Mochila 0/1

Pedro O. Pérez M., PhD.

Análisis y diseño de algoritmos avanzados
Tecnológico de Monterrey

pperezm@tec.mx

12-2021

Contenido

Mochila 0/1

Sean N objetos no fraccionables de pesos w_i, y beneficios v_i, y sea C el peso máximo que puede llevar la mochila. El problema consiste en llenar la mochila con objetos, tal que se maximice el beneficio. Por ejemplo, W = [10, 4, 6, 12], V = [100, 70, 50, 10], y C = 12. ¿Cuál es la beneficio máximo que podemos obtener?

Usando Programación Dinámica

Procedure 1 KNAPSACK_DP

Input: $W, V : Array, C : Integer$
$M : Matrix[0...S.length][0..C]$
$n \leftarrow W.length$
$INIT(M, 0)$
for $i \leftarrow 1$ to n **do**
  for $j \leftarrow 1$ to C **do**
    /* There is not space in the container */
    if $j < W[i]$ **then**
      $M[i][j] \leftarrow M[i-1][j]$
    else
      $M[i][j] \leftarrow MAX(V[i] + M[i-1][j - W[i]], M[i-1][j])$
    end if
  end for
end for
return $M[n][C]$

Usando Divide y Conquista

Procedure 2 KNAPSACK_D&C

Input: $W, V : Array, n, remainingWeigth : Integer$

```
if n ← 0 or W[n − 1] > remainingWeigth then
    return 0
else
    return MAX(
    KNAPSACK_D&C(W, V, n − 1, remainingWeigth),
    V[n − 1] + KNAPSACK_D&C(W, V, n − 1, remainingWeigth − W[n − 1]))
end if
```

Usando Backtracking

El problema de la Mochila al ser un problema de Selección puede ser resuelto con la técnica de Backtracking, pero al ser un problema de optimización tendremos que tomar esto en consideración para poder implementarlo. Lo primero que hay que diseñar es el árbol de búsqueda de soluciones, el cual tendrá la siguiente conformación:

- El árbol tendrá n+1 niveles como máximo, donde n es la cantidad de objetos.
- Cada nivel indica un objeto a incluir en la mochila.
- Cada nodo tiene dos hijos; el de la izquierda que indica que si incluye al siguiente objeto y el de la derecha que indica que no incluye al siguiente objeto.
- Cada nodo tendrá que almacenar el valor acumulado hasta es punto, el peso acumulado hasta ese punto, y valor posible a acumular.

Usando Backtracking

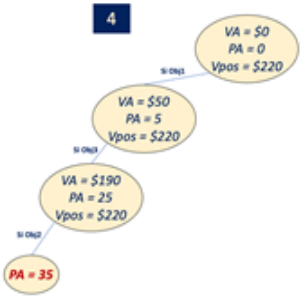
El criterio de selección del nodo será:

- Si el peso acumulado de los objetos incluidos no excede a la capacidad de la mochila.
- Si el valor posible a acumular es mayor al mejor valor acumulado hasta ese momento.

Para la estimación del posible valor a acumular, se sabe el peso y valor acumulador en el nivel i. A partir de este punto, se debe analizar cuantos de los siguientes objetos se podrían acumular sin exceder el peso de la mochila, asumiendo que el objeto en el nivel k es el que excede la mochila. Estratégicamente conviene que los objetos estén ordenados en forma descendente de acuerdo con su valor proporcional ($valor_i/peso_i$). El cálculo del valor posible será:

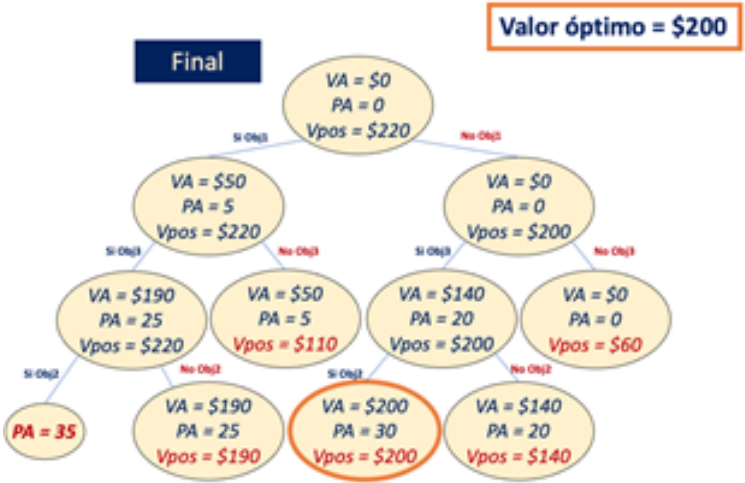
- El valor acumulado por los objetos ya incluidos, más...
- El valor de los objetos i+1 hasta k-1 (nos que caben sin exceder el peso soportado por la mochila, más...
- El valor proporcional del objeto k por el peso que resta de la mochila.

PESO MOCHILA = 30
Objeto$_1$, valor$_1$: \$50, peso$_1$: 5, valor$_1$/peso$_1$ = \$10
Objeto$_3$, valor$_3$: \$140, peso$_3$: 20, valor$_3$/peso$_3$ = \$7
Objeto$_2$, valor$_2$: \$60, peso$_2$: 10, valor$_2$/peso$_2$ = \$6



Valor óptimo = \$190
Valor posible a acumular:

PA < 30 && Vpos > Valor óptimo


Valor óptimo = $200
Final
VA = $0
PA = 0
Vpos = $220
Si Obj3
No Obj3
VA = $50
PA = 5
Vpos = $220
VA = $0
PA = 0
Vpos = $200
Si Obj3
No Obj3
Si Obj3
No Obj3
VA = $190
PA = 25
Vpos = $220
VA = $50
PA = 5
Vpos = $110
VA = $140
PA = 20
Vpos = $200
VA = $0
PA = 0
Vpos = $60
Si Obj2
No Obj2
Si Obj2
No Obj2
PA = 35
VA = $190
PA = 25
Vpos = $190
VA = $200
PA = 30
Vpos = $200
VA = $140
PA = 20
Vpos = $140

Procedure 3 CALCULATING_POSSIBLE_VALUE

Input: $W, V : Array, weight, i, n, auxV, auxW : Integer$

$k \leftarrow i + 1$
while $k < n$ **do**
$auxV \leftarrow V[k], auxW \leftarrow V[k], k \leftarrow k + 1$
end while
if $k < n$ **then**
$auxV \leftarrow auxV + ((weight - auxW) * V[k])$
end if
return $auxV$

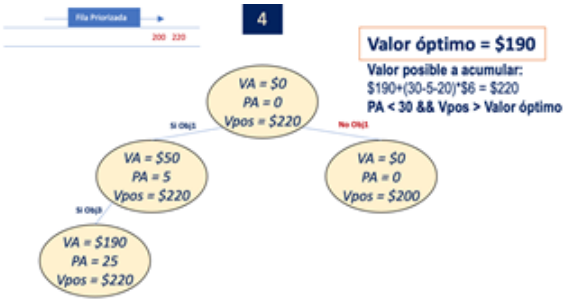
Procedure 4 KNAPSACK_BT

Input: $W, V : Array$
$weight, i, n, optimalValue : Integer$
$accumulatedValue, accumulatedWeight, possibleValue : Integer$
if $i < n$ **and** $accumulatedWeight <= weight$ **and** $possibleValue > optimalValue$ **then**
if $accumulatedValue > optimalValue$ **then**
$optimalValue \leftarrow accumulatedValue$
end if
$KNAPSACK\_BT(i + 1, accumulatedValue + V[i], accumulatedWeight + W[i], possibleValue)$
$auxV \leftarrow accumulatedValue, auxW \leftarrow accumulatedWeight$
$auxV \leftarrow CALCULATING\_POSSIBLE\_VALUE(W, V, wight, i, nauxV, auxW)$
$KNAPSACK\_BT(i + 1, accumulatedValue, accumulatedWeight, auxV)$
end if

Usando Ramificación y Poda (Branch and Bound)

El problema de la Mochila al ser un problema de Selección con Optimización cumple con los requisitos para poder ser resuelto con la técnica de Branch & Bound. El árbol de búsqueda de soluciones será idéntico al de Backtracking, donde se tendrán n+1 niveles, y cada nivel se trabajará con un objeto, los cuales serán ordenados descendentemente por valor proporcional ($valor_i/peso_i$). Cada nodo tendrá 2 hijos, el hijo de la derecha donde si incluye al siguiente objeto y el de la derecha donde no lo incluye. Los nodos almacenarán el valor y peso acumulados hasta el momento, así como su valor posible (el cual se calcula igual que Backtracking). La gran diferencia es que en Backtracking se va procediendo a profundidad y en Branch & Bound se trabaja en anchura con Best-First. Para esto, manejaremos una fila priorizada por mayor valor posible.

PESO MOCHILA = 30
Objeto$_1$, valor$_1$: \$50, peso$_1$: 5, valor$_1$/peso$_1$ = \$10
Objeto$_3$, valor$_3$: \$140, peso$_3$: 20, valor$_3$/peso$_3$ = \$7
Objeto$_2$, valor$_2$: \$60, peso$_2$: 10, valor$_2$/peso$_2$ = \$6



Procedure 5 KNAPSACK_BB

Input: W, V : *Array*
$weight, i, n, optimalValue$: *Integer*
$node$: *Node*, pq : *PriorityQueue* $< Node >$
$node.level \leftarrow -1$
$node.accumulatedValue \leftarrow 0$
$node.accumulatedWeight \leftarrow 0$
$node.possibleValue \leftarrow CALCULATING\_POSSIBLE\_VALUE(-1, 0, 0)$
pq.enqueue(node)
while not pq.empty() **do**
  NEXT_SLIDE
end while

```
node ← pq.dequeueMin()
if node.accumulatedValue > optimalValue then
  optimalValue ← node.accumulatedValue
end if
if node.possibleValue > optimalValue then
  node.level ← node.level + 1
  node.possibleValue ← CALCULATING_POSSIBLE_VALUE(
  node.level, node.accumulatedValue, node.possibleValue)
  if node.possibleValue > optimalValue and node.accumulatedWeight <=
  weight then
    pq.enqueue(node)
  end if
  NEXT_SLIDE
end if
```

```
if node.possibleValue > optimalValue then
  ...
  node.accumulatedValue ← node.accumulatedValue + V[node.level]
  node.accumulatedWeight ← node.accumulatedWeight + W[node.level]
  node.possibleValue ← CALCULATING_POSSIBLE_VALUE(
  node.level, node.accumulatedValue, node.possibleValue)
  if node.possibleValue > optimalValue and node.accumulatedWeight <=
  weight then
    pq.enqueue(node)
  end if
end if
```

SuperSale [1]

Hay una Superventa en un SuperHiperMarket. Cada persona puede llevar solo un objeto de cada tipo, es decir, un televisor, una zanahoria, pero a un precio muy bajo. Vamos con toda una familia a ese SuperHiperMarket.
Cada persona puede tomar tantos objetos como pueda llevar a cabo desde la Superventa. Hemos dado una lista de objetos con precios y su peso. También sabemos cuál es el peso máximo que toda persona puede soportar. ¿Cuál es el valor máximo de los objetos que podemos comprar en SuperSale?

[1] https://onlinejudge.org/external/101/10130.pdf

Entrada

La entrada consta de casos de prueba T. El número de ellos (1 <= T <= 1000) se da en la primera línea del archivo de entrada. Cada caso de prueba comienza con una línea que contiene un solo número entero N que indica el número de objetos (1 <= N <= 1000). Luego sigue N líneas, cada una con dos enteros: P y W. El primer entero (1 <= P <= 100) corresponde al precio del objeto. El segundo entero (1 <= W <= 30) corresponde al peso del objeto. La siguiente línea contiene un número entero (1 <= G <= 100) es el número de personas en nuestro grupo. Las siguientes líneas G contienen el peso máximo (1 <= MW <= 30) que puede soportar esta i-ésima persona de nuestra familia (1 <= i <= G).

Salida

Para cada caso de prueba, su programa tiene que determinar un número entero. Imprima el valor máximo de los bienes que podemos comprar con esa familia.

Ejemplo de entrada

```
6
64 26
85 22
52 4
99 18
39 13
54 9
4
23
20
20
26
```

Ejemplo de salida

```
514
```