

Approximation algorithms for the knapsack problem (1/2)

Outline

Why approximation algorithms for the KP?

Notation

Split item and residual capacity

The greedy algorithm

The Ext-Greedy algorithm

The H^2 algorithm

The H^ℓ algorithm

The Basic-FPTAS algorithm

Learning objectives

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

- ▶ execute the greedy, Ext-Greedy, H^2 , H^ℓ and Basic-FPTAS algorithms by hand on a KP instance;
- ▶ explain why the greedy algorithm has no constant guarantee, and why Ext-Greedy has guarantee 2;
- ▶ prove the guarantees $\frac{4}{3}$ of H^2 , $\frac{\ell+2}{\ell+1}$ of H^ℓ and $\frac{1}{1-\varepsilon}$ of the Basic-FPTAS algorithm;
- ▶ compare the running times of H^ℓ and of the Basic-FPTAS algorithm, and say why only the second one is an FPTAS;
- ▶ explain why an absolute performance guarantee for the KP forces an exact algorithm.

Prerequisites

- ▶ From this module, the chapter «Principles of approximation algorithms»: performance ratio, relative and absolute performance guarantees, tightness, approximation schemes, PTAS and FPTAS.
- ▶ From this module, the chapters «Knapsack problems» and «Formulations of knapsack problems»: the KP, the LP relaxation LP_{KP} and the bound of Dantzig, which uses the split item.
- ▶ From this module, the chapters «Constructive heuristics» and «DP for the 0/1 knapsack problem»: the greedy algorithm and the dynamic programming by profits, used by the Basic-FPTAS algorithm.

The problem of this chapter

The problem of this chapter is the
0/1 knapsack problem (KP):

its definition and its assumptions are on its page.

- ▶ We study heuristic algorithms for the KP that come with a **performance guarantee**
- ▶ From the greedy algorithm, which has none, up to a scheme that gets as close to the optimum as we want

Why approximation algorithms for the KP?



Introduction: Why approximation algorithms for the KP?

- ▶ We start from a challenge on a small instance: the most natural rule, the greedy one, misses a very simple solution
- ▶ We ask how bad the greedy rule can be, compared with the optimum
- ▶ The chapter answers: as bad as we like, and it shows a cheap cure, and better and better algorithms

A challenge

A knapsack of capacity $c = 10$ and four items with profits and weights

	$i = 1$	$i = 2$	$i = 3$	$i = 4$
p_i	6	8	19	4
w_i	2	3	9	5

- ▶ The greedy rule takes the items by non-increasing efficiency, whenever they fit: it collects the profit 18.
- ▶ Item 3 alone has profit 19.

How bad can the greedy rule be?

- ▶ Can the greedy rule be arbitrarily worse than the optimum?
- ▶ Try to build an instance in which it collects a profit of 2 and the optimal value is as large as you like.
- ▶ The answer is yes, and the cure is cheap: **compare with the best single item.**
- ▶ This chapter proves the guarantee of this fix, and of three better algorithms, up to a scheme that gets as close to the optimum as we want at a polynomial price.

Summary

The challenge:

- ▶ On a small instance, the greedy rule collects 18, while a single item has profit 19

The questions of the chapter:

- ▶ How far from the optimum can a heuristic algorithm be, in the worst case?
- ▶ Which algorithms have a guarantee, and at which price in running time?

Notation

Introduction: Notation

- ▶ We fix the notation used in the whole chapter: items, profits, weights and capacity
- ▶ We compare the value of a heuristic solution with the optimal value
- ▶ We assume once and for all that the items are sorted by efficiency

Items, profits and weights

- ▶ As on the problem page, c is the knapsack capacity and $\mathcal{I} = \{1, 2, \dots, n\}$ the set of items.
- ▶ In this chapter item $i \in \mathcal{I}$ has profit p_i and weight w_i .

Optimal value and heuristic value

- ▶ We denote by $z(I)$ the **optimal value** of the KP on the instance I .
- ▶ We denote by $\mathcal{A}(I)$ the profit of the feasible solution returned by a heuristic algorithm $\mathcal{A}$.
- ▶ Since every heuristic algorithm of this chapter returns a feasible solution,

$$\mathcal{A}(I) \leq z(I) \quad \text{and} \quad \frac{z(I)}{\mathcal{A}(I)} \geq 1$$

for every KP instance I .

Items sorted by efficiency

- ▶ We assume throughout that the items are sorted by non-increasing efficiency

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad \text{for all} \quad i \in \{1, 2, \dots, n-1\}.$$

- ▶ Sorting requires $O(n \log n)$ time, a polynomial function of the number n of items.

Summary

Notation:

- ▶ c capacity, $\mathcal{I} = \{1, 2, \dots, n\}$ items, p_i profits, w_i weights
- ▶ $z(I)$ optimal value, $\mathcal{A}(I) \leq z(I)$ value of a heuristic algorithm $\mathcal{A}$

Assumption:

- ▶ the items are sorted by non-increasing efficiency $\frac{p_i}{w_i}$, in $O(n \log n)$ time

Split item and residual capacity

Introduction: Split item and residual capacity

- ▶ The split item is the first item that does not fit when we take the items in order of efficiency
- ▶ It appears in every algorithm and in every proof of this chapter
- ▶ We define it, together with the residual capacity, and we compute both on an example

Definition 1: split item and residual capacity

Given a set $\mathcal{I} = \{1, 2, \dots, n\}$ of n items ordered by non-increasing efficiency, the **split item** is the item

$$s \in \mathcal{I} \quad \text{such that} \quad \sum_{i=1}^{s-1} w_i \leq c \quad \text{and} \quad \sum_{i=1}^s w_i > c$$

and the **residual capacity** is

$$rc = c - \sum_{i=1}^{s-1} w_i$$

Split item and residual capacity: remarks

- ▶ The split item is the first one that does not fit entirely in the knapsack if all the previous ones are selected.
- ▶ The residual capacity is the knapsack capacity remaining when all the items that precede the split item are selected.
- ▶ Sorting the items by non-increasing efficiency takes $O(n \log n)$ time, and the split item is then found by scanning them.
- ▶ The split item can also be found in $O(n)$ time, without sorting, by a weighted median search: the sorting is convenient, not necessary.

Example (1/2)

We consider a knapsack of capacity $c = 9$ and a set $\mathcal{I} = \{1, 2, 3, 4, 5, 6, 7\}$ of 7 items ($n = 7$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
p_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

The efficiencies of the items are:

$$\frac{p_1}{w_1} = \frac{6}{2}, \quad \frac{p_2}{w_2} = \frac{5}{3}, \quad \frac{p_3}{w_3} = \frac{8}{6}, \quad \frac{p_4}{w_4} = \frac{9}{7},$$

$$\frac{p_5}{w_5} = \frac{6}{5}, \quad \frac{p_6}{w_6} = \frac{7}{9}, \quad \frac{p_7}{w_7} = \frac{3}{4}.$$

Example (2/2)

- ▶ The items are sorted by non-increasing efficiency:
(1, 2, 3, 4, 5, 6, 7).
- ▶ The split item is $s = 3$ since:

$$\sum_{i=1}^2 w_i = 2 + 3 = 5 \leq 9 \quad \text{and} \quad \sum_{i=1}^3 w_i = 2 + 3 + 6 = 11 > 9$$

- ▶ The residual capacity is:

$$rc = c - \sum_{i=1}^2 w_i = 9 - 2 - 3 = 4$$

This instance is used again and again in the chapter.

Summary

Split item:

- ▶ the first item s that does not fit when all the previous ones are selected
- ▶ it can be found in $O(n \log n)$ time by sorting, or in $O(n)$ time by a weighted median search

Residual capacity:

- ▶ $rc = c - \sum_{i=1}^{s-1} w_i$, the capacity left by the items $1, 2, \dots, s-1$
- ▶ on the example: $s = 3$ and $rc = 4$

The greedy algorithm

Introduction: The greedy algorithm

- ▶ The greedy algorithm is the most natural constructive heuristic for the KP
- ▶ It is fast and simple, and it is the building block of all the other algorithms of the chapter
- ▶ We write its pseudo-code and run it on four instances: on the last ones it is far from the optimum

The greedy algorithm *GR*

- ▶ We assume items are sorted in non-increasing order of efficiency

$$\frac{p_i}{w_i} \geq \frac{p_{i+1}}{w_{i+1}} \quad \text{for all } i \in \{1, 2, \dots, n-1\}$$

- ▶ The *greedy* algorithm (denoted by *GR*) is a constructive heuristic for the KP.
- ▶ It builds a feasible solution by processing the items in non-increasing order of *efficiency*, defined as the ratio between profit and weight.
- ▶ Starting from an empty subset of items, each item is added to the solution if and only if the capacity constraint is not violated.

Algorithm: Greedy (1/2)

Input : KP instance I with n items, profits $p[1..n]$, weights $w[1..n]$, capacity c

Output : Feasible KP solution $x[1..n]$

```
1 function Greedy( $n, p, w, c$ )  
2    $x :=$  new array[1.. $n$ ]           // selected items  
3   for  $i := 1$  to  $n$  do  
4      $x[i] \leftarrow 0$  // item  $i$  is initially not selected  
5   end  
6    $rc := c$                          // residual capacity
```

Algorithm: Greedy (2/2)

function *Greedy*(n, p, w, c) (continued)

```
7   for  $i := 1$  to  $n$  do
8       if  $w[i] \leq rc$  then
9            $x[i] \leftarrow 1$                                 // select item  $i$ 
10           $rc \leftarrow rc - w[i]$  // update residual capacity
11      end
12  end
13  return  $x$ 
14 end
```

GR and the split item

- ▶ Given the split item s , GR selects all items that precede the split item, i.e.,

$$\{1, 2, \dots, s-1\} \subseteq S \quad \text{and} \quad GR(I) \geq \sum_{i=1}^{s-1} p_i.$$

- ▶ Moreover, item s is not selected by GR .
- ▶ GR may also select some items that follow the split item.

Next: four executions of GR , on four instances.

Example: execution of *GR* (1/3)

For the KP instance of the split item example ($c = 9$, $n = 7$), the items are sorted by non-increasing efficiency: (1, 2, 3, 4, 5, 6, 7).

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$
p_i	6	5	8	9	6	7	3
w_i	2	3	6	7	5	9	4

The execution of *GR* proceeds as follows:

1. Item 1 (weight $w_1 = 2$) is selected. The available capacity rc becomes $9 - 2 = 7$.
2. Item 2 (weight $w_2 = 3$) is selected. The available capacity rc becomes $7 - 3 = 4$.

Example: execution of *GR* (2/3)

3. Item 3 (weight $w_3 = 6$) is not selected since it does not fit into the available capacity $rc = 4$.
4. Item 4 (weight $w_4 = 7$) is not selected since it does not fit into the available capacity $rc = 4$.
5. Item 5 (weight $w_5 = 5$) is not selected since it does not fit into the available capacity $rc = 4$.
6. Item 6 (weight $w_6 = 9$) is not selected since it does not fit into the available capacity $rc = 4$.
7. Item 7 (weight $w_7 = 4$) is selected. The available capacity rc becomes $4 - 4 = 0$.

Example: execution of *GR* (3/3)

Hence *GR* returns the feasible solution $S = \{1, 2, 7\}$, of total profit

$$GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_7 = 6 + 5 + 3 = 14.$$

Therefore, 14 is a valid lower bound on the optimal value $z(I)$:

$$14 \leq z(I).$$

- ▶ In this instance the split item is $s = 3$.
- ▶ *GR* selects all items preceding the split item, i.e., items 1 and 2, does not select item 3 (the split item), and then also selects item 7, which follows the split item.

Example: execution of GR ($1/4$)

We consider a knapsack of capacity $c = 10$ and a set $\mathcal{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$
p_i	6	8	19	4
w_i	2	3	9	5

The efficiencies of the items are:

$$\frac{p_1}{w_1} = \frac{6}{2}, \quad \frac{p_2}{w_2} = \frac{8}{3}, \quad \frac{p_3}{w_3} = \frac{19}{9}, \quad \frac{p_4}{w_4} = \frac{4}{5}.$$

Thus the items are sorted by non-increasing efficiency: (1, 2, 3, 4).

Example: execution of *GR* (2/4)

The split item is $s = 3$ since:

$$\sum_{i=1}^2 w_i = 2 + 3 = 5 \leq 10 \quad \text{and} \quad \sum_{i=1}^3 w_i = 2 + 3 + 9 = 14 > 10$$

and the residual capacity is:

$$rc = c - \sum_{i=1}^2 w_i = 10 - 2 - 3 = 5.$$

Example: execution of *GR* (3/4)

The execution of *GR* proceeds as follows:

1. Item 1 (weight $w_1 = 2$) is selected. The available capacity rc becomes $10 - 2 = 8$.
2. Item 2 (weight $w_2 = 3$) is selected. The available capacity rc becomes $8 - 3 = 5$.
3. Item 3 (weight $w_3 = 9$) is not selected since it does not fit into the available capacity $rc = 5$.
4. Item 4 (weight $w_4 = 5$) is selected. The available capacity rc becomes $5 - 5 = 0$.

Example: execution of GR (4/4)

Hence GR returns the feasible solution $S = \{1, 2, 4\}$, of total profit

$$GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_4 = 6 + 8 + 4 = 18.$$

Therefore, 18 is a valid lower bound on the optimal value $z(I)$:
 $18 \leq z(I)$.

- ▶ In this instance the split item is $s = 3$: GR selects items 1 and 2, which precede it, does not select item 3 (the split item), and then selects item 4, which follows it.
- ▶ Item 3 has the maximum profit ($p_3 = 19$) and it is optimal to select it alone, but it is not selected by GR .

Example: execution of *GR* (1/4)

We consider a knapsack of capacity $c = 10$ and a set $\mathcal{I} = \{1, 2, 3, 4, 5\}$ of 5 items ($n = 5$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
p_i	4	6	5	12	3
w_i	1	2	2	6	2

The efficiencies of the items are:

$$\frac{p_1}{w_1} = \frac{4}{1} = 4, \quad \frac{p_2}{w_2} = \frac{6}{2} = 3, \quad \frac{p_3}{w_3} = \frac{5}{2} = 2.5,$$

$$\frac{p_4}{w_4} = \frac{12}{6} = 2, \quad \frac{p_5}{w_5} = \frac{3}{2} = 1.5.$$

Thus the items are sorted by non-increasing efficiency:
(1, 2, 3, 4, 5).

Example: execution of *GR* (2/4)

The split item is $s = 4$ since:

$$\sum_{i=1}^3 w_i = 1+2+2 = 5 \leq 10 \quad \text{and} \quad \sum_{i=1}^4 w_i = 1+2+2+6 = 11 > 10$$

and the residual capacity is:

$$rc = c - \sum_{i=1}^3 w_i = 10 - 5 = 5.$$

Example: execution of GR (3/4)

The execution of GR proceeds as follows:

1. Item 1 (weight $w_1 = 1$) is selected. The available capacity rc becomes $10 - 1 = 9$.
2. Item 2 (weight $w_2 = 2$) is selected. The available capacity rc becomes $9 - 2 = 7$.
3. Item 3 (weight $w_3 = 2$) is selected. The available capacity rc becomes $7 - 2 = 5$.
4. Item 4 (weight $w_4 = 6$) is not selected since it does not fit into the available capacity $rc = 5$.
5. Item 5 (weight $w_5 = 2$) is selected. The available capacity rc becomes $5 - 2 = 3$.

Example: execution of GR (4/4)

Hence GR returns the feasible solution $S = \{1, 2, 3, 5\}$, of total profit

$$GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_3 + p_5 = 4 + 6 + 5 + 3 = 18.$$

Therefore, 18 is a valid lower bound on the optimal value $z(I)$:
 $18 \leq z(I)$.

- ▶ In this instance the split item is $s = 4$: GR selects items 1, 2 and 3 (which precede the split item), skips item 4 (the split item), and then selects item 5.
- ▶ The optimal solution is $\tilde{S} = \{2, 3, 4\}$ with value $z(I) = 6 + 5 + 12 = 23$, which is better than the solution found by GR .

Example: execution of *GR* (1/5)

We consider a knapsack of capacity $c = 21$ and a set $\mathcal{I} = \{1, 2, 3, 4, 5, 6\}$ of 6 items ($n = 6$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$
p_i	4	6	4	3	2	20
w_i	1	2	2	2	2	21

The efficiencies of the items are:

$$\frac{p_1}{w_1} = \frac{4}{1} = 4, \quad \frac{p_2}{w_2} = \frac{6}{2} = 3, \quad \frac{p_3}{w_3} = \frac{4}{2} = 2,$$
$$\frac{p_4}{w_4} = \frac{3}{2} = 1.5, \quad \frac{p_5}{w_5} = \frac{2}{2} = 1, \quad \frac{p_6}{w_6} = \frac{20}{21}.$$

Thus the items are sorted by non-increasing efficiency:
(1, 2, 3, 4, 5, 6).

Example: execution of *GR* (2/5)

The split item is $s = 6$ since:

$$\sum_{i=1}^5 w_i = 1 + 2 + 2 + 2 + 2 = 9 \leq 21$$

$$\text{and } \sum_{i=1}^6 w_i = 1 + 2 + 2 + 2 + 2 + 21 = 30 > 21$$

and the residual capacity is:

$$rc = c - \sum_{i=1}^5 w_i = 21 - 9 = 12.$$

Example: execution of *GR* (3/5)

The execution of *GR* proceeds as follows:

1. Item 1 (weight $w_1 = 1$) is selected. The available capacity rc becomes $21 - 1 = 20$.
2. Item 2 (weight $w_2 = 2$) is selected. The available capacity rc becomes $20 - 2 = 18$.
3. Item 3 (weight $w_3 = 2$) is selected. The available capacity rc becomes $18 - 2 = 16$.

Example: execution of GR (4/5)

- Item 4 (weight $w_4 = 2$) is selected. The available capacity rc becomes $16 - 2 = 14$.
- Item 5 (weight $w_5 = 2$) is selected. The available capacity rc becomes $14 - 2 = 12$.
- Item 6 (weight $w_6 = 21$) is not selected since it does not fit into the available capacity $rc = 12$.

Hence GR returns the feasible solution $S = \{1, 2, 3, 4, 5\}$, of total profit

$$GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_3 + p_4 + p_5 = 4 + 6 + 4 + 3 + 2 = 19.$$

Therefore, 19 is a valid lower bound on the optimal value $z(I)$:
 $19 \leq z(I)$.

Example: execution of GR (5/5)

- ▶ The item with maximum profit is item 6 with $p_6 = 20$, which appears last in the efficiency-sorted order.
- ▶ GR selects all items from 1 to 5 (which precede the split item), but does not select item 6 (the split item itself) because its weight $w_6 = 21$ exceeds the residual capacity $rc = 12$.
- ▶ Item 6 alone would fit into the knapsack (since $w_6 = 21 \leq c = 21$) and would give a profit of 20, which is better than the greedy solution.
- ▶ The optimal solution is $\tilde{S} = \{6\}$ with value $z(I) = 20$.

Summary

The greedy algorithm GR :

- ▶ processes the items by non-increasing efficiency and adds each item that fits
- ▶ selects all the items $1, 2, \dots, s-1$ that precede the split item, so $GR(I) \geq \sum_{i=1}^{s-1} p_i$
- ▶ skips the split item s , but keeps scanning and may select items that follow it

Its weakness:

- ▶ it may miss the item of maximum profit, even when that item alone is optimal

The Ext-Greedy algorithm

Introduction: The Ext-Greedy algorithm

- ▶ The examples show the weakness of *GR*: it may miss the item of maximum profit
- ▶ Ext-Greedy cures it at almost no cost, comparing the greedy solution with the best single item
- ▶ This small change gives a performance guarantee, as we prove later in the chapter

The Ext-Greedy algorithm GR^e

- ▶ The *Ext-Greedy* algorithm (denoted by GR^e) is a constructive heuristic for the KP.
- ▶ It first computes a feasible solution by running the greedy algorithm GR .
- ▶ Then it compares this solution with the best feasible solution consisting of a single item.
- ▶ Finally, it returns the better of the two solutions.

Algorithm: Extended Greedy (1/3)

Input : KP instance I with n items, profits $p[1..n]$, weights $w[1..n]$, capacity c

Output : Feasible KP solution $x[1..n]$

```
1 function ExtendedGreedy( $n, p, w, c$ )
2    $x := \text{Greedy}(n, p, w, c)$            // greedy solution
3    $z := 0$                              // value of the greedy solution
4   for  $i := 1$  to  $n$  do
5     if  $x[i] = 1$  then
6        $z \leftarrow z + p[i]$            // add the profit of a
        selected item
7     end
8   end
```

Algorithm: Extended Greedy (2/3)

function *ExtendedGreedy*(n, p, w, c) (continued)

```
9   |    $i_m := 1$                                 // best single item
10  |    $p_{\max} := p[1]$ 
11  |   for  $i := 2$  to  $n$  do
12  |       |   if  $p[i] > p_{\max}$  then
13  |           |    $i_m \leftarrow i$                 // update best single item
14  |           |    $p_{\max} \leftarrow p[i]$           // update its profit
15  |       |   end
16  |   end
```

Algorithm: Extended Greedy (3/3)

function *ExtendedGreedy*(n, p, w, c) (continued)

```
17    $\bar{x} := \text{new array}[1..n]$     // best single-item solution
18   for  $i := 1$  to  $n$  do
19        $\bar{x}[i] \leftarrow 0$ 
20   end
21    $\bar{x}[i_m] \leftarrow 1$            // select only item  $i_m$ 
22   if  $z < p_{\max}$  then
23       for  $i := 1$  to  $n$  do
24            $x[i] \leftarrow \bar{x}[i]$  // copy component by component
25       end
26   end
27   return  $x$ 
28 end
```

Maximum profit and best single item

- ▶ Let $p_{\max}$ be the **maximum profit**

$$p_{\max} = \max \{ p_i : i \in \mathcal{I} \}.$$

- ▶ Let $\bar{S}$ be the **best single-item feasible solution**

$$\bar{S} = \{ i_m \} \text{ and } i_m \in \arg \max \{ p_i : i \in \mathcal{I} \}$$

Example: execution of GR^e

We start from the feasible solution computed by GR on the KP instance of the split item example ($c = 9$, $n = 7$):

$$S = \{1, 2, 7\}, \quad GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_7 = 6 + 5 + 3 = 14.$$

We now compute the maximum profit of a single item:

$$p_{\max} = \max \{ p_i : i \in \mathcal{I} \} = p_4 = 9, \quad \bar{S} = \{4\}.$$

Since $14 \geq 9$, GR^e keeps the greedy solution and returns

$$S = \{1, 2, 7\}.$$

Example: execution of GR^e

We start from the feasible solution computed by GR on the KP instance with $c = 10$ and 4 items, whose item 3 has profit 19:

$$S = \{1, 2, 4\}, \quad GR(I) = \sum_{i \in S} p_i = p_1 + p_2 + p_4 = 6 + 8 + 4 = 18.$$

We now compute the maximum profit of a single item:

$$p_{\max} = \max \{ p_i : i \in \mathcal{I} \} = p_3 = 19, \quad \bar{S} = \{3\}.$$

Since $18 < 19$, GR^e replaces the greedy solution with the best single-item feasible solution and returns

$$S = \{3\}.$$

Two bounds on $GR^e(I)$

- ▶ Given the split item s , we have

$$GR^e(I) \geq GR(I) \geq \sum_{i=1}^{s-1} p_i.$$

- ▶ In addition, by definition of GR^e , we have

$$GR^e(I) \geq p_{\max}.$$

- ▶ For every KP instance I , the objective values satisfy

$$GR(I) \leq GR^e(I) \leq z(I).$$

- ▶ This holds because GR^e returns the better between the greedy solution and the best single-item feasible solution, so it cannot be worse than GR , and any feasible solution value is at most the optimal value $z(I)$.

Summary

The Ext-Greedy algorithm GR^e :

- ▶ runs GR , then compares its solution with the best single item $\bar{S} = \{i_m\}$, of profit $p_{\max}$
- ▶ returns the better of the two solutions

Two bounds used later:

- ▶ $GR^e(I) \geq \sum_{i=1}^{s-1} p_i$ and $GR^e(I) \geq p_{\max}$
- ▶ $GR(I) \leq GR^e(I) \leq z(I)$

The H^2 algorithm

Introduction: The H^2 algorithm

- ▶ GR^e can still lose a lot, when it cannot pack together the items of largest profit
- ▶ The H^2 algorithm “guesses” two items of an optimal solution, by trying all the feasible pairs, and completes each guess with GR^e
- ▶ It pays more running time, and we will see that it gains a better guarantee

The H^2 algorithm

- ▶ The H^2 algorithm is a constructive heuristic for the KP; it is the algorithm called $G^{3/4}$ in the literature.
- ▶ It enumerates all feasible pairs of items (i, k) and “guesses” that the two items belong to an optimal solution.
- ▶ For each pair, it fixes i and k in the knapsack, builds a residual instance using the remaining capacity and keeping only items with profit at most $\min\{p_i, p_k\}$, and then completes the partial solution by running the Ext-Greedy algorithm GR^e on the residual instance.
- ▶ Finally, it returns the best solution found over all pairs, also comparing it with the best single-item feasible solution.

Algorithm: H^2 (1/2)

Input : KP instance I with item set $\mathcal{I} = \{1, 2, \dots, n\}$,
profits $p[1..n]$, weights $w[1..n]$, capacity c

Output : Feasible KP value $H^2(I)$

```
1 function  $H2(\mathcal{I}, p, w, c)$ 
2    $p_{\max} := \max\{p[j] : j \in \mathcal{I}\}$ 
3    $lb := p_{\max}$                                 // best value found
4    $\mathcal{P} := \{(i, k) \in \mathcal{I} \times \mathcal{I} : i < k, w[i] + w[k] \leq c\}$ 
```

Algorithm: H^2 (2/2)

function $H2(\mathcal{I}, p, w, c)$ (continued)

```
5   foreach  $(i, k)$  in  $\mathcal{P}$  do
6        $rc := c - w[i] - w[k]$            // residual capacity
7        $\mathcal{J}(i, k) := \{j \in \mathcal{I} \setminus \{i, k\} : p[j] \leq$ 
            $\min\{p[i], p[k]\}, w[j] \leq rc\}$ 
8        $l_{ik} :=$ 
           KP instance restricted to  $\mathcal{J}(i, k)$  with capacity  $rc$ 
9        $z_{res} := GR^e(l_{ik})$            // value of  $GR^e$  on the
           residual instance
10       $lb_{ik} := p[i] + p[k] + z_{res}$ 
11      if  $lb_{ik} > lb$  then
12           $lb \leftarrow lb_{ik}$ 
13      end
14  end
15  return  $lb$ 
16 end
```

Definition 2: feasible ordered pairs of items

Given a KP instance with item set $\mathcal{I} = \{1, 2, \dots, n\}$ and knapsack capacity c , the **set of all feasible ordered pairs of items** is

$$\mathcal{P} = \left\{ (i, k) \in \mathcal{I} \times \mathcal{I} : i < k, \quad w_i + w_k \leq c \right\}.$$

- ▶ The H^2 algorithm is based on the following idea: an optimal solution either consists of a single item, the one of maximum profit, or contains (at least) two items of “large” profit.
- ▶ The algorithm therefore *guesses* the two items of largest profit of an optimal solution, by enumerating all the feasible pairs in $\mathcal{P}$, and completes each guess on a residual instance.

The three operations of H^2 (1/2)

The H^2 algorithm considers the item pairs $(i, k) \in \mathcal{P}$, one at a time, and it executes three operations:

1. The **first operation** consists of inserting items i and k into the knapsack. This creates a **partial solution** of profit $p_i + p_k$ and a **residual capacity** $c - w_i - w_k$.
2. The **second operation** is the creation of a **residual instance** (denoted $I(i, k)$) containing only the items with profit smaller than or equal to the profits of the two inserted items. The **residual set of items** is:

$$\mathcal{I}(i, k) = \left\{ j \in \mathcal{J} \setminus \{i, k\} : \begin{aligned} p_j &\leq \min \{p_i, p_k\}, \\ w_j &\leq c - w_i - w_k \end{aligned} \right\}$$

The items with weight larger than the residual capacity are also discarded.

The three operations of H^2 (2/2)

3. The **third operation** is to complete the partial solution by executing the GR^e algorithm on the residual instance $I(i, k)$.

- For each item pair $(i, k) \in \mathcal{P}$, we have the lower bound:

$$lb(i, k) = p_i + p_k + GR^e(I(i, k))$$

- The global lower bound produced by the H^2 algorithm is:

$$H^2(I) = \max \left\{ \max_{(i,k) \in \mathcal{P}} \{ lb(i, k) \}, p_{\max} \right\}$$

A guess

This heuristic algorithm can be seen as “guessing” the two items with largest profit in an optimal solution (or the item with the largest profit).

Example: H^2 algorithm (1/4)

We consider a set $\mathcal{J} = \{1, 2, 3, 4, 5, 6, 7\}$ of 7 items ($n = 7$) with the following profits and weights:

item	p_j	w_j	$\frac{p_j}{w_j}$	efficiency
$j = 1$	6	2	$\frac{6}{2}$	$= 3$
$j = 2$	5	3	$\frac{5}{3}$	≈ 1.66
$j = 3$	8	6	$\frac{8}{6}$	≈ 1.33
$j = 4$	9	7	$\frac{9}{7}$	≈ 1.28
$j = 5$	6	5	$\frac{6}{5}$	$= 1.2$
$j = 6$	7	9	$\frac{7}{9}$	≈ 0.77
$j = 7$	3	4	$\frac{3}{4}$	$= 0.75$

and a knapsack capacity $c = 12$, so that $p_{\max} = 9$.

Example: H^2 algorithm (2/4)

- ▶ The optimal solution is $\tilde{S} = \{1, 2, 4\}$, of weight $2 + 3 + 7 = 12$ and value $z(I) = 6 + 5 + 9 = 20$.
- ▶ The split item is $s = 4$, and GR^e returns the solution $\{1, 2, 3\}$ with $GR^e(I) = 6 + 5 + 8 = 19$.

Next: the execution of H^2 , with initialization $lb = p_{\max} = 9$.

Example: H^2 algorithm (3/4)

Execution of H^2 (initialization $lb = p_{\max} = 9$):

- ▶ Pair (1, 2): the residual capacity is $rc = 12 - 2 - 3 = 7$, $\mathcal{J}(1, 2) = \{7\}$ and $GR^e(I(1, 2)) = 3$, hence $lb(1, 2) = 6 + 5 + 3 = 14 > lb$ and lb is updated to 14.
- ▶ Pair (1, 3): the residual capacity is $rc = 12 - 2 - 6 = 4$, $\mathcal{J}(1, 3) = \{2, 7\}$ and $GR^e(I(1, 3)) = 5$, hence $lb(1, 3) = 6 + 8 + 5 = 19 > lb$ and lb is updated to 19.
- ▶ Pair (1, 4): the residual capacity is $rc = 12 - 2 - 7 = 3$, $\mathcal{J}(1, 4) = \{2\}$ and $GR^e(I(1, 4)) = 5$, hence $lb(1, 4) = 6 + 9 + 5 = 20 > lb$ and lb is updated to 20.

Example: H^2 algorithm (4/4)

- ▶ Pair (1, 5): the residual capacity is $rc = 12 - 2 - 5 = 5$, $\mathcal{I}(1, 5) = \{2, 7\}$ and $GR^e(I(1, 5)) = 5$, hence $lb(1, 5) = 6 + 6 + 5 = 17 < lb$.
- ▶ Pair (1, 6): the residual capacity is $rc = 12 - 2 - 9 = 1$, $\mathcal{I}(1, 6) = \emptyset$, hence $lb(1, 6) = 6 + 7 = 13 < lb$.
- ▶ Pair (1, 7): the residual capacity is $rc = 12 - 2 - 4 = 6$, $\mathcal{I}(1, 7) = \emptyset$, hence $lb(1, 7) = 6 + 3 = 9 < lb$.
- ▶ The pairs (i, k) with $i \geq 2$ are processed in the same way, and none of them gives a value larger than 20.

Hence $H^2(I) = 20 = z(I)$: on this instance H^2 finds the optimal solution, which GR^e misses.

The running time of H^2

Proposition 1

The H^2 algorithm can be computed in $O(n^3)$ time, and in $O(n^2)$ time with a more careful implementation.

- ▶ We do not provide a formal proof but we only give the main ideas in what follows.
- ▶ The running time of the H^2 algorithm is basically determined by executing the GR^e algorithm, which requires $O(n)$ time, for every pair of items.
- ▶ This number of executions of GR^e is $\binom{n}{2}$, which is in $O(n^2)$, and hence the total running time is $O(n^3)$.
- ▶ With a cleverer implementation it can be reduced to $O(n^2)$.

Summary

The H^2 algorithm:

- ▶ guesses two items of largest profit of an optimal solution, trying every pair $(i, k) \in \mathcal{P}$
- ▶ for each pair: partial solution, residual instance $I(i, k)$ with items of profit at most $\min\{p_i, p_k\}$, completion by GR^e
- ▶ $H^2(I) = \max\{\max_{(i,k) \in \mathcal{P}} lb(i, k), p_{\max}\}$

Running time:

- ▶ $O(n^2)$ pairs, $O(n)$ for each GR^e : $O(n^3)$, or $O(n^2)$ with a careful implementation

The H^l algorithm

Introduction: The H^ℓ algorithm

- ▶ If guessing two items helps, guessing more items may help more
- ▶ The H^ℓ algorithm guesses the ℓ items of largest profit of an optimal solution, for a fixed integer ℓ
- ▶ The guarantee improves with ℓ , but the running time grows very quickly with ℓ

The H^ℓ algorithm

- ▶ The H^ℓ algorithm is a constructive heuristic for the KP.
- ▶ It first computes the best feasible solution made of at most $\ell - 1$ items.
- ▶ Then it enumerates all the feasible sets L of exactly ℓ items and “guesses” that they are the ℓ items of largest profit of an optimal solution.
- ▶ For each set, it packs the items of L , builds a residual instance with the remaining capacity and the items of profit at most $\min\{p_i : i \in L\}$, and completes the partial solution by running the Ext-Greedy algorithm GR^e on the residual instance.
- ▶ Finally, it returns the best solution found.

Algorithm: H^ℓ (1/2)

Input : KP instance I with item set $\mathcal{I} = \{1, 2, \dots, n\}$,
profits $p[1..n]$, weights $w[1..n]$, capacity c , integer ℓ
with $2 \leq \ell \leq n$

Output : Feasible KP value $H^\ell(I)$

```
1 function  $H(I, p, w, c, \ell)$ 
2    $lb := \max \{ \sum_{j \in S} p[j] : S \in \hat{T}(\ell) \}$     // feasible sets
   of at most  $\ell - 1$  items
```

Algorithm: H^ℓ (2/2)

function $H(\mathcal{I}, p, w, c, \ell)$ (continued)

```
3   foreach  $L$  in  $T(\ell)$     // feasible sets of exactly  $\ell$ 
    items
4   do
5        $rc := c - \sum_{i \in L} w[i]$     // residual capacity
6        $\mathcal{J}(L) := \{j \in \mathcal{I} \setminus L : p[j] \leq \min_{i \in L} p[i], w[j] \leq rc\}$ 
7        $lb_L := \sum_{i \in L} p[i] + GR^e(I(L))$     // complete the
        guess
8       if  $lb_L > lb$  then
9            $lb \leftarrow lb_L$ 
10      end
11  end
12  return  $lb$ 
13 end
```

From H^2 to H^ℓ

- ▶ The H^2 algorithm guesses the *two* items of largest profit of an optimal solution.
- ▶ The H^ℓ algorithm generalizes this idea: it guesses the ℓ items of largest profit, for a fixed integer ℓ with $2 \leq \ell \leq n$.
- ▶ For $\ell = 2$ it is exactly the H^2 algorithm.

Feasible sets of items

We define the collections of all feasible sets of exactly ℓ items and of at most $\ell - 1$ items:

$$T(\ell) = \left\{ L \subseteq \mathcal{I} : \sum_{j \in L} w_j \leq c, \quad |L| = \ell \right\}$$

and

$$\hat{T}(\ell) = \left\{ L \subseteq \mathcal{I} : \sum_{j \in L} w_j \leq c, \quad |L| \leq \ell - 1 \right\}$$

The three operations of H^ℓ

For each set $L \in T(\ell)$ the algorithm executes three operations:

1. it inserts all items $i \in L$ into the knapsack, which creates a **partial solution** of profit $\sum_{i \in L} p_i$ and a **residual capacity** $c - \sum_{i \in L} w_i$;
2. it creates the **residual instance** $I(L)$, whose **residual set of items** is

$$\mathcal{I}(L) = \left\{ j \in \mathcal{I} \setminus L : p_j \leq \min \{ p_i : i \in L \}, w_j \leq c - \sum_{i \in L} w_i \right\};$$

3. it completes the partial solution by running GR^e on $I(L)$, which gives the value

$$lb(L) = \sum_{i \in L} p_i + GR^e(I(L)).$$

The value returned by H^ℓ

The value returned by the algorithm is

$$H^\ell(I) = \max \left\{ \max_{L \in T(\ell)} lb(L), \max_{S \in \hat{T}(\ell)} \sum_{j \in S} p_j \right\}.$$

- ▶ The first term comes from the guesses of exactly ℓ items, each completed by GR^e .
- ▶ The second term is the best feasible solution made of at most $\ell - 1$ items.

Example: execution of H^ℓ (1/3)

We consider a knapsack of capacity $c = 12$ and a set $\mathcal{I} = \{1, 2, 3, 4, 5\}$ of 5 items ($n = 5$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
p_i	2	3	3	3	3
w_i	1	3	3	3	3

- ▶ The items are sorted by non-increasing efficiency, since $\frac{2}{1} = 2 > \frac{3}{3} = 1$.
- ▶ The optimal solution packs the four items of weight 3, with $z(I) = 12$.

Example: execution of H^ℓ (2/3)

With $\ell = 2$, i.e., the H^2 algorithm of the previous section.

- ▶ A pair of two items of profit 3, for instance $L = \{2, 3\}$, leaves residual capacity 6 and the residual set $\mathcal{I}(L) = \{1, 4, 5\}$.
- ▶ GR^e packs item 1 and item 4, of profit $2 + 3 = 5$, so $lb(L) = 3 + 3 + 5 = 11$.
- ▶ A pair containing item 1 leaves an empty residual set, since no other item has profit at most 2.
- ▶ Hence $H^2(I) = 11 < 12$.

Example: execution of H^ℓ (3/3)

With $\ell = 3$.

- ▶ A set of three items of profit 3, for instance $L = \{2, 3, 4\}$, leaves residual capacity 3 and the residual set $\mathcal{I}(L) = \{1, 5\}$.
- ▶ GR^e returns the best between item 1 alone (profit 2, then item 5 no longer fits) and item 5 alone (profit 3), so $lb(L) = 9 + 3 = 12$.
- ▶ Hence $H^3(I) = 12 = z(I)$.

Guessing one item more is enough to find the optimal solution on this instance, on which H^2 does not.

The running time of H^ℓ

Observation 1

The H^ℓ algorithm can be computed in $O(n^{\ell+1})$ time.

- ▶ The running time is determined by the executions of GR^e , which requires $O(n)$ time, one for every set in $T(\ell)$.
- ▶ There are at most $\binom{n}{\ell}$ such sets, which is in $O(n^\ell)$, hence the total running time is $O(n^{\ell+1})$.
- ▶ With a more careful implementation it can be reduced to $O(n^\ell)$.
- ▶ Besides storing the input data, no significant additional memory is used, and the total space requirement is only $O(n)$.

Summary

The H^ℓ algorithm:

- ▶ best feasible set of at most $\ell - 1$ items, then every feasible set $L \in T(\ell)$ of exactly ℓ items, completed by GR^e on the residual instance $I(L)$
- ▶ for $\ell = 2$ it is the H^2 algorithm
- ▶ on the example, $H^2(I) = 11$ and $H^3(I) = 12 = z(I)$

Running time and space:

- ▶ $O(n^{\ell+1})$ time, or $O(n^\ell)$ with a careful implementation; $O(n)$ space

The Basic-FPTAS algorithm

Introduction: The Basic-FPTAS algorithm

- ▶ The H^ℓ algorithm enumerates $O(n^\ell)$ sets of items, a number that grows very quickly with ℓ
- ▶ The Basic-FPTAS algorithm follows a completely different route: instead of guessing items, it rounds the profits and solves the rounded instance exactly
- ▶ We describe it, explain why only the profits are rounded, and run it on a small instance

The Basic-FPTAS algorithm FP^ϵ

- ▶ The *Basic-FPTAS* algorithm (denoted by FP^ϵ) replaces the item profits by scaled profits, which take few distinct values.
- ▶ Then it solves the scaled instance *to optimality* by dynamic programming by profits.
- ▶ The optimal solution of the scaled instance is returned as a feasible solution of the original instance, which in general is not optimal.

Algorithm: Basic-FPTAS

Input : KP instance I with n items, profits $p[1..n]$, weights $w[1..n]$, capacity c , accuracy $\varepsilon \in (0, 1)$

Output : Feasible KP solution $x[1..n]$

```
1 function BasicFPTAS( $n, p, w, c, \varepsilon$ )
2    $lb := GR^e(I)$  // value of the Ext-Greedy
   solution, a lower bound on  $z(I)$ 
3    $K := \varepsilon \cdot lb / n$  // scaling factor
4   for  $j := 1$  to  $n$  do
5      $p'[j] \leftarrow \lfloor p[j] / K \rfloor$  // scaled profit
6   end
7    $Q := \lfloor 2n / \varepsilon \rfloor$  // upper bound on the optimal value
   of the scaled instance
8    $x := DPProfits(n, p', w, c, Q)$  // scaled instance
   solved to optimality, table truncated at  $Q$ 
9   return  $x$ 
10 end
```

Scaling factor and scaled profits

- ▶ The **scaling factor** is

$$K = \frac{\varepsilon GR^e(I)}{n}.$$

- ▶ The **scaled profits** are obtained by rounding down the profits to a multiple of K :

$$p'_j = \left\lfloor \frac{p_j}{K} \right\rfloor \quad \text{for all } j \in \mathcal{J}.$$

- ▶ The scaled instance I' has the same weights and the same capacity as I , hence I and I' have exactly the same feasible solutions.

The profits are scaled, not the weights

- ▶ Rounding the weights down would make infeasible solutions look feasible.
- ▶ Rounding the weights up would discard solutions that are feasible.
- ▶ In both cases the scaled instance would answer a different question.
- ▶ Rounding the profits down, instead, keeps the set of feasible solutions untouched and only distorts the objective function.

The scaled instance is solved by dynamic programming by profits

- ▶ The scaled instance is solved by the **second DP algorithm** of the chapter «DP for the 0/1 knapsack problem».
- ▶ $u(j, q)$ is the minimum total weight of a subset of $\{1, 2, \dots, j\}$ of total profit exactly q , and the optimal value is the largest q with $u(n, q) \leq c$.
- ▶ The table is truncated at $Q = \lfloor 2n/\varepsilon \rfloor$. This is not smaller than the optimal value of the scaled instance, which is at most

$$\frac{z(I)}{K} \leq \frac{2 GR^e(I)}{K} = \frac{2n}{\varepsilon}$$

by the guarantee $z(I) \leq 2 GR^e(I)$ of GR^e .

- ▶ The table has size $n \times Q$, and the running time is $O(nQ) = O(n^2/\varepsilon)$.

Example: execution of FP^ε (1/4)

We consider a knapsack of capacity $c = 8$ and a set $\mathcal{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following profits and weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$
p_i	19	19	14	14
w_i	2	4	3	3

- ▶ The items are sorted by non-increasing efficiency, since $\frac{19}{2} = 9.5 > \frac{19}{4} = 4.75 > \frac{14}{3} \approx 4.67$.
- ▶ We run FP^ε with $\varepsilon = \frac{1}{2}$.

Example: execution of FP^ε (2/4)

Lower bound.

- ▶ GR packs item 1 (residual capacity $8 - 2 = 6$) and item 2 (residual capacity $6 - 4 = 2$), while items 3 and 4 do not fit, so $GR(I) = 19 + 19 = 38$.
- ▶ The best single item has profit $19 \leq 38$, hence $GR^e(I) = 38$.

Scaling. The scaling factor and the scaled profits are

$$K = \frac{\varepsilon GR^e(I)}{n} = \frac{\frac{1}{2} \cdot 38}{4} = 4.75,$$

$$p'_1 = p'_2 = \left\lfloor \frac{19}{4.75} \right\rfloor = 4, \quad p'_3 = p'_4 = \left\lfloor \frac{14}{4.75} \right\rfloor = 2.$$

The total scaled profit is $P' = 4 + 4 + 2 + 2 = 12$, not larger than $Q = \lfloor 2n/\varepsilon \rfloor = 16$, so the truncation does not act here.

Example: execution of FP^ϵ (3/4)

Dynamic programming by profits. On the scaled instance, the minimum weights $u(4, q)$ are

q	0	2	4	6	8	10	12
$u(4, q)$	0	3	2	5	6	9	12

The largest q with $u(4, q) \leq c = 8$ is $q = 8$, attained by the items $\{1, 2\}$ of total weight 6.

Example: execution of FP^ε (4/4)

Solution returned. FP^ε returns $S = \{1, 2\}$, of true profit

$$FP^\varepsilon(I) = 19 + 19 = 38.$$

- ▶ The optimal solution is $\tilde{S} = \{1, 3, 4\}$, of weight $2 + 3 + 3 = 8$ and value $z(I) = 19 + 14 + 14 = 47$.
- ▶ The scaling has made items 1 and 2 look as good as items 1, 3 and 4 together.

The performance ratio is

$$\frac{z(I)}{FP^\varepsilon(I)} = \frac{47}{38} \approx 1.24.$$

Summary

The Basic-FPTAS algorithm FP^ε :

- ▶ computes $GR^e(I)$ and the scaling factor $K = \varepsilon GR^e(I)/n$
- ▶ rounds down the profits, $p'_j = \lfloor p_j/K \rfloor$; the weights and the capacity are untouched
- ▶ solves the scaled instance exactly by dynamic programming by profits, with the table truncated at $Q = \lfloor 2n/\varepsilon \rfloor$

Running time:

- ▶ $O(nQ) = O(n^2/\varepsilon)$

Summary of this part (1/2)

Split item and greedy algorithms:

- ▶ the split item s is the first item that does not fit, taking the items by non-increasing efficiency; rc is the residual capacity
- ▶ GR adds every item that fits; GR^e returns the better of the greedy solution and the best single item

Guessing items:

- ▶ H^2 guesses two items of an optimal solution, H^ℓ guesses ℓ items, and both complete each guess with GR^e on a residual instance
- ▶ running times $O(n^3)$ and $O(n^{\ell+1})$

Summary of this part (2/2)

Rounding the profits:

- ▶ the Basic-FPTAS algorithm FP^ε scales the profits with $K = \varepsilon GR^e(I)/n$ and solves the scaled instance exactly by dynamic programming by profits, in $O(n^2/\varepsilon)$ time

Next:

- ▶ the worst case performance ratio of each algorithm, the approximation schemes, and the absolute performance guarantee