

# Tema 1: Algoritmos devoradores

A. Salguero, F. Palomo, I. Medina


Universidad de Cádiz

Diseño de Algoritmos  
Curso 2017/18

# Esquema general

Los algoritmos *devoradores* o *voraces* (*greedy*) se emplean normalmente en problemas de optimización, y responden al siguiente esquema general:

```

devorador :  $C \rightarrow S$  
 $S \leftarrow \emptyset$ 
mientras  $\neg \text{solución}(S) \wedge C \neq \emptyset$ 
     $p \leftarrow \text{selección}(C)$ 
     $C \leftarrow C - \{p\}$ 
    si  $\text{factible}(S \cup \{p\})$ 
         $S \leftarrow S \cup \{p\}$ 
  
```

Hay múltiples variantes de este esquema. Por ejemplo, a veces  $C$  y  $S$  son secuencias y no conjuntos. Las operaciones  $-$  y  $\cup$  se convierten entonces en la *extracción* e *inserción* en secuencias, respectivamente.

# Elementos

Se distinguen los siguientes elementos:

- Un *conjunto de candidatos*.
- Un *conjunto de candidatos seleccionados*.
- Una *función solución* que comprueba si un conjunto de candidatos es una solución (posiblemente no óptima).
- Una *función de selección* que indica el *candidato más prometedor* de los que quedan.
- Una *función de factibilidad* que comprueba si un conjunto de candidatos se puede ampliar para obtener una solución (no necesariamente óptima).
- Una *función objetivo* que asocia un valor a una solución, y que queremos optimizar.
- Un *objetivo*, *minimizar* o *maximizar*.

# Cuestiones

El candidato más prometedor puede no ser único. Una vez elegido se elimina y nunca más vuelve a ser considerado.

A cada paso, se toma la mejor opción sin preocuparse del futuro: si se incluye un candidato en la solución ya no sale de ella; si se excluye, no se le vuelve a tener en cuenta.

Si al terminar,  $S$  no es solución, el algoritmo no ha podido encontrar una solución. Esto puede ser debido a que no exista o a que el algoritmo sea incapaz de encontrarla:

*¿Qué causa es la responsable?*

Incluso en el caso de que encuentre una solución queda el problema de la optimalidad:

*¿Es la solución siempre óptima?*

# Cuestiones

Sólo se puede responder a estas cuestiones con un estudio particular para cada problema y algoritmo.

Existen problemas para los que existe un *algoritmo devorador óptimo* con el que siempre se encuentra una solución óptima. En otros debemos conformarnos con un *algoritmo devorador aproximado* que no garantiza la optimalidad de la solución encontrada.

En tal caso es deseable que el error, esto es, la diferencia entre los valores de la función objetivo para la solución óptima y la obtenida, sea lo menor posible o, al menos, que esté acotado.

A veces, hay que hallar todas las soluciones y no sólo una.

Entonces, el esquema básico ha de modificarse. La unicidad o no de la solución debe ser estudiada teóricamente.

# El problema de la mochila

Dado un conjunto  $O$  de objetos, cada uno con un valor  $v$  y un peso  $p$ , y una mochila con una capacidad,  $c$ , que limita el peso total que puede transportar, se desea hallar la composición de la mochila que maximiza el valor de la carga.

En su *versión continua*, los objetos pueden fraccionarse y el problema puede resolverse óptimamente con un algoritmo devorador.

Existen tres posibles estrategias de selección de objetos:

- En orden decreciente de valor
- En orden creciente de peso
- En orden decreciente de relación valor/peso

Sólo esta última produce un algoritmo óptimo.

# Elementos

Se distinguen los siguientes elementos:

- Conjunto inicial de candidatos: los objetos.
- Función solución: ¿se ha llenado ya la mochila?
- Función de selección: elige *un* objeto con máxima relación valor/peso.
- Función de factibilidad: ¿se puede introducir el objeto sin exceder la capacidad?
- Función objetivo: el valor total de los objetos devueltos.
- Objetivo: maximizar.

A cada paso, el objeto seleccionado tiene la mayor relación valor/peso. Si cabe, se introduce por completo en la mochila, si no, se rellena la mochila con la fracción que quepa del mismo.

# Algoritmo

```
mochila :  $O \times c \rightarrow S$   
 $C \leftarrow O$   
 $S \leftarrow \emptyset$   
mientras  $c \neq 0 \wedge C \neq \emptyset$   
     $\langle v, p \rangle \leftarrow \text{selecciona-objeto}(C)$   
     $C \leftarrow C - \{\langle v, p \rangle\}$   
    si  $p \leq c$   
         $S \leftarrow S \cup \{\langle v, p \rangle\}$   
         $c \leftarrow c - p$   
    si no  
         $S \leftarrow S \cup \{\langle v \cdot c/p, c \rangle\}$   
         $c \leftarrow 0$ 
```



# Algoritmo

*selecciona-objeto* :  $C \rightarrow v \times p$

$r \leftarrow -\infty$

para todo  $\langle a, b \rangle \in C$

    si  $a/b > r$

$r \leftarrow a/b$

$\langle v, p \rangle \leftarrow \langle a, b \rangle$

# Análisis

Analicemos el algoritmo. Sea  $n = |O|$ :

- En el peor caso, el bucle realiza  $i = |C|$  iteraciones.
- La función de selección realiza  $\Theta(i)$  operaciones elementales, donde  $i$  varía desde  $n$  hasta, en el peor caso, 1.

Por lo tanto,  $t(n) \in O(n^2)$  operaciones elementales.

Preordenando los objetos en orden decreciente de relación valor/peso, la selección de los objetos se convierte en una operación elemental.

Con esta mejora, el tiempo dominante es el de la ordenación y puede obtenerse un algoritmo con  $t(n) \in \Theta(n \log n)$ .

# El problema del cambio de monedas

Sea  $M$  un conjunto de monedas y  $c$  una cantidad a devolver. Por cada tipo de moneda de valor  $v$  se dispone de un suministro de  $k$  unidades. Se desea hallar la composición del cambio que posee el menor número de monedas.

Una estrategia posible de selección de monedas consiste en elegir, de las que quedan, la de mayor valor.

Esto se hace a diario en máquinas expendedoras, cajeros automáticos, etc. Si es posible devolver el cambio con las monedas disponibles, se actualiza el conjunto para su próximo empleo.

Conviene seleccionar todas las monedas del mismo valor de una vez: puede tratarse  $M$  como un *multiconjunto* o como un *conjunto de pares*.

# El problema del cambio de monedas

Esta estrategia no produce, en general, un algoritmo devorador óptimo.

## Ejemplo

Para un cambio de 15 unidades con monedas de 1, 5, 10 y 12 unidades, el algoritmo devolvería una moneda de 12 y tres de 1. Sin embargo, el cambio óptimo está formado por dos monedas: una de 10 y otra de 5.

Sin embargo, para determinados conjuntos de monedas el algoritmo sí es óptimo.

De hecho, se demuestra que si se dispone de un *número suficiente* de monedas de 1, 5, 10, 25, 50, 100, 200 y 500 unidades, el algoritmo encuentra una solución óptima.

Igualmente ocurre con monedas de 1, 5, 10, 20, 50, 100 y 200 unidades.

# Elementos

Se distinguen los siguientes elementos:

- Conjunto inicial de candidatos: las monedas.
- Función solución: ¿se ha devuelto ya el cambio?
- Función de selección: elige las monedas de mayor valor.
- Función de factibilidad: ¿se pueden devolver las monedas sin exceder el cambio?
- Función objetivo: el número de monedas devueltas.
- Objetivo: minimizar.

A cada paso, las monedas seleccionadas tienen el mayor valor. Se devuelven tantas monedas de dicho valor como se puedan suministrar sin exceder el cambio a devolver.

# Algoritmo

$\text{cambio} : M \times c \rightarrow S$

$C \leftarrow M$

$S \leftarrow \emptyset$

mientras  $c \neq 0 \wedge C \neq \emptyset$

$\langle v, k \rangle \leftarrow \text{selecciona-monedas}(C)$

$C \leftarrow C - \{\langle v, k \rangle\}$

$k \leftarrow \text{mín}(k, c \text{ div } v)$

    si  $k > 0$

$S \leftarrow S \cup \{\langle v, k \rangle\}$

$c \leftarrow c - v \cdot k$

$\text{selecciona-monedas} : C \rightarrow v \times k$

$v \leftarrow -\infty$

para todo  $\langle a, b \rangle \in C$

    si  $a > v$

$\langle v, k \rangle \leftarrow \langle a, b \rangle$

# Análisis

Analicemos el algoritmo. Sea  $n = |M|$ :

- En el peor caso, el bucle realiza  $i = |C|$  iteraciones.
- La función de selección realiza  $\Theta(i)$  operaciones elementales, donde  $i$  varía desde  $n$  hasta, en el peor caso, 1.

Por lo tanto,  $t(n) \in O(n^2)$  operaciones elementales.

Preordenando las monedas en orden decreciente de valor, su selección se convierte en una operación elemental.

Con esta mejora, el tiempo dominante es el de la ordenación y puede obtenerse un algoritmo con  $t(n) \in \Theta(n \log n)$ .

# Árbol de expansión mínimo

Dado un grafo  $G = \langle V, A \rangle$  conexo y ponderado con valores no negativos en sus aristas, se trata de calcular un subgrafo  $\langle V, S \rangle$  conexo de forma que la suma de los valores de sus aristas sea mínima.

Se demuestra que un subgrafo tal es un árbol. Éste se denomina «de expansión» por unir todos los vértices y «mínimo» por ser mínimo el valor total de sus aristas.

## Nota

Un grafo puede tener más de un árbol de expansión mínimo.

Supondremos la existencia de un orden en  $A$  inducido por la función de ponderación  $p$ .

$$\{i, j\} \leq \{k, l\} \iff p(i, j) \leq p(k, l)$$



# Elementos

Distinguimos:

- Conjunto inicial de candidatos: las aristas.
- Función solución: ¿se tiene ya un árbol de expansión?
- Función de selección: elige *una* arista de valor mínimo, quizás con alguna restricción adicional.
- Función de factibilidad: ¿se puede incluir la arista sin introducir ciclos?
- Función objetivo: el valor total de las aristas devueltas.
- Objetivo: minimizar.

La comprobación de factibilidad es costosa: los algoritmos intentarán simplificarla manteniendo el subgrafo acíclico por construcción.

# Algoritmo de Kruskal

El algoritmo de Kruskal construye un árbol de expansión mínimo a partir de un bosque de  $G$  (inicialmente, vacío). La idea es la siguiente:

*árbol-expansión-mínimo* :  $V \times A \rightarrow S$

$C \leftarrow A$

$S \leftarrow \emptyset$

mientras  $\neg \text{árbol-expansión}(V, S) \wedge \overbrace{C \neq \emptyset}^{\text{innecesario}}$

$a \leftarrow \text{mín}(C)$

$C \leftarrow C - \{a\}$

si *acíclico*( $V, S \cup \{a\}$ )

$S \leftarrow S \cup \{a\}$

Se emplean *estructuras de partición* para simplificar la comprobación de aciclicidad. Supondremos  $\langle V, A \rangle$  convenientemente aritmetizado.

Para obtener un algoritmo eficiente, se preordena  $A$ . Una alternativa a la preordenación es emplear un *montículo invertido* para  $A$ .

# Algoritmo de Kruskal

La siguiente versión actúa sobre un grafo representado mediante su lista de aristas.

```
Kruskal :  $V \times A \rightarrow S$   
 $C \leftarrow A$   
 $S \leftarrow \emptyset$   
 $n \leftarrow |V|$   
 $p \leftarrow \text{partición-inicial}(n)$   
ordena( $C$ )  
mientras  $|S| \neq n - 1$   
     $\{i, j\} \leftarrow \text{extrae-primer}(C)$   
     $e_1 \leftarrow \text{búsqueda}(p, n, i)$   
     $e_2 \leftarrow \text{búsqueda}(p, n, j)$   
    si  $e_1 \neq e_2$   
        unión( $p, n, e_1, e_2$ )  
         $S \leftarrow S \cup \{\{i, j\}\}$ 
```

Al ser  $G$  conexo, sabremos que  $\langle V, S \rangle$  es un árbol de expansión cuando  $|S| = n - 1$ , condición que habrá de cumplirse

# Análisis

El tamaño de la entrada viene dado por  $n = |V|$  y  $a = |A|$ . En general, el tiempo puede depender de ambos parámetros.

Análisis:

- Se ordenan las aristas.
- Se crea una partición inicial.
- Se ejecutan  $2a$  búsquedas como máximo.
- Se ejecutan  $n - 1$  uniones exactamente.

En el peor caso:

$$t(n, a) = t_O(a) + t_P(n) + 2at_B(n) + (n - 1)t_U(n)$$

# Análisis

El tiempo también depende de la representación escogida. Por ejemplo, si se emplea un bosque con control de alturas:

$$\begin{aligned} t(n, a) &\in O(a \log a) + O(n) + O(a \log n) + O(n) \\ &= O(a \log n) \end{aligned}$$

Recuérdese que  $\log n^2 = 2 \log n$ , y que al ser  $G$  conexo:

$$\underbrace{n - 1}_{\text{árbol}} \leq a \leq \underbrace{\frac{n(n - 1)}{2}}_{\text{grafo completo (clique)}}$$

Por lo tanto, dependiendo de la densidad del grafo, se obtiene que:

$$t(n, a) \in \begin{cases} O(n \log n), & a \in \Theta(n) \\ O(n^2 \log n), & a \in \Theta(n^2) \end{cases}$$

# Algoritmo de Prim

El algoritmo de Prim construye un árbol de expansión mínimo a partir de un subárbol de  $G$ . Inicialmente, éste tiene un único vértice (el primero, por ejemplo). La idea es la siguiente:

*árbol-expansión-mínimo* :  $V \times A \rightarrow S$   
 $C \leftarrow A$   
 $\langle U, S \rangle \leftarrow \langle \{1\}, \emptyset \rangle$   
 mientras  $\neg \text{árbol-expansión}(U, S) \wedge \overbrace{C \neq \emptyset}^{\text{innecesario}}$   
      $\{i, j\} \leftarrow \text{selecciona-arista}(U, C)$   
      $C \leftarrow C - \{\{i, j\}\}$   
      $\langle U, S \rangle \leftarrow \langle U \cup \{j\}, S \cup \{\{i, j\}\} \rangle$

Asegurar la aciclicidad es sencillo. Basta seleccionar *una* arista  $\{i, j\} \in C$  de valor mínimo que tenga uno de sus vértices en el subárbol ( $i \in U$ ) y el otro no ( $j \notin U$ ).

La cuestión está en elegir el vértice externo eficientemente.

Supondremos  $\langle V, A \rangle$  convenientemente aritmetizado.

# Algoritmo de Prim

La siguiente versión actúa sobre un grafo representado mediante su matriz de pesos.

*Prim* :  $p \times n \rightarrow S$

$C \leftarrow \emptyset$

desde  $j \leftarrow 2$  hasta  $n$

$C \leftarrow C \cup \{j\}$

$c[j] \leftarrow 1$

$d[j] \leftarrow p[1, j]$

$S \leftarrow \emptyset$

mientras  $C \neq \emptyset$

$k \leftarrow \text{selecciona-vértice}(C, d)$

$C \leftarrow C - \{k\}$

$S \leftarrow S \cup \{\{c[k], k\}\}$

para todo  $j \in C$

si  $p[k, j] < d[j]$

$c[j] \leftarrow k$

$d[j] \leftarrow p[k, j]$

# Algoritmo de Prim

La función de selección calcula el vértice de  $C$  más cercano a  $V - C$ .

*selecciona-vértice* :  $C \times d \rightarrow k$

$v \leftarrow \infty$

para todo  $j \in C$

si  $d[j] < v$

$v \leftarrow d[j]$

$k \leftarrow j$

Análisis:

- La función de selección realiza  $|C|$  comparaciones, variando  $|C|$  en el exterior desde  $n - 1$  hasta 1.
- El bucle interno realiza  $|C|$  comparaciones, con  $|C|$  variando desde  $n - 2$  hasta 0.

Por lo tanto,  $t(n) \in \Theta(n^2)$  comparaciones.



# Caminos mínimos

Dado un grafo orientado y ponderado con valores no negativos, se trata de encontrar el *camino mínimo* que conduce de un vértice a otro.

El algoritmo de Dijkstra los calcula desde un vértice origen a todos los demás. Con la matriz de pesos, sus *valores* se calculan como sigue:

*Dijkstra* :  $p \times n \times i \rightarrow d$

$C \leftarrow \emptyset$

desde  $j \leftarrow 1$  hasta  $n$

$C \leftarrow C \cup \{j\}$

$d[j] \leftarrow p[i, j]$

$C \leftarrow C - \{i\}$

mientras  $C \neq \emptyset$

$k \leftarrow \text{selecciona-vértice}(C, d)$

$C \leftarrow C - \{k\}$

para todo  $j \in C$

$d[j] \leftarrow \text{mín}(d[j], d[k] + p[k, j])$

# Análisis

La función de selección calcula el vértice de  $C$  más cercano al origen.

*selecciona-vértice* :  $C \times d \rightarrow k$

$v \leftarrow \infty$

para todo  $j \in C$

si  $d[j] < v$

$v \leftarrow d[j]$

$k \leftarrow j$

Análisis:

- La función de selección realiza  $|C|$  comparaciones, variando  $|C|$  en el exterior desde  $n - 1$  hasta 1.
- El bucle interno realiza  $|C|$  comparaciones, con  $|C|$  variando desde  $n - 2$  hasta 0.

Por lo tanto,  $t(n) \in \Theta(n^2)$  comparaciones.

# Recuperación de la solución

La siguiente modificación permite recuperar los caminos mínimos.

*Dijkstra* :  $p \times n \times i \rightarrow c \times d$

$C \leftarrow \emptyset$

desde  $j \leftarrow 1$  hasta  $n$

$C \leftarrow C \cup \{j\}$

$c[j] \leftarrow i$

$d[j] \leftarrow p[i, j]$

$C \leftarrow C - \{i\}$

mientras  $C \neq \emptyset$

$k \leftarrow \text{selecciona-vértice}(C, d)$

$C \leftarrow C - \{k\}$

para todo  $j \in C$

si  $d[k] + p[k, j] < d[j]$

$c[j] \leftarrow k$

$d[j] \leftarrow d[k] + p[k, j]$

El predecesor de  $j$  en el camino mínimo de  $i$  a  $j$  es  $c[j]$ . Esto permite reconstruir el camino.

# Referencias



Brassard, Gilles & Bratley, Paul.

*Fundamentos de Algoritmia.*

Prentice-Hall. 1997.



Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L. & Stein, Clifford.

*Introduction to Algorithms.*

MIT Press. 2001. 2ª ed.



Manber, Udi.

*Introduction to Algorithms. A Creative Approach.*

Addison-Wesley. 1989.



Sedgwick, Robert.

*Algorithms.*

Addison-Wesley. 1988. 2ª edición.