PROBLEMA DE LA «0-1-MOCHILA»

DANIEL STEVEN FLORIDO, JUAN SEBASTIAN GALEANO

Resumen. En este documento se presenta el problema la 0-1-mochila con programación dinámica. Además, se presenta el análisis del problema, diseño de la solución y los algoritmos dinámicos que la solucionan.

Parte 1. Análisis y Diseño del problema

1. Análisis

1.1. Problema de la mochila. El problema de la mochila es una problema de optimización combinatoria, es decir, que busca la mejor solución entre un conjunto de soluciones. Se modela como una mochila que aguanta un peso máximo W y un conjunto de objetos i que tienen un peso w_i y un valor V_i, cada objeto puede repetirse un número k de veces. Por lo tanto, se busca que se guarde en la mochila el conjunto de objetos que tengan el mayor valor sin sobrepasar el peso máximo de la mochila.

1.2. Problema de 0-1-Mochila. El problema de 0-1-Mochila se define exactamente igual que el anterior pero con la peculiaridad de que los objetos no pueden repetirse y no se puede guardar una fracción de dicho objeto, es decir que $k = 1$ para cada objeto, el nombre se refiere a que el objeto puede estar 1 o no estar 0 en la mochila en la solución final. A continuación se modelan formalmente los conjuntos para la solución de este problema:

- $W \leftarrow$ peso máximo que soporta la mochila, por definición se dice que $W > 0$ ya que no tiene sentido tener una mochila que no soporte objetos.
- $w \leftarrow$ conjunto de pesos relacionado con el número de objetos, se define que $w_i > 0$ ya que no tiene sentido tener objetos con peso negativo o directamente que no pesan.
- $v \leftarrow$ conjunto de valores relacionado con el número de objetos, se que $v_i > 0$ ya que no vamos a meter objetos que no aporten valor o que nos hagan perder dicho valor.
- se definen los objetos como una secuencias $1 < i \leq n$ siendo n el número de objetos que se tienen, también se define n como el tamaño de los conjuntos w y v.

la solución se define matemáticamente como:

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

Date: 13 de abril de 2023.

tal que:

$$\sum_{i=1}^{n} w_i \leq W$$

siendo n el numero de objetos que cumplen con dicha relación.

2. Diseño

A continuación se presentan las condiciones de entrada para el correcto funcionamiento de la solución y su respectiva salida:

Definición. Entradas:

1. $w \leftarrow$ conjunto de pesos de los objetos a guardar en la mochila y se debe garantizar que los pesos individuales w_i sean mayores a cero.
2. $v \leftarrow$ conjunto de valor de los objetos a guardar en la mochila y se debe garantizar que los valores individuales v_i sean mayores a cero.
3. $W \leftarrow$ numero natural mayor a cero que representa el valor máximo del peso que aguanta la mochila y se debe garantizar que $W > 0$

Definición. Salidas:

1. $G \leftarrow$ numero natural que representa el valor máximo que se puede llevar en la mochila.
2. $I \leftarrow$ el conjunto de objetos que sumados dan el valor máximo G

Parte 2. Algoritmos

3. Solución recurrente

3.1. Algoritmo recurrente. Mediante este algoritmo se llama a otra función auxiliar, como parte de la programación dinámica se añade este paso pero la salida solo sera el valor máximo G mas no el conjunto de solución I

Algorithm 1 01KNAPSACK Recurrente

```
1: procedure 01KNAPSACK(w, v, W)
2:     return 01KNAPSACKAUX(w, v, |w|, W)
3: end procedure
```

Mediante el siguiente algoritmo se definen tres casos bases, el primero donde no hay mas ítem donde su valor sera 0, el segundo donde el peso del objeto sea mayor al peso actual de la mochila donde el valor sera el valor del anterior objeto a este, es decir que no se añadió el objeto, el tercero donde el peso del objeto es menor o igual al actual de la mochila en este caso se añade el objeto y se revisa si es el valor máximo entre el valor anterior y el valor del de lo que queda por meter sin que sean mayores al peso actual.

Algorithm 2 Recurrente auxiliar

```
1: procedure 01KNAPSACKAUX(w, v, i, j)
2:     if i = 0 then
3:         return 0
4:     else if w_i > j then
5:         return 01KNAPSACKAUX(w, v, i − 1, j)
6:     else
7:         return max(
8:         01KNAPSACKAUX(w, v, i − 1, j),
9:         v_i+ 01KNAPSACKAUX(w, v, i − 1, j − w_i))
10:    end if
11: end procedure
```

4. Solución Tabla de memorización

4.1. Algoritmo Tabla de memorizacion. Tomamos en cuenta las variables que se se mueven en este caso i y j. i representa los objetos y j el valor del peso que puede aguantar actualmente la mochila. Por lo tanto, creamos una tabla de memorización de tamaño $i * j$ y los retornos ahora serán guardados en la tabla. Dicha tabla sera inicializada en -inf.

Algorithm 3 Tabla de memorización

```
1: procedure 01KNAPSACKM(w, v, W)
2:     M ← matriz[|w| + 1][W + 1]
3:     M ← −∞
4:     return 01KNAPSACKMAUX(w, v, M, |w|, W)
5: end procedure
```

A continuación el algoritmo de Auxiliar que devuelve solo la tabla de memorizacion

Algorithm 4 Memorización auxiliar

```
1: procedure 01KNAPSACKAUX(w, v, M, i, j)
2:     if M_(i,j) = −inf then
3:         if i = 0 then
4:             M_(i,j) ← 0
5:         else if w_i > j then
6:             M_(i,j) ←01KNAPSACKAUX(w, v, M, i − 1, j)
7:         else
8:             M_(i,j) ←max(
9:             01KNAPSACKAUX(w, v, M, i − 1, j),
10:            v_i+ 01KNAPSACKAUX(w, v, M, i − 1, j − w_i))
11:        end if
12:    end if
13:    return M_(i,j)
14: end procedure
```

5. Solución Iterativa

5.1. Algoritmo Iterativo. A continuación se va a pasar la solución a una función iterativa donde no se necesite otra función auxiliar, se inicializa igual la tabla de memorización pero en este caso llena de 0 y se recorre desde los casos base. Ademas, se agregan dos nuevas condiciones, si $i = 1$ quiere decir que es el primer elemento que se agrega por lo que se confirmara si cabe en la mochila o no y se agregara o no respectivamente, luego se agrega $i > 1$ para el resto de objetos que funciona como en los algoritmos anteriores. Por lo tanto, la solución estará ubicada en $M_{|w|,W}$

Algorithm 5 Algoritmo iterativo

```
1: procedure 01KNAPSACKI(w, v, W)
2:     M ← matriz[|w| + 1][W + 1]
3:     M ← 0
4:     for i ← 1 to |w + 1| do
5:         for j ← 1 to |W| do
6:             if i = 1 then
7:                 if w_i < j then
8:                     M_(i,j) ← v_i
9:                 end if
10:            else if i > 1 then
11:                if w_i > j then
12:                    M_(i,j) ← M_(i−1,j)
13:                else
14:                    M_(i,j) ← Max(M_(i−1,j), v_i + M_(i−1,j−w_i))
15:                end if
16:            end if
17:        end for
18:    end for
19:    return M_(|w|,W)
20: end procedure
```

6. Solución Completa con backtraking

6.1. Algoritmo con backtraking. A continuación se presenta el algoritmo que representa el algoritmo que resuelve el problema totalmente, ya que, muestra los objetos con los que se obtiene el mayor valor sin sobrepasar el peso máximo de la mochila. Para esto agregamos un bucle adicional que recorre la tabla de memorización desde el resultado, si el valor $M_{(i,j)}$ es igual al valor $M_{(i-1,j)}$ entonces se ignora el valor, si son distintos; guardamos $i - 1$ y cambiamos los valores de j por $j - w_i$ e i por $i - 1$ y seguimos iterando

Algorithm 6 Algoritmo Backtraking

```
1: procedure 01KNAPSACKBACK(w, v, W)
2:     M ← matriz[|w| + 1][W + 1]
3:     M ← 0
4:     for i ← 1 to |w + 1| do
5:         for j ← 1 to |W| do
6:             if i = 1 then
7:                 if w_i < j then
8:                     M_(i,j) ← v_i
9:                 end if
10:            else if i > 1 then
11:                if w_i > j then
12:                    M_(i,j) ← M_(i−1,j)
13:                else
14:                    M_(i,j) ← Max(M_(i−1,j), v_i + M_(i−1,j−w_i))
15:                end if
16:            end if
17:        end for
18:    end for
19:    j ← W
20:    S ← subsecuence
21:    for i ← |w + 1| to 1 i − − do
22:        if M_(i−1,j) ≠ M_(i,j) then
23:            if M_(i,j) = v_i + M_(i−1,j−w_i) then
24:                S ← S + (i, w_i)
25:                j ← j − w_i
26:            end if
27:        end if
28:    end for
29:    return M_(|w|,W), S
30: end procedure
```