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 el peso máximo de ésta.

El problema de la mochila 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 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 $i \in \{1, \ldots, n\}$. Tendremos también una mochila que soporta un peso máximo $M > 0$.

Definimos la función indicadora $x_i \in \{0, 1\}$ que representará 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, de las cuales podremos obtener también el valor óptimo.

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, es decir, $x_i \in [0, 1]$. 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

A la hora de evaluar los algoritmos generaremos ficheros donde guardar los casos de prueba que vayamos a ejecutar y posteriormente los ejecutaremos calculando el tiempo que tardan y comprobando la solución.

Para producir los casos de prueba lo haremos de manera aleatoria. Generaremos un número de objetos a los cuales les limitaremos el peso y valor para evitar desbordamientos en casos muy grandes. Además, se guardará también el peso máximo de la mochila y el número de objetos. Incluimos a continuación el código para generar un caso de prueba donde los pesos de los objetos y el tamaño máximo de la mochila son números reales. El caso en que estos números sean enteros es análogo.

```
1  /**
2   * Genera un fichero de nombre nombreFichero con el tamanyo de la mochila
3   * M y el numero de objetos nObjetos en la primera linea separados por un
4   * espacio. Produce nObjetos aleatorios con la restriccion de que el peso y
5   * el valor esten en (0, maxPesoObjeto) o (0, maxValorObjeto).
6   * Escribe los objetos a partir de la segunda linea. Cada linea de
7   * objeto son dos numeros, peso y valor, separados por un espacio.
8   *
9   * @param nombreFichero Nombre del fichero donde guardar los datos.
10  * @param nObjetos Numero de objetos a generar.
11  * @param maxPesoObjeto Peso maximo de cada objeto.
12  * @param maxValorObjeto Valor maximo de cada objeto.
13  * @param M Peso maximo soportado por la mochila.
14  */
15 void generaCasoPruebaMochilaReal(std::string nombreFichero, int nObjetos,
16                                  double maxPesoObjeto, double maxValorObjeto,
17                                  double M) {
18     double peso, valor;
19     std::ofstream out;
20     out.open(nombreFichero);
21
22     out << M << ' ' << nObjetos << '\n';
23     for (int i = 0; i < nObjetos; ++i) {
24         peso = ((double) rand() / (double) RAND_MAX) * maxPesoObjeto;
25         valor = ((double) rand() / (double) RAND_MAX) * maxValorObjeto;
26
27         out << peso << ' ' << valor << '\n';
28     }
29
30     out.close();
31 }
```

Posteriormente leeremos los ficheros que hemos generado, guardando los objetos y los valores del caso de prueba en las estructuras que utilizaremos para llamar al algoritmo. El código en el caso de construir objetos con pesos reales es el siguiente:

```
/**
 * Construye el vector de objetos a partir de los datos de nombreFichero,
 * se presupone estructura coherente con el metodo generaCasoPruebaMochila.
 * Devuelve tambien el tamanyo maximo de la mochila M.
 *
 * @param nombreFichero Nombre del fichero del cual leer.
 * @param M Tamanyo maximo de la mochila.
 * @param objetos Vector de objetos. Se presupone vacio.
 */
void leeCasoPruebaMochilaReal(std::string nombreFichero, double &M,
                              std::vector<ObjetoReal> &objetos) {
    std::ifstream in;
    in.open(nombreFichero);
    if (!in.is_open()) {
        std::cout << "ERROR. No se ha podido abrir el fichero.\n";
        return;
    }

    int nObjetos;
    in >> M >> nObjetos;
    objetos.resize(nObjetos);

    for (int i = 0; i < nObjetos; ++i) {
        in >> objetos[i].peso >> objetos[i].valor;
    }

    in.close();
}
```

Finalmente ejecutaremos el algoritmo en cuestión varias veces para evitar las pequeñas variaciones de los tiempos, ajenas a los algoritmos, entre unas realizaciones y otras. Devolveremos por pantalla el valor de la solución y el tiempo transcurrido. El código para ejecutar un caso de prueba sobre el algoritmo voraz lo incluimos a continuación.

```
/**
 * Ejecuta el algoritmo voraz sobre el caso de prueba casoPrueba contenido en
 * el fichero nombreFichero un numero de veces dado por nIt. Muestra por
 * pantalla el valor de la solucion obtenida por el algoritmo y el tiempo
 * que ha tardado.
 *
 * @param casoPrueba Nombre del caso de prueba.
 * @param nombreFichero Fichero de donde cargar el caso de prueba.
 * @param nIt Numero de veces a ejecutar el algoritmo.
 */
void casoPruebaVoraz(std::string const &casoPrueba, std::string const &
nombreFichero, const int nIt) {
    std::vector<ObjetoReal> objetos;
    double M;
    double valorSol;

    std::cout << casoPrueba << "\n\n";
    for (int i = 1; i <= nIt; ++i) {
```

```
19
20          leeCasoPruebaMochilaReal(nombreFichero, M, objetos);
21
22          std::vector<double> solucion(objetos.size());
23
24          auto t1 = std::chrono::steady_clock::now();
25
26          mochilaVoraz(objetos, M, solucion, valorSol);
27
28          auto t2 = std::chrono::steady_clock::now();
29          auto time_span =
30            std::chrono::duration_cast<std::chrono::duration<double>>(t2 - t1);
31
32          std::cout << "Vuelta: " << i << " ValorSol: " << valorSol << '\n';
33          std::cout << "El algoritmo ha tardado " << time_span.count()
34                    << " segundos.\n\n";
35      }
36      std::cout << "--------------------------------\n";
37  }
```

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 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 $\sum_{i=1}^{n} v_i z_i \geq \sum_{i=1}^{n} v_i y_i$:

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

2.1.3. Código

```
1  /**
2   * Resuelve el problema de la mochila con objetos fraccionables mediante un
3   * algoritmo voraz. Presuponemos que la suma de los pesos de todos los
4   * objetos > M.
5   *
6   * Coste en tiempo: O(n logn), n = numero de objetos.
7   *
```

```
 * @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
    valorSol = 0;
    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. En el caso peor obtendremos un coste cuadrático $O(n^2)$, que viene dado también por la ordenación. 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, luego obtenemos un coste en memoria de $O(n)$.

2.2. Programación dinámica

En este apartado implementaremos un algoritmo de programación dinámica para el problema de la mochila en su versión 0-1. Tendremos que tener la restricción de que los pesos de los objetos y el peso total de la mochila sean números enteros. Sin embargo esto no supone una restricción real para el algoritmo ya que, siempre que los pesos sean números racionales, podremos

multiplicar todos los pesos por una constante para que resulten en números enteros (por ejemplo, suponiendo los pesos escritos como fracciones, por el mínimo común múltiplo de los denominadores). Con este algoritmo obtendremos una solución exponencial con respecto al tamaño de los datos de entrada.

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.

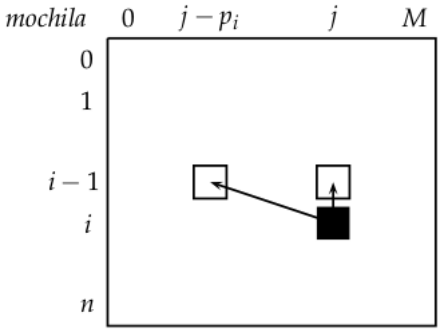
Tomamos como casos base:

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

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. Para ello recorremos la tabla por filas de forma descendente (cada vez el intervalo $[0, i]$ es más grande) y cada fila la recorremos de izquierda a derecha (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 ascendente empezando por n y para cada objeto comprobaremos si lo hemos cogido o no.

2.2.2. Código

```
/**
 * 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 la tabla 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

Veamos 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 nM) 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

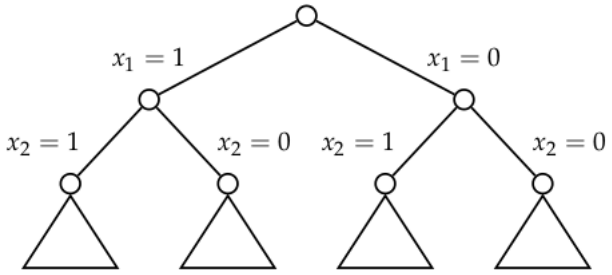
tenemos un coste $O(nM)$ dado también por la tabla. Aunque pueda parecer que estamos ante un algoritmo polinómico, en realidad tenemos un coste exponencial con respecto al tamaño de los datos de entrada. Esto se debe a que M es un número que representaremos en una cierta base b y esta representación, $\log_b(M)$, es logarítmica frente a M. De aquí deducimos que M es exponencial frente al tamaño de su representación.

2.3. Ramificación y poda

En este apartado implementaremos un algoritmo de ramificación y poda para resolver el problema de la mochila en la versión 0-1. Al igual que en programación dinámica, obtendremos un coste exponencial ($O(n2^n)$).

2.3.1. Descripción de la solución

A la hora de resolver el problema desde el punto de vista de la programación dinámica, tomaremos un árbol de decisiones binario que represente si se coge o no cada objeto.



A partir de esto seguiremos el esquema optimista/pesimista. La cola de prioridad donde iremos introduciendo los nodos será de máximos y tomaremos como prioridad el valor óptimo que calculemos. Así, la estructura de cada nodo será la siguiente:

```
1  struct Nodo {
2      std::vector<bool> sol;
3      int k;
4      double pesoAc, valorAc;
5      double valorOpt;     //Prioridad
6  };
7
8  bool operator<(Nodo const &n1, Nodo const &n2) {
9      return n1.valorOpt < n2.valorOpt;
10 }
```

Para el nodo X se cumplirá:

$$valorOpt(X) \geq valorFinal(X) \geq valorPes(X)$$

donde $valorFinal(X)$ será el valor que tendrá la mochila en la mejor solución alcanzable desde X. Además, para cualquier solución Y a la que podamos llegar desde X se cumple que:

$$valorOpt(X) \geq valorFinal(X) \geq valor(Y)$$

A la hora de calcular $valorOpt(X)$, como el problema es de maximización, tendremos que calcular una cota superior de la mejor solución alcanzable. Para ello utilizaremos el algoritmo voraz que resuelve el problema cuando los objetos se pueden partir. Como esa solución es óptima y tiene menos restricciones que la solución 0-1, no puede haber ninguna solución sin fraccionar objetos que sea mejor.

Para $valorPes(X)$ completaremos una posible solución. Incorporaremos a la mochila todos los objetos que se pueda sobre los que todavía no hayamos decidido. Para ello los tomaremos en el orden de densidades del algoritmo voraz.

2.3.2. Código

```
1  /**
2   * Calcula las estimaciones optimista y pesimista segun el estado en el que
3   * nos encontremos. Presupone que los objetos estan ordenados en orden
4   * decreciente de su densidad (valor/peso).
5   *
6   * Coste: O(n-k), n = numero de objetos, k = indice por el que vamos.
7   *
8   * @param objetos Conjunto de objetos que tenemos disponibles.
9   * @param d Vector ordenado en orden decreciente de las densidades.
10  * @param M Peso maximo que soporta la mochila.
11  * @param k Indice del objeto por el que vamos.
12  * @param pesoAc Peso acumulado en la mochila.
13  * @param valorAc Valor acumulado en la mochila.
14  * @param opt Cota optimista.
15  * @param pes Cota pesimista.
16  */
17 void calculoEst(std::vector<ObjetoReal> const &objetos, std::vector<Densidad>
18 const &d, double M, int k, double pesoAc, double valorAc, double &opt,
19                 double &pes) {
20     double hueco = M - pesoAc;
21     const size_t n = objetos.size();
22     pes = opt = valorAc;
23     k++;
24     for (k; k < n && objetos[d[k].obj].peso <= hueco; ++k) {
25         //Cogemos el objeto k entero
26         hueco -= objetos[d[k].obj].peso;
27         opt += objetos[d[k].obj].valor;
28         pes += objetos[d[k].obj].valor;
29     }
30     if (k < n) { //Quedan objetos por probar y objetos[k].peso > hueco
31         //Fraccionamos el objeto k (solucion voraz)
32         opt += (hueco / objetos[d[k].obj].peso) * objetos[d[k].obj].valor;
```

```
33          //Extendemos a una solucion en la version 0-1
34          k++;
35          for (k; k < n && hueco > 0; ++k) {
36              if (objetos[d[k].obj].peso <= hueco) {
37                  hueco -= objetos[d[k].obj].peso;
38                  pes += objetos[d[k].obj].valor;
39              }
40          }
41      }
42  }
43
44  /**
45   * Resuelve el problema de la mochila 0-1 mediante un algoritmo de
46   * ramifiacion y poda.
47   *
48   * Coste: O(n 2^n) en tiempo y espacio, n = numero de objetos.
49   *
50   * @param objetos Conjunto de objetos que tenemos disponibles.
51   * @param M Peso maximo que soporta la mochila.
52   * @param solMejor Indica si se coge el objeto o no.
53   * @param valorMejor Valor de la mochila con los objetos dados por solMejor.
54   */
55  void mochilaRamPoda(std::vector<ObjetoReal> const &objetos, double M,
56                      std::vector<bool> &solMejor, double &valorMejor) {
57      Nodo X, Y;
58      std::priority_queue<Nodo> C;
59      const size_t n = objetos.size();
60      double pes;
61
62      //Calculamos las densidades de cada objeto
63      std::vector<Densidad> d(n);
64      for (size_t i = 0; i < n; ++i) {
65          d[i].densidad = objetos[i].valor / objetos[i].peso;
66          d[i].obj = i;     //Para saber a que objeto corresponde
67      }
68
69      //Ordenamos de mayor a menor las densidades
70      std::sort(d.begin(), d.end(), std::greater<Densidad>());
71
72      //Generamos la raiz
73      Y.k = -1;   //Empezamos en -1 para que vaya de [0, n)
74      Y.pesoAc = 0;
75      Y.valorAc = 0;
76      Y.sol.resize(n, false);
77      calculoEst(objetos, d, M, Y.k, Y.pesoAc, Y.valorAc, Y.valorOpt,
78                 valorMejor);
79
80      C.push(Y);
81      while (!C.empty() && C.top().valorOpt >= valorMejor) {
82          Y = C.top();
83          C.pop();
84          X.k = Y.k + 1;
85          X.sol = Y.sol;
86
87          //Si cabe probamos a meter el objeto en la mochila
88          if (Y.pesoAc + objetos[d[X.k].obj].peso <= M) {
89              X.sol[d[X.k].obj] = true;
90              X.pesoAc = Y.pesoAc + objetos[d[X.k].obj].peso;
91              X.valorAc = Y.valorAc + objetos[d[X.k].obj].valor;
92              X.valorOpt = Y.valorOpt;
93              if (X.k == n - 1) {
94                  solMejor = X.sol;
```

```
95                      valorMejor = X.valorAc;
96                  } else {
97                      C.push(X);
98                  }
99              }
100
101             //Probamos a no meter el objeto en la mochila
102             calculoEst(objetos, d, M, X.k, Y.pesoAc, Y.valorAc, X.valorOpt, pes);
103             if (X.valorOpt >= valorMejor) {
104                 X.sol[d[X.k].obj] = false;
105                 X.pesoAc = Y.pesoAc;
106                 X.valorAc = Y.valorAc;
107                 if (X.k == n - 1) {
108                     solMejor = X.sol;
109                     valorMejor = X.valorAc;
110                 } else {
111                     C.push(X);
112                     valorMejor = std::max(valorMejor, pes);
113                 }
114             }
115         }
116 }
```

2.3.3. Análisis de costes

Analizaremos ahora los costes en tiempo y memoria del algoritmo. En cuanto al tiempo tendremos un coste en el caso peor $O(n2^n)$, donde n es el número de objetos. Cada iteración del bucle tendrá un coste lineal en n y el bucle se realizará en el caso peor 2^n veces. Esto lo podemos razonar desde la estructura de árbol binario que se va generando. En cuanto al coste en espacio tenemos un coste lineal en n por cada nodo y a lo sumo 2^{n-1} nodos ya que cada vez vamos sacando uno de la cola de prioridad y hay como mucho 2^{n-1} hojas en cada momento. Luego el coste en memoria es también de $O(n2^n)$.

2.4. Algoritmo genético

En este apartado implementaremos un algoritmo genético para resolver el problema de la mochila 0-1. Al tratarse de un algoritmo heurístico no se asegura una solución óptima aunque generalmente obtendremos una buena solución en un tiempo razonable.

2.4.1. Descripción de la solución y código de las funciones

Veamos las estructuras y las funciones que hemos utilizado a la hora de implementar el algoritmo genético. Representaremos cada solución como un cromosoma que contendrá el vector de soluciones y su valor asociado. Como estamos tratando de maximizar el valor de la mochila, un cromosoma será mejor que otro si su valor es mayor. Más adelante necesitaremos también

ordenar toda la población y lo haremos mediante un vector auxiliar de índices así que implementamos también una estructura comparadora.

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

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

A la hora de calcular la aptitud de un cromosoma calcularemos el valor total de los objetos que tiene. Como tenemos que tener en cuenta la restricción de que el total de los pesos de la mochila no puede superar el peso máximo M, será aquí donde impongamos esto. Si una solución de las que hemos obtenido al inicio o después de un cruce o mutación supera en peso a M, iremos quitando objetos de manera aleatoria hasta que cumplamos dicha condición. Para inicializar la población lo haremos también de manera aleatoria.

```
/**
 * 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, descartamos aleatoriamente hasta que quepa
    size_t r = rand() %c.crom.size();
```

```
27      while (pesoAc > M) {
28          if (c.crom[r]) {
29              c.crom[r] = false;
30              pesoAc -= objetos[r].peso;
31              valorAc -= objetos[r].valor;
32          }
33          r = (r + 1) %c.crom.size();
34      }
35
36      c.valor = valorAc;
37  }
38
39  /**
40   * Inicializa la poblacion de forma aleatoria.
41   *
42   * Coste: O(mn), n = numero de objetos, m = tamanyo de la poblacion.
43   *
44   * @param poblacion Conjunto de cromosomas.
45   * @param nObjetos Numero de objetos que tenemos disponibles.
46   */
47  void iniPoblacion(std::vector<Cromosoma> &poblacion,
48                    size_t nObjetos) {
49      for (Cromosoma &c : poblacion)
50          for (int i = 0; i < nObjetos; ++i)
51              c.crom.push_back(rand() % 2);
52  }
```

Estudiemos la elección de los cromosomas a cruzarse para obtener la siguiente generación. La idea básica detrás de lo que vamos a hacer es escoger con una probabilidad mayor los mejores cromosomas (para que el algoritmo converja más rápido), pero sin dejar de lado los menos aptos (para evitar converger en un mínimo local). Además utilizaremos elitismo, es decir, un porcentaje de los mejores cromosomas se cruzarán siempre. De esta forma nos aseguramos de que no perdemos los mejores candidatos que tenemos por el momento.

Así, primero ordenaremos la población, luego aplicaremos elitismo y finalmente completaremos el conjunto de los cromosomas seleccionando de forma aleatoria. Para elegir con mayor probabilidad los mejores candidatos, dividiremos la población en cuatro intervalos y seleccionaremos de forma ponderada individuos de cada intervalo. Cabe destacar que se puede seleccionar cada cromosoma más de una vez.

```
1   /**
2    * Selecciona los individuos que formaran parte de la siguiente generacion.
3    * Escoge un porcentaje (PROB_ELITISMO) de los mejores cromosomas de la
4    * generacion anterior y los mantiene. El resto se seleccionan segun su
5    * aptitud dando mas posibilidades (PROB_xCUARTIL) a los mejores cromosomas.
6    *
7    * Coste: O(n logn), n = numero de objetos.
8    *
9    * @param poblacion Conjunto de cromosomas con las aptitudes ya calculadas.
10   * @param seleccionados Vector donde almacenaremos los individuos
11   * seleccionados a formar parte de la siguiente generacion. Presuponemos que
```

```
 * ya esta creado con el mismo tamanyo que poblacion.
 */
void funcSeleccion(std::vector<Cromosoma> &poblacion,
                   std::vector<Cromosoma> &seleccionados) {
    std::vector<int> ind(poblacion.size());
    for (int i = 0; i < ind.size(); ++i)
        ind[i] = i;
    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 los cromosomas mas aptos
    for (i; i < n; ++i) {
        double r = (double) rand() / (double) RAND_MAX; //Entre 0 y 1
        size_t j = rand() %n / 4;  //Posicion inferior al primer cuartil
        if (r > PROB_3CUARTIL) {    //Desplazamos la posicion a entre el
            j += (3 * n) / 4;        // tercer cuartil y el final
        } else if (r > PROB_2CUARTIL) { //Entre el 2 y el 3
            j += n / 2;
        } else if (r > PROB_1CUARTIL) { //Entre el 1 y el 2
            j += n / 4;
        }
        seleccionados[i] = poblacion[ind[j]];
    }
}
```

Veamos cómo haremos los cruces y las mutaciones de los individuos seleccionados. Sólo se cruzarán un porcentaje, que tomaremos alto, de los cromosomas, y de esta manera algunos se transmitirán intactos a la siguiente generación. Para cruzar los cromosomas los iremos cogiendo por parejas, escogeremos un punto aleatorio de la cadena del cromosoma e intercambiaremos la información hasta dicho punto. Las mutaciones se harán sólo en un porcentaje muy bajo de los individuos. Si un cromosoma debe mutarse, invertiremos de forma aleatoria un porcentaje de sus elementos (si antes un objeto se cogía ahora no y viceversa).

```
/**
 * 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
```

```
15
16              //Elegimos el punto de cruce simple
17              size_t k = rand() % seleccionados[i].crom.size();
18
19              for (int j = 0; j < k; ++j)    //Cruzamos el intervalo
20                  std::swap(seleccionados[i].crom[j],
21                            seleccionados[i - 1].crom[j]);
22          }
23      }
24  }
25
26  /**
27   * Muta un porcentaje PORC_MUTACION de los elementos de cada cromosoma con
28   * probabilidad PROB_MUTACION.
29   *
30   * Coste: O(n), n = numero de objetos.
31   *
32   * @param seleccionados Cromosomas seleccionados para mutar.
33   */
34  void funcMutacion(std::vector<Cromosoma> &seleccionados) {
35      for (Cromosoma &c : seleccionados) {
36          double r = (double) rand() / (double) RAND_MAX;
37          if (r <= PROB_MUTACION) { //Si se debe mutar
38
39              int numMut = rand() % (int) std::ceil(c.crom.size() *
40                                                    PORC_MUTACION);
41              for (int i = 0; i < numMut; ++i) {
42                  size_t j = rand() % c.crom.size();
43                  c.crom[j] = !c.crom[j];
44              }
45          }
46      }
47  }
```

Como condición de terminación iremos comprobando si se ha mejorado la mejor solución que teníamos hasta el momento o la media de la población en alguna de las últimas generaciones. Si se ha mejorado alguna de ambas se seguirá con la ejecución y sino terminará. Añadiremos también un máximo de generaciones tras las cuales el algoritmo se detendrá aunque no se cumpla la condición anterior. Valoramos también el tomar como condición de terminación que la media de la población esté muy cerca del mejor valor. Sin embargo, tras probar ambas, nos decantamos por la primera ya que de esta segunda forma el algoritmo terminaba demasiado rápido aunque tomásemos diferencias muy pequeñas.

```
1   /**
2    * Comprueba si se cumple alguna condicion de terminacion. Si se ha superado
3    * el maximo de generaciones o si no se ha mejorado ni la media ni la mejor
4    * solucion en las ultimas TAM_ULT generaciones devuelve true.
5    *
6    * Coste: O(1).
7    *
8    * @param ultMedias Contiene las ultimas TAM_ULT mejores medias.
9    * @param ultMejores Contiene los ultimos TAM_ULT mejores valores.
10   * @param generacionAct Generacion por la que vamos.
11   * @return True si se cumple alguna condicion de terminacion, false en caso
12   * contrario.
```

```
 */
bool condTerminacion(std::deque<double> &ultMedias,
                     std::deque<double> &ultMejores, int generacionAct) {
    //Nunca terminamos en las primeras generaciones
    if (generacionAct < TAM_ULT)
        return false;

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

    //Comprobamos si se ha mejorado la media o el valor mejor ultimamente
    bool mejora = false;
    for (int i = 0; i < TAM_ULT - 1 && !mejora; ++i) {
        if (ultMedias[i] > ultMedias.back() ||
            ultMejores[i] > ultMejores.back())
            mejora = true;
    }

    return !mejora;
}
```

Utilizaremos una función auxiliar para actualizar los valores de la mejor solución que tenemos y las medias que usamos en la condición de parada.

```
/**
 * 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 Contiene las TAM_ULT ultimas medias.
 * @param ultMejores Contiene los TAM_ULT ultimos mejores valores.
 * @param solMejor Mejor solucion hasta el momento.
 * @param valorMejor Valor de la mejor solucion hasta el momento.
 */
void calcMejores(std::vector<Cromosoma> const &poblacion, std::deque<double>
&ultMedias, std::deque<double> &ultMejores, std::vector<bool> &solMejor,
                 double &valorMejor) {

    ultMedias.pop_back();
    ultMejores.pop_back();

    //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.push_front(mejor);
    ultMedias.push_front(sumaVal / poblacion.size());
    if (mejor > valorMejor) {
        valorMejor = mejor;
```

```
37          solMejor = solMejorAux;
38      }
39  }
```

Finalmente, nuestro programa principal será el encargado de inicializar la población e ir evolucionando las sucesivas generaciones hasta que se cumpla la condición de terminación. Devolverá como parámetro el mejor valor que hayamos encontrado en todas las generaciones y los objetos que componían la mochila para ese valor.

2.4.2. Código

```
1   /**
2    * Resuelve el problema de la mochila 0-1 mediante un algoritmo genetico. No
3    * se asegura la solucion optima. Se suele obtener una solucion buena en un
4    * tiempo razonable.
5    *
6    * Coste tiempo: O(nm * MAX_GENERACIONES), n = numero de objetos, m = tamanyo
7    *                   de la poblacion.
8    * Coste espacio: O(m)
9    *
10   * @param objetos Conjunto de objetos que tenemos disponibles.
11   * @param M Peso maximo que soporta la mochila.
12   * @param solMejor Indica si se coge el objeto o no.
13   * @param valorMejor Valor de la mochila con los objetos dados por solMejor.
14   */
15  void mochilaGenetico(std::vector<ObjetoReal> const &objetos, double M,
16                       std::vector<bool> &solMejor, double &valorMejor) {
17
18      //Inicializamos las estructuras
19      std::vector<Cromosoma> poblacion(TAM_POBL);
20      std::vector<Cromosoma> seleccionados(TAM_POBL);
21      std::deque<double> ultMedias(TAM_ULT);
22      std::deque<double> ultMejores(TAM_ULT);
23      valorMejor = -1;
24      for (Cromosoma &c : seleccionados)
25          c.crom.resize(objetos.size());
26
27      //Generamos la poblacion inicial y calculamos sus aptitudes
28      iniPoblacion(poblacion, objetos.size());
29      for (Cromosoma &c : poblacion)
30          funcAptitud(c, objetos, M);
31      calcMejores(poblacion, ultMedias, ultMejores, solMejor, valorMejor);
32
33      //Mientras que no se cumpla la condicion de terminacion vamos
34      // evolucionando las sucesivas generaciones
35      for (int generacionAct = 0;
36           !condTerminacion(ultMedias, ultMejores, generacionAct);
37           generacionAct++) {
38
39          funcSeleccion(poblacion, seleccionados);
40          funcCruce(seleccionados);
41          funcMutacion(seleccionados);
42          poblacion = seleccionados;
43          for (Cromosoma &c : poblacion)
44              funcAptitud(c, objetos, M);
45          calcMejores(poblacion, ultMedias, ultMejores, solMejor, valorMejor);
```

```
46      }
47  }
```

2.4.3. Análisis de costes

Analizaremos ahora los costes en tiempo y memoria del algoritmo. Veamos primero el coste en espacio: Guardaremos la población actual y los cromosomas seleccionados para formar parte de la siguiente, luego tendremos un coste $O(m)$ respecto a los datos de entrada (tenemos un vector de n objetos), siendo m el tamaño de la población que elijamos.

En cuanto al coste en tiempo, se trata de un algoritmo heurístico que en el caso peor terminará cuando complete el máximo de generaciones. Luego llamando n al numero de objetos, m al tamaño que elijamos de la población y g al número máximo de generaciones, obtendremos un coste $O(g \max\{nm, n \log n\})$. Justificamos el coste viendo que el bucle se ejecutará un máximo de g veces y cada iteración tiene un coste $\max\{nm, n \log n\}$ dado por las funciones de cruce y selección respectivamente.

3. Comparación

Comparemos ahora los distintos algoritmos que hemos implementado ejecutando una batería de casos de prueba según hemos construido en el apartado 1.2. Construiremos tres casos de prueba para cada número de objetos y los ejecutaremos 3 veces cada uno con los distintos algoritmos, salvo en el caso del algoritmo genético. Para este algoritmo, con el fin de poder estudiar mejor su comportamiento, ejecutaremos cada caso 10 veces. El peso y el valor máximo de cada objeto será 100 y el peso máximo soportado por la mochila será 2500. Estos valores son completamente arbitrarios ya que, salvo casos extremos, el comportamiento de los algoritmos es independiente de estos factores y la relación entre ellos.

Generaremos casos de 1000, 100000 y 1000000 objetos. Probaremos el algoritmo voraz con las tres cantidades, los algoritmos de ramificación y poda y programación dinámica solo con 1000 y 100000 objetos y el algoritmo genético solo con 1000 objetos. Estos números y el hecho de ejecutar los algoritmos solo para algunos tamaños de casos de prueba lo hemos deducido empíricamente viendo los tiempos que tardaba cada algoritmo.

Mostramos a continuación en la tabla 1 los datos recopilados en las diferentes ejecuciones. Los tiempos que se muestran son la media de las tres ejecuciones de cada caso.

	Voraz		**Ram. y Poda (Real)**	
Caso	Tiempo	Valor	Tiempo	Valor
1000A	0,000249	12803,3	0,0171	12801
1000B	0,000262	12445,7	0,0130	12444,5
1000C	0,000252	12905,5	0,0110	12904,1
100000A	0,0422	128475	38,6	128474
100000B	0,0421	133659	66,8	133659
100000C	0,0423	129030	33,8	129029
1000000A	0,460	410797		
1000000B	0,454	408655		
1000000C	0,457	407775		

Tabla 1: Tiempos (en segundos) y valores de las ejecuciones de los distintos casos de prueba sobre los algoritmos voraz y ramificación y poda en su variante con números reales.

Observamos que, como era de esperar, el algoritmo voraz es el más rápido con diferencia e incluso tomando tamaños de 1000000 de elementos tarda menos de medio segundo. En comparación tenemos el algoritmo de ramificación y poda cuando trabaja con números reales (veremos a continuación por qué diferenciamos). Este algoritmo exponencial ya para el caso de 100000 de elementos tarda entre medio minuto y un minuto dependiendo del caso en particular. Esta diferencia considerable entre unos casos (el caso 100000A y 100000C) y otros (el 100000B) del mismo tamaño la podemos achacar a la cantidad de ramas que podrá podar el algoritmo en su ejecución.

Si nos fijamos ahora en los valores que devuelven ambos algoritmos veremos como el valor del algoritmo voraz siempre será mayor o igual que el del algoritmo de ramificación y poda. Esto era esperable desde el planteamiento del problema para ambos casos ya que en el primero permitimos trocear los objetos y en el segundo no. Aún así, cabe destacar la poca diferencia que hay entre ambos valores. Esto se deberá a la forma de construir los casos de prueba, de forma aleatoria, ya que se pueden construir fácilmente casos pequeños donde esta diferencia sea muy grande.

	Ram. y Poda (Int)		**Prog. Din.**	
Caso	Tiempo	Valor	Tiempo	Valor
1000A	0,0602	13165,8	0,0520	13165,8
1000B	0,0644	12972,5	0,0512	12972,5
1000C	0,0632	13647,5	0,0511	13647,5
100000A	5,6010	159232	5,4551	159232
100000B	5,5077	156013	5,4382	156013
100000C	5,5036	157786	5,4287	157786

Tabla 2: Tiempos (en segundos) y valores de las ejecuciones de los distintos casos de prueba sobre los algoritmos.

A la hora de comparar los dos algoritmos exponenciales, vemos que el de programación dinámica es sustancialmente más rápido que el de ramificación y poda cuando este último trabaja con números reales. Sin embargo, sorprendentemente, si ejecutamos este último algoritmo con números enteros en lugar de reales, los tiempos bajan drásticamente como podemos ver en la comparación de la tabla 2. Así, vemos que en este caso ambos algoritmos tienen resultados sin diferencias notables entre uno y otro. Evidentemente, ambos algoritmos devuelven el mismo valor como resultado.

Finalmente nos falta por analizar el comportamiento del algoritmo genético. En la tabla 3 hemos recogido los tiempos y los valores devueltos en cada ejecución de los distintos casos de prueba de tamaño 1000, así como las medias correspondientes para poder estimar mejor los resultados. Como se puede ver a simple vista, tanto los tiempos como los valores obtenidos distan mucho de lo que desearíamos. Aunque teóricamente este algoritmo es polinómico, en la práctica hemos comprobado que se comporta bastante peor que los algoritmos exponenciales que hemos estudiado.

Variando los parámetros (probabilidades, iteraciones, etc.) asociados al algoritmo genético según se recoge en la tabla 4, conseguimos distintos tiempos y valores. Para obtener valores en torno al 50 % del resultado óptimo, el algoritmo tarda 1,5 segundos aproximadamente. Si ya buscamos una precisión del 75 %, el tiempo se dispara a 4,5 segundos.

	Genético			
	Parámetros A		Parámetros B	
Caso	Tiempo	Valor	Tiempo	Valor
1000A	1,3657	6365,26	4,1778	8404,04
	0,8804	5698,52	4,8370	8083,7
	1,5510	7309,02	5,8133	9103,2
	0,8246	5592,46	3,2265	8385,36
	1,6826	6509,06	4,2955	8745,99
	1,5860	6647,46	6,2247	9080,07
	1,3431	6935,55	1,6927	7163,77
	0,9354	6224,15	2,3283	7437,2
	1,6642	6948,62	5,4778	8840,18
	0,3325	4644,46	4,2357	8303,36
Media	1,2165	6287,456	4,2309	8354,687
1000B	1,7015	7019,75	5,9562	8004,94
	0,4515	4870,14	5,5291	8201,76
	1,2480	6470,11	4,4462	7746,2
	0,9441	5456,98	5,2488	9069,21
	1,7079	6849,38	2,6198	7443,36
	1,6683	7066,49	6,8396	9391,17
	1,6927	6785,64	2,9504	7626,07
	0,7423	5186,24	3,5481	8588,01
	1,4724	6222,11	4,2377	8088,93
	2,8176	7136,79	4,7617	8474,92
Media	1,4446	6306,363	4,6137	8263,457
1000C	1,6137	6788,4	3,3136	8592,67
	1,6749	7192,69	2,9546	7163,36
	0,9530	5311,1	2,1096	6864,59
	2,6441	8479,58	3,6007	7977,89
	1,9831	7029,83	2,7865	6779,97
	1,8745	6610,03	7,1270	9328,72
	1,9290	7331,75	5,3016	9095,04
	0,7222	5305,44	5,3340	8846,7
	1,3151	6265,06	7,0775	9920,19
	1,8414	7367,8	4,6577	8702,84
Media	1,6551	6768,168	4,4263	8327,197

Tabla 3: Tiempos (en segundos) y valores de las distintas ejecuciones de cada caso de prueba sobre el algoritmo genético en sus versiones con parámetros A y B.

	Parámetros A	Parámetros B
MAX_GENERACIONES	1000	2000
TAM_POBL	100	100
TAM_ULT	10	30
PROB_MUTACION	0,05	0,05
PORC_MUTACION	0,01	0,01
PROB_CRUCE	0,85	0,95
PROB_ELITISMO	0,1	0,05
PROB_1CUARTIL	0,5	0,5
PROB_2CUARTIL	0,8	0,8
PROB_3CUARTIL	0,95	0,95

Tabla 4: Parámetros para las dos ejecuciones del algoritmo genético. Los nombres se corresponden con los del código descrito en la sección 2.4.

Concluimos que, al contrario de lo que podría parecer en un principio, este algoritmo genético en particular no nos es útil para resolver el problema. Sin embargo, obtenemos tiempos muy razonables para cantidades grandes de objetos con los algoritmos exponenciales, además de hallar la solución óptima. Por lo tanto vemos que, entre los algoritmos que hemos estudiado, los más adecuados al problema de la mochila son estos últimos. Sin diferenciar entre programación dinámica y ramificación y poda salvo en el caso de escoger parámetros enteros o reales.

Referencias

[1] E. Horowitz, S. Sahni y S. Rajasekaran. *Computer Algorithms*. Tercera edición. Computer Science Press, 1998. Capítulos 4, 5 y 8.

[2] Hristakeva-Shrestha. *Solving the 0-1 Knapsack Problem with Genetic Algorithms*. Simpson College. `http://www.micsymposium.org/mics_2004/Hristake.pdf`

[3] N. Martí Oliet, Y. Ortega Mallén y J. A. Verdejo López. *Estructuras de datos y métodos algorítmicos: ejercicios resueltos*. Colección Prentice Practica, Pearson/Prentice Hall, 2003. Capítulo 13.

[4] N. Martí Oliet, Y. Ortega Mallén y A. Verdejo. *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. Segunda edición. Garceta, 2013. Capítulos 12 y 15.

[5] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Third, revised and extended edition. Springer, 1995.

[6] M. Mitchell. *An Introduction to Genetic Algorithms*. Fifth printing. The MIT Press, 1999.

[7] R. Neapolitan. *Foundations of Algorithms*. Quinta edición. Jones and Barlett, 2015. Capítulos 4 y 10.

[8] R. Neapolitan y K. Naimipour. *Foundations of Algorithms using C++ pseudocode*. Jones and Barlett Publishers, 2004. Capítulos 3 y 6.

[9] Artículos de la enciclopedia libre Wikipedia:

- Problema de la mochila:
  `https://es.wikipedia.org/wiki/Problema_de_la_mochila`
- Lista de 21 problemas NP-completos de Karp:
  `https://es.wikipedia.org/wiki/Lista_de_21_problemas_NP-completos_de_Karp`
- Algoritmos genéticos:
  `https://en.wikipedia.org/wiki/Genetic_algorithm`