# El problema de la mochila

David Mallasén Quintana

**Resumen**

Implementación y comparación de diferentes algoritmos para resolver el problema de la mochila en sus distintas variantes. Se incluye una introducción al problema y el código de las resoluciones en C++.

# Índice



# 1. Introducción

## 1.1. Descripción del problema y variantes

El problema de la mochila es un problema de optimización combinatoria, es decir, que busca la mejor solución entre un conjunto finito de posibles

soluciones. Supondremos que tenemos una mochila con un peso limitado y que queremos llenarla con una serie de objetos dados por su peso y su valor. El objetivo del problema será maximizar el valor total de los objetos que metamos en la mochila sin exceder su peso máximo.

Es uno de los 21 problemas NP-completos de Richard Karp, lista elaborada en 1972 y perteneciente a su trabajo Reducibility Among Combinatorial Problems". Esto surgió como profundización del trabajo de Stephen Cook, quien en 1971 había demostrado uno de los resultados más importantes y pioneros de la complejidad computacional: la NP-completitud del Problema de satisfacibilidad booleana (SAT). El descubrimiento de Karp de que todos estos importantes problemas eran NP-completos motivó el estudio de la NP-completitud y de la indagación en la famosa pregunta de si $P = NP$.

### 1.1.1. Definición formal del problema

Supongamos que tenemos $n$ objetos numerados del 1 al $n$, cada uno con un peso $p_i > 0$ y un valor $v_i > 0$ para cada $i \in \{1 \dots n\}$. Tendremos también una mochila que soporta un peso máximo $M > 0$.

Definimos la función $x_i \in \{0, 1\}$ que indicará si se ha cogido el objeto $i$ ($x_i = 1$) o no ($x_i = 0$). El problema consiste en maximizar

$$\sum_{i=1}^{n} v_i x_i$$

con la restricción de $\sum_{i=1}^{n} p_i x_i < M$. La solución del problema vendrá dada por el conjunto de las $x_i$.

El caso en el que todos los objetos caben juntos en la mochila no tiene mucho interés ya que la solución consistiría en añadirlos todos. Por tanto consideraremos el caso en el que $\sum_{i=1}^{n} p_i > M$.

Para el método voraz que veremos en la sección 2.1 tomaremos la variante en que los objetos se pueden fraccionar. En este caso siempre obtendremos una solución óptima, lo demostraremos en 2.1.2, en la que $\sum_{i=1}^{n} p_i x_i = M$.

## 1.2. Desarrollo de los casos de prueba

# 2. Implementación de los algoritmos

## 2.1. Método voraz

En este apartado implementaremos una solución voraz al problema de la mochila. En el caso del método voraz obtendremos una solución de forma muy eficiente ($O(n \log n)$). Sin embargo tendremos que imponer la restricción de que los objetos sean fraccionables para que podamos asegurar una solución óptima.

### 2.1.1. Descripción de la solución

Primero ordenaremos los objetos según su densidad $d_i = \frac{v_i}{p_i}$. A la hora de construir la solución iremos cogiendo los objetos enteros en orden decreciente de densidad mientras quepan. Finalmente, si sobra hueco, fraccionaremos el objeto de mayor densidad que nos quede para terminar de rellenar toda la mochila.

### 2.1.2. Demostración de optimalidad

Sea $X = (x_1, \ldots, x_n)$ la solución construida por el algoritmo voraz como hemos indicado anteriormente. Como hemos supuesto al principio que $\sum_{i=1}^{n} p_i > M$, $\exists j \in \{1, \ldots, n\}$ tal que $x_j < 1$. Por la forma en la que construimos la solución sabemos que $0 \leq x_j < 1$ y $x_i = \begin{cases} 1 & \text{si } i < j \\ 0 & \text{si } i > j \end{cases}$. Supongamos que la solución $X$ no es óptima y procedamos mediante el método de reducción de diferencias. Comparamos con una solución óptima $Y = (y_1, \ldots, y_n)$.

Sea $k = \text{mín}\{i : y_i \neq x_i\}$. Por como funciona el algoritmo se debe cumplir que $k \leq j$, veamos que $y_k < x_k$:

- Si $\underline{k < j}$: $x_k = 1$ y, por tanto, $y_k < x_k$.
- Si $\underline{k = j}$: $y_i = 1$ para $1 \leq i < k$ por lo que $y_k > x_k$ implicaría $\sum_{i=1}^{n} p_i y_i > M$, cosa que no puede suceder. Por como hemos elegido $k$, $y_k \neq x_k$, luego debe ser $y_k < x_k$.
- Si $\underline{k > j}$: Por como hemos construido la solución voraz, $\sum_{i=1}^{n} p_i y_i > M$, luego este caso no se puede dar.

Modificamos la solución óptima aumentando $y_k$ hasta que $y_k = x_k$ y decrementando los $y_{k+1}, \ldots, y_n$ de forma que el peso de la mochila siga siendo $M$. Obtenemos así $Z = (z_1, \ldots, z_n)$ que cumplirá $z_i = x_i$ para $1 \leq i \leq k$. También tendremos, por como hemos modificado $Y$ para conseguir $Z$, que:

$$\sum_{i=k+1}^{n} p_i(y_i - z_i) = p_k(z_k - y_k) \qquad (*)$$

Finalmente veamos que $Z$ también es óptima. Para ello, como $Y$ lo era, basta ver que no hemos empeorado la situación, es decir, que $\sum_{i=1}^{n} v_i z_i \geq \sum_{i=1}^{n} v_i y_i$:

$$\begin{aligned}
\sum_{i=1}^{n} v_i z_i &= \sum_{i=1}^{n} v_i y_i + v_k(z_k - y_k) - \sum_{i=k+1}^{n} v_i(y_i - z_i) = \\
&= \sum_{i=1}^{n} v_i y_i + \frac{v_k}{p_k} p_k(z_k - y_k) - \sum_{i=k+1}^{n} \frac{v_i}{p_i} p_i(y_i - z_i) \overset{\frac{v_k}{p_k} \geq \frac{v_i}{p_i} \implies -\frac{v_i}{p_i} \geq -\frac{v_k}{p_k}}{\geq} \\
&\geq \sum_{i=1}^{n} v_i y_i + (p_k(z_k - y_k) - \sum_{i=k+1}^{n} p_i(y_i - z_i)) \frac{v_k}{p_k} \overset{(*)}{=} \\
&= \sum_{i=1}^{n} v_i y_i
\end{aligned}$$

### 2.1.3. Código

```
1
2  /**
3   * Resuelve el problema de la mochila con objetos fraccionables mediante un
4   * algoritmo voraz. Presuponemos que la suma de los pesos de todos los
5   * objetos > M.
6   *
7   * Coste en tiempo: O(n logn), n = numero de objetos.
8   *
9   * @param objetos Conjunto de objetos que tenemos disponibles.
10  * @param M Peso maximo que soporta la mochila.
11  * @param solucion Indica cuanto se debe coger de cada objeto [0, 1].
12  * @param valorSol Valor de la mochila con los objetos dados por solucion.
13  */
14 void mochilaVoraz(std::vector<ObjetoReal> const &objetos, double M,
15                   std::vector<double> &solucion, double &valorSol) {
16     const size_t n = objetos.size();
17
18     //Calculamos las densidades de cada objeto
19     std::vector<Densidad> d(n);
20     for (size_t i = 0; i < n; ++i) {
21         d[i].densidad = objetos[i].valor / objetos[i].peso;
22         d[i].obj = i;     //Para saber a que objeto corresponde
23     }
24
```

```
25      //Ordenamos de mayor a menor las densidades
26      std::sort(d.begin(), d.end(), std::greater<Densidad>());
27
28      //Cogemos los objetos mientras quepan enteros
29      size_t i;
30      for (i = 0; i < n && M - objetos[d[i].obj].peso >= 0; ++i) {
31          valorSol += objetos[d[i].obj].valor;
32          M -= objetos[d[i].obj].peso;
33          solucion[d[i].obj] = 1;
34      }
35
36      //Si aun no se ha llenado la mochila completamos partiendo el objeto
37      if (M > 0) {
38          solucion[d[i].obj] = M / objetos[d[i].obj].peso;
39          valorSol += objetos[d[i].obj].valor * solucion[d[i].obj];
40      }
41  }
```

### 2.1.4. Análisis de costes

Analizaremos ahora los costes en tiempo y memoria del algoritmo. En cuanto al tiempo tenemos un coste medio $O(n\log(n))$, donde $n$ es el numero de objetos, que viene dado por la ordenación de las densidades. Los dos bucles son de coste lineal y el resto de operaciones son constantes. En cuanto al espacio usamos un vector de tamaño $n$ para almacenar las densidades pero como es del orden del tamaño de los datos obtenemos un coste en memoria de $O(1)$.

## 2.2. Programación dinámica

### 2.2.1. Descripción de la solución

Veamos la forma de abordar el problema desde el punto de vista de la programación dinámica. Primero definimos la función:

*mochila(i, j)* = máximo valor que podemos poner en la mochila de peso máximo j considerando los objetos del 1 al i.

Tomamos como casos base:

$$mochila(0, j) = 0 \qquad 0 \leq j \leq M$$
$$mochila(i, 0) = 0 \qquad 0 \leq i \leq n$$

Y como función recursiva:

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

Así pues vamos probando cada objeto y, si no cabe, no lo cogemos pero si cabe tomamos el máximo entre cogerlo y no cogerlo.

### 2.2.2. Código

```
1
2  /**
3   * Resuelve el problema de la mochila 0-1 mediante un algoritmo de
4   * programacion dinamica. El peso de cada objeto y el peso maximo de la
5   * mochila deben ser enteros positivos.
6   *
7   * Coste: O(nM) en tiempo y espacio, n = numero de objetos, M = peso que
8   * soporta la mochila.
9   *
10  * @param objetos Conjunto de objetos que tenemos disponibles.
11  * @param M Peso maximo que soporta la mochila.
12  * @param solucion Indica si se coge el objeto o no.
13  * @param valorSol Valor de la mochila con los objetos dados por solucion.
14  */
15 void mochilaProgDin(std::vector<ObjetoInt> const &objetos, int M,
16                     std::vector<bool> &solucion, double &valorSol) {
17     const size_t n = objetos.size();
18
19     //Creamos e inicializamos a 0 la tabla con la que resolvemos el problema
20     std::vector<std::vector<double>> mochila(n + 1, std::vector<double>
21             (M + 1, 0));
22
23     //Rellenamos la tabla
24     //objetos[i - 1] ya que mochila va de [1..n] y objetos va de [0, n)
25     for (size_t i = 1; i <= n; ++i) {
26         for (int j = 1; j <= M; ++j) {
27             if (objetos[i - 1].peso > j)     //Si no cabe no lo cogemos
28                 mochila[i][j] = mochila[i - 1][j];
29             else     //Si cabe tomamos el maximo entre cogerlo y no cogerlo
30                 mochila[i][j] =
31                         std::max(mochila[i - 1][j],
32                                  mochila[i - 1][j - objetos[i - 1].peso] +
33                                  objetos[i - 1].valor);
34         }
35     }
36     valorSol = mochila[n][M];
37
38     //Calculamos que objetos hemos cogido
39     for (size_t i = n; i >= 1; --i) {
40         if (mochila[i][M] == mochila[i - 1][M]) //No cogido el objeto i
41             solucion[i - 1] = false;
42         else {     //Cogido el objeto i
43             solucion[i - 1] = true;
44             M -= objetos[i - 1].peso;
45         }
46     }
47 }
```