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 $x_i = \left\{ \begin{array}{ll} 1 & \text{si } i < j \\ 0 & \text{si } i > j \end{array} \right.$. 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 $1 \le i \le k$. 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 $\sum_{i=1}^{n} v_i z_i \ge \sum_{i=1}^{n} v_i y_i$:

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

2.1.3. Código

```

/**
 * Resuelve el problema de la mochila con objetos fraccionables mediante un
 * algoritmo voraz. Presuponemos que la suma de los pesos de todos los
 * objetos > M.
 *
 * Coste en tiempo: O(n logn), n = numero de objetos.
 *
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param M Peso maximo que soporta la mochila.
 * @param solucion Indica cuanto se debe coger de cada objeto [0, 1].
 * @param valorSol Valor de la mochila con los objetos dados por solucion.
 */
void mochilaVoraz(std::vector<ObjetoReal> const &objetos, double M,
                  std::vector<double> &solucion, double &valorSol) {
    const size_t n = objetos.size();

    //Calculamos las densidades de cada objeto
    std::vector<Densidad> d(n);
    for (size_t i = 0; i < n; ++i) {
        d[i].densidad = objetos[i].valor / objetos[i].peso;
        d[i].obj = i;      //Para saber a que objeto corresponde
    }

```

```
    //Ordenamos de mayor a menor las densidades
    std::sort(d.begin(), d.end(), std::greater<Densidad>());

    //Cogemos los objetos mientras quepan enteros
    size_t i;
    for (i = 0; i < n && M - objetos[d[i].obj].peso >= 0; ++i) {
        valorSol += objetos[d[i].obj].valor;
        M -= objetos[d[i].obj].peso;
        solucion[d[i].obj] = 1;
    }

    //Si aun no se ha llenado la mochila completamos partiendo el objeto
    if (M > 0) {
        solucion[d[i].obj] = M / objetos[d[i].obj].peso;
        valorSol += objetos[d[i].obj].valor * solucion[d[i].obj];
    }
}
```

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 número 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

TODO————————————

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:

$$\begin{aligned} mochila(0, j) &= 0 \qquad 0 \leq j \leq M \\ mochila(i, 0) &= 0 \qquad 0 \leq i \leq n \end{aligned}$$

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. Para ello recorremos la tabla por filas de forma ascendente (cada vez el intervalo $[0, i]$ es más grande) y cada fila la recorremos también de forma ascendente (cada vez la mochila soporta un peso mayor j hasta llegar a M). De esta forma el valor que buscamos lo tendremos en la posición (n, M).

Para calcular qué objetos hemos cogido una vez que hemos obtenido la solución haremos el proceso inverso. Recorreremos las filas de forma descendente y para cada objeto comprobaremos si lo hemos cogido o no.

2.2.2. Código

```cpp

/**
 * Resuelve el problema de la mochila 0-1 mediante un algoritmo de
 * programacion dinamica. El peso de cada objeto y el peso maximo de la
 * mochila deben ser enteros positivos.
 *
 * Coste: O(nM) en tiempo y espacio, n = numero de objetos, M = peso que
 * soporta la mochila.
 *
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param M Peso maximo que soporta la mochila.
 * @param solucion Indica si se coge el objeto o no.
 * @param valorSol Valor de la mochila con los objetos dados por solucion.
 */
void mochilaProgDin(std::vector<ObjetoInt> const &objetos, int M,
                    std::vector<bool> &solucion, double &valorSol) {
    const size_t n = objetos.size();

    //Creamos e inicializamos a 0 la tabla con la que resolvemos el problema
    std::vector<std::vector<double>> mochila(n + 1, std::vector<double>
            (M + 1, 0));

    //Rellenamos la tabla
    //objetos[i - 1] ya que mochila va de [1..n] y objetos va de [0, n)
    for (size_t i = 1; i <= n; ++i) {
        for (int j = 1; j <= M; ++j) {
            if (objetos[i - 1].peso > j)    //Si no cabe no lo cogemos
                mochila[i][j] = mochila[i - 1][j];
            else    //Si cabe tomamos el maximo entre cogerlo y no cogerlo
                mochila[i][j] =
                        std::max(mochila[i - 1][j],
                                mochila[i - 1][j - objetos[i - 1].peso] +
                                objetos[i - 1].valor);
        }
    }
    valorSol = mochila[n][M];

    //Calculamos que objetos hemos cogido
    for (size_t i = n; i >= 1; --i) {
        if (mochila[i][M] == mochila[i - 1][M]) //No cogido el objeto i
            solucion[i - 1] = false;
        else {    //Cogido el objeto i
```

```
            solucion[i - 1] = true;
            M -= objetos[i - 1].peso;
        }
    }
}
```

2.2.3. Análisis de costes

Vemos los costes en tiempo y memoria del algoritmo. En cuanto al tiempo tenemos un coste $O(nM)$, donde n es el numero de objetos. Esto lo obtenemos al recorrer la tabla (de tamaño nxM) realizando operaciones constantes en cada posición. El coste de recuperar los objetos seleccionados será lineal en n así que no empeora el orden que ya tenemos. En cuanto a la memoria tenemos un coste $O(nM)$ dado también por la tabla. Aunque pueda parecer que estamos ante un algoritmo polinómico en realidad tenemos un coste exponencial con respecto a los datos de entrada ya que M (lineal) lo es con respecto al número que representa (?). TODO (PREGUNTAR!!!!!!!!!!!!!!!!!!!!!!!)

2.3. Ramificación y poda

TODO——————————-

2.3.1. Descripción de la solución

2.3.2. Código

```
struct Nodo {
    std::vector<bool> sol;
    int k;
    double pesoAc, valorAc;
    double valorOpt;    //Prioridad
};

bool operator<(Nodo const &n1, Nodo const &n2) {
    return n1.valorOpt < n2.valorOpt;
}

/**
 * Calcula las estimaciones optimista y pesimista segun el estado en el que
 * nos encontremos. Presupone que los objetos estan ordenados en orden
 * decreciente de su densidad (valor/peso).
 *
 * Coste: O(n-k), n = numero de objetos, k = indice por el que vamos.
 *
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param d Vector ordenado en orden creciente de las densidades de los objetos.
 * @param M Peso maximo que soporta la mochila.
 * @param k Indice del objeto por el que vamos.
 * @param pesoAc Peso acumulado en la mochila.
 * @param valorAc Valor acumulado en la mochila.
 * @param opt Cota optimista.
```

```
 * @param pes Cota pesimista.
 */
void calculoEst(std::vector<ObjetoReal> const &objetos, std::vector<Densidad>
const &d, double M, int k, double pesoAc, double valorAc, double &opt,
                double &pes) {
    double hueco = M - pesoAc;
    const size_t n = objetos.size();
    pes = opt = valorAc;
    k++;
    for (k; k < n && objetos[d[k].obj].peso <= hueco; ++k) {
        //Cogemos el objeto k entero
        hueco -= objetos[d[k].obj].peso;
        opt += objetos[d[k].obj].valor;
        pes += objetos[d[k].obj].valor;
    }
    if (k < n) { //Quedan objetos por probar y objetos[k].peso > hueco
        //Fraccionamos el objeto k (solucion voraz)
        opt += (hueco / objetos[d[k].obj].peso) * objetos[d[k].obj].valor;
        //Extendemos a una solucion en la version 0-1
        k++;
        for (k; k < n && hueco > 0; ++k) {
            if (objetos[d[k].obj].peso <= hueco) {
                hueco -= objetos[d[k].obj].peso;
                pes += objetos[d[k].obj].valor;
            }
        }
    }
}

/**
 * Resuelve el problema de la mochila 0-1 mediante un algoritmo de
 * ramifiacion y poda.
 *
 * Coste: O(n 2^n) en tiempo y espacio, n = numero de objetos.
 *
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param M Peso maximo que soporta la mochila.
 * @param solMejor Indica si se coge el objeto o no.
 * @param valorMejor Valor de la mochila con los objetos dados por solMejor.
 */
void mochilaRamPoda(std::vector<ObjetoReal> const &objetos, double M,
                    std::vector<bool> &solMejor, double &valorMejor) {
    Nodo X, Y;
    std::priority_queue<Nodo> C;
    const size_t n = objetos.size();
    double pes;

    //Calculamos las densidades de cada objeto
    std::vector<Densidad> d(n);
    for (size_t i = 0; i < n; ++i) {
        d[i].densidad = objetos[i].valor / objetos[i].peso;
        d[i].obj = i;     //Para saber a que objeto corresponde
    }

    //Ordenamos de mayor a menor las densidades
    std::sort(d.begin(), d.end(), std::greater<Densidad>());

    //Generamos la raiz
    Y.k = -1;    //Empezamos en -1 para que vaya de [0, n)
    Y.pesoAc = 0;
    Y.valorAc = 0;
    Y.sol.resize(n, false);
```

```
    calculoEst(objetos, d, M, Y.k, Y.pesoAc, Y.valorAc, Y.valorOpt, valorMejor);

    C.push(Y);
    while (!C.empty() && C.top().valorOpt >= valorMejor) {
        Y = C.top();
        C.pop();
        X.k = Y.k + 1;
        X.sol = Y.sol;

        //Si cabe probamos a meter el objeto en la mochila
        if (Y.pesoAc + objetos[d[X.k].obj].peso <= M) {
            X.sol[d[X.k].obj] = true;
            X.pesoAc = Y.pesoAc + objetos[d[X.k].obj].peso;
            X.valorAc = Y.valorAc + objetos[d[X.k].obj].valor;
            X.valorOpt = Y.valorOpt;
            if (X.k == n) {
                solMejor = X.sol;
                valorMejor = X.valorAc;
            } else {
                C.push(X);
            }
        }

        //Probamos a no meter el objeto en la mochila
        calculoEst(objetos, d, M, X.k, Y.pesoAc, Y.valorAc, X.valorOpt, pes);
        if (X.valorOpt >= valorMejor) {
            X.sol[d[X.k].obj] = false;
            X.pesoAc = Y.pesoAc;
            X.valorAc = Y.valorAc;
            if (X.k == n) {
                solMejor = X.sol;
                valorMejor = X.valorAc;
            } else {
                C.push(X);
                valorMejor = std::max(valorMejor, pes);
            }
        }
    }
}
```

2.3.3. Análisis de costes

2.4. Algoritmo genético

TODO————————————

2.4.1. Descripción de la solución

2.4.2. Código

```
struct Cromosoma {
    std::vector<bool> crom;
    double valor;
};

bool operator<(Cromosoma const &c1, Cromosoma const &c2) {
```

```cpp
    return c1.valor < c2.valor;
}

struct IndCompValor {
    std::vector<Cromosoma> *poblacion;

    IndCompValor(std::vector<Cromosoma> *poblacion) {
        this->poblacion = poblacion;
    }

    bool operator()(int i1, int i2) {
        return poblacion->at(i1).valor > poblacion->at(i2).valor;
    }
};

const int MAX_GENERACIONES = 50;
const int TAM_ULT = 5;
const double PROB_MUTACION = 0.001;
const double PROB_CRUCE = 0.85;
const double PROB_ELITISMO = 0.05;
const double PROB_1CUARTIL = 0.5;
const double PROB_2CUARTIL = 0.8;
const double PROB_3CUARTIL = 0.95;

/**
 * Calcula la aptitud de un cromosoma. Tomamos la aptitud de cada cromosoma como
 * el valor de los objetos que tiene. Si sobrepasa el limite de peso se quitan
 * objetos aleatoriamente hasta que el cromosoma sea valido.
 *
 * Coste: O(n), n = numero de objetos.
 *
 * @param c Cromosoma a evaluar.
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param M Peso maximo que soporta la mochila.
 */
void funcAptitud(Cromosoma &c, std::vector<ObjetoReal> const &objetos,
                 double M) {
    double pesoAc, valorAc;
    pesoAc = valorAc = 0;

    //Calculamos lo que tenemos en la mochila
    for (int i = 0; i < c.crom.size(); ++i) {
        if (c.crom[i]) {
            pesoAc += objetos[i].peso;
            valorAc += objetos[i].valor;
        }
    }

    //Si no cabe en la mochila, vamos descartando aleatoriamente hasta que quepa
    size_t r = rand() % c.crom.size();
    while (pesoAc > M) {
        if (c.crom[r]) {
            c.crom[r] = false;
            pesoAc -= objetos[r].peso;
            valorAc -= objetos[r].valor;
        }
        r = (r + 1) % c.crom.size();
    }

    c.valor = valorAc;
}

```

```
/**
 * Inicializa la poblacion de forma aleatoria.
 *
 * Coste: O(mn), n = numero de objetos, m = tamanyo de la poblacion.
 *
 * @param poblacion Conjunto de cromosomas.
 * @param objetos Conjunto de objetos que tenemos disponibles.
 */
void iniPoblacion(std::vector<Cromosoma> &poblacion,
                  std::vector<ObjetoReal> const &objetos) {
    for (Cromosoma &c : poblacion)
        for (int i = 0; i < objetos.size(); ++i)
            c.crom.push_back(rand() % 2);
}

/*
 * Suponemos que seleccionados ya esta creado con el mismo tamanyo que poblacion
 * Suponemos que los cromosomas ya tienen su valor calculado
 * Coste: n^2, reducir a n logn ordenando y luego escogiendo segun
 * cierta probabilidad
 * @param poblacion
 * @param seleccionados

void funcSeleccion(std::vector<Cromosoma> const &poblacion,
                   std::vector<Cromosoma> &seleccionados) {
    //Calculamos la suma de todos los valores
    double valorAc = 0;
    for (Cromosoma const &c : poblacion)
        valorAc += c.valor;

    //Elegimos el elemento cuyo valor acumulado supere un numero aleatorio
    // entre 0 y la suma de todos los valores
    for (int i = 0; i < poblacion.size(); ++i) {
        double r = valorAc * ((double) rand() / (double) RAND_MAX);

        double valorAux = 0;
        int j;
        for (j = 0; valorAux < r; ++j)
            valorAux += poblacion[j].valor;

        if (j > 0) j--;
        seleccionados[i] = poblacion[j];
    }
}*/

/**
 * Selecciona los individuos que formaran parte de la siguiente generacion.
 * Escoge un porcentaje (PROB_ELITISMO) de los mejores cromosomas de la
 * generacion anterior y los mantiene. El resto se seleccionan segun su
 * aptitud dando mas posibilidades (PROB_xCUARTIL) a los mejores cromosomas.
 *
 * Coste: O(n logn), n = numero de objetos.
 *
 * @param poblacion Conjunto de cromosomas con las aptitudes ya calculadas.
 * @param seleccionados Vector donde almacenaremos los individuos
 * seleccionados a formar parte de la siguiente generacion. Presuponemos que
 * ya esta creado con el mismo tamanyo que @param poblacion.
 */
void funcSeleccion(std::vector<Cromosoma> &poblacion,
                   std::vector<Cromosoma> &seleccionados) {
    std::vector<int> ind(poblacion.size());
    IndCompValor comp(&poblacion);
```

```
    const size_t n = seleccionados.size();

    //Ordena la poblacion usando una estructura de indices auxiliar
    sort(ind.begin(), ind.end(), comp);

    //Selecciona primero a los mejores para preservarlos (elitismo)
    int nElit = (int) ceil(PROB_ELITISMO * n);
    int i = 0;
    for (i; i < nElit; ++i)
        seleccionados[i] = poblacion[ind[i]];

    //Completa seleccionando con mayor probabilidad a los cromosomas mas aptos
    for (i; i < n; ++i) {
        double r = (double) rand() / (double) RAND_MAX;
        size_t j = rand() % n / 4;
        if (r > PROB_3CUARTIL) {
            j += (3 * n) / 4;
        } else if (r > PROB_2CUARTIL) {
            j += n / 2;
        } else if (r > PROB_1CUARTIL) {
            j += n / 4;
        }
        seleccionados[i] = poblacion[ind[j]];
    }
}

/**
 * Cruza los elementos de la poblacion usando cruce simple. Solo cruza un
 * porcentage de los elementos (PROB_CRUCE), el resto no los modifica.
 *
 * Coste: O(nm), n = numero de objetos, m = tamanyo de la poblacion.
 *
 * @param seleccionados Cromosomas seleccionados para cruzarse.
 */
void funcCruce(std::vector<Cromosoma> &seleccionados) {
    //Cogemos los elementos de dos en dos
    for (size_t i = 1; i < seleccionados.size(); i += 2) {

        double r = (double) rand() / (double) RAND_MAX;
        if (r <= PROB_CRUCE) {  //Si se deben cruzar

            //Elegimos el punto de cruce simple
            size_t k = rand() % seleccionados[i].crom.size();

            bool aux;
            for (int j = 0; j < k; ++j) {//Cruzamos el intervalo que corresponde
                aux = seleccionados[i].crom[j];
                seleccionados[i].crom[j] = seleccionados[i - 1].crom[j];
                seleccionados[i - 1].crom[j] = aux;
            }
        }
    }
}

/**
 * Muta de 1 a 3 elementos de cada cromosoma con probabilidad PROB_MUTACION.
 *
 * Coste: O(n), n = numero de objetos.
 *
 * @param seleccionados Cromosomas seleccionados para mutar.
 */
void funcMutacion(std::vector<Cromosoma> &seleccionados) {
```

```
    for (Cromosoma &c : seleccionados) {
        double r = (double) rand() / (double) RAND_MAX;
        if (r <= PROB_MUTACION) { //Si se debe mutar

            int numMut = (rand() % 3) + 1;
            for (int i = 0; i < numMut; ++i) {  //Mutamos de 1 a 3 elementos
                size_t j = rand() % c.crom.size();
                c.crom[j] = !c.crom[j];
            }
        }
    }
}

/**
 * Comprueba si se cumple alguna condicion de terminacion. Si se ha superado
 * el maximo de generaciones o si no se ha mejorado ni la media ni la mejor
 * solucion en las ultimas TAM_ULT generaciones devuelve true.
 *
 * Coste: O(1).
 *
 * @param ultMedias Vector que contiene las ultimas TAM_ULT mejores medias.
 * @param ultMejores Vector que contiene los ultimos TAM_ULT mejores valores.
 * @param generacionAct Generacion por la que vamos.
 * @return True si se cumple alguna condicion de terminacion, false en caso
 * contrario.
 */
bool condTerminacion(std::vector<double> &ultMedias,
                     std::vector<double> &ultMejores, int generacionAct) {
    //Nunca terminamos en las primeras generaciones
    if (generacionAct < TAM_ULT)
        return false;

    //Si se ha superado el maximos de generaciones
    if (generacionAct >= MAX_GENERACIONES)
        return true;

    //Comprobamos si se ha mejorado la media o el valor mejor ultimamente
    bool mejora = false;
    double mediaAnt = ultMedias[0], mejorAnt = ultMejores[0];

    for (int i = 1; i < TAM_ULT && !mejora; ++i) {
        if (ultMedias[i] > mediaAnt || ultMejores[i] > mejorAnt)
            mejora = true;
    }

    return !mejora;
}

/**
 * Calcula el mejor valor, junto con su solucion, y la media de esta
 * generacion. Actualiza los valores de las ultimas generaciones y los
 * valores mejores hasta el momento.
 *
 * Coste: O(n), n = numero de objetos.
 *
 * @param poblacion Conjunto de cromosomas con las aptitudes ya calculadas.
 * @param ultMedias Vector con las TAM_ULT ultimas medias.
 * @param ultMejores Vector con los TAM_ULT ultimos mejores valores.
 * @param solMejor Solucion mejor hasta el momento.
 * @param valorMejor Valor de la solucion mejor hasta el momento.
 */
void calcMejores(std::vector<Cromosoma> const &poblacion, std::vector<double>
```

```cpp
&ultMedias, std::vector<double> &ultMejores, std::vector<bool> &solMejor,
                 double &valorMejor) {
    //Desplazamos los ultimos valores para actualizar
    for (int i = TAM_ULT - 1; i > 0; --i) {
        ultMedias[i] = ultMedias[i - 1];
        ultMejores[i] = ultMejores[i - 1];
    }

    //Calculamos los parametros de esta generacion
    double sumaVal = 0, mejor = -1;
    std::vector<bool> solMejorAux(solMejor.size());
    for (Cromosoma const &c : poblacion) {
        sumaVal += c.valor;
        if (c.valor > mejor) {
            mejor = c.valor;
            solMejorAux = c.crom;
        }
    }

    //Actualizamos
    ultMejores[0] = mejor;
    ultMedias[0] = sumaVal / poblacion.size();
    if (mejor > valorMejor) {
        valorMejor = mejor;
        solMejor = solMejorAux;
    }
}

/**
 * Resuelve el problema de la mochila 0-1 mediante un algoritmo genetico. No
 * se asegura la solucion optima. Se suele obtener una solucion buena en un
 * tiempo razonable.
 *
 * Coste: O(nm * MAX_GENERACIONES), n = numero de objetos, m = tamanyo de la
 * poblacion.
 *
 * @param objetos Conjunto de objetos que tenemos disponibles.
 * @param M Peso maximo que soporta la mochila.
 * @param solMejor Indica si se coge el objeto o no.
 * @param valorMejor Valor de la mochila con los objetos dados por solMejor.
 */
void mochilaGenetico(std::vector<ObjetoReal> const &objetos, double M,
                     std::vector<bool> &solMejor, double &valorMejor) {
    const size_t n = objetos.size();

    std::vector<Cromosoma> poblacion(n);
    std::vector<Cromosoma> seleccionados(n);
    std::vector<double> ultMedias(TAM_ULT);
    std::vector<double> ultMejores(TAM_ULT);

    valorMejor = -1;
    for (Cromosoma &c : seleccionados)
        c.crom.resize(n);

    iniPoblacion(poblacion, objetos);
    for (Cromosoma &c : poblacion)
        funcAptitud(c, objetos, M);
    calcMejores(poblacion, ultMedias, ultMejores, solMejor, valorMejor);
    for (int generacionAct = 0; !condTerminacion(ultMedias, ultMejores,
                                                  generacionAct); generacionAct++) {
        funcSeleccion(poblacion, seleccionados);
        funcCruce(seleccionados);
```

```
        funcMutacion(seleccionados);
        poblacion = seleccionados;
        for (Cromosoma &c : poblacion)
            funcAptitud(c, objetos, M);
        calcMejores(poblacion, ultMedias, ultMejores, solMejor, valorMejor);
    }
}
```

2.4.3. Análisis de costes

3. Comparación

Referencias