Práctica 3. Knapsack problem

Autor: Marcos Hidalgo Baños **Fecha:** 11 de noviembre del 2022

Descripción del problema.

El problema de la *mochila con múltiples ítems* trata de obtener la **mejor solución** dada una situación similar a la del llenado de una mochila (cuya capacidad es limitada) con objetos de peso y cantidad determinadas, descartando todas aquellas combinaciones que aún cumpliendo dichas restricciones no sean la óptima.

Se nos propone implementar un total de tres clases JAVA las cuales emplean determinadas técnicas del diseño de algoritmos para dar solución a nuestro problema. Todas ellas heredan de la clase padre Mochila.java y reescribirán su método resolver. Estas clases a entregar son:

- **MochilaFB**, que escogerá la mejor combinación de todas las obtenidas, comprobando que cumplen las restricciones, de manera progresiva desde la primera hasta la última.
- **MochilaPD**, que creará una tabla de soluciones parciales en base a la ecuación de recurrencia del propio problema. Finalmente, escogerá aquellas celdas que cumplan unas determinadas condiciones, puesto que pertenecerán a la solución.
- **MochilaAV**, que empleará el concepto densidad (valor/peso) como medida para distinguir si un objeto es buen candidato o no a ser incorporado a la solución.

Código implementado y decisiones de diseño.

➔ Un aspecto aplicable a todas las soluciones, es la pobreza a la hora de modularizar las partes del código mediante métodos bien definidos (enfoque demasiado secuencial).

➔ En PD y AV, realizo un preprocesado del array de items (dado como un listado) pasándolo a un array de enteros y así poder trabajar directamente con los índices de los objetos.

➔ En AV, introduzco algunas variantes que serán comentadas junto con el propio algoritmo.

MochilaFB.java

Solución naive del problema de las mochilas múltiples.

Emplea una llamada recursiva a la función **calcSol** que comprueba que no haya más unidades del mismo objeto que puedan ser introducidos como parte de la solución.

```
public class MochilaFB extends Mochila {


    public SolucionMochila resolver(ProblemaMochila pm) {
        SolucionMochila sm = new SolucionMochila();
        sm = calcSol(sm, pm.getItems(), new ArrayList<Integer>(), 0, pm.pesoMaximo);
        return sm;
    }

    private SolucionMochila calcSol(SolucionMochila sm, ArrayList<Item> items,
                                    ArrayList<Integer> res, int iteracion, int maxW) {

        int mejorSuma = 0, mejorPeso = 0;

        if (iteracion == items.size()) {    // Ultima iteracion de la lista de items
            for (int i = 0; i < res.size(); i++) {  // Vemos la solucion que tenemos
                mejorSuma += items.get(i).valor*res.get(i); // Obtenemos su suma de valores
                mejorPeso += items.get(i).peso*res.get(i); // Obtenemos su peso de los items
            }
            if (sm.sumaValores < mejorSuma && maxW >= mejorPeso) {  // Encontramos una mejor solucion
                sm.sumaValores = mejorSuma; // Actualizamos su suma de valores
                sm.sumaPesos   = mejorPeso; // Actualizamos su peso de los items
                sm.solucion    = new ArrayList<>(res); // Devolvemos la mejor solucion
            }
            return sm;

        } else {
            if (iteracion == 0) {   //Primera iteracion
                res = new ArrayList<Integer>(); // Creamos la solucion que iremos llenando
                for (int i = 0; i < items.size(); i++) { // Inicializamos con ceros dicha solucion
                    res.add(0);
                }
            }
            for (int i = 0; i <= items.get(iteracion).unidades; i++) { // Para cada item con unidades
                res.set(iteracion, i);  // Avanzamos la iteracion correspondiente
                sm = calcSol(sm, items, res, iteracion+1, maxW); // Continuamos visitando soluciones
            }
            return sm;
        }
    }
}
```

MochilaPD.java

En esta primera parte del algoritmo, se genera el vector de ítems y la tabla que se rellenará en función de las condiciones de las ecuaciones de recurrencia. Comprobamos tal y como indica la última rama, cuál es el mayor número de unidades de un objeto que podemos introducir.

```java
public class MochilaPD extends Mochila {

    public SolucionMochila resolver(ProblemaMochila pm) {
        SolucionMochila sm = new SolucionMochila();

        ArrayList<Item> items = new ArrayList<>();
        items.add(new Item(0, 0, 0, 0));
        items.addAll(pm.getItems());
        Item it;

        // Inicializamos la tabla de programacion dinamica
        int[][] tabla = new int[items.size()][pm.getPesoMaximo()+1];
        int valor, test;

        // Rellenamos la tabla de programacion dinamica
        for (int i = 0; i < tabla.length; i++) {
            it = items.get(i);
            valor = 0;
            for (int j = 0; j < tabla[i].length; j++) {
                // Aplicamos la ecuacion de recurrencia
                /** Rama1. Filas y columnas a cero */
                if (i==0 || j==0) {
                    tabla[i][j] = 0;
                /** Rama2. Actual igual al de fila anterior */
                } else if (j < it.peso) {
                    tabla[i][j] = tabla[i-1][j];
                /** Rama3. Maximo k unidades del item */
                } else {
                    int k = 0;
                    // Vemos el mayor valor de k podemos obtener
                    while (k <= it.unidades && j-(k*it.peso) >= 0) {
                        test = tabla[i-1][j-(k*it.peso)] + k*it.valor;
                        if (valor < test) { // Nos quedamos con el mayor
                            valor = test;
                        }
                        k++;
                    }
                    tabla[i][j] = valor;
                }
            }
        }
```

MochilaPD.java (Continuación)

Finalmente, inicializamos y rellenamos el vector solución junto con el propio objeto solución.

```
        // Inicializamos la solucion
        sm.solucion = new ArrayList<>();
        for (int i = 0; i < items.size()-1; i++) {
            sm.solucion.add(0);
        }

        // Rellenamos la solucion
        int j = pm.pesoMaximo;
        int i = items.size()-1;
        while (i > 0 && j > 0) { // Desde el ultimo item al primero
            if (tabla[i-1][j] < tabla[i][j]) { // Es parte de la solucion
                sm.solucion.set(i-1, sm.solucion.get(i-1) + 1);
                j = j-items.get(i).peso;
                if (sm.solucion.get(i-1) == items.get(i).unidades) {
                    i--; // Si es el ultimo item, pasamos al siguiente
                }
            } else { // No esta en la solucion
                i--;
            }

        }

        // Finalmente, actualizamos el peso y valor total de la solucion
        sm.sumaPesos = 0;
        for (int t = 0; t < sm.solucion.size(); t++) {
            sm.sumaPesos += sm.solucion.get(t) * items.get(t+1).peso;
        }
        sm.sumaValores = tabla[items.size()-1][pm.getPesoMaximo()];
        return sm;
    }
}
```

MochilaAV.java

Como algoritmo de ordenación utilizamos Mergesort, pero introducimos un nuevo vector de índices que nos permite trackear la nueva posición de cada ítem tras ser reordenado (es su identificador). Además, no empleo un vector de densidades, sino que en el propio Mergesort hago que el criterio de ordenación sea la densidad entre los pares de ítems.

```
public class MochilaAV extends Mochila {

    public SolucionMochila resolver(ProblemaMochila pm) {
        SolucionMochila sm = new SolucionMochila();

        // Obtenemos los objetos del listado
        Item[] items = new Item[pm.items.size()];
        // Junto con su indice correspondiente
        int indices[] = new int[pm.items.size()];
        for (int i = 0; i < pm.items.size(); i++) {
            items[i] = pm.items.get(i);
            indices[i] = i;
        }

        // Ordenamos dichos arrays mediante mergesort
        mergeSort(items, indices, 0, items.length-1);

        // Inicializamos la solucion
        sm.sumaPesos = 0; sm.sumaValores = 0;
        sm.solucion = new ArrayList<>();
        for (int i = 0; i < items.length; i++) {
            sm.solucion.add(0);
        }

        int it = 0;
        // Rellenamos la solucion
        while (it < items.length) {
            if (sm.sumaPesos + items[it].peso <= pm.pesoMaximo) { // Es parte de la solucion
                sm.sumaPesos += items[it].peso;
                sm.solucion.set(indices[it], sm.solucion.get(indices[it])+1);
                sm.sumaValores += items[it].valor;
                if (sm.solucion.get(indices[it]) == items[it].unidades) {
                    it++; // Si es el ultimo item, pasamos al siguiente
                }
            } else { // No esta en la solucion
                it++;
            }
        }

        return sm;
    }
```

MochilaAV.java (Continuación)

Implementación tradicional de mergesort, pero incluye algunas modificaciones ya comentadas.

```
    private void mergeSort(Item[] items, int[] indices, int inf, int sup) {
        if (inf < sup){
            mergeSort(items,indices,inf,(inf+sup)/2);
            mergeSort(items,indices,(inf+sup)/2+1,sup);
            mezclar(items,indices,inf,(inf+sup)/2,sup);
        }
    }

    private void mezclar(Item[] items, int[] indices, int inf, int medio, int sup) {
        int i = inf; int j = medio+1;
        Item[] b = new Item[sup-inf+1];
        int[] idx = new int[sup-inf+1];
        int k = 0;

        while (i <= medio && j <= sup){
            Double densidad1 = (double) items[i].valor/items[i].peso;
            Double densidad2 = (double)items[j].valor/items[j].peso;
            if (densidad1 >= densidad2) {
                    b[k] = items[i];
                    idx[k] = indices[i];
                    i++;
            } else {
                b[k] = items[j];
                idx[k] = indices[j];
                j++;
            }
            k++;
        }
        while (i <= medio) {
            b[k] = items[i];
            idx[k] = indices[i];
            i++; k++;
        }
        while (j <= sup) {
            b[k] = items[j];
            idx[k] = indices[j];
            j++; k++;
        }
        k = 0;
        for (int f = inf; f <= sup; f++) {
            items[f] = b[k];
            indices[f] = idx[k];
            k++;
        }
    }
}
```

Análisis de la complejidad del programa.

Como es de esperar, partimos del principio que el algoritmo de Fuerza Bruta debería ser de mayor complejidad temporal que sus semejantes de Programación Dinámica y Algoritmos Voraces. Esto se puede corroborar de la siguiente manera:

En Fuerza Bruta.

Realizamos siempre una primera comprobación para saber si estamos en el límite de unidades de un objeto, para luego o bien analizar la solución que tenemos o bien seguir probando combinaciones. Esto dará lugar, en el peor de los casos, en el que la mejor solución sea la última por comprobar, a las **combinaciones de n elementos con m posibles valores**.

En Programación Dinámica.

Gracias a que realizamos el llenado de la tabla de soluciones parciales, podemos reducir el esfuerzo computacional en gran medida. Las operaciones que consumen mayor esfuerzo son:

- Creación y llenado de la tabla (y procesamiento del vector de ítems) → **O(2n * m)**
- Inicialización de la solución y su llenado → **O(3n)**

En Algoritmos Voraces.

De manera bastante similar a PD, podemos realizar bastantes mejoras a nuestro algoritmo:

- Creación y llenado del vector items e índices → **O(n)**
- Ordenación con Mergesort del vector items → **O(n logn)**
- Inicialización de la solución y su llenado → **O(2n)**

Enlace al repositorio de Github con el código fuente.

github.com/MarkosHB/Analisis-y-diseno-de-Algoritmos/blob/main/Knapsack

- Marcos Hidalgo Baños -