# Algoritmos de Aproximación

Daniel Valverde Menasalvas

24 de diciembre de 2018

## 1. Introducción

Hay muchos problemas NP-completos que son demasiado importantes como para abandonarlos simplemente por no poder obtener una solución óptima. Como poco, tendremos tres posibles enfoques para este tipo de problemas:

1. Utilizar un algoritmo de coste exponencial en casos con pocos datos de entrada.
2. Aislar casos especiales que se puedan resolver con coste polinómico.
3. Utilizar algoritmos que encuentren soluciones cercanas a la óptima con coste polinómico.

Un algoritmo que devuelve soluciones cercanas a la óptima se llama algoritmo de aproximación. En este trabajo se expondrán algoritmos de aproximación con coste polinómico para varios problemas NP-completos.

### 1.1. Radio de comportamiento para algoritmos de aproximación

Decimos que un algoritmo tiene, para un problema, un **radio de comportamiento** $\rho(n)$ si, para cualquier entrada de tamaño $n$, el coste $C$ de la solución producida por el algoritmo y el coste $C*$ de la solución óptima cumplen:

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

Esta definición sirve para problemas de maximización y minimización. Un radio de comportamiento de 1 indica una solución óptima y uno muy grande indica una mala aproximación. A los algoritmos que consiguen un radio de aproximación de $\rho(n)$ los llamamos $\rho(n)-$**algoritmos de aproximación**.

Un **esquema de aproximación** para un problema de optimización es un algoritmo de aproximación que toma como input un valor $\epsilon > 0$ de forma que, dado un $\epsilon$, el esquema es un $(1+\epsilon)-$algoritmo de aproximación. Un esquema

de aproximación es un **algoritmo de aproximación en tiempo polinómico** si dado un $\epsilon > 0$, el esquema tiene coste polinómico en el tamaño de los datos de entrada.

Decimos que un esquema de aproximación es un **esquema de aproximación completamente polinómico** si es polinómico en $1/\epsilon$ y en el tamaño de los datos de entrada. Un ejemplo sería $O((1/\epsilon)^2 n^3)$. De esta manera, si somos menos exigentes con la precisión de la solución del algoritmo (aumentamos $\epsilon$) veremos una disminución en el tiempo de ejecución.

## 2. El problema del recubrimiento por vértices (vertex-cover)

El recubrimiento por vértices de un grafo no dirigido $G = (V, E)$ es un subconjunto $V' \subseteq V$ tal que si $(u, v)$ es una arista de $G$, entonces $u \in V' \vee v \in V'$. El tamaño de $V'$ es el número de vértices que contiene.

El problema del recubrimiento por vértices es encontrar un recubrimiento de tamaño mínimo (**recubrimiento óptimo por vertices**) en un grafo no dirigido. Aunque es difícil encontrar una solución óptima a este problema, no lo es tanto encontrar una solución aproximada. El siguiente algoritmo toma un grafo no dirigido $G$ y devuelve un recubrimiento por vértices cuyo tamaño es, a lo sumo, el doble del óptimo ($\rho(n) = 2$). Es posible demostrar que el algoritmo es un 2-algoritmo de aproximación en tiempo polinómico.

**Algorithm 1** APPROX-VERTEX-COVER(G)

```
1: C ← ∅
2: E' ← G[E]
3: while E' ≠ ∅ do
4:     sea (u, v) una arista arbitraria de E
5:     C ← C ∪ {u, v}
6:     eliminar de E' toda arista incidente en u o v
7: end while
8: return C
```

Si usamos una lista de adyacencia para representar E', el tiempo de ejecución del algoritmo es $O(V + E)$.

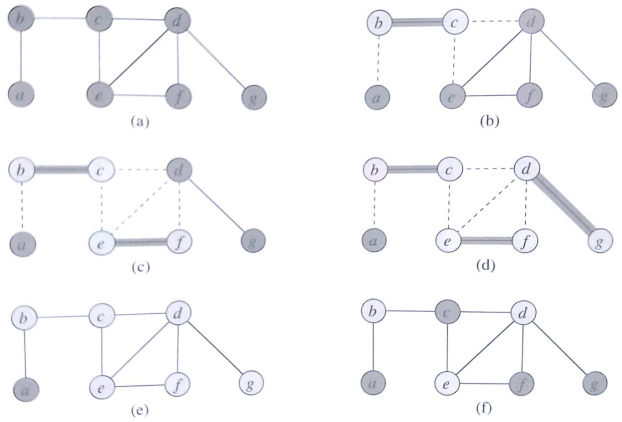


Figura 1: Ejemplo gráfico del algoritmo APPROX-VERTEX-COVER

## 3. El problema del viajante

Dado un grafo G = (V,E) no dirigido y valorado positivamente, El problema del viajante se basa en encontrar un ciclo hamiltoniano de G con coste mínimo. Este problema es NP-completo incluso si la función de costes cumple la desigualdad triangular.

Decimos que la función de costes del grafo, c(u,v), satisface la **desigualdad triangular** si para cualesquiera $u, v, w \in V$, $c(u, w) \leq c(u, v) + c(v, w)$. Esta propiedad se satisface numerosas veces de manera natural, por ejemplo, si el grafo está compuesto por puntos del plano y c es la distancia entre ellos. A continuación se expone un 2-algoritmo de aproximación(omitiremos la demostración de esta propiedad) para el problema del viajante cumpliendo la desigualdad triangular.

**Algorithm 2** APPROX-TSP-TOUR(G,c)

1: elegimos un vértice $r \in V[G]$ para ser raíz
2: generamos un árbol recubridor mínimo T para G desde r usando el algoritmo de Prim
3: sea L la lista de vértices de T en pre-orden
4: **return** el ciclo hamiltoniano H que visita los vértices en el orden L

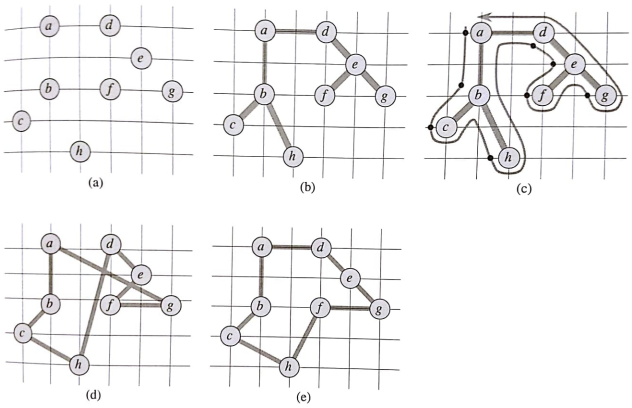


Figura 2: Ejemplo gráfico del algoritmo APPROX-TSP-TOUR

Si quitamos la condición de que c cumpla la desigualdad triangular, no se pueden encontrar buenas aproximaciones de este problema a no ser que $P = NP$. Es posible demostrar que, si $P \neq NP$, dado cualquier $\rho \geq 1$, no hay ningún $\rho(n)$-algoritmo de aproximación con tiempo polinómico para el problema general del viajante.

## 4. El problema del conjunto de cobertura

El problema del conjunto de cobertura es un problema de optimización que modela muchos problemas de selección de recursos, entre ellos el del recubrimiento por vértices. Sin embargo, el algoritmo mostrado en 2 no es aplicable aquí, por lo que intentaremos otros enfoques con un radio de aproximación logarítmico.

Una instancia $(X, F)$ del problema del conjunto de cobertura consiste de un conjunto finito $X$ y una familia $F$ de subconjuntos de $X$, tal que cada elemento de $X$ pertenece a, al menos, un subconjunto en $F$. El problema se trata de encontrar un subconjunto $C \subseteq F$ cuyos elementos cubran todo $X$.

El problema del conjunto de cobertura es una abstracción de diversos problemas típicos de combinatoria. Un ejemplo sencillo sería el de tener un conjunto $X$ de habilidades necesarias para realizar un trabajo y una serie de personas, cada una con alguna de esas habilidades. El problema consistiría en obtener un equipo de personas con el menor número de integrantes posible tal que, para cada habilidad, haya al menos una persona en el equipo que la tenga.

Propondremos un algoritmo voraz para resolver el problema. Este consiste en coger, en cada etapa del problema, el subconjunto $S$ que cubra el mayor

número de elementos no cubiertos. El algoritmo es el siguiente:

**Algorithm 3** GREEDY-SET-COVER$(X, F)$

1: $U \leftarrow X$
2: $C \leftarrow \emptyset$
3: **while** $U \neq \emptyset$ **do**
4: elegir un $S$ que maximice $|S \cap U|$
5: $U \leftarrow U - S$
6: $C \leftarrow C \cup \{S\}$
7: **end while**
8: **return** $C$

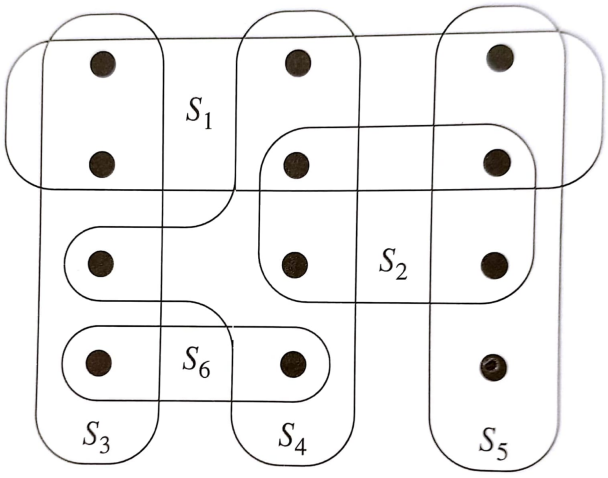


Figura 3: Ejemplo gráfico del problema del conjunto de cobertura. El algoritmo GREEDY-SET-COVER elegiría los conjuntos: $S_1, S_4, S_5$. $S_3$(podría cambiarse $S_3$ por $S_6$, esta elección depende de la implementación del algoritmo).

Hay una implementación de este algoritmo de orden $O(|X||F|\,\text{mín}(|X|, |F|))$. Esto es debido a que el bucle de las líneas 3-6 se ejecuta, a lo sumo, mín$(|X|, |F|)$ veces y el cuerpo del bucle puede ser implementado en $O(|X||F|)$. Puede probarse que este algoritmo es un $\rho(n)$-algoritmo de aproximación en tiempo polinómico, donde $\rho(n) = H(max|S| : S \in F)$, siendo $H(d) = \sum_{i=1}^{d} 1/i$. Consecuencia directa de esto último es que el algoritmo es también un $(ln|X| + 1)$-algoritmo de aproximación en tiempo polinómico.

# 5. Aleatorización y programación lineal

En esta sección estudiaremos dos técnicas útiles para diseñar algoritmos de aproximación: aleatorización y programación lineal.

## 5.1. Un algoritmo de aproximación aleatorizado para la satisfacibilidad de MAX-3-CNF

Una **3-formal normal conjuntiva (3-CNF)** es una expresión booleana formada por cláusulas unidas por AND de 3 literales unidos por OR. Un ejemplo sería $(x_1 \vee x_3 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$. El problema de la satisfacibilidad de 3-CNF se trata de determinar la satisfacibilidad de una fórmula puesta en 3-CNF, este es un problema NP-completo.

Si una fórmula 3-CNF no es satisfactible, puede que queramos saber 'como de cerca' esta de serlo, es decir, cual es el número máximo de cláusulas que se pueden satisfacer a la vez. Este problema es denominado **la satisfacibilidad de MAX-3-CNF**. El objetivo es encontrar una asignación para cada variable que maximice el número de cláusulas evaluando a 1. Es posible probar que, dada una instancia de la satisfacibilidad de MAX-3-CNF con n variables y m cláusulas, el algoritmo aleatorizado que, independientemente, pone cada variable a 1 con probabilidad 1/2 y a 0 con probabilidad 1/2 es un (8/7)-algoritmo de aproximación aleatorizado.

## 5.2. Aproximando el recubrimiento por vértices valorados usando programación linear

El **problema del mínimo recubrimiento por vértices valorados** es similar al problema del recubrimiento por vértices salvo que, en este caso, cada grafo tiene asociado un valor positivo, y buscamos un recubrimiento que tenga el menor valor posible. El valor de un recubrimiento es la suma de los valores de los vértices por los que está compuesto.

Si en este problema usamos el algoritmo de la sección 2 o uno aleatorizado obtendríamos soluciones con ratios de aproximación demasiado grandes. Sin embargo, podemos conseguir un límite inferior de la solución del problema usando programación lineal. Después 'redondearemos' esta solución para obtener un recubrimiento por vértices.

Llamaremos w(v) a la valoración del vértice v y asociaremos una variable binaria x(v) a cada vértice de V. Interpretaremos que v forma parte de nuestro recubrimiento si y solo si $x(v) = 1$. De esta manera podemos interpretar el problema del recubrimiento por vértices valorados como el siguiente problema de programación binaria:

minimizar

$$\sum_{v \in V} w(v)x(v)$$

sujeto a

$$x(u) + x(v) \geq 1 \forall u, v \in V$$

$$x(v) \in \{0, 1\} \forall v \in V$$

Encontrar soluciones para este problema es NP-complejo, sin embargo, podemos cambiar la condición de que $x(v) \in \{0, 1\}$ por $0 \leq x(v) \leq 1$. De esta manera obtenemos el siguiente problema de programación lineal denominado **problema de programación lineal relajado**:

minimizar

$$\sum_{v \in V} w(v)x(v)$$

sujeto a

$$x(u) + x(v) \geq 1 \forall v, u \in V$$

$$x(v) \leq 1 \forall v \in V$$

$$x(v) \geq 0 \forall v \in V$$

Una solución factible del problema binario es también solución factible del problema lineal. De esta manera, una solución óptima del problema lineal es límite inferior de la solución del problema binario(es necesariamente menor o igual) y, por lo tanto, límite inferior del recubrimiento por vértices valorados.

El algoritmo mostrado a continuación es un 2-algoritmo de aproximación de tiempo polinómico para el problema del recubrimiento por vértices valorados:

**Algorithm 4** APPROX-MIN-WEIGHT-VC(G,w)

1: $C \leftarrow \emptyset$
2: Computar una solución X para el problema de programación lineal
3: **for** each $v \in V$ **do**
4:  **if** $X(v) \geq 1/2$ **then**
5:  $C \leftarrow C \cup \{v\}$
6:  **end if**
7: **end for**
8: **return** C

Este algoritmo genera un recubrimiento eligiendo los vértices a los que la solución X del problema lineal había asignado valor 1/2 o más. A esto nos referíamos cuando decíamos que 'redondeábamos' la solución del problema de programación lineal.

# 6. El problema de la suma de subconjuntos

Dado un par $(S, t)$, donde S es un conjunto de n enteros positivos y t es otro entero positivo, el problema de la suma de subconjuntos consiste en determinar si existe un subconjunto de S cuya suma sea exactamente t. El problema es NP-completo.

Este problema lleva asociado otro problema de optimización que consiste en encontrar un subconjunto de S cuya suma sea lo mayor posible, sin llegar a ser mayor que t. Una aplicación práctica de este problema sería un camión con el que queremos transportar unas cajas de diferentes pesos de la forma más eficiente posible, es decir, llevando en cada viaje el máximo peso posible que pueda soportar el camión.

## 6.1. Un algoritmo exacto de coste exponencial

Antes de mostrar el algoritmo, son necesarias unas definiciones. La función MERGE-LISTS devuelve una lista formada por los integrantes de las dos listas que recibe como argumentos. Estos estarán ordenados y no se permitirán duplicados. La función tiene coste $O(|L|, |L'|)$ y se omitirá su pseudocódigo debido a su sencillez, aunque sí se incluirá su implementación en el código de C++. Definimos también el conjunto $S + x = \{s + x : s \in S\}$.

A continuación se muestra un algoritmo que computa, con coste exponencial, todas las posibles sumas de subconjuntos de S (al menos las que no excedan t) y devuelve la que más se acerque a t sin pasarse.

**Algorithm 5** EXACT-SUBSET-SUM(S,t)

1: $n \leftarrow |S|$
2: $L_0 = \{0\}$
3: **for** $i \leftarrow 1$ to n **do**
4: $\quad L_i \leftarrow MERGE - LISTS(L_{i-1}, L_{i-1} + x_i)$
5: $\quad$ Borramos de $L_i$ todo elemento mayor que t
6: **end for**
7: **return** Mayor elemento en $L_n$

## 6.2. Un esquema de aproximación completamente polinómico

A partir de la solución exponencial podremos crear un esquema de aproximación completamente polinómico para el problema de la suma de subconjuntos.

Para ello, simplificaremos las listas $L_i$ en cada iteración del bucle for. La simplificación se hará conforme a un parámetro $0 < \delta < 1$. Si $L$ es la lista inicial y $L'$ es la lista simplificada para cada elemento $y \in L$ existirá un $z \in L'$ tal que:

$$\frac{y}{1+\delta} \leq z \leq y$$

El algoritmo de simplificación es lineal en el número de objetos de la lista L. Se expone a continuación un ejemplo: dado $\delta = 0{,}1$ y $L =< 10, 11, 12, 15, 20, 21, 22, 23, 24, 29 >$, la lista simplificada es $L' =< 10, 12, 15, 20, 23, 29 >$.

Una vez definido el método de simplificación, podemos mostrar el algoritmo de aproximación para el problema de la suma de subconjuntos. Es posible demostrar que este algoritmo es un esquema de aproximación completamente polinómico para el problema dado.

**Algorithm 6** TRIM(L,$\delta$)

```
1: m ← |L|
2: L' ←< y_1 >
3: last ← y_1
4: for i ← 2 to m do
5:    if y_i > last(1 + δ) then
6:       Añadir y_i al final de L' {y_i ≥ last ya que L está ordenada}
7:       last ← y_i
8:    end if
9: end for
10: return L'
```

**Algorithm 7** APPROX-SUBSET-SUM(S,t,$\epsilon$)

```
1: n ← |S|
2: L_0 = {0}
3: for i ← 1 to n do
4:    L_i ← MERGE − LISTS(L_{i−1}, L_{i−1} + x_i)
5:    L_i ← TRIM(L_i, ϵ/2n)
6:    Borramos de L_i todo elemento mayor que t
7: end for
8: return Mayor elemento en L_n
```

# Referencias


[1] Cormen,T.H, *Introduction to Algorithms* 2ª ed. The Massachusetts Institute of Technology, 2001.