DP for the multiple knapsack problem

Outline

A DP algorithm
- Number of steps
- The growth with the number of knapsacks

Typical mistakes

In summary

Learning objectives

At the end of this chapter you will be able to:

- write the recurrence $v(j, k_1, \ldots, k_m)$ over the items and the residual capacities of the m knapsacks;
- run the DP algorithm by hand on an instance with $m = 2$ knapsacks, one matrix for each item;
- reconstruct an optimal solution, with the knapsack of every selected item, from the tables s;
- count the $\Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$ steps and explain why the number of states grows exponentially with m.

Prerequisites

- From this module, the chapter «DP for the 0/1 knapsack problem»: the recurrence $v(j, k)$ of the 0/1 knapsack problem (KP) over items and capacities, the matrix s of the decisions and the reconstruction of a solution.
- From the module *Combinatorial Optimization Problems*, the chapter «Knapsack problems»: the multiple knapsack problem, with m knapsacks of capacities $c_1, \ldots, c_m$.
- From the module *Algorithms*: pseudocode conventions and the size of the input.

The problem of this chapter

The problem of this chapter is the
multiple knapsack problem (MKP):

its definition and its assumptions are on its page.

- We write a DP algorithm that gives the optimal value of an MKP instance
- The same algorithm also gives an optimal solution: the knapsack of every selected item

A DP algorithm

Introduction: A DP algorithm

- In this section we write a DP algorithm for the MKP, in two steps: first the optimal value, then an optimal solution
- The algorithm follows the DP algorithm of the KP, but the capacity becomes a vector: one residual capacity for each knapsack
- We run it by hand on an instance with $m = 2$ knapsacks and four items, we count its steps, and we see how the number of states grows with the number of knapsacks

Knapsacks, solutions and unit vectors

- We write $i \in \mathscr{K} = \{1, 2, \ldots, m\}$ for a generic knapsack, because in the DP chapters the letter k denotes a capacity. Knapsack i has capacity c_i.
- A solution assigns each item j to one knapsack or leaves it out: it is described by the vector $(a_1, a_2, \ldots, a_n)$, where
  - $a_j = i$ if item j is inserted into knapsack i,
  - $a_j = 0$ if item j is not selected.
- The vector $\boldsymbol{e}_i$ is the i-th unit vector of length m: $\boldsymbol{k} - w_j \boldsymbol{e}_i$ is the vector $\boldsymbol{k}$ with the component k_i decreased by w_j.

Example: MKP instance

We consider $m = 2$ knapsacks with capacities $c_1 = 5$ and $c_2 = 4$, and a set $\mathscr{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following profits and weights:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	3	9	7	5
w_j	3	4	4	2

We use this MKP instance as the example of the whole chapter.

The two steps of the DP algorithm

- Step 1: computation of the optimal value
  - a recursive formula gives the optimal value of each subinstance, with its own residual capacities
- Step 2: computation of an optimal solution
  - the tables s store, for each item, the knapsack into which it is inserted, or 0

Step 1: the subinstances

For any $j \in \{0, 1, \ldots, n\}$ and any vector $\boldsymbol{k} = (k_1, k_2, \ldots, k_m)$ with $k_i \in \{0, 1, \ldots, c_i\}$ for every $i \in \mathscr{K}$, let

$$v(j, \boldsymbol{k}) = v(j, k_1, \ldots, k_m)$$

be the optimal value of the MKP subinstance given by:

1. m knapsacks, knapsack i with capacity k_i,
2. a set $\{1, 2, \ldots, j\}$ of available items,
3. for each item $h \in \{1, 2, \ldots, j\}$, a profit p_h and a weight w_h.

Step 1: the recursive formula

The recursive formula to compute $v(j, \boldsymbol{k})$ reads as follows:

$$v(j, \boldsymbol{k}) = \begin{cases} 0 & \text{if } j = 0, \\ v(j-1, \boldsymbol{k}) & \text{if } j \geq 1 \text{ and } w_j > k_i \text{ for all } i, \\ \max\left\{ v(j-1, \boldsymbol{k}), \right. & \text{if } j \geq 1 \text{ and} \\ \quad \left. p_j + \max\limits_{i:\, w_j \leq k_i} v(j-1, \boldsymbol{k} - w_j \boldsymbol{e}_i) \right\} & w_j \leq k_i \text{ for some } i. \end{cases}$$

Reading the recursive formula

- The components of $\boldsymbol{k}$ are the **residual capacities** of the knapsacks: in the top-down view, the decisions on the items $j+1, \ldots, n$ have already used $c_i - k_i$ units of knapsack i.
- With $m = 1$ the recursive formula is the recurrence of the KP.
- The base cases are $v(0, \boldsymbol{k}) = 0$ for every vector $\boldsymbol{k}$, since no item is available.

Optimal substructure

- The formula generates at most $m + 1$ **independent subproblems**:
  - item j is left out, or
  - it is inserted into one of the knapsacks i with $w_j \leq k_i$, and then only the capacity of knapsack i decreases.

$\rightsquigarrow$ **Optimal Substructure verified!**

Overlapping subproblems and the bottom-up approach

- Different assignments of the items $j+1, \ldots, n$ can leave the same residual capacities, and therefore the same value $v(j, \boldsymbol{k})$ may be needed several times.

$$\rightsquigarrow \textbf{Overlapping Subproblems verified!}$$

- The value $v(j, \boldsymbol{k})$ depends only on values of row $j-1$.
- Therefore, the **bottom-up approach** computes the rows $j = 0, 1, \ldots, n$ in this order, and in each row all the vectors $\boldsymbol{k}$ in any order.
- With $m = 2$ the row j is a matrix, with one row for each k_1 and one column for each k_2.

Step 2: computation of an optimal solution

- For each item j and vector $\boldsymbol{k}$ we store:
  - $s[j][\boldsymbol{k}] = i$ if item j is inserted into knapsack i in the optimal solution found for $(j, \boldsymbol{k})$,
  - $s[j][\boldsymbol{k}] = 0$ if item j is left out.
- The set of all the vectors $\boldsymbol{k}$ is

$$\mathscr{C} = \{0, 1, \ldots, c_1\} \times \{0, 1, \ldots, c_2\} \times \cdots \times \{0, 1, \ldots, c_m\}.$$

Next: the pseudo-code of the **DP algorithm using the bottom-up approach**,
which stores the tables s together with the tables v.

Algorithm: MKP_DP (1/2)

Input : An MKP instance given by n, m, arrays $p[1..n]$, $w[1..n]$, and the capacity vector $\boldsymbol{c}$

Output : The optimal value of the MKP instance and the tables $s[j][\boldsymbol{k}]$

```
1 function MKP_DP(n, m, p, w, c)
2 |   v := new array[0..n][0..c_1] ··· [0..c_m];
  |     s := new array[0..n][0..c_1] ··· [0..c_m]
3 |   foreach k ∈ 𝒞 do
4 |   |   v[0][k] ← 0; s[0][k] ← 0
5 |   end
```

Algorithm: MKP_DP (2/2)

function *MKP_DP*$(n, m, p, w, \boldsymbol{c})$ (continued)

```
6   |  for j := 1 to n do
7   |  |  foreach k ∈ 𝒞 do
8   |  |  |  v[j][k] ← v[j − 1][k]; s[j][k] ← 0
9   |  |  |  for i := 1 to m do
10  |  |  |  |  if w[j] ≤ k_i and
    |  |  |  |    p[j] + v[j − 1][k − w[j] e_i] > v[j][k] then
11  |  |  |  |  |  v[j][k] ← p[j] + v[j − 1][k − w[j] e_i];
    |  |  |  |  |    s[j][k] ← i
12  |  |  |  |  end
13  |  |  |  end
14  |  |  end
15  |  end
16  |  return v[n][c], s
17  end
```

How MKP_DP works

- For each vector $\boldsymbol{k}$, the DP algorithm first leaves item j out, and then tries to insert it into each knapsack $i = 1, 2, \ldots, m$ in which it fits.

- In case of ties the first option met is kept:
  - leaving item j out,
  - then the knapsack of smallest index,

  because the value is updated only when it is strictly larger.

Example: execution of the DP algorithm (1/7)

We consider the MKP instance of the example above, with capacities $c_1 = 5$ and $c_2 = 4$:

	$j = 1$	$j = 2$	$j = 3$	$j = 4$
p_j	3	9	7	5
w_j	3	4	4	2

- The row $j = 0$ is $v[0][\boldsymbol{k}] = 0$ for every $\boldsymbol{k}$.
- For each item j we compute the matrix $v[j][k_1][k_2]$, with one row for each $k_1 \in \{0, 1, \ldots, 5\}$ and one column for each $k_2 \in \{0, 1, \ldots, 4\}$.
- First, the computations of the cell $\boldsymbol{k} = \boldsymbol{c} = (5, 4)$ for each item, and of the cells used by the reconstruction.

Example: execution of the DP algorithm (2/7)

Items $j = 1$ and $j = 2$:

$$v[1][(5,4)] = \max\{v[0][(5,4)],\ p_1 + v[0][(2,4)],\ p_1 + v[0][(5,1)]\}$$
$$= \max\{0,\ 3+0,\ 3+0\} = 3$$

$$v[1][(3,0)] = \max\{v[0][(3,0)],\ p_1 + v[0][(0,0)]\}$$
$$= \max\{0,\ 3+0\} = 3$$

$$v[2][(5,4)] = \max\{v[1][(5,4)],\ p_2 + v[1][(1,4)],\ p_2 + v[1][(5,0)]\}$$
$$= \max\{3,\ 9+3,\ 9+3\} = 12$$

$$v[2][(3,4)] = \max\{v[1][(3,4)],\ p_2 + v[1][(3,0)]\}$$
$$= \max\{3,\ 9+3\} = 12$$

Example: execution of the DP algorithm (3/7)

Items $j = 3$ and $j = 4$:

$$\begin{aligned} v[3][(5,4)] &= \max\{v[2][(5,4)],\ p_3 + v[2][(1,4)],\ p_3 + v[2][(5,0)]\} \\ &= \max\{12,\ 7+9,\ 7+9\} = 16 \end{aligned}$$

$$\begin{aligned} v[3][(3,4)] &= \max\{v[2][(3,4)],\ p_3 + v[2][(3,0)]\} \\ &= \max\{12,\ 7+3\} = 12 \end{aligned}$$

$$\begin{aligned} v[4][(5,4)] &= \max\{v[3][(5,4)],\ p_4 + v[3][(3,4)],\ p_4 + v[3][(5,2)]\} \\ &= \max\{16,\ 5+12,\ 5+9\} = 17 \end{aligned}$$

Example: execution of the DP algorithm (4/7)

Item $j = 1$, with $p_1 = 3$ and $w_1 = 3$:

$v[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	3
$k_1 = 1$	0	0	0	3	3
$k_1 = 2$	0	0	0	3	3
$k_1 = 3$	3	3	3	3	3
$k_1 = 4$	3	3	3	3	3
$k_1 = 5$	3	3	3	3	3

Example: execution of the DP algorithm (5/7)

Item $j = 2$, with $p_2 = 9$ and $w_2 = 4$:

$v[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	12
$k_1 = 5$	9	9	9	12	12

Example: execution of the DP algorithm (6/7)

Item $j = 3$, with $p_3 = 7$ and $w_3 = 4$:

$v[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	3	9
$k_1 = 1$	0	0	0	3	9
$k_1 = 2$	0	0	0	3	9
$k_1 = 3$	3	3	3	3	12
$k_1 = 4$	9	9	9	12	16
$k_1 = 5$	9	9	9	12	16

Example: execution of the DP algorithm (7/7)

Item $j = 4$, with $p_4 = 5$ and $w_4 = 2$:

$v[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	5	5	9
$k_1 = 1$	0	0	5	5	9
$k_1 = 2$	5	5	5	8	14
$k_1 = 3$	5	5	8	8	14
$k_1 = 4$	9	9	14	14	16
$k_1 = 5$	9	9	14	14	17

Example: the tables s (1/4)

For the instance of the example above, these are the matrices $s[j][k_1][k_2]$ computed by the DP algorithm: 0 means that item j is left out, 1 and 2 the knapsack into which it is inserted.

Item $j = 1$, with $p_1 = 3$ and $w_1 = 3$:

$s[1][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	2	2
$k_1 = 1$	0	0	0	2	2
$k_1 = 2$	0	0	0	2	2
$k_1 = 3$	1	1	1	1	1
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Example: the tables s (2/4)

Item $j = 2$, with $p_2 = 9$ and $w_2 = 4$:

$s[2][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	2
$k_1 = 1$	0	0	0	0	2
$k_1 = 2$	0	0	0	0	2
$k_1 = 3$	0	0	0	0	2
$k_1 = 4$	1	1	1	1	1
$k_1 = 5$	1	1	1	1	1

Example: the tables s (3/4)

Item $j = 3$, with $p_3 = 7$ and $w_3 = 4$:

$s[3][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	0	0	0
$k_1 = 1$	0	0	0	0	0
$k_1 = 2$	0	0	0	0	0
$k_1 = 3$	0	0	0	0	0
$k_1 = 4$	0	0	0	0	1
$k_1 = 5$	0	0	0	0	1

Example: the tables s (4/4)

Item $j = 4$, with $p_4 = 5$ and $w_4 = 2$:

$s[4][k_1][k_2]$	$k_2 = 0$	$k_2 = 1$	$k_2 = 2$	$k_2 = 3$	$k_2 = 4$
$k_1 = 0$	0	0	2	2	0
$k_1 = 1$	0	0	2	2	0
$k_1 = 2$	1	1	1	1	1
$k_1 = 3$	1	1	2	1	1
$k_1 = 4$	0	0	2	2	0
$k_1 = 5$	0	0	2	2	1

Reconstruction of an optimal solution

- Once the tables s have been computed, an optimal solution is reconstructed:
  - scanning the items from $j = n$ down to $j = 1$,
  - starting from $\boldsymbol{k} = \boldsymbol{c}$ and $a_j = 0$ for every item j:

$$a_j \leftarrow s[j][\boldsymbol{k}], \qquad \text{and, if } a_j = i \geq 1, \quad \boldsymbol{k} \leftarrow \boldsymbol{k} - w_j \boldsymbol{e}_i.$$

Algorithm: MKP_Reconstruct

Input : The number n of items, the capacity vector $\boldsymbol{c}$, the array $w[1..n]$, and the tables s

Output : An optimal solution of the MKP instance, as the vector of the knapsacks

```
1  function MKP_Reconstruct(n, c, w, s)
2      a := new array[1..n]
3      k := c
4      for j := n downto 1 do
5          a[j] ← s[j][k]
6          if a[j] ≥ 1 then
7              k ← k − w[j] e_{a[j]}
8          end
9      end
10     return a
11 end
```

Example: computing an optimal solution (1/2)

After the execution of the DP algorithm on the instance of the example above, we start from $j = 4$ and $\boldsymbol{k} = (5, 4)$:

Iteration	j	$\boldsymbol{k}$	$s[j][\boldsymbol{k}]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	Action
–	–	–	–	0	0	0	0	$\boldsymbol{k} = (5, 4)$
1	4	$(5, 4)$	1	0	0	0	1	$a[4] \leftarrow 1$, $\boldsymbol{k} \leftarrow (3, 4)$
2	3	$(3, 4)$	0	0	0	0	1	$a[3] \leftarrow 0$, $\boldsymbol{k} = (3, 4)$
3	2	$(3, 4)$	2	0	2	0	1	$a[2] \leftarrow 2$, $\boldsymbol{k} \leftarrow (3, 0)$
4	1	$(3, 0)$	1	1	2	0	1	$a[1] \leftarrow 1$, $\boldsymbol{k} \leftarrow (0, 0)$

Example: computing an optimal solution (2/2)

- The vector $a = (1, 2, 0, 1)$ is an optimal solution of the MKP instance of the example:
  - knapsack 1 contains the items $\{1, 4\}$, of total weight $3 + 2 = 5 \leq c_1$,
  - knapsack 2 contains the items $\{2\}$, of total weight $4 \leq c_2$.

  The optimal value is $v(4, 5, 4) = 17$.

- Filling the knapsacks one after the other is not enough:
  - the best filling of knapsack 1 alone is $\{2\}$, of profit 9,
  - the best filling of knapsack 2 with the remaining items is $\{3\}$, of profit 7,

  for a total of $16 < 17$.

How many steps does the DP algorithm execute?

We count how many times each line
of the pseudo-code of MKP_DP is executed.

Number of steps (1/3)

- Let

$$N = |\mathscr{C}| = \prod_{i=1}^{m} (c_i + 1)$$

  be the number of vectors $\boldsymbol{k}$.

- Let $S(n, m, \boldsymbol{c})$ denote the number of steps executed by `MKP_DP` in the worst case.

- Inserting item j into knapsack i changes only the component k_i, so lines 10 and 11 count one step each.

Number of steps (2/3)

- Lines 2 and 16 are executed 1 time each.
- Lines 3 and 4 are executed N times each.
- Line 6 is executed n times.
- Lines 7 and 8 are executed nN times each.
- Lines 9, 10 and 11 are executed at most $m\,nN$ times each.

Number of steps (3/3)

Therefore, the total number of steps in the worst case is

$$S(n, m, \boldsymbol{c}) = 3m\,nN + 2nN + 2N + n + 2,$$

and, since

$$3m\,nN \leq S(n, m, \boldsymbol{c}) \leq 10m\,nN,$$

we have

$$S(n, m, \boldsymbol{c}) = \Theta\left(m\,n \prod_{i=1}^{m} (c_i + 1)\right).$$

The number of steps of the DP algorithm

The DP algorithm for the MKP executes

$$\Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$$

steps in the worst case.

- For a fixed number m of knapsacks this is $\Theta(n \prod_{i=1}^{m} c_i)$, a **pseudo-polynomial** number of steps.
- The reconstruction executes $\Theta(n)$ steps.

The growth with the number of knapsacks (1/2)

- Each new knapsack multiplies the number of states by $c_i + 1$.
- If all the capacities are equal to c, the table has

$$(n+1)(c+1)^m$$

  cells.

- The number of states grows **exponentially** with m, while the size of the input grows only by one number, the capacity c_i.

The growth with the number of knapsacks (2/2)

- With $n = 50$ items and capacities $c_i = 100$, the table has:

m	number of cells
1	$51 \cdot 101 = 5151$
2	$51 \cdot 101^2 = 520\,251$
5	$51 \cdot 101^5 \approx 5.4 \cdot 10^{11}$

- Knapsacks with the same capacity give symmetric states: if $c_1 = c_2$, the vectors $(k_1, k_2, \dots)$ and $(k_2, k_1, \dots)$ have the same optimal value, and the DP algorithm computes both.

Summary (1/2)

Step 1: the optimal value:

- the state is the item j and the vector $\boldsymbol{k}$ of the residual capacities
- item j is left out or inserted into one knapsack i with $w_j \leq k_i$: only k_i decreases by w_j
- base cases $v(0, \boldsymbol{k}) = 0$; with $m = 1$ the recurrence of the KP
- optimal substructure and overlapping subproblems: bottom-up, row j after row $j - 1$

Step 2: an optimal solution:

- $s[j][\boldsymbol{k}]$ is the knapsack of item j, or 0 if item j is left out
- on the example: $a = (1, 2, 0, 1)$, of value $v(4, 5, 4) = 17$

Summary (2/2)

Number of steps:

- $S(n, m, \boldsymbol{c}) = \Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$ for MKP_DP, $\Theta(n)$ for the reconstruction
- pseudo-polynomial for a fixed number m of knapsacks

The growth with m:

- each new knapsack multiplies the number of states by $c_i + 1$
- the number of states grows exponentially with m

Typical mistakes

Introduction: Typical mistakes

- We collect the mistakes that are made most often with the DP algorithm for the MKP
- Two of them concern the capacities, two the choices of the recurrence
- Check them before running the DP algorithm by hand

Typical mistakes (1/2)

Observation 1: Decreasing all the capacities

Item j goes into one knapsack only: $\boldsymbol{k} - w_j \boldsymbol{e}_i$ decreases the component k_i, not all of them as in the multidimensional knapsack problem.

Observation 2: Using one capacity $c_1 + \cdots + c_m$

The KP with the total capacity gives only an upper bound: its solution may not be divisible among the knapsacks.

Typical mistakes (2/2)

Observation 3: Filling the knapsacks one after the other

An optimal filling of knapsack 1*, followed by an optimal filling of knapsack* 2 *with the remaining items, can be worse than the optimum.*

Observation 4: Forgetting the option «item left out»

The term $v(j-1, \boldsymbol{k})$ *is always a candidate, also when item* j *fits into some knapsack.*

Summary

The capacities:

- item j decreases only the component k_i of the knapsack into which it is inserted
- one total capacity $c_1 + \cdots + c_m$ gives only an upper bound

The choices:

- filling the knapsacks one after the other can miss the optimum
- the option «item left out», $v(j-1, \boldsymbol{k})$, is always a candidate

In summary

Introduction: In summary

- We collect the key ideas of the chapter on the DP algorithm for the MKP
- The dashboard gathers the result on the example of the chapter

Summary (1/2)

- The state of the DP algorithm is the item j and the vector $\boldsymbol{k}$ of the residual capacities of the m knapsacks.
- Item j is left out or inserted into one knapsack i in which it fits: $m + 1$ options, and only k_i decreases.
- The DP algorithm executes $\Theta(m\, n \prod_{i=1}^{m}(c_i + 1))$ steps: the number of states grows exponentially with m.
- The tables s store the knapsack of each item, and an optimal solution is read backwards from $(n, \boldsymbol{c})$.

Summary (2/2)

Dashboard of the MKP example ($\boldsymbol{c} = (5, 4)$, four items):

Method	Lower bound	Upper bound	Origin
DP algorithm	17	17	optimal value, solution $a = (1, 2, 0, 1)$