i] = \sum_{i=1}^{m} \frac{7}{8} = \frac{7m}{8}.$$

5. Como m es una cota superior de el número de cláusulas que satisfacen el problema, tenemos que el ratio de aproximación es como mucho $\frac{m}{\frac{7m}{8}} = \frac{8}{7}$.

□

2.4. Problema de la cobertura de vértices con pesos

En el problema de la cobertura de la cobertura de vértices con pesos nos dan un grafo no dirigido $G = (V, E)$ en el que cada vértices $v \in V$ tiene asociado un peso positivo $w(v)$. Para cualquier cobertura de vértices $V' \subseteq V$, se define el peso de la cobertura como $w(V') = \sum_{v \in V'} w(v)$. Queremos encontrar la cobertura de mínimo peso.

El propio problema de la cobertura de vértices no se puede aproximar con un radio inferior a 1,1666, con lo cual este tampoco se va a poder. Pero vamos uno 2-aproximado utilizando un método distinto a los que hemos visto: la programación lineal.

Vamos a asociar la variable $x(v)$ a cada vértice en $v \in V$, donde $x(v)$ vale o 1 o 0, y $x(v) = 1$ si y solo si ponemos v en la cobertura. Así, para cada arista (u, v), tendrá alguno de sus vértices en la cobertura, y esto es $x(u) + x(v) \geq 1$. Gracias a esto podemos establecer el problema de programación lineal binario para encontrar la cobertura mínima como:

$$\begin{aligned} min \quad & \textstyle\sum_{v \in V} w(v)x(v) & \\ s.a.: \quad & x(u) + x(v) \geq 1 & \text{para cada } (u, v) \in E \\ & x(v) \in \{0, 1\} & \text{para cada } v \in V \end{aligned}$$

El caso particular donde todos los $w(v) = 1$, nos queda el problema de optimización NP-difícil de la cobertura de vértices.

Si cambiamos la condición $x(v) \in \{0, 1\}$ por $0 \leq x(v) \leq 1$, tenemos la relajación del problema:

$$\begin{aligned} min \quad & \textstyle\sum_{v \in V} w(v)x(v) & \\ s.a.: \quad & x(u) + x(v) \geq 1 & \text{para cada } (u, v) \in E \\ & x(v) \leq 1 & \text{para cada } v \in V \\ & x(v) \geq 0 & \text{para cada } v \in V \end{aligned}$$

Cualquier solución del problema de programación lineal binario es también solución del problema relajado. Entonces, el valor de una solución óptima del relajado es una cota inferior del valor

de la solución óptima del problema de programación lineal 0-1, y por lo tanto una cota inferior del problema de minimización de la cobertura de vértices con pesos.

Algoritmo 2: Algoritmo cobertura de vértices con pesos [1]

Entrada: G grafo, w función pesos
Salida: $\mathcal{C}$ cobertura mínima

```
C = ∅;
computar x̄ solución óptima del problema relajado;
para v ∈ V hacer
    si x̄ ≥ 1/2 entonces
        C = C ∪ {v};
    fin
fin
devolver C
```

Teorema. *El algoritmo es un algoritmo polinómico 2-aproximado.*

Demostración. 1. Es polinómico porque el problema de programación lineal se puede resolver en tiempo polinómico, y el for recorre los vértices del grafo, luego es polinómico.

2. Sea C^* una cobertura óptima y sea z^* el valor óptimo del problema relajado. Como una cobertura óptima es una posible solución al problema de programación lineal, $z^* \leq w(C^*)$.

3. Al redondear los valores de las variables $\bar{x}(v)$, vamos a ver que el conjunto C que escogemos es una cobertura de vértices y satisface $w(C) \leq 2z^*$.

   a) Para ver que es cobertura, dad $(u, v) \in E$ se cumple que $x(u) + x(v) \geq 1$, lo que implica que $x(u) \geq \frac{1}{2}$ o $x(v) \geq \frac{1}{2}$. Así, al menos uno de los dos vertices está en la cobertura y toda arista está cubierta.

   b) Ahora, consideramos el peso de la cobertura:

$$\begin{aligned} z^* &= \sum_{v \in V} w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq \frac{1}{2}} w(v)\bar{x}(v) \\ &\geq \sum_{v \in V : \bar{x}(v) \geq \frac{1}{2}} w(v)\frac{1}{2} \\ &= \sum_{v \in C} w(v)\frac{1}{2} \\ &= \frac{1}{2}\sum_{v \in C} w(v) \\ &= \frac{1}{2} \cdot w(C). \end{aligned}$$

   Por lo tanto, combinando lo obtenido:

$$w(C) \leq 2z^* \leq 2w(C^*).$$

□

2.5. Otros ejemplos

Para los siguientes problemas $\mathcal{NP}$ - difíciles también existen algoritmos aproximados absolutos:

1. **Steiner tree**: Dado un grafo no dirigido, valorado $\mathcal{G} =< V, E >$ y un subconjunto de vértices $\mathcal{S}$, encontrar el árbol de mínimo coste que recubra $\mathcal{S}$, permitiendo que estén en él cualquier vértice de V que no esté en $\mathcal{S}$. Al igual que el problema del viajante, admite diferentes variantes como la euclídea (grafo completo y se da la desigualdad triangular). Este problema tiene numerosas aplicaciones en circuitos eléctricos y redes de telecomunicaciones. Existe un algoritmo 2 - aproximado para el problema euclídeo.[2]

Figura 3: Grafos de ejemplo

2. **k-center**: Dado un grafo no dirigido, valorado $\mathcal{G} =< V, E >$ y un subconjunto de vértices $\mathcal{S}$ con $|S| = k$ tal que se minimiza la máxima distancia de un vértice v a $\mathcal{S}$. Este problema tiene algoritmo 2-aproximado si es euclídeo.[3]

[2] Calcular el ARM de $\mathcal{S}$
[3] Utiliza una poda compleja

3. Problemas con algoritmos $\rho(n)$ - aproximados ($\rho(n)$-APX)

Dentro de este tipos de problemas destaca el del recubrimiento, pues a partir de él se pueden obtener otros con aplicaciones en diferentes áreas.

3.1. Problema del recubrimiento

El problema del recubrimiento es un problema de optimización que consiste en dado un conjunto finito X y una familia $\mathcal{F}$ de subconjuntos de X que recubre X (esto es $X = \cup_{S\in\mathcal{F}} S$), encontrar el subconjunto mínimo $\mathcal{C} \subseteq \mathcal{F}$ que recubra todo X, es decir, $X = \cup_{S\in\mathcal{C}} S$. Veámoslo en una imagen:

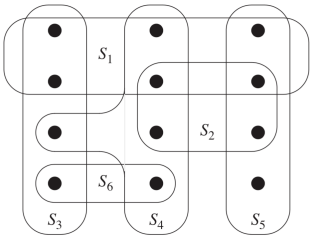


Figura 4: X son los 12 puntos, $\mathcal{F} = \{S_1, S_2, S_3, S_4, S_5, S_6\}$
y el recubrimiento mínimo es $\mathcal{C} = \{S_3, S_4, S_5\}$.

El problema de decisión asociado al de optimización es $\mathcal{NP}$-completo (también es $\mathcal{NP}$-difícil) luego el de optimización es $\mathcal{NP}$-difícil.

Un ejemplo práctico relacionado con este problema: supongamos que tenemos un conjunto de personas que van a trabajar en un problema que requiere de un conjunto de habilidades X. Queremos formar un equipo de la mínima cantidad de personas posibles tal que para cada habilidad de X, hay al menos una persona que tiene esa habilidad.

Una aproximación para este problema es el siguiente algoritmo voraz:

Algoritmo 3: GREEDY-SET-COVER ALGORITHM [1]

Entrada: X conjunto, $\mathcal{F}$ recubrimiento finito
Salida: $\mathcal{C}$ recubrimiento mínimo

```
U = X;
C = ∅;
mientras U ≠ ∅ hacer
    elegir S ∈ F que maximice |S ∩ U|;
    U = U − S;
    C = C ∪ {S}
fin
devolver C
```

Como el número de vueltas del bucle while está acotado por mín$(|X|, |\mathcal{F}|)$, este algoritmo es del orden de $O(|X||\mathcal{F}|\,\text{mín}(|X|, |\mathcal{F}|)$.

Teorema. *Este algoritmo es $\rho(n)$-aproximado donde $\rho(n) = H(\text{máx}\{|S| : S \in \mathcal{F}\}$. ($H_n = \Sigma_{k=1}^{n} \frac{1}{k} = \log n + O(1)$).*

Demostración. Vamos a asignar un coste de 1 a cada conjunto seleccionado por el algoritmo, y lo distribuimos entre sus elementos cubiertos la primera vez, para luego usar estos costes para obtener la relación entre la cobertura óptima C^* y la del algoritmo C.

Denotamos S_i el i-ésimo conjunto seleccionado por el algoritmo voraz; asignamos un coste de 1 añadir el conjunto S_i a C. Distribuimos este coste entre los elementos cubiertos la primera vez. Sea c_x el coste de cada elemento $x \in X$. A cada elemento se le asigna el coste sólo una vez, y este

coste es:

$$c_x = \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Con cada vuelta del algoritmo, conseguimos que $|C| = \sum_{x \in X} c_x$. Cada elemento $x \in X$ está en al menos un conjunto de C^* con lo que

$$\sum_{S \in C^*} \sum_{x \in S} c_x \geq \sum_{x \in X} c_x.$$

Combinando estas ecuaciones tenemos

$$|C^*| \leq \sum_{S \in C^*} \sum_{x \in S} c_x.$$

El resto de la demostración consiste en probar $\sum_{x \in S} c_x \leq H(|S|)$ para cualquier S de la familia $\mathcal{F}$, para así obtener

$$|C^*| \leq \sum_{S \in C^*} H(|S|) \leq |C^*| \cdot H(\text{máx}\{|S| : S \in \mathcal{F}\}).$$

Seas $S \in \mathcal{F}$ y sea $i = 1, 2, \ldots, |C|$, definimos $u_i(S) = |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_i|$, es decir, los elementos de S que quedan por cubrir en la i-esima iteración. Definimos $u_0(S) = |S|$ y sea k el momento donde $u_k(S) = 0$. Se cumple que $u_0(S) \geq u_1(S) \geq \cdot \geq u_k(S)$, así que $u_{i-1}(S) \geq u_i$ y $u_{i-1}(S) - u_i(S)$ son los elementos de S que se cubren por primera vez por S_i en la i-esima iteración. Así:

$$\sum_{x \in S} c_x = \sum_{i=1}^{k} (u_{i-1}(S) - u_i(S)) \cdot \frac{1}{|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})|}.$$

Se observa que $|S_i \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| \geq |S \setminus (S_1 \cup S_2 \cup \cdots \cup S_{i-1})| = u_{i-1}$ porque el algoritmo voraz escoge S_i en la i-ésima iteración, así que S no puede cubrir más elementos que S_i.

Así, nos queda

$$\sum_{x \in S} c_x \leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}}.$$

Por lo tanto tenemos lo siguiente:

$$\begin{aligned}
\sum_{x \in S} c_x &\leq \sum_{i=1}^{k} (u_{i-1} - u_i) \cdot \frac{1}{u_{i-1}} \\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{u_{i-1}} \\
&= \sum_{i=1}^{k} \sum_{j=u_i+1}^{u_{i-1}} \frac{1}{j} \qquad (j \leq u_{i-1}) \\
&= \sum_{i=1}^{k} \left(\sum_{j=1}^{u_{i-1}} \frac{1}{j} - \sum_{j=1}^{u_i} \frac{1}{j} \right) \\
&= \sum_{i=1}^{k} (H(u_{i-1}) - H(u_i)) \\
&= H(u_0) - H(u_k) \\
&= H(u_0) \\
&= H(|S|)
\end{aligned}$$

como queríamos ver. □

El problema del recubrimiento con costes tiene esta demostración para ver el coste:

- Obsérvese que $coste(C) = \sum_{x \in X} c_x$.
- Supongamos $x_1, x_2, \ldots, x_n$ el orden en el que el algoritmo escoge los elementos de X, así $coste(C) = \sum_{k=1}^{n} c_{x_k}$.
- Para ver $\sum_{k=1}^{n} c_{x_k} \leq H_n \cdot coste(C^*)$ es suficiente probar que para cada $k \in \{1, 2, \ldots, n\}$,

$$c_{x_k} \leq \frac{coste(C^*)}{n-k+1}$$

.

- Sea la iteración en la cual se cubre x_k. Al comienzo de la iteración, U contiene todos los elementos aún sin cubrir, que son al menos $n-k+1$. Ahora, la cobertura óptima de X cubre todo X, en particular cubre U, por lo tanto existe un conjunto tal que $\alpha \leq \frac{coste(C^*)}{|U|}$. Esto es porque, imagina que cogemos conjuntos del recubrimiento óptimo en lugar de minimizar α. Si mantenemos los costes de los elementos como siempre, entones 1 elemento $x_0 \in U$ debe tener $c_{x_0} \leq \frac{coste(C^*)}{|U|}$, ya que si no de otra forma la cuenta total del recubrimiento óptimo acabaría siendo $> coste(C^*)$, lo que es absurdo. Pero entonces el conjunto S que cubre x_0 es precisamente el que estamos buscando, ya que α solo puede incrementar en las iteraciones, ya que cada vez hay menos elementos disponibles sobre a los cuales distribuir su coste.
- Entonces, la existencia de un conjunto con $\alpha \leq \frac{coste(C^*)}{|U|}$ significa que cogemos el conjunto que minimiza α, el conjunto que seleccionamos debe tener $\alpha \leq \frac{coste(C^*)}{|U|}$. Como α es el coste c_{x_k}, tenemos $c_{x_k} \leq \frac{coste(C^*)}{|U|}$ y como $|U| \geq n-k+1$ obtenemos lo que queríamos.

En cuanto a la cota inferior del radio de aproximación de este problema, se ha demostrado que no se puede aproximar con radio $1+\epsilon$ para ningún ϵ (si $\mathcal{P} \neq \mathcal{NP}$)

3.2. Aplicación al shortest-superstring

Vamos a ver ahora cómo podemos relacionar el problema del set-cover con otro problemas interesante, el shortest-superstring. No es la mejor aproximación que se conoce, pero sí es interesante de ver.

Las aplicaciones del shortest-superstring son muchas, entre otras el análisis de la cadena de ADN para descifrar su información. La cadena de ADN humano se puede ver como una cadena larga formada por cuatro letras, $\Sigma = \{A, G, C, T\}$. Por lo general, se conocen subcadenas cortas, las cuales se superponen encima de otras al ser una cadena muy larga. Se piensa que la cadena más corta de ADN conteniendo todas estas subcadenas es una buena aproximación a la cadena de ADN original. [4]

El problema es el siguiente: dado un alfabeto finito Σ y un conjunto de n cadenas $S = \{s_1, \ldots, s_n\} \subset \Sigma^*$, encontrar la menor cadena $s \in \Sigma^*$ que contenga a cada s_i como subcadena para cada $i = 1, \ldots, n$. Sin perdida de generalidad se puede asumir que ninguna s_i es subcadena de otra s_j, con $i \neq j$.

Este problema es NP-difícil. Un algoritmo de aproximación voraz da un ratio de aproximación de 2, pero nosotros vamos a ver uno $2H_n$ basado en el algoritmo visto para el set-cover.

Dado una instancia $S \subset \Sigma^*$ del shortest-superstring, queremos construir la instancia (X, F) para el set-cover. En el shortest-superstring, el conjunto que queremos recubrir es S, así que tomaremos $X = S$. Ahora queremos encontrar F, así que para $\sigma \in \Sigma^*$ definimos $set(\sigma) = \{\tau \in S : \tau$es subcadena de$\sigma\}$. Como F tiene que ser finito no podemos elegir el conjunto de todos los $set(\sigma)$, y no podemos limitar los σ a S porque la concatenación de todas las cadenas de S sería una solución poco útil. Así que para cada par s_i, s_j, donde los k últimos caracteres de s_i son los primeros de s_j, definimos σ_{ijk} la superposición de las dos cadenas. Definimos I como el conjunto

de todos los σ_{ijk} válidos, es decir, las buenas superstring para los pares de cadenas en S. Entonces definimos $F = \{set(\sigma) : \sigma \in S \cup I\}$, y el coste asociado a cada $set(\sigma)$ es su longitud.

Entonces podemos ejecutar el algoritmo que consiste en:

1. Conseguimos la instancia (X, F) del set-cover.
2. Ejecutamos el algoritmo voraz del set-cover, que nos devuelve un conjunto de la forma $\{set(\sigma_1), \ldots, set(\sigma_k)\}$.
3. Devolvemos $s := \sigma_1 \cdot \sigma_2 \cdot \cdots \cdot \sigma_k$.

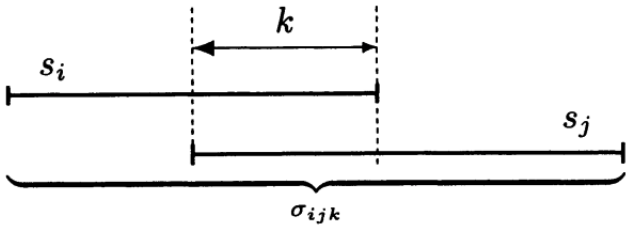


Figura 5: Definición de σ_{ijk}

4. Problemas PTAS (POLINOMIAL TIME APROXIMATION SCHEMES)

Estos problemas tienen algoritmos aproximados cuya solución puede estar arbitrariamente cerca de la óptima. Sin embargo, el coste del algoritmo está en el orden polinómico de n, dependiendo arbitrariamente de ϵ. Por lo tanto, son los que ofrecen una aproximación más exacta, pero su coste puede ser muy elevado. Presentamos un problema para el cuál existe un algoritmo **FPTAS**, es decir, su coste es polinómico respecto a n y $\frac{1}{\epsilon}$.
Como ya hemos visto en ejemplos anteriores, no se puede implementar un **FPTAS** para todos los problemas $\mathcal{NP}$-difíciles.De hecho, si un problema tiene un algoritmo **FPTAS**, entonces es $\mathcal{NP}$ - difícil débil. Estos son los que aceptan algoritmos pseudo-polinómicos, cuyos costes son polinómicos en el valor de la entrada, pero arbitrarios respecto al número de dígitos.
Por ejemplo, si para comprobar la primalidad de un número n vemos si es divisible entre $2, 3, ...\sqrt{n}$, este algoritmo es pseudo-polinómico pero no polinómico, pues es exponencial respecto al número de dígitos. Los problemas para los que se ha demostrado que no pueden tener algoritmos pseudo-polinómicos se denominan $\mathcal{NP}$ - difíciles fuertes.
Para obtener algoritmos **FPTAS**, la técnica más utilizada es transformar la entrada de una programación dinámica. A continuación mostramos un ejemplo.

4.1. Problema de la mochila

Tenemos n objetos con pesos p_i y beneficio b_i, y un peso máximo M. Se trata de maximizar $\sum_{i=1}^{n} x_i \cdot b_i$, donde x_i vale 1 si cogemos el objeto y 0 si no lo cogemos; verificando que $\sum_{i=1}^{n} x_i \cdot p_i \leq M$

Para este problema podemos obtener un algoritmo aproximado de tal forma que dado $\epsilon > 0$, se tiene un radio de aproximación de $\frac{1}{1-\epsilon}$, es decir, arbitrariamente cerca de la solución óptima. Este algoritmo se basa en la programación dinámica. Recordamos como era:

Se define mochila(i,j) = máximo beneficio en la mochila de peso máximo j considerando los objetos del 1 al i.

$$mochila(i,j) = \begin{cases} mochila(i-1,j) & \text{si } p_i\text{>j} \\ \text{máx}(mochila(i-1,j), mochila(i-1,j-p_i)+b_i) & \text{c.c} \end{cases}$$

Este algoritmo tenía coste de $O(n \cdot M)$, que no es polinómico pues M puede ser todo lo grande que quiera. Sin embargo, el algoritmo aproximado utiliza una variante de esta programación dinámica. Definimos mochila2(i,j)= mínimo peso con los objetos 1 a i y con beneficio j.

$$\begin{cases} B = \underset{1 \leq i \leq n}{\text{máx}} (b_i) \\ C = n \cdot B \\ mochila2(i,j) = \begin{cases} mochila2(i-1,j) & \text{si } p_i\text{>j} \\ \text{mín}(mochila2(i-1,j), mochila2(i-1,j-b_i)+p_i) & \text{c.c} \end{cases} \end{cases}$$

Buscamos el mayor j que verifica que $mochila2(n,j) \leq M$. El coste de este algoritmo está en el $O(n \cdot C)$. Al igual que antes, este coste no tiene por qué ser polinómico, pero podemos reescalar los beneficios de tal forma que C sí que sea polinómico. Para ello fijamos $\epsilon > 0$.

$$\hat{b_i} = \lfloor \frac{b_i \cdot n}{\epsilon \cdot B} \rfloor$$

Con estos valores aplicamos el siguiente algoritmo:

Algoritmo 4: ALGORITMO ϵ - APROXIMADO PARA LA MOCHILA [2]

Entrada: $\{n, p_1 \dots p_n, b_1 \dots b_n, M, \epsilon\ \}$
Salida: $(x_1 \dots x_n)$

```
Descartar los b_i tales que p_i > M;
B = máx_{1≤i≤n} (b_i);
i = 1;
mientras i ≤ n hacer
    b̂_i = ⌊(b_i · n)/(ε · B)⌋;
fin
(x_1 ... x_n) = PDinamica2(n, p_1 ... p_n, b̂_i ... b_n, M̂);
devolver (x_1 ... x_n)
```

Al aplicar la programación dinámica sobre los $\hat{b_i}$, tenemos que el número de columnas de la matriz es $n \cdot \hat{B}$, donde $\hat{B} = \max_{1 \leq i \leq n} (\hat{b_i}) \leq \frac{n}{\epsilon}$. Por lo tanto, el algoritmo tiene coste de $O(\frac{n^3}{\epsilon})$. Este algoritmo es polinómico, pero debemos tener en cuenta que cuanto mejor sea la aproximación, más costoso nos será.

Por último, tenemos que comprobar que el algoritmo es $\frac{1}{1-\epsilon}$ - aproximado.

Demostración. Supongamos que la solución óptima del problema es el conjunto S, con beneficio total C^* ($C^* = \sum_{i \in S} b_i$). El beneficio total de los valores reescalados de S verifica:

$$\sum_{i \in S} \hat{b_i} = \sum_{i \in S} \lfloor \frac{b_i \cdot n}{\epsilon \cdot B} \rfloor \geq \sum_{i \in S} (b_i \cdot \frac{n}{\epsilon \cdot B} - 1) \geq C^* \cdot \frac{n}{\epsilon \cdot B} - n$$

Por otro lado, sea $\hat{S}$ el conjunto que conforma la solución óptima del problema con los valores reescalados, verifica que

$$\sum_{i \in \hat{S}} \hat{b_i} \geq \sum_{i \in S} \hat{b_i}$$

Además , también se verifica que

$$b_i \geq \hat{b_i} \cdot \frac{\epsilon \cdot B}{n}$$

Así pues, tenemos que el coste de la solución del algoritmo, dado por C, cumple el radio de aproximación $\frac{1}{1-\epsilon}$:

$$C = \sum_{i \in \hat{S}} b_i \geq \sum_{i \in \hat{S}} \hat{b_i} \cdot \frac{\epsilon \cdot B}{n} \geq (C^* \cdot \frac{n}{\epsilon \cdot B} - n) \cdot \frac{\epsilon \cdot B}{n} = C^* - \epsilon \cdot B \geq C^* \cdot (1 - \epsilon)$$

□

5. Problemas sin algoritmos aproximados

5.1. Problema del viajante genérico

En este caso tenemos un grafo $\mathcal{G}$ que no tiene por qué verificar la desigualdad triangular. Vamos a ver que no existe un algoritmo aproximado para este problema. $\mathcal{NP}$-difícil, a menos que $\mathcal{P} = \mathcal{NP}$

Teorema.

$$\mathcal{P} \neq \mathcal{NP} \Rightarrow \forall \rho \geq 1, \nexists \mathcal{A} \text{ } \textit{algoritmo } \rho \textit{ - aproximado para el problema del viajante genérico.}$$

Demostración. Vamos a probarlo por reducción al absurdo. Supongamos que $\exists \rho \geq 1$ tal que $\exists \mathcal{A}$ algoritmo ρ - aproximado. Vamos a ver que dicho $\mathcal{A}$ sirve para resolver el problema de decisión del ciclo hamiltoniano, que ya hemos probado que es $\mathcal{NP}$ - completo.

Sea $\mathcal{G} =< V, E >$ una entrada para el problema del ciclo hamiltoniano, queremos saber si existe un ciclo hamiltoniano en $\mathcal{G}$ usando el algoritmo $\mathcal{A}$ Transformamos $\mathcal{G}$ en una entrada adecuada para el problema del viajante genérico, de tal forma que $\mathcal{G}' =< V, E' >$ es el completado de $\mathcal{G}$ y el coste de sus aristas viene dado por:

$$c(u, v) = \begin{cases} 1 & \text{si } (u, v) \in E \\ \rho \cdot |V| + 1 & \text{c.c} \end{cases}$$

Ahora bien, si $\mathcal{G}$ tuviera un ciclo hamiltoniano, la solución óptima de $\mathcal{G}'$ tendría coste $|V|$. Sin embargo, si no tuviera ninguno entonces todas las soluciones de $\mathcal{G}'$ tendrían coste C' que verifica que:

$$C' \geq (\rho \cdot |V| + 1) + (|V| - 1) = \rho \cdot |V| + V > \rho \cdot |V|$$

Por lo tanto, el coste de un ciclo hamiltoniano en $\mathcal{G}'$ que no lo es en $\mathcal{G}$ es al menos $\rho+1$ veces el coste de uno que sí lo es en $\mathcal{G}$. Aplicamos el algoritmo $\mathcal{A}$ a $\mathcal{G}'$, que es un algoritmo ρ - aproximado,por lo que devuelve un ciclo hamiltoniano cuyo coste dista del óptimo como mucho ρ:

1. Si $\mathcal{G}$ tuviera un ciclo hamiltoniano (que sería solución óptima del problema del viajante), $\mathcal{A}$ debería devolverlo, pues si no estaría devolviendo uno que dista como poco $\rho + 1$ del óptimo.

2. Si Si $\mathcal{G}$ no tuviera un ciclo hamiltoniano, $\mathcal{A}$ devuelve uno de coste mayor que $\rho \cdot |V|$.

Por lo tanto, podemos usar $\mathcal{A}$ para resolver el problema de decisión del ciclo hamiltoniano. Como $\mathcal{A}$ es polinómico, tenemos que $\mathcal{P} = \mathcal{NP}$!!! [1] □

6. Bibliografía

Referencias

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[2] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms.* McGraw-Hill Higher Education, 2006.

[3] R. Peña Marí. *Algoritmos y estructuras de datos : con programas verificados en Dafny.* Garceta, Madrid, 2019.

[4] V. Vazirani. *Approximation Algorithms.* Springer Berlin Heidelberg, 2013.

[5] Y. Xiao, H. Chen, and F. Li. *Handbook on Sensor Networks.* World Scientific, 2010.