

SAPIENZA
UNIVERSITÀ DI ROMA

Approximation Algorithms

Volume 1 — Approximation algorithms

Approximation algorithms: performance ratios and approximation schemes, with algorithms and worst-case analyses for knapsack, bin packing, the metric TSP and parallel machine scheduling, and the limits of approximation for vertex coloring, maximum clique and the general TSP.

Fabio Furini

fabiofurini.github.io

Department of Computer, Control and Management Engineering — Sapienza University of Rome

Online version and downloadable material:
fabiofurini.github.io/approximation-algorithms/
github.com/fabiofurini/approximation-algorithms

Academic year 2026–2027

Contents

1 Principles of approximation algorithms **4**

1.1 Approximation algorithms . . . 5
1.2 Approximation schemes . . . 8
1.3 Typical mistakes . . . 10
1.4 In summary . . . 11

Knapsack problems **12**

2 Approximation algorithms for the knapsack problem **13**

2.1 Notation . . . 14
2.2 Split item and residual capacity . . . 14
2.3 The greedy algorithm . . . 15
2.4 The Ext-Greedy algorithm . . . 20
2.5 The H^2 algorithm . . . 23
2.6 The H^ℓ algorithm . . . 26
2.7 The Basic-FPTAS algorithm . . . 28
2.8 Performance guarantees . . . 30
2.9 Absolute performance guarantee . . . 42
2.10 Typical mistakes . . . 44
2.11 In summary . . . 44
2.12 `C++` implementation . . . 45

Packing problems **55**

3 Approximation algorithms for bin packing **56**

3.1 Notation . . . 57
3.2 The Next Fit . . . 57
3.3 The Any Fit class . . . 59
3.4 Half-full property . . . 70
3.5 Performance guarantees . . . 73
3.6 Typical mistakes . . . 80
3.7 In summary . . . 80

3.8 C++ implementation 81

Coloring problems **90**

4 Vertex coloring: limits of approximation **91**
4.1 Notation 92
4.2 The VCP is not approximable 92
4.3 Special cases 93
4.4 Examples 97
4.5 Sources 98
4.6 Typical mistakes 99
4.7 In summary 99

Clique problems **101**

5 Maximum clique: limits of approximation **102**
5.1 Notation 103
5.2 The MCP is not approximable 103
5.3 Special cases 105
5.4 Example 107
5.5 Sources 107
5.6 Typical mistakes 108
5.7 In summary 108

Routing problems **109**

6 The TSP: the general case and the metric case **110**
6.1 The general TSP is not approximable 111
6.2 A special case: the metric TSP 112
6.3 The double tree and the Christofides algorithms 112
6.4 Structural properties 115
6.5 Performance guarantees 117
6.6 Sources 120
6.7 Typical mistakes 120
6.8 In summary 120

Scheduling problems **122**

7 Approximation for $P||C_{\max}$ **123**
7.1 Notation 124
7.2 The algorithms $LIST$ and LPT 124

7.3 A property of list scheduling 127
7.4 Performance guarantees 127
7.5 Sources 130
7.6 Typical mistakes 130
7.7 In summary 131

Chapter 1

Principles of approximation algorithms

Prerequisites

- From the module *Heuristic Algorithms*, the chapter «Heuristic algorithms: definitions»: heuristic algorithms, feasible solutions and the value of the solution returned by a heuristic algorithm.
- From the module *Combinatorial Optimization Problems*, the chapters on the combinatorial optimization problems: instances, optimal solutions and the optimal value $z(I)$ of an instance I.

Objectives

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

- compute the performance ratio, the additive error and the relative error of a heuristic algorithm on an instance, for minimization and maximization problems;
- state when a number is a relative or an additive performance guarantee, and when it is tight;
- prove that a guarantee is tight by exhibiting an instance, or a sequence of instances;
- define an approximation scheme, a PTAS and an FPTAS, and translate the accuracy ε into a performance guarantee.

1.1 Approximation algorithms

Definition 1.1: heuristic value

Let $\mathscr{A}$ be a **heuristic algorithm** for an **optimization problem** $\mathscr{P}$. For every **instance** I of $\mathscr{P}$:

1. $z(I)$ is the **optimal value** of $\mathscr{P}$ on I (assumed positive);
2. $\mathscr{A}(I)$ is the **heuristic value**, i.e., the objective value of the feasible solution returned by $\mathscr{A}$ on I (assumed positive).

- We assume that $\mathscr{A}$ always outputs a **feasible solution**. Hence, for every instance I,

$$\begin{cases} \mathscr{A}(I) \geq z(I) \quad \text{and} \quad \dfrac{\mathscr{A}(I)}{z(I)} \geq 1 & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \mathscr{A}(I) \leq z(I) \quad \text{and} \quad \dfrac{z(I)}{\mathscr{A}(I)} \geq 1 & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

In general, a heuristic algorithm may sometimes generate solutions with an almost optimal value, but in other cases it may produce significantly inferior solutions. Accordingly, we are interested in "measuring" (whenever possible) the quality of the solutions provided by a heuristic algorithm.

- We adopt a **worst-case behaviour** point of view: we evaluate a heuristic algorithm by looking at the largest possible deviation, over all instances, between the heuristic value and the optimal value.

- There are two standard ways to measure this deviation:

1. **Relative performance guarantee** (multiplicative guarantee): in this case we measure the quality of the heuristic solution by looking at the *relative error*, that is, how far the heuristic value is from the optimum *in relative terms*, using ratios.

2. **Absolute performance guarantee** (additive guarantee): in this case we measure the quality of the heuristic solution by looking at the *absolute error*, that is, the difference between the heuristic value and the optimal value.

1.1.1 Relative performance guarantee

Definition 1.2: performance ratio

The **performance ratio** of $\mathscr{A}$ on an instance I is

$$\begin{cases} \dfrac{\mathscr{A}(I)}{z(I)} & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \dfrac{z(I)}{\mathscr{A}(I)} & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

- Under the feasibility assumption, the performance ratio is at least 1 for every instance I.

Definition 1.3: α-approximation algorithm

A heuristic algorithm $\mathscr{A}$ for an optimization problem $\mathscr{P}$ is an **α-approximation algorithm** if there exists a constant $\alpha \in [1, +\infty)$ such that, for every instance I of $\mathscr{P}$,

$$\begin{cases} \dfrac{\mathscr{A}(I)}{z(I)} \leq \alpha & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \dfrac{z(I)}{\mathscr{A}(I)} \leq \alpha & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

Any such constant α is called a **relative performance guarantee** of $\mathscr{A}$.

- A relative performance guarantee α for $\mathscr{A}$ is **tight** if no smaller constant $\alpha' < \alpha$ is also a relative performance guarantee for $\mathscr{A}$ (i.e., the same inequalities hold for all instances with α' in place of α).
- Tightness is usually proved by exhibiting either an instance I for which the performance ratio attains α, namely

  $$\frac{\mathscr{A}(I)}{z(I)} = \alpha \quad \text{(minimization)} \qquad \text{or} \qquad \frac{z(I)}{\mathscr{A}(I)} = \alpha \quad \text{(maximization)},$$

  or a sequence of instances $\{I_n\}$ such that the corresponding performance ratios get arbitrarily close to α.

Definition 1.4: Worst-case performance ratio

The **worst-case performance ratio** of $\mathscr{A}$ is

$$r(\mathscr{A}) = \begin{cases} \sup\left\{ \dfrac{\mathscr{A}(I)}{z(I)} : I \text{ is an instance of } \mathscr{P} \right\} & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \sup\left\{ \dfrac{z(I)}{\mathscr{A}(I)} : I \text{ is an instance of } \mathscr{P} \right\} & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

- The value $r(\mathscr{A})$ is the smallest constant α for which $\mathscr{A}$ is an α-approximation algorithm. The relative performance guarantee α is tight if and only if $\alpha = r(\mathscr{A})$.
- If $r(\mathscr{A}) < \infty$, then $\mathscr{A}$ admits a finite relative performance guarantee (equivalently, it has a bounded performance ratio). If $r(\mathscr{A}) = \infty$, then $\mathscr{A}$ has no relative worst-case guarantee.
- Terminology used in the literature
  - a finite relative performance guarantee α is also called an **approximation factor** (or **approximation guarantee**)
  - an **α-approximation algorithm** is also called a **constant-factor approximation algorithm**
  - $r(\mathscr{A})$ is also called the **(worst-case) approximation ratio** of $\mathscr{A}$

1.1.2 Additive performance guarantee

- The **additive error** (or **absolute error**) of $\mathscr{A}$ on an instance I is

$$\begin{cases} \mathscr{A}(I) - z(I) & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ z(I) - \mathscr{A}(I) & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

  Under the feasibility assumption, this quantity is non-negative for every instance I.

Definition 1.5: β-additive approximation algorithm

A heuristic algorithm $\mathscr{A}$ for an optimization problem $\mathscr{P}$ is a **β-additive approximation algorithm** if there exists a constant $\beta \in [0, +\infty)$ such that, for every instance I of $\mathscr{P}$,

$$\begin{cases} \mathscr{A}(I) - z(I) \leq \beta & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ z(I) - \mathscr{A}(I) \leq \beta & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

Any such constant β is called an **additive performance guarantee** of $\mathscr{A}$.

- An additive performance guarantee β for $\mathscr{A}$ is **tight** if no smaller constant $\beta' < \beta$ is also an additive performance guarantee for $\mathscr{A}$.
- Tightness is usually proved by exhibiting either an instance I for which the additive error attains β, namely

$$\mathscr{A}(I) - z(I) = \beta \quad \text{(minimization)} \qquad \text{or} \qquad z(I) - \mathscr{A}(I) = \beta \quad \text{(maximization)},$$

  or a sequence of instances $\{I_n\}$ such that the corresponding additive errors get arbitrarily close to β.

Definition 1.6: Worst-case additive error

The **worst-case additive error** of $\mathscr{A}$ is

$$e(\mathscr{A}) = \begin{cases} \sup\{\mathscr{A}(I) - z(I) : I \text{ instance of } \mathscr{P}\} & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \sup\{z(I) - \mathscr{A}(I) : I \text{ instance of } \mathscr{P}\} & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

- The value $e(\mathscr{A})$ is the smallest constant β for which $\mathscr{A}$ is a β-additive approximation algorithm. The additive performance guarantee β is tight if and only if $\beta = e(\mathscr{A})$.
- If $e(\mathscr{A}) < \infty$, then $\mathscr{A}$ admits a finite additive performance guarantee (equivalently, it has bounded additive error). If $e(\mathscr{A}) = \infty$, then $\mathscr{A}$ has no additive (absolute) worst-case guarantee.
- Terminology used in the literature
  - a finite additive performance guarantee β is also called an **absolute performance guarantee** or an **additive approximation guarantee**
  - $e(\mathscr{A})$ is also called the **worst-case absolute (additive) error** of $\mathscr{A}$

1.2 Approximation schemes

- The **relative error** (also called **percentage error**) of $\mathscr{A}$ on an instance I is

$$\begin{cases} \dfrac{\mathscr{A}(I) - z(I)}{z(I)} & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \dfrac{z(I) - \mathscr{A}(I)}{z(I)} & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

  Under the feasibility assumption, this quantity is non-negative for every instance I, and it is equal to 0 if and only if $\mathscr{A}(I) = z(I)$.
- The relative error divides by the optimal value $z(I)$, and must not be confused with the gap of the chapter «Combinatorial optimization: definitions» of the module *Combinatorial Optimization Problems*, which divides by the lower bound and is never smaller (chapter «Heuristic algorithms: definitions» of the module *Heuristic Algorithms*).

Definition 1.7: Approximation scheme

An **approximation scheme** for an optimization problem $\mathscr{P}$ is a family of heuristic algorithms $\{\mathscr{A}_\varepsilon\}_{\varepsilon \in (0,1)}$ such that, for every $\varepsilon \in (0,1)$ and for every instance I of $\mathscr{P}$,

$$\begin{cases} \dfrac{\mathscr{A}_\varepsilon(I) - z(I)}{z(I)} \leq \varepsilon & \text{if } \mathscr{P} \text{ is a minimization problem,} \\ \dfrac{z(I) - \mathscr{A}_\varepsilon(I)}{z(I)} \leq \varepsilon & \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

- For minimization problems,

$$\frac{\mathscr{A}_\varepsilon(I) - z(I)}{z(I)} \leq \varepsilon \qquad \text{for every instance } I \text{ of } \mathscr{P}.$$

implies

$$\mathscr{A}_\varepsilon(I) - z(I) \leq \varepsilon\, z(I) \quad \Longleftrightarrow \quad \mathscr{A}_\varepsilon(I) \leq (1+\varepsilon)\, z(I).$$

Dividing by $z(I) > 0$,

$$\frac{\mathscr{A}_\varepsilon(I)}{z(I)} \leq 1 + \varepsilon \qquad \text{for every instance } I \text{ of } \mathscr{P}.$$

Hence, $\mathscr{A}_\varepsilon$ is a $(1+\varepsilon)$-approximation algorithm.

- For maximization problems,

$$\frac{z(I) - \mathscr{A}_\varepsilon(I)}{z(I)} \leq \varepsilon \qquad \text{for every instance } I \text{ of } \mathscr{P}.$$

implies

$$z(I) - \mathscr{A}_\varepsilon(I) \leq \varepsilon\, z(I) \quad \Longleftrightarrow \quad \mathscr{A}_\varepsilon(I) \geq (1-\varepsilon)\, z(I).$$

Since $0 < \varepsilon < 1$ and $z(I) > 0$, dividing by $(1-\varepsilon)z(I)$ gives

$$\frac{z(I)}{\mathscr{A}_\varepsilon(I)} \leq \frac{1}{1-\varepsilon} \qquad \text{for every instance } I \text{ of } \mathscr{P}.$$

Hence, $\mathscr{A}_\varepsilon$ is a $\left(\frac{1}{1-\varepsilon}\right)$-approximation algorithm.

It is quite natural that the running time of an approximation scheme increases as ε decreases: the smaller the allowed error, the higher the required running time.

- The quantity $\frac{1}{\varepsilon}$ is often called the **accuracy**. Indeed, from the defining inequalities of an approximation scheme we obtain

$$\frac{\mathscr{A}_\varepsilon(I) - z(I)}{z(I)} \leq \varepsilon \implies \mathscr{A}_\varepsilon(I) - z(I) \leq \varepsilon\, z(I) \implies \frac{z(I)}{\mathscr{A}_\varepsilon(I) - z(I)} \geq \frac{1}{\varepsilon} \qquad \text{(minimization)},$$

$$\frac{z(I) - \mathscr{A}_\varepsilon(I)}{z(I)} \leq \varepsilon \implies z(I) - \mathscr{A}_\varepsilon(I) \leq \varepsilon\, z(I) \implies \frac{z(I)}{z(I) - \mathscr{A}_\varepsilon(I)} \geq \frac{1}{\varepsilon} \qquad \text{(maximization)}.$$

In the special case where $\mathscr{A}_\varepsilon(I) = z(I)$, the denominators above are zero; in this case we interpret the corresponding fractions as $+\infty$, and the inequalities hold trivially.

- Not all approximation schemes are equally useful in practice: we distinguish them according to how the running time depends on the input size and on the accuracy $\frac{1}{\varepsilon}$.

Definition 1.8: Polynomial-time approximation scheme (PTAS)

An approximation scheme $\{\mathscr{A}_\varepsilon\}_{\varepsilon\in(0,1)}$ is a **polynomial-time approximation scheme** if, for every fixed $\varepsilon \in (0,1)$, the running time of $\mathscr{A}_\varepsilon$ is polynomial in the size of the instance.

- Thus, the running time is polynomial in the input size, but it may grow very fast as ε decreases (i.e., as the required accuracy increases).

Definition 1.9: Fully polynomial-time approximation scheme (FPTAS)

An approximation scheme $\{\mathscr{A}_\varepsilon\}_{\varepsilon\in(0,1)}$ is a **fully polynomial-time approximation scheme** if the running time of $\mathscr{A}_\varepsilon$ is polynomial both in the size of the instance and in the accuracy $\frac{1}{\varepsilon}$.

1.3 Typical mistakes

Warning

- **Inverting the ratio.** The performance ratio is $\frac{\mathscr{A}(I)}{z(I)}$ for a minimization problem and $\frac{z(I)}{\mathscr{A}(I)}$ for a maximization problem: in both cases it is at least 1.
- **Proving a guarantee with examples.** An instance on which the ratio equals α shows that no constant below α is a guarantee, but it does not prove that α is one: this needs a proof for *every* instance.
- **Calling a guarantee tight without an instance.** A guarantee α is tight only if some instance, or a sequence of instances, reaches α in the limit.
- **Confusing ε and α.** An approximation scheme with relative error ε is a $(1+\varepsilon)$-approximation algorithm for a minimization problem, but a $\frac{1}{1-\varepsilon}$-approximation algorithm for a maximization problem.
- **PTAS and FPTAS.** The running time of a PTAS may grow like $n^{1/\varepsilon}$; only for an FPTAS it is polynomial in $\frac{1}{\varepsilon}$ as well.

1.4 In summary

Summary

- A relative performance guarantee bounds the ratio $\mathscr{A}(I)/z(I)$ (minimization) or $z(I)/\mathscr{A}(I)$ (maximization) over all instances; an additive guarantee bounds the difference.
- A guarantee is tight if it is reached by an instance, or approached by a sequence of instances; the smallest guarantee is the worst-case ratio $r(\mathscr{A})$, or the worst-case additive error $e(\mathscr{A})$.
- An approximation scheme has relative error at most ε for every $\varepsilon \in (0, 1)$; it is a PTAS if the running time is polynomial in the size of the instance for every fixed ε, and an FPTAS if it is polynomial in $\frac{1}{\varepsilon}$ too.

Dashboard. A guarantee bounds the gap between a heuristic solution (a lower bound on $z(I)$ for a maximization problem, an upper bound for a minimization problem) and the optimal value, on every instance. The dashboards of the chapters of this volume collect, for the reference instances, the values of the heuristic algorithms and their guarantees; for the VCP, the MCP and the general TSP no constant guarantee exists, unless $P = NP$.

Knapsack problems

Chapter 2

Approximation algorithms for the knapsack problem

Prerequisites

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

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.

The problem of this chapter is the 0/1 knapsack problem (KP): its definition and its assumptions are on its page.

2.1 Notation

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

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

$$\mathscr{A}(I) \leq z(I) \qquad \text{and} \qquad \frac{z(I)}{\mathscr{A}(I)} \geq 1$$

  for every KP instance I.

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

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

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

2.2 Split item and residual capacity

Definition 2.1: split item and residual capacity

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

$$s \in \mathscr{I} \quad \text{such that} \quad \sum_{i=1}^{s-1} w_i \leq c \quad \text{and} \quad \sum_{i=1}^{s} w_i > c \tag{2.1}$$

and the **residual capacity** is

$$rc = c - \sum_{i=1}^{s-1} w_i \tag{2.2}$$

- The split item is the first one that does not fit entirely in the knapsack if all the previous ones are all 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. It is worth knowing that 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 2.1: split item and residual capacity

We consider a knapsack of capacity $c = 9$ and a set $\mathscr{I} = \{1, 2, 3, 4, 5, 6, 7\}$ 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:

$$\frac{p_1}{w_1} = \frac{6}{2}, \quad \frac{p_2}{w_2} = \frac{5}{3}, \quad \frac{p_3}{w_3} = \frac{8}{6}, \quad \frac{p_4}{w_4} = \frac{9}{7}, \quad \frac{p_5}{w_5} = \frac{6}{5}, \quad \frac{p_6}{w_6} = \frac{7}{9}, \quad \frac{p_7}{w_7} = \frac{3}{4}.$$

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

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

and the residual capacity is:

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

2.3 The greedy algorithm

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

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

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.

- The pseudo-code of GR reads as follows:

Algorithm

Algorithm: Greedy

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] ← 0                                  // item i is initially not selected
5      end
6      rc := c                                                // residual capacity
7      for i := 1 to n do
8          if w[i] ≤ rc then
9              x[i] ← 1                                                // select item i
10             rc ← rc − w[i]                          // update residual capacity
11         end
12     end
13     return x
```

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

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

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

Example 2.2: Execution of GR

For the KP instance of Example 2.1, the items are sorted by non-increasing efficiency:

$$(1,\ 2,\ 3,\ 4,\ 5,\ 6,\ 7).$$

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 $7 - 3 = 4$.
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 $4 - 4 = 0$.

Hence GR returns the feasible solution:

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

The total profit of the feasible solution is:

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

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

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

In this instance the split item is $s = 3$. Accordingly, 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 2.3: Execution of GR

We consider a knapsack of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, 3, 4\}$ 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). 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:

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

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 $5 - 5 = 0$.

Hence GR returns the feasible solution:

$$S = \{1, 2, 4\}.$$

The total profit of the feasible solution is:

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

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

$$18 \leq z(I).$$

In this instance the split item is $s = 3$. Accordingly, GR selects all items preceding the split item, i.e., items 1 and 2, does not select item 3 (the split item), and then selects item 4, which follows the split item. 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 2.4: Execution of GR

We consider a knapsack of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, 3, 4, 5\}$ 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:

$$\frac{p_1}{w_1} = \frac{4}{1} = 4, \quad \frac{p_2}{w_2} = \frac{6}{2} = 3, \quad \frac{p_3}{w_3} = \frac{5}{2} = 2.5, \quad \frac{p_4}{w_4} = \frac{12}{6} = 2, \quad \frac{p_5}{w_5} = \frac{3}{2} = 1.5.$$

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

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

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 $5 - 2 = 3$.

Hence GR returns the feasible solution:

$$S = \{1, 2, 3, 5\}.$$

The total profit of the feasible solution is:

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

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

$$18 \leq z(I).$$

In this instance the split item is $s = 4$. Accordingly, 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 2.5: Execution of GR

We consider a knapsack of capacity $c = 21$ and a set $\mathscr{I} = \{1, 2, 3, 4, 5, 6\}$ 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:

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

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

$$\sum_{i=1}^{5} w_i = 1 + 2 + 2 + 2 + 2 = 9 \leq 21 \quad \text{and} \quad \sum_{i=1}^{6} w_i = 1 + 2 + 2 + 2 + 2 + 21 = 30 > 21$$

and the residual capacity is:

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

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 $18 - 2 = 16$.
4. Item 4 (weight $w_4 = 2$) is selected. The available capacity rc becomes $16 - 2 = 14$.
5. Item 5 (weight $w_5 = 2$) is selected. The available capacity rc becomes $14 - 2 = 12$.
6. 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:

$$S = \{1, 2, 3, 4, 5\}.$$

The total profit of the feasible solution is:

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

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

$$19 \leq z(I).$$

In this instance, the item with maximum profit is item 6 with $p_6 = 20$, which appears last in the efficiency-sorted order. The greedy algorithm 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 $z(I) = 20$.

2.4 The Ext-Greedy algorithm

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.

- The pseudo-code of GR^e reads as follows:

Algorithm

Algorithm: Extended Greedy

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

Function *ExtendedGreedy*(n, p, w, c)
  $x := Greedy(n, p, w, c)$ // greedy solution
  $z := 0$ // value of the greedy solution
  for $i := 1$ **to** n **do**
    if $x[i] = 1$ **then**
      $z \leftarrow z + p[i]$ // add the profit of a selected item
    end
  end
  $i_m := 1$ // best single item
  $p_{\max} := p[1]$
  for $i := 2$ **to** n **do**
    if $p[i] > p_{\max}$ **then**
      $i_m \leftarrow i$ // update best single item
      $p_{\max} \leftarrow p[i]$ // update its profit
    end
  end
  $\bar{x} :=$ new array$[1..n]$ // best single-item solution
  for $i := 1$ **to** n **do**
    $\bar{x}[i] \leftarrow 0$
  end
  $\bar{x}[i_m] \leftarrow 1$ // select only item i_m
  if $z < p_{\max}$ **then**
    for $i := 1$ **to** n **do**
      $x[i] \leftarrow \bar{x}[i]$ // copy component by component
    end
  end
  return x

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

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

  and let $\bar{S}$ be the **best single-item feasible solution**

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

Example 2.6: Execution of GR^e

We start from the feasible solution computed by GR on the KP instance of Example 2.1 (Example 2.2):

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

We now compute the maximum profit of a single item:

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

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

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

Example 2.7: Execution of GR^e

We start from the feasible solution computed by GR on the KP instance of Example 2.3:

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

We now compute the maximum profit of a single item:

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

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

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

- 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 $z(I)$.

2.5 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

Algorithm: H^2

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

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

Function *H2*$(\mathscr{I}, p, w, c)$
  $p_{\max} := \max\{p[j] : j \in \mathscr{I}\}$
  $lb := p_{\max}$ `// best value found`
  $\mathscr{P} := \{(i, k) \in \mathscr{I} \times \mathscr{I} : i < k,\ w[i] + w[k] \leq c\}$
  foreach (i, k) ***in*** $\mathscr{P}$ **do**
    $rc := c - w[i] - w[k]$ `// residual capacity`
    $\mathscr{I}(i, k) := \{j \in \mathscr{I} \setminus \{i, k\} : p[j] \leq \min\{p[i], p[k]\},\ w[j] \leq rc\}$
    $I_{ik} :=$ KP instance restricted to $\mathscr{I}(i, k)$ with capacity rc
    $z_{res} := GR^e(I_{ik})$ `// value of` GR^e `on the residual instance`
    $lb_{ik} := p[i] + p[k] + z_{res}$
    if $lb_{ik} > lb$ **then**
      $lb \leftarrow lb_{ik}$
    end
  end
  return lb

Definition 2.2: feasible ordered pairs of items

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

$$\mathscr{P} = \left\{ \ (i, k) \in \mathscr{I} \times \mathscr{I} : \ i < k, \ \ w_i + w_k \leq c \ \right\}. \tag{2.3}$$

- 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 $\mathscr{P}$, and completes each guess on a residual instance.

- The H^2 algorithm considers the item pairs $(i,k) \in \mathscr{P}$, 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 item** is:

  $$\mathscr{I}(i,k) = \left\{ j \in \mathscr{I} \setminus \{i,k\} : \quad p_j \leq \min\{p_i, p_k\}, \quad w_j \leq c - w_i - w_k \right\}$$

  The items with weight larger than the residual capacity are also discarded.
  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 $(i,k) \in \mathscr{P}$, we have the lower bound:

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

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

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

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 2.8: H^2 algorithm

We consider a set $\mathscr{I} = \{1,2,3,4,5,6,7\}$ 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 $p_{\max} = 9$. 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 $GR^e(I) = 6 + 5 + 8 = 19$.

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

- Pair $(1,2)$: the residual capacity is $rc = 12 - 2 - 3 = 7$, $\mathscr{I}(1,2) = \{7\}$ and $GR^e\big(I(1,2)\big) = 3$, hence $lb(1,2) = 6 + 5 + 3 = 14 > lb$ and lb is updated to 14.
- Pair $(1,3)$: the residual capacity is $rc = 12 - 2 - 6 = 4$, $\mathscr{I}(1,3) = \{2,7\}$ and $GR^e\big(I(1,3)\big) = 5$, hence $lb(1,3) = 6 + 8 + 5 = 19 > lb$ and lb is updated to 19.
- Pair $(1,4)$: the residual capacity is $rc = 12 - 2 - 7 = 3$, $\mathscr{I}(1,4) = \{2\}$ and $GR^e\big(I(1,4)\big) = 5$, hence $lb(1,4) = 6 + 9 + 5 = 20 > lb$ and lb is updated to 20.
- Pair $(1,5)$: the residual capacity is $rc = 12 - 2 - 5 = 5$, $\mathscr{I}(1,5) = \{2,7\}$ and $GR^e\big(I(1,5)\big) = 5$, hence $lb(1,5) = 6 + 6 + 5 = 17 < lb$.
- Pair $(1,6)$: the residual capacity is $rc = 12 - 2 - 9 = 1$, $\mathscr{I}(1,6) = \varnothing$, hence $lb(1,6) = 6 + 7 = 13 < lb$.
- Pair $(1,7)$: the residual capacity is $rc = 12 - 2 - 4 = 6$, $\mathscr{I}(1,7) = \varnothing$, 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.

Proposition 2.3

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 execution 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 $O(n^2)$.

2.6 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

Algorithm: H^ℓ

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

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

Function $Hl(\mathscr{I}, p, w, c, \ell)$
  $lb := \max\left\{\sum_{j\in S} p[j] : S \in \widehat{T}(\ell)\right\}$ `// feasible sets of at most` $\ell - 1$ `items`
  foreach L ***in*** $T(\ell)$ `// feasible sets of exactly` ℓ `items`
  do
    $rc := c - \sum_{i\in L} w[i]$ `// residual capacity`
    $\mathscr{I}(L) := \left\{j \in \mathscr{I} \setminus L : p[j] \le \min_{i\in L} p[i],\ w[j] \le rc\right\}$
    $lb_L := \sum_{i\in L} p[i] + GR^e\big(I(L)\big)$ `// complete the guess`
    if $lb_L > lb$ **then**
      $lb \leftarrow lb_L$
    end
  end
  return lb

- 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 $2 \le \ell \le n$. For $\ell = 2$ it is exactly the H^2 algorithm.

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

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

and

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

- 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
  $$\mathscr{I}(L) = \left\{ j \in \mathscr{I} \setminus L : \quad p_j \leq \min\{p_i : i \in L\}, \quad w_j \leq c - \sum_{i \in L} w_i \right\};$$
  3. it completes the partial solution by running GR^e on $I(L)$, which gives the value
  $$lb(L) = \sum_{i \in L} p_i + GR^e\big(I(L)\big).$$

- The value returned by the algorithm is
$$H^\ell(I) = \max \left\{ \max_{L \in T(\ell)} lb(L), \quad \max_{S \in \widehat{T}(\ell)} \sum_{j \in S} p_j \right\}.$$

Example 2.9: Execution of H^ℓ

We consider a knapsack of capacity $c = 12$ and a set $\mathscr{I} = \{1, 2, 3, 4, 5\}$ 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 $z(I) = 12$.

With $\ell = 2$, i.e., the H^2 algorithm of Section 2.5. A pair of two items of profit 3, for instance $L = \{2, 3\}$, leaves residual capacity 6 and the residual set $\mathscr{I}(L) = \{1, 4, 5\}$: 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 $H^2(I) = 11 < 12$.

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 $\mathscr{I}(L) = \{1, 5\}$: 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.

Observation 2.4

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 $O(n)$.

2.7 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.

The *Basic-FPTAS* algorithm (denoted by FP^ε) replaces the item profits by scaled profits, which take few distinct values, and then 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

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 $x[1..n]$

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

- The **scaling factor** is
$$K = \frac{\varepsilon\, GR^e(I)}{n},$$
and the **scaled profits** are obtained by rounding down the profits to a multiple of K:
$$p'_j = \left\lfloor \frac{p_j}{K} \right\rfloor \qquad \text{for all} \qquad j \in \mathscr{I}.$$
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, and rounding them 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 second DP algorithm of the chapter «DP for the 0/1 knapsack problem» of the module *Dynamic Programming*: $u(j, q)$ is the minimum total weight of a subset of $\{1, 2, \ldots, j\}$ of total profit exactly q, and the optimal value is the largest q with $u(n, q) \le c$.

- 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 $\frac{z(I)}{K} \le \frac{2\,GR^e(I)}{K} = \frac{2n}{\varepsilon}$ by the guarantee $z(I) \le 2\,GR^e(I)$ of GR^e. The table has size $n \times Q$, and the running time is $O(n\,Q) = O(n^2/\varepsilon)$.

Example 2.10: Execution of FP^ε

We consider a knapsack of capacity $c = 8$ and a set $\mathscr{I} = \{1, 2, 3, 4\}$ 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 $\varepsilon = \frac{1}{2}$.

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 $19 \le 38$, hence
$$GR^e(I) = 38.$$

Scaling. The scaling factor and the scaled profits are
$$K = \frac{\varepsilon\, GR^e(I)}{n} = \frac{\frac{1}{2} \cdot 38}{4} = 4.75, \qquad p'_1 = p'_2 = \left\lfloor \frac{19}{4.75} \right\rfloor = 4, \qquad p'_3 = p'_4 = \left\lfloor \frac{14}{4.75} \right\rfloor = 2.$$

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.

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.

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

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

2.8 Performance guarantees

- We now determine the worst case performance ratio of the algorithms of the previous sections. Recall that, the KP being a maximization problem, the performance ratio of an algorithm $\mathscr{A}$ on an instance I is $z(I)/\mathscr{A}(I) \geq 1$, and that $r(\mathscr{A})$ is its supremum over all the instances.

2.8.1 The worst case performance ratio of GR

Proposition 2.5

There are KP instances I such that:

$$\frac{z(I)}{GR(I)} \to +\infty \qquad \text{as } c \to +\infty. \tag{2.4}$$

Proof. Fix an integer $\vartheta > 2$ and consider the KP instance I_ϑ with knapsack capacity $c = \vartheta$ and the items set $\mathscr{I} = \{1, 2\}$, where the item profits and weights are

$$p_1 = 2, \quad w_1 = 1, \qquad \text{and} \qquad p_2 = \vartheta, \quad w_2 = \vartheta.$$

An optimal solution packs item 2, and therefore

$$z(I_\vartheta) = \vartheta.$$

Since

$$\frac{p_1}{w_1} = 2 \; > \; \frac{p_2}{w_2} = \frac{\vartheta}{\vartheta} = 1,$$

GR packs item 1 first, leaving available capacity $\vartheta - 1$, so item 2 cannot be added. Thus

$$GR(I_\vartheta) = 2.$$

Hence,

$$\lim_{\vartheta \to +\infty} \frac{z(I_\vartheta)}{GR(I_\vartheta)} = \lim_{\vartheta \to +\infty} \frac{\vartheta}{2} = +\infty.$$

□

- The proposition shows that there is no constant $\alpha \in [1, +\infty)$ such that

$$\frac{z(I)}{GR(I)} \leq \alpha \qquad \text{for all instances } I.$$

  Equivalently, the ratio $\frac{z(I)}{GR(I)}$ can be made arbitrarily large by a suitable choice of the instance.

  In other words, there is no constant $\alpha \in [1, +\infty)$ such that GR is an α-approximation algorithm and the worst-case performance ratio of GR is unbounded:

$$r(GR) = +\infty.$$

- A key drawback of GR is that it may fail to select the item with maximum profit (even when that single item is optimal). In particular, by prioritizing efficiency p_i/w_i, GR can fill the knapsack with high-efficiency items and leave insufficient available capacity to include the maximum-profit item.

2.8.2 The worst case performance ratio of GR^e

Proposition 2.6

For every KP instance I, we have:

$$\frac{z(I)}{GR^e(I)} \leq 2. \tag{2.5}$$

Proof. Assume items are sorted by non-increasing efficiency. Let s be the split item and rc be the residual capacity. We have

$$\underbrace{\sum_{i=1}^{s-1} p_i \; + \; p_s \frac{rc}{w_s}}_{= z\left(\mathrm{LP_{KP}}\right)} \leq \sum_{i=1}^{s-1} p_i \; + \; p_s \qquad \text{since} \qquad rc < w_s,$$

where $z(\mathrm{LP_{KP}})$ is the optimal solution value of the LP relaxation of the natural ILP formulation of the KP. Accordingly,

$$z(I) \leq z(\mathrm{LP_{KP}}) \leq \sum_{i=1}^{s-1} p_i + p_s \leq \sum_{i=1}^{s-1} p_i + p_{\max},$$

since by definition of $p_{\max}$ we have $p_{\max} \geq p_s$.

Moreover,

$$GR^e(I) \geq GR(I) \geq \sum_{i=1}^{s-1} p_i \quad \text{and} \quad GR^e(I) \geq p_{\max}.$$

Summing these two inequalities yields

$$2\,GR^e(I) = GR^e(I) + GR^e(I) \;\geq\; \sum_{i=1}^{s-1} p_i + p_{\max}.$$

Therefore,

$$z(I) \leq \sum_{i=1}^{s-1} p_i + p_{\max} \;\leq\; 2\,GR^e(I).$$

Dividing by $GR^e(I) > 0$ yields

$$\frac{z(I)}{GR^e(I)} \leq 2.$$

□

Proposition 2.7

There are KP instances I such that:

$$\frac{z(I)}{GR^e(I)} \to 2 \quad \text{as} \quad c \to +\infty. \tag{2.6}$$

Proof. Fix an integer $\vartheta > 2$ and consider the KP instance I_ϑ with capacity $c = 2\vartheta$ and 3 items:

$$p_1 = 2,\ w_1 = 1, \qquad p_2 = \vartheta,\ w_2 = \vartheta, \qquad p_3 = \vartheta,\ w_3 = \vartheta.$$

Items 2 and 3 fit together into the knapsack since $w_2 + w_3 = 2\vartheta = c$; hence an optimal solution packs items 2 and 3 and

$$z(I_\vartheta) = 2\vartheta.$$

Moreover,

$$\frac{p_1}{w_1} = 2 \;>\; \frac{p_2}{w_2} = \frac{p_3}{w_3} = 1,$$

so GR packs item 1 first. After packing item 1, the available capacity is $2\vartheta - 1$, hence exactly one of items 2 and 3 can be added. Therefore GR returns a feasible solution of

value $\vartheta + 2$. Since $p_{\max} = \max\{p_1, p_2, p_3\} = \vartheta$, we have $\vartheta + 2 > p_{\max}$, and thus

$$GR^e(I_\vartheta) = \vartheta + 2.$$

Consequently,

$$\lim_{\vartheta \to +\infty} \frac{z(I_\vartheta)}{GR^e(I_\vartheta)} = \lim_{\vartheta \to +\infty} \frac{2\vartheta}{\vartheta + 2} = \lim_{\vartheta \to +\infty} \frac{2}{1 + \frac{2}{\vartheta}} = 2.$$

□

From Proposition 2.6 and Proposition 2.7 it follows that the worst case performance ratio of GR^e is two, i.e., we have

$$r(GR^e) = 2$$

and GR^e is a 2-approximation algorithm.

- A key strength of GR^e is that it mitigates the main weakness of GR: even if the greedy packing misses a very profitable item because of the efficiency ordering, GR^e explicitly compares the greedy solution with the best single-item feasible solution and returns the better one, thereby guaranteeing a constant-factor approximation algorithm.
- However, the guarantee $r(GR^e) = 2$ is tight: there are instances in which GR^e is misled by efficiency and, after selecting an early high-efficiency item, it no longer has enough capacity to pack together the two items with the largest profits (which form the optimal solution), yielding a ratio arbitrarily close to 2.

- Getting a solution which is in the worst-case only half as good as the optimal solution does not sound very impressive.
- Hence, it is worthwhile trying to find a better approximation algorithm, possibly at the cost of an increased running time.

- In the previous worst-case example the GR^e made the "mistake" of excluding item 3 which had a very high profit value. To avoid such a mistake we need more advanced heuristic algorithms.

2.8.3 The worst case performance ratio of H^2

Proposition 2.8

For every KP instance I, we have:

$$\frac{z(I)}{H^2(I)} \leq \frac{4}{3}. \tag{2.7}$$

Proof. If $z(I)$ is generated by a single item, we have $H^2(I) = z(I)$.

Otherwise consider a feasible pair $(\tilde{\imath}, \tilde{k}) \in \mathscr{P}$ which contributes the highest profit in an optimal solution. The residual instance is $I(\tilde{\imath}, \tilde{k})$ and the residual set of items is $\mathscr{I}(\tilde{\imath}, \tilde{k})$. The H^2 algorithm considers this pair of items during its execution and applies GR^e on the residual instance $I(\tilde{\imath}, \tilde{k})$.

The pair contains the two items of largest profit of an optimal solution, so every other item of that solution has profit at most $\min\{p_{\tilde{\imath}}, p_{\tilde{k}}\}$ and fits into the residual capacity: it belongs to $\mathscr{I}(\tilde{\imath}, \tilde{k})$. The optimal solution value can therefore be written as:

$$z(I) = z\big(I(\tilde{\imath}, \tilde{k})\big) + p_{\tilde{\imath}} + p_{\tilde{k}} \tag{2.8}$$

where $z\big(I(\tilde{\imath}, \tilde{k})\big)$ is the optimal solution value of the residual instance $I(\tilde{\imath}, \tilde{k})$.

From of the operations of H^2, we have:

$$H^2(I) \quad \geq \quad p_{\tilde{\imath}} + p_{\tilde{k}} + GR^e\big(I(\tilde{\imath}, \tilde{k})\big) \tag{2.9}$$

and since GR^e is a 2-approximation algorithm, i.e., $GR^e(I) \geq \frac{1}{2}\, z(I)$, we have:

$$GR^e\big(I(\tilde{\imath}, \tilde{k})\big) \quad \geq \quad \frac{1}{2}\, z\big(I(\tilde{\imath}, \tilde{k})\big)$$

Accordingly:

$$H^2(I) \geq p_{\tilde{\imath}} + p_{\tilde{k}} + \frac{1}{2}\, z\big(I(\tilde{\imath}, \tilde{k})\big)$$

Moreover, due to (2.8), we have:

$$H^2(I) \geq p_{\tilde{\imath}} + p_{\tilde{k}} + \frac{1}{2}\big(z(I) - p_{\tilde{\imath}} - p_{\tilde{k}}\big) = \frac{1}{2}\big(z(I) + p_{\tilde{\imath}} + p_{\tilde{k}}\big)$$

We consider two possible cases:

- Case i):

  $$p_{\tilde{\imath}} + p_{\tilde{k}} \geq \frac{1}{2}\, z(I)$$

  We have in this case:

  $$H^2(I) \geq \frac{1}{2}\big(z(I) + p_{\tilde{\imath}} + p_{\tilde{k}}\big)$$

  $$\geq \frac{1}{2}\left(z(I) + \frac{1}{2}\, z(I)\right) = \frac{3}{4}\, z(I)$$

and accordingly

$$H^2(I) \geq \frac{3}{4}\, z(I)$$

- Case ii):

$$p_{\tilde{\imath}} + p_{\tilde{k}} < \frac{1}{2}\, z(I)$$

Since their profits sum to less than $\frac{1}{2}\, z(I)$, at least one of the two items $\tilde{\imath}$, $\tilde{k}$ has a profit smaller than $\frac{1}{4}\, z(I)$.

From $\mathrm{LP_{KP}}$ of the residual instance $I(\tilde{\imath}, \tilde{k})$, we have:

$$\begin{aligned} z\big(I(\tilde{\imath}, \tilde{k})\big) &\leq z\big(\mathrm{LP_{KP}}\big) \\ &\leq GR^e\big(I(\tilde{\imath}, \tilde{k})\big) + p_s \\ &\leq GR^e\big(I(\tilde{\imath}, \tilde{k})\big) + \frac{1}{4}\, z(I) \end{aligned}$$

where s is the split item of $I(\tilde{\imath}, \tilde{k})$ and $p_s < \frac{1}{4} z(I)$ since all its items have a profit at most $\min\{p_{\tilde{\imath}}, p_{\tilde{k}}\} < \frac{1}{4} z(I)$ (by construction of the residual instance). If all the items of the residual instance fit into the residual capacity there is no split item: then GR^e packs all of them, $z\big(I(\tilde{\imath}, \tilde{k})\big) = GR^e\big(I(\tilde{\imath}, \tilde{k})\big)$, and the inequalities hold with $p_s := 0$.

We have:

$$\begin{aligned} z(I) &= p_{\tilde{\imath}} + p_{\tilde{k}} + z\big(I(\tilde{\imath}, \tilde{k})\big) \\ &\leq \underbrace{p_{\tilde{\imath}} + p_{\tilde{k}} + GR^e\big(I(\tilde{\imath}, \tilde{k})\big)}_{\leq H^2(I),\ \text{eq.(2.9)}} + \frac{1}{4}\, z(I) \\ &\leq H^2(I) + \frac{1}{4}\, z(I) \end{aligned}$$

and accordingly:

$$H^2(I) \geq \frac{3}{4}\, z(I)$$

In both cases $H^2(I) \geq \frac{3}{4}\, z(I)$, that is, dividing by $H^2(I) > 0$,

$$\frac{z(I)}{H^2(I)} \leq \frac{4}{3}.$$

□

Proposition 2.9

There are KP instances I_M such that:

$$\frac{z(I_M)}{H^2(I_M)} \to \frac{4}{3} \qquad \text{as} \qquad M \to +\infty. \tag{2.10}$$

Proof. Let us consider the KP instance I_M with capacity $c = 4\,M$ and 5 items:

$$p_1 = 2, \quad w_1 = 1, \quad \text{and} \quad p_2 = p_3 = p_4 = p_5 = M, \quad w_2 = w_3 = w_4 = w_5 = M,$$

where $M \in \mathbb{Z}_{\geq 1}, M > 2$ is an integer number. An optimal solution packs the four big items, since $w_2 + w_3 + w_4 + w_5 = 4\,M = c$, hence

$$z(I_M) = 4\,M.$$

The efficiencies are

$$\frac{p_1}{w_1} = 2 \quad > \quad \frac{p_2}{w_2} = \frac{p_3}{w_3} = \frac{p_4}{w_4} = \frac{p_5}{w_5} = \frac{M}{M} = 1$$

so item 1 comes first in the efficiency order.

We consider the two possible kinds of pair examined by H^2:

1. a pair of two big items: the residual capacity is $2\,M$ and the residual instance keeps item 1 and the two remaining big items, so GR^e packs item 1 and one big item, for a total value of $3\,M + 2$;
2. a pair consisting of item 1 and one big item: the residual instance is empty, because no item has profit at most $\min\{2, M\} = 2$, for a total value of $M + 2$.

The first kind of pair gives the larger value, hence

$$H^2(I_M) = 3\,M + 2$$

Consequently,

$$\lim_{M\to\infty} \frac{z(I_M)}{H^2(I_M)} = \lim_{M\to\infty} \frac{4\,M}{3\,M+2} = \lim_{M\to\infty} \frac{4}{3 + \frac{2}{M}} = \frac{4}{3}.$$

□

From Proposition 2.8 and Proposition 2.9 it follows that the worst case performance ratio of H^2 is four thirds, i.e., we have

$$r(H^2) = \frac{4}{3}$$

and H^2 is a $\frac{4}{3}$-approximation algorithm. Guessing two items instead of one brings the ratio

down from the 2 of GR^e to $\frac{4}{3}$.

2.8.4 The worst case performance ratio of H^ℓ

Proposition 2.10

For every KP instance I, we have:

$$\frac{z(I)}{H^\ell(I)} \leq \frac{\ell+2}{\ell+1}. \tag{2.11}$$

Proof. If an optimal solution contains at most $\ell-1$ items, the algorithm finds it among the sets of $\widehat{T}(\ell)$, and $H^\ell(I) = z(I)$.

Otherwise, let $\tilde{L} \in T(\ell)$ be the set of the ℓ items of largest profit of an optimal solution. The algorithm considers $\tilde{L}$ during its execution and applies GR^e to the residual instance $I(\tilde{L})$. All the other items of the optimal solution belong to $\mathscr{I}(\tilde{L})$, hence

$$z(I) = \sum_{i\in\tilde{L}} p_i + z\big(I(\tilde{L})\big), \tag{2.12}$$

where $z\big(I(\tilde{L})\big)$ is the optimal value of the residual instance. From the operations of H^ℓ, and since GR^e is a 2-approximation algorithm,

$$H^\ell(I) \;\geq\; \sum_{i\in\tilde{L}} p_i + GR^e\big(I(\tilde{L})\big) \;\geq\; \sum_{i\in\tilde{L}} p_i + \frac{1}{2}\,z\big(I(\tilde{L})\big) \;=\; \frac{1}{2}\Big(z(I) + \sum_{i\in\tilde{L}} p_i\Big),$$

where the last equality follows from (2.12). We consider two cases.

- Case i): $\displaystyle\sum_{i\in\tilde{L}} p_i \geq \frac{\ell}{\ell+2}\,z(I)$. Then

$$H^\ell(I) \;\geq\; \frac{1}{2}\left(z(I) + \frac{\ell}{\ell+2}\,z(I)\right) = \frac{\ell+1}{\ell+2}\,z(I).$$

- Case ii): $\displaystyle\sum_{i\in\tilde{L}} p_i < \frac{\ell}{\ell+2}\,z(I)$. Then at least one item of $\tilde{L}$ has profit smaller than $\frac{1}{\ell+2}\,z(I)$, and so do all the items of the residual instance, whose profits are at most $\min\{p_i : i \in \tilde{L}\}$. Let s be the split item of $I(\tilde{L})$ (if there is none, GR^e packs all the items of $I(\tilde{L})$, and the inequalities below hold with $p_s := 0$). The LP relaxation of the residual instance gives

$$z\big(I(\tilde{L})\big) \;\leq\; z\big(\mathrm{LP}_{\mathrm{KP}}\big) \;\leq\; GR^e\big(I(\tilde{L})\big) + p_s \;<\; GR^e\big(I(\tilde{L})\big) + \frac{1}{\ell+2}\,z(I),$$

and therefore, by (2.12),

$$z(I) \;<\; \sum_{i\in\tilde{L}} p_i + GR^e\big(I(\tilde{L})\big) + \frac{1}{\ell+2}\,z(I) \;\leq\; H^\ell(I) + \frac{1}{\ell+2}\,z(I),$$

that is, $H^\ell(I) > \frac{\ell+1}{\ell+2}\, z(I)$.

In both cases $H^\ell(I) \geq \frac{\ell+1}{\ell+2}\, z(I)$, and dividing by $H^\ell(I) > 0$ gives the statement. □

Proposition 2.11

There are KP instances I_M such that:

$$\frac{z(I_M)}{H^\ell(I_M)} \to \frac{\ell+2}{\ell+1} \qquad \text{as} \qquad M \to +\infty. \tag{2.13}$$

Proof. Fix an integer $M > 2$ and consider the KP instance I_M with capacity $c = (\ell+2)\, M$ and $n = \ell + 3$ items, where

$$p_1 = 2, \quad w_1 = 1, \quad \text{and} \quad p_j = w_j = M \quad \text{for} \quad j \in \{2, 3, \ldots, n\}.$$

An optimal solution packs the $\ell + 2$ items of weight M, which fill the knapsack, hence $z(I_M) = (\ell + 2)\, M$. Item 1 comes first in the efficiency order, since $\frac{p_1}{w_1} = 2 > 1$. The sets L examined by H^ℓ are of two kinds:

1. L made of ℓ items of profit M: the residual capacity is $2M$, the residual instance keeps item 1 and the two remaining items of profit M, and GR^e packs item 1 and one of them, for a total value of $\ell M + M + 2 = (\ell + 1)\, M + 2$;
2. L containing item 1 and $\ell - 1$ items of profit M: the residual instance is empty, since no item has profit at most $\min\{2, M\} = 2$, for a total value of $(\ell - 1)\, M + 2$.

The sets of at most $\ell - 1$ items give at most $(\ell - 1)\, M + 2$ as well. Hence $H^\ell(I_M) = (\ell + 1)\, M + 2$ and

$$\lim_{M\to\infty} \frac{z(I_M)}{H^\ell(I_M)} = \lim_{M\to\infty} \frac{(\ell+2)\, M}{(\ell+1)\, M + 2} = \lim_{M\to\infty} \frac{\ell+2}{\ell+1+\frac{2}{M}} = \frac{\ell+2}{\ell+1}.$$

For $\ell = 2$ this is the instance used for H^2. □

From Proposition 2.10 and Proposition 2.11 it follows that the worst case performance ratio of H^ℓ is

$$r(H^\ell) = \frac{\ell+2}{\ell+1}$$

and H^ℓ is a $\frac{\ell+2}{\ell+1}$-approximation algorithm. For $\ell = 2$ we find again $r(H^2) = \frac{4}{3}$, and the ratio tends to 1 as ℓ grows.

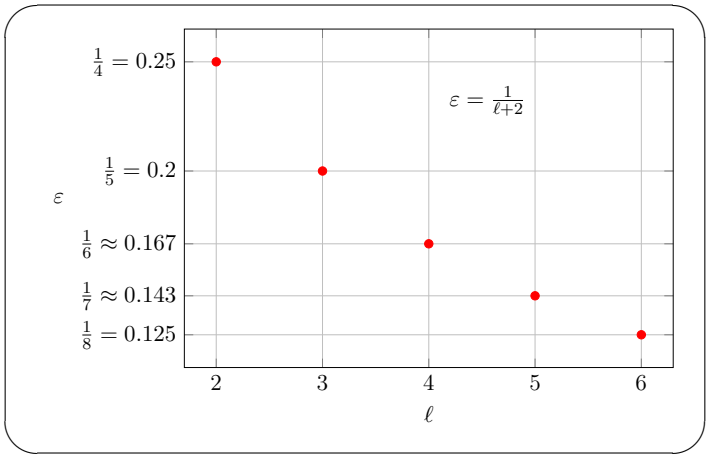
2.8.5 Approximation schemes

- We recall the definitions of the chapter «Principles of approximation algorithms». The **relative error** of an algorithm $\mathscr{A}$ on a KP instance I is $\frac{z(I)-\mathscr{A}(I)}{z(I)}$. An **approximation scheme** is a family of algorithms $\{\mathscr{A}_\varepsilon\}_{\varepsilon\in(0,1)}$ such that, for every ε and every instance I, the relative error of $\mathscr{A}_\varepsilon$ is at most ε; for a maximization problem this means that $\mathscr{A}_\varepsilon$ is a $\frac{1}{1-\varepsilon}$-approximation algorithm. The scheme is a **PTAS** if, for every fixed ε, the running time of $\mathscr{A}_\varepsilon$ is polynomial in the size of the instance, and an **FPTAS** if it is polynomial both in the size of the instance and in the accuracy $\frac{1}{\varepsilon}$.

- By Proposition 2.10, $H^\ell(I) \geq \frac{\ell+1}{\ell+2}\, z(I)$, hence the relative error of H^ℓ is at most

$$1 - \frac{\ell+1}{\ell+2} = \frac{1}{\ell+2},$$

  which tends to 0 as ℓ grows:



- The relation can be read the other way round: given the relative error ε to be attained, it is enough to choose ℓ with $\frac{1}{\ell+2} \leq \varepsilon$, i.e., $\ell \geq \frac{1}{\varepsilon} - 2$.

By choosing

$$\ell(\varepsilon) = \min\left\{\max\left\{\left\lceil\frac{1}{\varepsilon}\right\rceil - 2,\ 2\right\},\ n\right\}$$

we obtain the H^ε algorithm, i.e., H^ℓ with $\ell = \ell(\varepsilon)$, whose relative error is at most ε for every $\varepsilon \in (0,1)$.

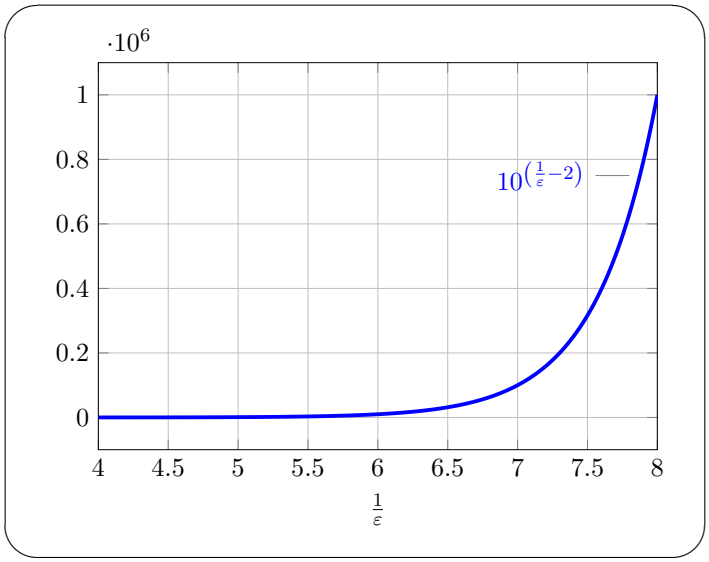
- For $\varepsilon \geq \frac{1}{4}$ we have $\ell(\varepsilon) = 2$, and H^ε is the H^2 algorithm. The size ℓ of the guessed sets cannot exceed n, and with $\ell = n$ the algorithm enumerates all the feasible solutions and is exact.

Observation 2.12

The family $\{H^\varepsilon\}_{\varepsilon\in(0,1)}$ is a PTAS for the 0/1 knapsack problem.

- For every fixed ε, $\ell(\varepsilon)$ is a constant, and by Observation 2.4 the running time $O\big(n^{\ell(\varepsilon)+1}\big)$ is polynomial in the size n of the instance.

- However, H^ε is **not** an FPTAS: the exponent $\ell(\varepsilon) = \left\lceil \frac{1}{\varepsilon} \right\rceil - 2$ grows with the accuracy $\frac{1}{\varepsilon}$, so the running time is not polynomial in $\frac{1}{\varepsilon}$. The plot shows, for $n = 10$ items and $4 \le \frac{1}{\varepsilon} \le 8$, the number $n^{\frac{1}{\varepsilon}-2}$, a lower bound on $n^{\ell(\varepsilon)}$ (the cap $\ell(\varepsilon) \le n$ is not active, since $\frac{1}{\varepsilon} \le 12$):



- The H^ε algorithm is a straightforward method and quite easy to implement, but its running time explodes as the required accuracy grows. Even for a moderate relative error $\varepsilon = \frac{1}{10}$, i.e., a deviation of 10 percent from the optimal value, we have $\ell(\varepsilon) = 8$ and a running time of $O(n^9)$, or still $O(n^8)$ with the improved implementation.

2.8.6 The worst case performance ratio of the Basic-FPTAS

Proposition 2.13

For every KP instance I and every $\varepsilon \in (0,1)$, we have:

$$\frac{z(I)}{FP^{\varepsilon}(I)} \leq \frac{1}{1-\varepsilon}. \tag{2.14}$$

Proof. Let $\tilde{S}$ be an optimal solution of I and let S be the solution returned by FP^{ε}, i.e., an optimal solution of the scaled instance I'. Since I and I' have the same feasible solutions, $\tilde{S}$ is feasible for I' as well, and by optimality of S for I',

$$\sum_{j \in S} p'_j \;\geq\; \sum_{j \in \tilde{S}} p'_j.$$

By definition of the scaled profits, $K\,p'_j \leq p_j$ and $K\,p'_j > p_j - K$ for every item j. Therefore

$$FP^{\varepsilon}(I) = \sum_{j \in S} p_j \;\geq\; K \sum_{j \in S} p'_j \;\geq\; K \sum_{j \in \tilde{S}} p'_j \;>\; \sum_{j \in \tilde{S}} \left(p_j - K\right) \;=\; z(I) - |\tilde{S}|\,K.$$

Since an optimal solution contains at most n items and $K = \varepsilon\, GR^{e}(I)/n$,

$$|\tilde{S}|\,K \;\leq\; n\,K \;=\; \varepsilon\, GR^{e}(I) \;\leq\; \varepsilon\, z(I),$$

where the last inequality holds because $GR^{e}(I)$ is the value of a feasible solution. Combining the two displays,

$$FP^{\varepsilon}(I) \;\geq\; z(I) - \varepsilon\, z(I) \;=\; (1-\varepsilon)\, z(I),$$

and dividing by $FP^{\varepsilon}(I) > 0$ gives the statement. On the instance of Example 2.10, with $\varepsilon = \frac{1}{2}$, the ratio is $\frac{47}{38} \approx 1.24$, well below the bound $\frac{1}{1-\varepsilon} = 2$. □

Observation 2.14

The FP^{ε} algorithm can be computed in $O\left(n^2/\varepsilon\right)$ time.

Proof. Let $\tilde{q}$ be the optimal value of the scaled instance I'. Every feasible solution of I' is feasible for I, hence

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

where $z(I) \leq 2\,GR^{e}(I)$ is exactly the relative performance guarantee of GR^{e}. Since $\tilde{q}$ is

an integer, $\tilde{q} \leq Q = \lfloor 2n/\varepsilon \rfloor$, and truncating the table of the dynamic program at Q does not change the result.
The table has $n+1$ rows and $Q+1$ columns, so the dynamic program runs in $O(n\,Q) = O(n^2/\varepsilon)$ time, and the scaling and the call to GR^e take $O(n)$ and $O(n \log n)$ time.
Without the truncation, the table would reach the total scaled profit $P' = \sum_j p'_j$. Since $p_j \leq GR^e(I)$ gives $p'_j \leq \frac{n}{\varepsilon}$, we only have $P' \leq \frac{n^2}{\varepsilon}$, which can be reached, and the running time would be $O(n\,P') = O(n^3/\varepsilon)$. □

From Proposition 2.13 and Observation 2.14 it follows that the family $\{FP^\varepsilon\}_{\varepsilon \in (0,1)}$ is an **approximation scheme** whose running time is polynomial both in the size n of the instance and in the accuracy $\frac{1}{\varepsilon}$, i.e., an **FPTAS** for the 0/1 knapsack problem.

- The guarantee of GR^e is used twice, and in two different roles: $GR^e(I) \leq z(I)$ fixes the scaling factor, and $z(I) \leq 2\,GR^e(I)$ bounds the size of the dynamic programming table. A better heuristic in place of GR^e would shorten the table, not improve the guarantee.

2.9 Absolute performance guarantee

- Unfortunately, there is only a negative result concerning absolute performance guarantees for the 0/1 knapsack problem.
- The following theorem says that if an approximation algorithm would have an absolute performance guarantee k for some fixed k, then this algorithm could be used to solve any instance of a 0/1 knapsack problem to optimality.

  Let $f(n, c)$ be a function describing the running time of an algorithm for the KP with n items and capacity c.

Theorem 2.15

Let $\mathscr{A}$ be an approximation algorithm for the KP with absolute performance guarantee k and running time in $O\big(f(n, c)\big)$ for any instance with n items and capacity c. Then $\mathscr{A}$ also computes the optimal solution of any such instance in $O(f(n, c))$ time.

Proof. Consider an approximation algorithm $\mathscr{A}$ for the KP with absolute performance guarantee k and let I be given by

$$p_1, \ldots, p_n, \quad w_1, \ldots, w_n, \quad c.$$

We construct the new instance I' with

$$p'_1, \ldots, p'_n, \quad w'_1, \ldots, w'_n, \quad c'$$

such that

$$p'_j = (k+1)\,p_j, \quad w'_j = w_j, \quad \forall j \in \mathscr{I}, \quad \text{and} \quad c' = c.$$

- Both instances I and I' have exactly the same set of feasible solutions.
- Since the objective function value of a solution of I' is exactly $k+1$ times as large as the objective function value of the same solution for I, the two instances also have the same optimal solutions.

Applying algorithm $\mathscr{A}$ to the instance I' yields a solution of value $\mathscr{A}(I')$ with

$$z(I') - \mathscr{A}(I') \leq k$$

using the absolute performance guarantee of $\mathscr{A}$.

Interpreting the output of $\mathscr{A}$ as a solution for I yields a feasible solution of value $\mathscr{A}(I)$, and

$$\mathscr{A}(I') = (k+1)\,\mathscr{A}(I) \qquad \text{and} \qquad z(I') = (k+1)\,z(I)$$

Inserting in the above inequality yields

$$(k+1)\,z(I) - (k+1)\,\mathscr{A}(I) \leq k \iff z(I) - \mathscr{A}(I) \leq \frac{k}{k+1}$$

But since all profit values are integer, also $z(I) - \mathscr{A}(I)$ must be integer and hence this inequality implies

$$z(I) - \mathscr{A}(I) \leq \left\lfloor \frac{k}{k+1} \right\rfloor \Longrightarrow z(I) - \mathscr{A}(I) \leq 0 \Longrightarrow z(I) = \mathscr{A}(I)$$

Since the approximate solution of I' was computed in $O(f(n,c))$ running time an optimal solution of I could be constructed within the same time. □

- This theorem indicates that an algorithm with fixed absolute performance guarantee for the KP would have the same unpleasant running time as an optimal algorithm.

  It is very unlikely that an exact algorithm for KP with a running time polynomial in n exists.

- Hence, also approximation algorithms with fixed absolute performance guarantee are bound to have a pseudo-polynomial running time which is unpleasantly dependent on the magnitude of the input values.

2.10 Typical mistakes

Warning

- **Stopping the greedy algorithm at the split item.** GR skips the split item s but keeps scanning, and it may select items that follow s (Example 2.2).
- **Confusing i_m and s.** In GR^e the item i_m is the item of maximum profit $p_{\max}$, not the split item s: the two coincide only by chance.
- **Forgetting to sort by efficiency.** All the algorithms of this chapter assume the items sorted by non-increasing $\frac{p_j}{w_j}$; the split item and the greedy solution depend on the order.
- **Calling H^ℓ an FPTAS.** Its running time $O(n^{\ell+1})$ has the exponent $\ell(\varepsilon) = \lceil \frac{1}{\varepsilon} \rceil - 2$ growing with $\frac{1}{\varepsilon}$: H^ε is a PTAS, not an FPTAS.
- **Scaling the weights.** The Basic-FPTAS algorithm rounds down the profits only; the weights and the capacity are untouched, so that the feasible solutions are the same.
- **Counting the whole table of the dynamic program.** The bound $O(n^2/\varepsilon)$ holds because the table is truncated at $\lfloor 2n/\varepsilon \rfloor$; the total scaled profit can be as large as $\frac{n^2}{\varepsilon}$.

2.11 In summary

Summary

- GR has no constant guarantee, GR^e (the better of GR and the best single item) has $r(GR^e) = 2$, H^2 has $r(H^2) = \frac{4}{3}$ and H^ℓ has $r(H^\ell) = \frac{\ell+2}{\ell+1}$.
- $\{H^\varepsilon\}$ is a PTAS with running time $O(n^{\ell+1})$; the Basic-FPTAS algorithm rounds down the profits, solves the scaled instance exactly by dynamic programming, and is an FPTAS: it guarantees $\frac{1}{1-\varepsilon}$ in time $O(n^2/\varepsilon)$.
- No algorithm with a fixed absolute performance guarantee exists, unless it solves the KP exactly.

Dashboard of the KP instance of Example 2.1 ($c = 9$, $p = (6, 5, 8, 9, 6, 7, 3)$, $w = (2, 3, 6, 7, 5, 9, 4)$). The bounds on $z(I)$ found so far:

Source	Lower bound	Upper bound	From
Dantzig bound	–	16	$\lfloor \frac{49}{3} \rfloor$, with $s = 3$ and $rc = 4$
This chapter, GR and GR^e	14	–	$S = \{1, 2, 7\}$
This chapter, H^2 and H^3	15	–	$S = \{1, 4\}$
This chapter, FP^ε with $\varepsilon = \frac{1}{2}$	15	–	$K = 1$, the scaled instance is the original one
Best bounds	15	16	UB$-$LB $= 1$; the optimal value is 15

2.12 C++ implementation

- In this section we use 0-based indexing for the items. The algorithmic logic is unchanged.
- The following is a C++ implementation of the **greedy algorithm**.

```cpp
// items are assumed to be sorted by non-increasing efficiency
// x[i] stores the selection decision for item i
// the function returns the profit of the solution x

int GreedyKP(int n, int c, int* w, int* p, int* x)
{
    for (int i = 0; i < n; ++i)
    {
        x[i] = 0;                       // item i is initially not selected
    }
    int rc = c;                         // residual capacity

    for (int i = 0; i < n; ++i)
    {
        if (w[i] <= rc)
        {
            x[i] = 1;                   // select item i
            rc = rc - w[i];             // update residual capacity
        }
    }

    // profit of the solution x, returned for convenience
    int z = 0;
    for (int i = 0; i < n; ++i)
    {
        if (x[i] == 1) z = z + p[i];
    }
    return z;
}
```

This is the step-by-step execution of the C++ implementation of GR on the KP instance of Example 2.1.

Initialization

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	-	-	-	-	-	-	-

$rc = 9$

1. Iteration 1 – Item 0 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	-	-	-	-	-	-

$rc = 7$

2. Iteration 2 – Item 1 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	-	-	-	-	-

$rc = 4$

3. Iteration 3 – Item 2 is not selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	-	-	-	-

$rc = 4$

4. Iteration 4 – Item 3 is not selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	0	-	-	-

$rc = 4$

5. Iteration 5 – Item 4 is not selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	0	0	-	-

$rc = 4$

6. Iteration 6 – Item 5 is not selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	0	0	0	-

$rc = 4$

7. Iteration 7 – Item 6 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	0	0	0	1

$rc = 0$

The value computed is

$$z = 14$$

This is the step-by-step execution of the C++ implementation of GR on the KP instance of Example 2.3.

Initialization

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	-	-	-	-

$rc = 10$

1. Iteration 1 – Item 0 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	1	-	-	-

$rc = 8$

2. Iteration 2 – Item 1 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	1	1	-	-

$rc = 5$

3. Iteration 3 – Item 2 is not selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	1	1	0	-

$rc = 5$

4. Iteration 4 – Item 3 is selected

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	1	1	0	1

$rc = 0$

The value computed is

$$z = 18$$

- The following is a `C++` implementation of the **extended greedy algorithm**.

```cpp
// items are assumed to be sorted by non-increasing efficiency
// x[i] stores the selection decision for item i
// the function returns the profit of the solution x

int ExtendedGreedyKP(int n, int c, int* w, int* p, int* x)
{
    GreedyKP(n, c, w, p, x);              // greedy solution

    int z = 0;                            // value of the greedy solution
    for (int i = 0; i < n; ++i)
    {
        if (x[i] == 1)
        {
            z = z + p[i];                 // add the profit of a selected item
        }
    }

    int im = 0;                           // best single item
    int pm = p[0];
    for (int i = 1; i < n; ++i)
    {
        if (p[i] > pm)
        {
            im = i;                       // update best single item
            pm = p[i];                    // update its profit
        }
    }

    int* xhat = new int[n];               // best single-item solution
    for (int i = 0; i < n; ++i)
    {
        xhat[i] = 0;
    }
    xhat[im] = 1;                         // select only item im

    if (z < pm)
    {
        for (int i = 0; i < n; ++i)
        {
            x[i] = xhat[i];               // copy component by component
        }
        z = pm;
    }

    delete[] xhat;
    return z;
}
```

- This is the step-by-step execution of the C++ implementation of GR^e on the KP instance of Example 2.1.

Initialization

The solution computed by GR is

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$	$i=5$	$i=6$
	1	1	0	0	0	0	1

The value computed is

$$z = 14$$

The maximum profit is:

$$p_{\max} = 9,$$

and the item of maximum profit is (0-based indexing)

$$i_m = 3.$$

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

- This is the step-by-step execution of the C++ implementation of GR^e on the KP instance of Example 2.3.

Initialization

The solution computed by GR is

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	1	1	0	1

The value computed is

$$z = 18$$

The maximum profit is:

$$p_{\max} = 19,$$

and the item of maximum profit is (0-based indexing)

$$i_m = 2.$$

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

$x[i]$	$i=0$	$i=1$	$i=2$	$i=3$
	0	0	1	0

The updated value is

$$z = 19.$$

- The following is a `C++` implementation of the H^2 **algorithm**. It calls `ExtendedGreedyKP` on the residual instance of every feasible pair, exactly as the pseudo-code does, and it keeps the best solution found.

```cpp
// items are assumed to be sorted by non-increasing efficiency
// the function returns H2(I), the value of the best solution found;
// in addition, x stores that solution, which the pseudo-code does not record

int H2KP(int n, int c, int* w, int* p, int* x)
{
    int im = 0;
    for (int j = 1; j < n; ++j)
    {
        if (p[j] > p[im]) im = j;
    }
    int pmax = p[im];
    int lb = pmax;                                  // best value found
    for (int j = 0; j < n; ++j) x[j] = 0;           // record the solution
    x[im] = 1;

    // residual instance I_ik: items, profits, weights and its solution
    int* idx = new int[n];
    int* pr  = new int[n];
    int* wr  = new int[n];
    int* xr  = new int[n];

    for (int i = 0; i < n; ++i)
    {
        for (int k = i + 1; k < n; ++k)
        {
            if (w[i] + w[k] > c) continue;          // the pair (i,k) is not in
    P

            int cbar = c - w[i] - w[k];             // residual capacity
            int pmin = (p[i] < p[k] ? p[i] : p[k]);

            // residual set I(i,k)
            int m = 0;
            for (int j = 0; j < n; ++j)
            {
                if (j != i && j != k && p[j] <= pmin && w[j] <= cbar)
                {
                    idx[m] = j; pr[m] = p[j]; wr[m] = w[j]; m++;
                }
            }

            // value of GR^e on the residual instance
            int zres = (m > 0) ? ExtendedGreedyKP(m, cbar, wr, pr, xr) : 0;
            int lbik = p[i] + p[k] + zres;

            if (lbik > lb)
            {
                lb = lbik;
                for (int j = 0; j < n; ++j) x[j] = 0;   // record the solution
                x[i] = 1; x[k] = 1;
                for (int j = 0; j < m; ++j) if (xr[j] == 1) x[idx[j]] = 1;
            }
        }
    }

    delete[] idx; delete[] pr; delete[] wr; delete[] xr;
```

```
    return lb;
}
```

- On the KP instance of Example 2.8, with $n = 7$ items and capacity $c = 12$, the function returns $lb = 20$ with $x = (1, 1, 0, 1, 0, 0, 0)$, i.e., the solution $S = \{1, 2, 4\}$ already found in that example, which is optimal for that instance.
- The two nested loops over i and k perform $O(n^2)$ iterations, and each iteration builds the residual instance and runs `ExtendedGreedyKP` in $O(n)$ time: this is the $O(n^3)$ running time of Proposition 2.3.
- The following is a `C++` implementation of the H^ℓ **algorithm**. The function `NextSubset` enumerates the subsets of exactly r items in lexicographic order; the preprocessing examines the subsets of at most $\ell - 1$ items, and each subset of exactly ℓ items is completed by `ExtendedGreedyKP` on its residual instance, as in the pseudo-code.

```
// items are assumed to be sorted by non-increasing efficiency
// l is the number of guessed items, with 2 <= l <= n
// the function returns H^l(I), the value of the best solution found;
// in addition, x stores that solution, which the pseudo-code does not record

// next subset of r indices of {0,...,n-1} in lexicographic order (false at the end)
static bool NextSubset(int* L, int r, int n)
{
    int t = r - 1;
    while (t >= 0 && L[t] == n - r + t) t--;
    if (t < 0) return false;
    L[t]++;
    for (int u = t + 1; u < r; ++u) L[u] = L[u - 1] + 1;
    return true;
}

int HlKP(int n, int c, int* w, int* p, int l, int* x)
{
    int* L    = new int[l];
    char* inL = new char[n];
    int* idx  = new int[n];
    int* pr   = new int[n];
    int* wr   = new int[n];
    int* xr   = new int[n];

    // lb := best value of the feasible sets of at most l-1 items
    int lb = 0;
    for (int j = 0; j < n; ++j) x[j] = 0;
    for (int r = 1; r <= l - 1; ++r)
    {
        for (int t = 0; t < r; ++t) L[t] = t;
        do
        {
            int wS = 0, pS = 0;
            for (int t = 0; t < r; ++t) { wS += w[L[t]]; pS += p[L[t]]; }
```

```
            if (wS <= c && pS > lb)
            {
                lb = pS;
                for (int j = 0; j < n; ++j) x[j] = 0;   // record the solution
                for (int t = 0; t < r; ++t) x[L[t]] = 1;
            }
        } while (NextSubset(L, r, n));
    }

    // foreach L in T(l): the feasible sets of exactly l items
    for (int t = 0; t < l; ++t) L[t] = t;
    do
    {
        int wL = 0, pL = 0, pmin = p[L[0]];
        for (int j = 0; j < n; ++j) inL[j] = 0;
        for (int t = 0; t < l; ++t)
        {
            wL += w[L[t]]; pL += p[L[t]]; inL[L[t]] = 1;
            if (p[L[t]] < pmin) pmin = p[L[t]];
        }
        if (wL > c) continue;                          // L is not in T(l)

        int cbar = c - wL;                             // residual capacity

        // residual set I(L)
        int m = 0;
        for (int j = 0; j < n; ++j)
        {
            if (!inL[j] && p[j] <= pmin && w[j] <= cbar)
            {
                idx[m] = j; pr[m] = p[j]; wr[m] = w[j]; m++;
            }
        }

        // complete the guess
        int lbL = pL + ((m > 0) ? ExtendedGreedyKP(m, cbar, wr, pr, xr) : 0);

        if (lbL > lb)
        {
            lb = lbL;
            for (int j = 0; j < n; ++j) x[j] = inL[j];  // record the solution
            for (int j = 0; j < m; ++j) if (xr[j] == 1) x[idx[j]] = 1;
        }
    } while (NextSubset(L, l, n));

    delete[] L; delete[] inL; delete[] idx; delete[] pr; delete[] wr; delete[]
    xr;
    return lb;
}
```

- The following is a `C++` implementation of **dynamic programming by profits**, the second DP algorithm of the chapter «DP for the 0/1 knapsack problem» of the module *Dynamic Programming*, used by the Basic-FPTAS algorithm. With the 0-based indexing of the code, the cell `u[i][q]` stores the minimum total weight of a subset of the first i items of total profit exactly q, and an optimal solution is reconstructed backwards. The parameter `qmax` truncates the table: only the profits $q \leq \min\{P, \texttt{qmax}\}$ are computed,

where P is the total profit.

```
// dynamic programming by profits (second DP algorithm of the chapter "DP for the 0/1 knapsack problem")
// u(i,q) = minimum total weight of a subset of {0,...,i-1} with total profit exactly q
// the table is truncated at the profit qmax: only the values q = 0,...,min{P, qmax} are computed,
// where P is the total profit; qmax must be at least the optimal value of the instance
// the function returns the optimal value, i.e., the largest q with u(n,q) <= c,
// and stores an optimal solution in x

int DPProfits(int n, int c, int* w, int* p, int qmax, int* x)
{
    const int INF = 2147483647;
    int P = 0;                                  // total profit
    for (int i = 0; i < n; ++i) P += p[i];
    if (P > qmax) P = qmax;                     // truncation of the table

    // table u[i][q], i = 0,...,n and q = 0,...,P (P = min{total profit, qmax})
    int** u = new int*[n + 1];
    for (int i = 0; i <= n; ++i) u[i] = new int[P + 1];

    u[0][0] = 0;
    for (int q = 1; q <= P; ++q) u[0][q] = INF; // no item: only q = 0 is attained

    for (int i = 1; i <= n; ++i)
    {
        for (int q = 0; q <= P; ++q)
        {
            u[i][q] = u[i - 1][q];              // item i-1 is not selected
            if (p[i - 1] <= q && u[i - 1][q - p[i - 1]] != INF
                && w[i - 1] + u[i - 1][q - p[i - 1]] < u[i][q])
            {
                u[i][q] = w[i - 1] + u[i - 1][q - p[i - 1]];   // item i-1 is selected
            }
        }
    }

    // optimal value: the largest profit attained within the capacity
    int qstar = 0;
    for (int q = 0; q <= P; ++q) if (u[n][q] <= c) qstar = q;

    // reconstruct an optimal solution backwards
    int q = qstar;
    for (int i = n; i >= 1; --i)
    {
        if (u[i][q] == u[i - 1][q]) x[i - 1] = 0;
        else { x[i - 1] = 1; q -= p[i - 1]; }
    }

    for (int i = 0; i <= n; ++i) delete[] u[i];
```

```
48      delete[] u;
49      return qstar;
50  }
```

- The following is a C++ implementation of the **Basic-FPTAS algorithm**. The accuracy ε is passed as a fraction $\frac{\texttt{eps_num}}{\texttt{eps_den}}$, so that the scaled profits $\left\lfloor \frac{p_j}{K} \right\rfloor = \left\lfloor \frac{p_j \, n \, \texttt{eps_den}}{\texttt{eps_num} \, GR^e(I)} \right\rfloor$ are computed in exact integer arithmetic, without rounding errors, and the table of `DPProfits` is truncated at $\lfloor 2n/\varepsilon \rfloor$.

```
1   // items are assumed to be sorted by non-increasing efficiency
2   // the accuracy is eps = eps_num / eps_den, with 0 < eps < 1
3   // x[i] stores the selection decision for item i
4   // the function returns the profit, in the original instance, of the solution x
5
6   int BasicFPTAS(int n, int c, int* w, int* p, int eps_num, int eps_den, int* x)
7   {
8       int lb = ExtendedGreedyKP(n, c, w, p, x);   // value of the Ext-Greedy
        solution
9
10      // scaling factor K = eps * lb / n and scaled profits p'[j] = floor(p[j] /
        K);
11      // with eps = eps_num / eps_den, p[j] / K = p[j] * n * eps_den / (eps_num *
         lb),
12      // which is computed in exact integer arithmetic
13      int* ps = new int[n];                       // scaled profits p'
14      for (int j = 0; j < n; ++j)
15      {
16          ps[j] = (int)(((long long)p[j] * n * eps_den) / ((long long)eps_num *
        lb));
17      }
18
19      // the optimal value of the scaled instance is at most floor(2n / eps): the
         table is truncated there
20      int qmax = (int)((2LL * n * eps_den) / eps_num);
21
22      DPProfits(n, c, w, ps, qmax, x);            // scaled instance solved to
        optimality
23
24      // profit of the solution x in the original instance
25      int z = 0;
26      for (int j = 0; j < n; ++j)
27      {
28          if (x[j] == 1) z = z + p[j];
29      }
30
31      delete[] ps;
32      return z;
33  }
```

- All the functions of this section are in the folder `KP_HEUR_CODE/src`. The command `make test` compiles them and runs every example of this chapter and every instance of the exercises, checking the value found against the expected one.

Packing problems

Chapter 3

Approximation algorithms for bin packing

Prerequisites

- From this module, the chapter «Principles of approximation algorithms»: performance ratio, relative performance guarantee, tight guarantee and worst-case performance ratio $r(\mathscr{A})$.
- From the module *Combinatorial Optimization Problems*, the chapter «Packing problems»: the BPP, its instances and the counting lower bound $\left\lceil \frac{\sum_{i \in \mathscr{I}} w_i}{c} \right\rceil$ on the optimal value.
- From the module *Heuristic Algorithms*, the chapter «Constructive heuristics»: heuristic algorithms that build a solution one item at a time.

Objectives

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

- execute the Next Fit, First Fit, Best Fit and Worst Fit algorithms by hand, and write the item-bin assignment and the number of bins;
- decide whether an algorithm belongs to the Any Fit class, and whether a solution satisfies the half-full property;
- prove that Next Fit and every Any Fit algorithm have a relative performance guarantee of 2;
- describe the families of instances that make the guarantee 2 tight for Next Fit and Worst Fit, and state the worst-case ratio $\frac{17}{10}$ of First Fit and Best Fit;
- read and run the `C++` implementations of the four algorithms.

The problem of this chapter is the bin packing problem (BPP): its definition and its assumptions are on its page.

3.1 Notation

- As on the problem page, c is the bin capacity and $\mathscr{I} = \{1, 2, \ldots, n\}$ the set of items; in this chapter item $i \in \mathscr{I}$ has weight w_i.

- A **BPP feasible solution** is a partition of $\mathscr{I}$ into nonempty **packing patterns**, i.e., subsets of items each assigned to one bin, such that the total weight of every packing pattern does not exceed the bin capacity c.

- We denote by $z(I)$ the **optimal value** of the BPP on the instance I, i.e., the minimum number of bins, and by $\mathscr{A}(I)$ the number of bins used by the feasible solution returned by a heuristic algorithm $\mathscr{A}$. Since every heuristic algorithm of this chapter returns a feasible solution,

$$\mathscr{A}(I) \geq z(I) \qquad \text{and} \qquad \frac{\mathscr{A}(I)}{z(I)} \geq 1$$

  for every BPP instance I.

3.2 The Next Fit

The *Next Fit Algorithm* (denoted by NF) is a constructive heuristic for the BPP. It processes the items in the order in which they are given in the input instance, maintaining exactly one open bin at any time. For each item, NF packs it into the currently open bin if it fits; otherwise, it closes the bin, opens a new one, and places the item there.

- **Data structure for the solution.** We use a vector $bin = (bin_1, bin_2, \ldots, bin_n)$ of length n to store the solution. At position i, the vector contains the number of the bin to which item i is assigned. More formally:

$$bin_i = b \quad \Longleftrightarrow \quad \text{item } i \text{ is packed into bin } b.$$

  For example, if $bin_3 = 2$, then item 3 is assigned to bin 2. In addition, we maintain a variable nb that stores the total number of bins used, i.e., at the end of the algorithm $NF(I) = nb$.

- The pseudo-code of NF reads as follows:

Algorithm

Algorithm: Next Fit

Input: BPP instance I with n items, weights $w[1..n]$, capacity c
Output: Item-bin assignment $bin[1..n]$ and number of used bins nb

```
1  Function NextFit(n, w, c)
2      bin := new array[1..n]                    // item-bin assignment
3      nb := 1                                   // current bin
4      rc := c                                   // residual capacity
5      for i := 1 to n do
6          if w[i] > rc then
7              nb ← nb + 1                       // open new bin
8              rc ← c
9          end
10         bin[i] ← nb                           // assign item to bin
11         rc ← rc − w[i]
12     end
13     return bin, nb
```

Example 3.1: Execution of NF

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
w_i	6	5	4	7	3

Initially $nb := 1$, $rc := c$. The NF algorithm produces the following feasible solution:

1. Item 1 (weight $w[1] = 6$): $w[1] = 6 \leq rc = 10$, so no new bin is opened. Item 1 is packed into bin $nb = 1$, and $rc \leftarrow 10 - 6 = 4$.
2. Item 2 (weight $w[2] = 5$): $w[2] = 5 > rc = 4$, so $nb \leftarrow 2$ and $rc \leftarrow c = 10$. Item 2 is packed into bin $nb = 2$, and $rc \leftarrow 10 - 5 = 5$.
3. Item 3 (weight $w[3] = 4$): $w[3] = 4 \leq rc = 5$, so no new bin is opened. Item 3 is packed into bin $nb = 2$, and $rc \leftarrow 5 - 4 = 1$.
4. Item 4 (weight $w[4] = 7$): $w[4] = 7 > rc = 1$, so $nb \leftarrow 3$ and $rc \leftarrow c = 10$. Item 4 is packed into bin $nb = 3$, and $rc \leftarrow 10 - 7 = 3$.
5. Item 5 (weight $w[5] = 3$): $w[5] = 3 \leq rc = 3$, so no new bin is opened. Item 5 is packed into bin $nb = 3$, and $rc \leftarrow 3 - 3 = 0$.

Hence NF returns:

$$bin = \begin{pmatrix} 1 & 2 & 2 & 3 & 3 \end{pmatrix}, \qquad nb = 3.$$

and therefore

$$NF(I) = nb = 3.$$

- Given the output of an algorithm, namely the vector $bin = (bin_1, bin_2, \ldots, bin_n)$ and the number of bins nb, we can obtain the **packing pattern** P_b of each bin b, i.e., the set of items assigned to it, as follows:

$$P_b = \{\ i \in \{1, 2, \ldots, n\} : bin_i = b\ \}, \qquad \text{for } b \in \{1, 2, \ldots, nb\}.$$

  In other words, P_b is the set of all items i such that $bin_i = b$; P_b is the packing pattern of bin b, not the bin itself.

Example 3.2: Packing patterns from bin vector

Consider the instance of the previous example where NF returns:

$$bin = \begin{pmatrix} 1 & 2 & 2 & 3 & 3 \end{pmatrix}, \qquad nb = 3.$$

The packing patterns are:

$$P_1 = \{i \in \{1, 2, 3, 4, 5\} : bin_i = 1\} = \{1\},$$
$$P_2 = \{i \in \{1, 2, 3, 4, 5\} : bin_i = 2\} = \{2, 3\},$$
$$P_3 = \{i \in \{1, 2, 3, 4, 5\} : bin_i = 3\} = \{4, 5\}.$$

These are exactly the sets of items packed into the bins constructed by the algorithm:

- Bin 1 contains item 1 with weight $w_1 = 6$;
- Bin 2 contains items 2 and 3 with total weight $w_2 + w_3 = 5 + 4 = 9$;
- Bin 3 contains items 4 and 5 with total weight $w_4 + w_5 = 7 + 3 = 10$.

3.3 The Any Fit class

A constructive heuristic for the BPP that keeps all the bins opened so far, and opens a new bin *only when* the current item fits into none of them, is called an *Any Fit Algorithm* (denoted by AF). Any Fit is therefore not a single algorithm but a *family* of algorithms: what the condition leaves free is the rule used to select one bin among those in which the item fits, and each rule gives a different member of the family.

Definition 3.1: Any Fit algorithm

A constructive heuristic $\mathscr{A}$ for the BPP is an **Any Fit algorithm** if, denoting by $P_1, P_2, \ldots, P_{nb}$ the packing patterns of the bins opened so far, it opens a new bin for

item i only when

$$\sum_{h \in P_b} w_h + w_i > c \qquad \text{for every } b \in \{1, 2, \ldots, nb\},$$

whereas the choice of the bin, when more than one can accommodate the item, is left unspecified.

- **The items of two bins never fit together.** Let b and b' be two bins opened by an Any Fit algorithm, with $b < b'$, let f be the first item packed into bin b', and let W_b be the total weight of the items contained in bin b at the moment in which f was processed. Bin b' was opened because f did not fit into any of the bins opened so far, and in particular not into bin b, hence $W_b + w_f > c$. The total weight of a bin never decreases during the execution, so $\sum_{i \in P_b} w_i \geq W_b$, and $f \in P_{b'}$ gives $\sum_{i \in P_{b'}} w_i \geq w_f$. Therefore

$$\sum_{i \in P_b} w_i + \sum_{i \in P_{b'}} w_i \;\geq\; W_b + w_f \;>\; c$$

  for any two distinct bins b and b': accordingly, no two bins of an Any Fit solution can be merged. Only the Any Fit condition is used here, the selection rule plays no role.

- **NF is not an Any Fit algorithm.** NF keeps a single open bin and closes it as soon as an item does not fit, even when that item would fit into a bin closed earlier.

Example 3.3: NF is not an Any Fit algorithm

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$
w_i	3	8	3	8

Initially $nb := 1$, $rc := c$. The NF algorithm produces the following feasible solution:

1. Item 1 (weight $w[1] = 3$): $w[1] = 3 \leq rc = 10$, so no new bin is opened. Item 1 is packed into bin $nb = 1$, and $rc \leftarrow 10 - 3 = 7$.
2. Item 2 (weight $w[2] = 8$): $w[2] = 8 > rc = 7$, so $nb \leftarrow 2$ and $rc \leftarrow c = 10$. Item 2 is packed into bin $nb = 2$, and $rc \leftarrow 10 - 8 = 2$.
3. Item 3 (weight $w[3] = 3$): $w[3] = 3 > rc = 2$, so $nb \leftarrow 3$ and $rc \leftarrow c = 10$. Item 3 is packed into bin $nb = 3$, and $rc \leftarrow 10 - 3 = 7$.
4. Item 4 (weight $w[4] = 8$): $w[4] = 8 > rc = 7$, so $nb \leftarrow 4$ and $rc \leftarrow c = 10$. Item 4 is packed into bin $nb = 4$, and $rc \leftarrow 10 - 8 = 2$.

Hence NF returns:

$$bin = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}, \qquad nb = 4.$$

At the third iteration item 3 would fit into bin 1, whose residual capacity is 7, and NF opens a new bin all the same: NF violates the Any Fit condition.

An Any Fit algorithm, on the contrary, cannot open a third bin for item 3, because the item fits into bin 1: it packs item 3 there and then opens a new bin for item 4, which fits in neither bin, using 3 bins instead of 4. This holds for every algorithm of the class, whatever the selection rule.

3.3.1 The First Fit

The *First Fit Algorithm* (denoted by FF) is a constructive heuristic for the BPP. It processes the items in the order in which they are given in the input instance, maintaining all bins opened so far. For each item, FF scans the bins in the order they were opened and packs the item into the *first* bin in which it fits; if no current bin can accommodate the item, FF opens a new bin and places the item there.

- FF is an Any Fit algorithm: it opens a new bin only when the item fits into none of the bins opened so far, and the rule left unspecified by Definition 3.1 is here resolved by taking the first bin in which the item fits.

- The pseudo-code of FF reads as follows:

Algorithm

Algorithm: First Fit

Input: BPP instance I with n items, weights $w[1..n]$, capacity c
Output: Item-bin assignment $bin[1..n]$ and number of used bins nb

```
1  Function FirstFit(n, w, c)
2      bin := new array[1..n]                    // item-bin assignment
3      rc := new array[1..n]                     // residual capacities
4      nb := 1                                   // number of bins
5      rc[1] ← c                                 // initialize first bin
6      for i := 1 to n do
7          s := −1                               // selected bin
8          for b := 1 to nb do
9              if w[i] ≤ rc[b] then
10                 s ← b
11                 break
12             end
13         end
14         if s = −1 then
15             nb ← nb + 1                       // open new bin
16             rc[nb] ← c
17             s ← nb
18         end
19         bin[i] ← s                            // assign item
20         rc[s] ← rc[s] − w[i]
21     end
22     return bin, nb
```

Example 3.4: Execution of FF

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
w_i	6	5	4	7	3

The FF algorithm produces the following feasible solution:

1. Item 1 (weight 6): bin 1 is open at the start ($nb = 1$, $rc[1] = c = 10$) and $w[1] \leq rc[1]$, so $s = 1$ and FF packs item 1 into bin 1, and $rc[1] \leftarrow 10 - 6 = 4$.
2. Item 2 (weight 5): $b = 1$: $w[2] = 5 \leq rc[1] = 4$? No. No bin fits ($s = -1$), so FF opens bin 2 ($rc[2] \leftarrow 10$). Item 2 is packed into bin 2, and $rc[2] \leftarrow 10 - 5 = 5$.

3. Item 3 (weight 4): $b = 1$: $w[3] = 4 \leq rc[1] = 4$? Yes, so $s \leftarrow 1$ and the scan stops. Item 3 is packed into bin 1, and $rc[1] \leftarrow 4 - 4 = 0$.

4. Item 4 (weight 7): $b = 1$: $w[4] = 7 \leq rc[1] = 0$? No. $b = 2$: $w[4] = 7 \leq rc[2] = 5$? No. No bin fits ($s = -1$), so FF opens bin 3 ($rc[3] \leftarrow 10$). Item 4 is packed into bin 3, and $rc[3] \leftarrow 10 - 7 = 3$.

5. Item 5 (weight 3): $b = 1$: $w[5] = 3 \leq rc[1] = 0$? No. $b = 2$: $w[5] = 3 \leq rc[2] = 5$? Yes, so $s \leftarrow 2$ and the scan stops. Item 5 is packed into bin 2, and $rc[2] \leftarrow 5 - 3 = 2$.

Hence FF returns:

$$bin = \begin{pmatrix} 1 & 2 & 1 & 3 & 2 \end{pmatrix}, \qquad nb = 3.$$

and therefore

$$FF(I) = nb = 3.$$

Example 3.5: Packing patterns from bin vector

Consider the instance of the previous example where FF returns:

$$bin = \begin{pmatrix} 1 & 2 & 1 & 3 & 2 \end{pmatrix}, \qquad nb = 3.$$

The packing patterns are:

$$P_1 = \{i \in \{1,2,3,4,5\} : bin_i = 1\} = \{1,3\},$$
$$P_2 = \{i \in \{1,2,3,4,5\} : bin_i = 2\} = \{2,5\},$$
$$P_3 = \{i \in \{1,2,3,4,5\} : bin_i = 3\} = \{4\}.$$

These are exactly the sets of items packed into the bins constructed by the algorithm:

- Bin 1 contains items 1 and 3 with total weight $w_1 + w_3 = 6 + 4 = 10$;
- Bin 2 contains items 2 and 5 with total weight $w_2 + w_5 = 5 + 3 = 8$;
- Bin 3 contains item 4 with weight $w_4 = 7$.

3.3.2 The Best Fit

The *Best Fit Algorithm* (denoted by BF) is a constructive heuristic for the BPP. It processes the items in the order in which they are given in the input instance, maintaining all bins opened so far. For each item, BF considers all the bins in which the item fits and packs it into the one with the *smallest residual capacity* (in case of ties, the bin opened first); if no bin can accommodate the item, BF opens a new bin and places the item there.

- The rationale is to fill the bins as tightly as possible: by using the bin where the item leaves the least space unused, BF keeps the bins with a larger residual capacity available for the larger items that may still arrive.

- BF is an Any Fit algorithm: it opens a new bin only when the item fits into none of the bins opened so far, and the rule left unspecified by Definition 3.1 is here resolved by taking the bin with the smallest residual capacity.

- The pseudo-code of BF reads as follows:

Algorithm

Algorithm: Best Fit

Input: BPP instance I with n items, weights $w[1..n]$, capacity c
Output: Item-bin assignment $bin[1..n]$ and number of used bins nb

```
1  Function BestFit(n, w, c)
2      bin := new array[1..n]                          // item-bin assignment
3      rc := new array[1..n]                           // residual capacities
4      nb := 1                                         // number of bins
5      rc[1] ← c                                       // initialize first bin
6      for i := 1 to n do
7          minrc := c + 1                              // smallest residual capacity found so far
8          s := −1                                     // selected bin
9          for b := 1 to nb do
10             if w[i] ≤ rc[b] and rc[b] < minrc then
11                 s ← b
12                 minrc ← rc[b]
13             end
14         end
15         if s = −1 then
16             nb ← nb + 1                             // open new bin
17             rc[nb] ← c
18             s ← nb
19         end
20         bin[i] ← s                                  // assign item
21         rc[s] ← rc[s] − w[i]
22     end
23     return bin, nb
```

Example 3.6: Execution of BF

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
w_i	6	5	4	7	3

The BF algorithm produces the following feasible solution:

1. Item 1 (weight 6): bin 1 is open at the start ($nb = 1$, $rc[1] = c = 10$) and $w[1] \leq rc[1]$, so $s = 1$ and BF packs item 1 into bin 1, and $rc[1] \leftarrow 10 - 6 = 4$.
2. Item 2 (weight 5): $minrc := 11$, $s := -1$. $b = 1$: $w[2] = 5 \leq rc[1] = 4$? No. No bin fits ($s = -1$), so BF opens bin 2 ($rc[2] \leftarrow 10$). Item 2 is packed into bin 2, and $rc[2] \leftarrow 10 - 5 = 5$.

3. Item 3 (weight 4): $minrc := 11$, $s := -1$. $b = 1$: $w[3] = 4 \leq rc[1] = 4$ and $rc[1] = 4 < 11$, so $s \leftarrow 1$, $minrc \leftarrow 4$. $b = 2$: $w[3] = 4 \leq rc[2] = 5$ and $rc[2] = 5 < minrc = 4$? No. Hence $s = 1$: item 3 is packed into bin 1 (the one leaving the smallest residual capacity, 0, against 1 for bin 2), and $rc[1] \leftarrow 4 - 4 = 0$.

4. Item 4 (weight 7): $b = 1$: $w[4] = 7 \leq rc[1] = 0$? No. $b = 2$: $w[4] = 7 \leq rc[2] = 5$? No. No bin fits ($s = -1$), so BF opens bin 3 ($rc[3] \leftarrow 10$). Item 4 is packed into bin 3, and $rc[3] \leftarrow 10 - 7 = 3$.

5. Item 5 (weight 3): $minrc := 11$, $s := -1$. $b = 1$: $w[5] = 3 \leq rc[1] = 0$? No. $b = 2$: $w[5] = 3 \leq rc[2] = 5$ and $rc[2] = 5 < 11$, so $s \leftarrow 2$, $minrc \leftarrow 5$. $b = 3$: $w[5] = 3 \leq rc[3] = 3$ and $rc[3] = 3 < minrc = 5$, so $s \leftarrow 3$, $minrc \leftarrow 3$. Hence $s = 3$: item 5 is packed into bin 3 (the one leaving the smallest residual capacity, 0, against 2 for bin 2), and $rc[3] \leftarrow 3 - 3 = 0$.

Hence BF returns:

$$bin = \begin{pmatrix} 1 & 2 & 1 & 3 & 3 \end{pmatrix}, \qquad nb = 3.$$

and therefore

$$BF(I) = nb = 3.$$

Example 3.7: Packing patterns from bin vector

Consider the instance of the previous example where BF returns:

$$bin = \begin{pmatrix} 1 & 2 & 1 & 3 & 3 \end{pmatrix}, \qquad nb = 3.$$

The packing patterns are:

$$P_1 = \{i \in \{1,2,3,4,5\} : bin_i = 1\} = \{1,3\},$$
$$P_2 = \{i \in \{1,2,3,4,5\} : bin_i = 2\} = \{2\},$$
$$P_3 = \{i \in \{1,2,3,4,5\} : bin_i = 3\} = \{4,5\}.$$

These are exactly the sets of items packed into the bins constructed by the algorithm:

- Bin 1 contains items 1 and 3 with total weight $w_1 + w_3 = 6 + 4 = 10$;
- Bin 2 contains item 2 with weight $w_2 = 5$;
- Bin 3 contains items 4 and 5 with total weight $w_4 + w_5 = 7 + 3 = 10$.

3.3.3 The Worst Fit

The *Worst Fit Algorithm* (denoted by WF) is a constructive heuristic for the BPP. It processes the items in the order in which they are given in the input instance, maintaining all bins opened so far. For each item, WF considers all the bins in which the item fits and packs it into the one with the *largest residual capacity* (in case of ties, the bin opened first); if no bin can accommodate the item, WF opens a new bin and places the item there.

- The rationale is the opposite of the one of BF: by packing each item into the emptiest bin, WF avoids creating bins whose residual capacity is so small that no future item can use it, and it keeps the free space spread over the open bins.

- WF is an Any Fit algorithm: it opens a new bin only when the item fits into none of the bins opened so far, and the rule left unspecified by Definition 3.1 is here resolved by taking the bin with the largest residual capacity.

- The pseudo-code of WF reads as follows:

Algorithm

Algorithm: Worst Fit

Input: BPP instance I with n items, weights $w[1..n]$, capacity c
Output: Item-bin assignment $bin[1..n]$ and number of used bins nb

```
1  Function WorstFit(n, w, c)
2      bin := new array[1..n]                          // item-bin assignment
3      rc := new array[1..n]                           // residual capacities
4      nb := 1                                         // number of bins
5      rc[1] ← c                                       // initialize first bin
6      for i := 1 to n do
7          maxrc := −1                                 // largest residual capacity found so far
8          s := −1                                     // selected bin
9          for b := 1 to nb do
10             if w[i] ≤ rc[b] and rc[b] > maxrc then
11                 s ← b
12                 maxrc ← rc[b]
13             end
14         end
15         if s = −1 then
16             nb ← nb + 1                             // open new bin
17             rc[nb] ← c
18             s ← nb
19         end
20         bin[i] ← s                                  // assign item
21         rc[s] ← rc[s] − w[i]
22     end
23     return bin, nb
```

Example 3.8: Execution of WF

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

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
w_i	6	5	4	7	3

The WF algorithm produces the following feasible solution:

1. Item 1 (weight 6): bin 1 is open at the start ($nb = 1$, $rc[1] = c = 10$) and $w[1] \leq rc[1]$, so $s = 1$ and WF packs item 1 into bin 1, and $rc[1] \leftarrow 10 - 6 = 4$.
2. Item 2 (weight 5): $maxrc := -1$, $s := -1$. $b = 1$: $w[2] = 5 \leq rc[1] = 4$? No. No bin fits ($s = -1$), so WF opens bin 2 ($rc[2] \leftarrow 10$). Item 2 is packed into bin 2, and $rc[2] \leftarrow 10 - 5 = 5$.

3. Item 3 (weight 4): $maxrc := -1$, $s := -1$. $b = 1$: $w[3] = 4 \leq rc[1] = 4$ and $rc[1] = 4 > -1$, so $s \leftarrow 1$, $maxrc \leftarrow 4$. $b = 2$: $w[3] = 4 \leq rc[2] = 5$ and $rc[2] = 5 > maxrc = 4$, so $s \leftarrow 2$, $maxrc \leftarrow 5$. Hence $s = 2$: item 3 is packed into bin 2 (the one with the largest residual capacity), and $rc[2] \leftarrow 5 - 4 = 1$.

4. Item 4 (weight 7): $b = 1$: $w[4] = 7 \leq rc[1] = 4$? No. $b = 2$: $w[4] = 7 \leq rc[2] = 1$? No. No bin fits ($s = -1$), so WF opens bin 3 ($rc[3] \leftarrow 10$). Item 4 is packed into bin 3, and $rc[3] \leftarrow 10 - 7 = 3$.

5. Item 5 (weight 3): $maxrc := -1$, $s := -1$. $b = 1$: $w[5] = 3 \leq rc[1] = 4$ and $rc[1] = 4 > -1$, so $s \leftarrow 1$, $maxrc \leftarrow 4$. $b = 2$: $w[5] = 3 \leq rc[2] = 1$? No. $b = 3$: $w[5] = 3 \leq rc[3] = 3$ and $rc[3] = 3 > maxrc = 4$? No. Hence $s = 1$: item 5 is packed into bin 1 (the one with the largest residual capacity), and $rc[1] \leftarrow 4 - 3 = 1$.

Hence WF returns:

$$bin = \begin{pmatrix} 1 & 2 & 2 & 3 & 1 \end{pmatrix}, \qquad nb = 3.$$

and therefore

$$WF(I) = nb = 3.$$

Example 3.9: Packing patterns from bin vector

Consider the instance of the previous example where WF returns:

$$bin = \begin{pmatrix} 1 & 2 & 2 & 3 & 1 \end{pmatrix}, \qquad nb = 3.$$

The packing patterns are:

$$P_1 = \{i \in \{1,2,3,4,5\} : bin_i = 1\} = \{1,5\},$$
$$P_2 = \{i \in \{1,2,3,4,5\} : bin_i = 2\} = \{2,3\},$$
$$P_3 = \{i \in \{1,2,3,4,5\} : bin_i = 3\} = \{4\}.$$

These are exactly the sets of items packed into the bins constructed by the algorithm:

- Bin 1 contains items 1 and 5 with total weight $w_1 + w_5 = 6 + 3 = 9$;
- Bin 2 contains items 2 and 3 with total weight $w_2 + w_3 = 5 + 4 = 9$;
- Bin 3 contains item 4 with weight $w_4 = 7$.

3.4 Half-full property

- The results of this section do not depend on the rule used to select the bin: they are proved using *only* the Any Fit condition, and therefore hold for every algorithm of the family, including FF, BF and WF.
- The **half-full property** of a feasible BPP solution states that, with the exception of at most one bin, all bins are filled to more than half of their capacity, or equivalently, that for any pair of distinct bins at least one of them is filled to more than half capacity.
- The half-full property alone does *not* guarantee that two bins cannot be merged: for instance, with $c = 10$ two bins of total weight 6 and 1 satisfy the property but can be merged into a single bin. Its role is different: it is exactly what is needed to prove a relative performance guarantee of 2.

Definition 3.2: Half-full property

Given a feasible BPP solution with k bins and packing patterns $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ (where P_b is the set of items packed into bin b, for $b \in \{1, 2, \ldots, k\}$), we say that the solution satisfies the **half-full property** if one of the following equivalent conditions holds:

- there exists at most one bin $b \in \{1, 2, \ldots, k\}$ such that
$$\sum_{i \in P_b} w_i \leq \frac{c}{2}.$$
- for any two distinct bins $b, b' \in \{1, 2, \ldots, k\}$, at least one of them is filled to more than half of its capacity, i.e.,
$$\sum_{i \in P_b} w_i > \frac{c}{2} \quad \text{or} \quad \sum_{i \in P_{b'}} w_i > \frac{c}{2}.$$

Example 3.10: Half-full property

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

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

Initially $nb := 1$, $rc := c$. The NF algorithm produces the following feasible solution:

1. Item 1 (weight $w[1] = 5$): $w[1] = 5 \leq rc = 10$, so no new bin is opened. Item 1 is packed into bin $nb = 1$, and $rc \leftarrow 10 - 5 = 5$.

2. Item 2 (weight $w[2] = 5$): $w[2] = 5 \leq rc = 5$, so no new bin is opened. Item 2 is packed into bin $nb = 1$, and $rc \leftarrow 5 - 5 = 0$.
3. Item 3 (weight $w[3] = 5$): $w[3] = 5 > rc = 0$, so $nb \leftarrow 2$ and $rc \leftarrow c = 10$. Item 3 is packed into bin $nb = 2$, and $rc \leftarrow 10 - 5 = 5$.
4. Item 4 (weight $w[4] = 5$): $w[4] = 5 \leq rc = 5$, so no new bin is opened. Item 4 is packed into bin $nb = 2$, and $rc \leftarrow 5 - 5 = 0$.

Hence NF returns:

$$bin = \begin{pmatrix} 1 & 1 & 2 & 2 \end{pmatrix}, \qquad nb = 2.$$

The packing patterns are $P_1 = \{1, 2\}$, $P_2 = \{3, 4\}$, with total weights:

$$\sum_{i \in P_1} w_i = 5 + 5 = 10 > \frac{c}{2} = 5, \qquad \sum_{i \in P_2} w_i = 5 + 5 = 10 > \frac{c}{2} = 5.$$

Since there is *no* bin with total weight at most $\frac{c}{2}$, the solution **satisfies** the half-full property.

Proposition 3.3

For every BPP instance, the heuristic BPP solutions produced by *any* Any Fit algorithm satisfy the half-full property. In particular this holds for FF, BF and WF.

Proof. Let $\mathscr{A}$ be an Any Fit algorithm, let $k := \mathscr{A}(I)$, and let $P_1, P_2, \ldots, P_k$ be the packing patterns produced by $\mathscr{A}$, indexed in the order the bins are opened. If $k = 1$ there is nothing to prove, so assume $k \geq 2$.

Suppose, by contradiction, that there exist two distinct bins b and b' such that

$$\sum_{i \in P_b} w_i \leq \frac{c}{2} \qquad \text{and} \qquad \sum_{i \in P_{b'}} w_i \leq \frac{c}{2}.$$

Summing the two inequalities gives

$$\sum_{i \in P_b} w_i + \sum_{i \in P_{b'}} w_i \leq \frac{c}{2} + \frac{c}{2} = c,$$

which contradicts the property of the Any Fit class proved in Section 3.3, namely that the items of two distinct bins never fit together into a single bin. Therefore there can be at most one bin with total weight at most $\frac{c}{2}$.

Only the Any Fit condition is used, so the statement holds for every algorithm of the class. □

Example 3.11: Half-full property

As an illustration of Proposition 3.3, we consider a bin of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, 3, 4\}$ of 4 items ($n = 4$), with the following item weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$
w_i	7	7	3	4

Initially $nb := 1$, $rc[1] := c = 10$. The FF algorithm produces the following feasible solution:

1. Item 1 (weight 7): $b = 1$: $w[1] = 7 \leq rc[1] = 10$? Yes, so $s \leftarrow 1$. Item 1 is packed into bin 1, and $rc[1] \leftarrow 10 - 7 = 3$.
2. Item 2 (weight 7): $b = 1$: $w[2] = 7 \leq rc[1] = 3$? No. No bin fits, so FF opens bin 2 ($rc[2] \leftarrow 10$). Item 2 is packed into bin 2, and $rc[2] \leftarrow 10 - 7 = 3$.
3. Item 3 (weight 3): $b = 1$: $w[3] = 3 \leq rc[1] = 3$? Yes, so $s \leftarrow 1$. Item 3 is packed into bin 1, and $rc[1] \leftarrow 3 - 3 = 0$.
4. Item 4 (weight 4): $b = 1$: $w[4] = 4 \leq rc[1] = 0$? No. $b = 2$: $w[4] = 4 \leq rc[2] = 3$? No. No bin fits, so FF opens bin 3 ($rc[3] \leftarrow 10$). Item 4 is packed into bin 3, and $rc[3] \leftarrow 10 - 4 = 6$.

Hence FF returns:

$$bin = \begin{pmatrix} 1 & 2 & 1 & 3 \end{pmatrix}, \qquad nb = 3.$$

The packing patterns are $P_1 = \{1, 3\}$, $P_2 = \{2\}$, $P_3 = \{4\}$, with total weights:

$$\sum_{i \in P_1} w_i = 7 + 3 = 10 > \frac{c}{2} = 5, \qquad \sum_{i \in P_2} w_i = 7 > \frac{c}{2} = 5, \qquad \sum_{i \in P_3} w_i = 4 \leq \frac{c}{2} = 5.$$

Since there is *exactly one* bin (bin 3) with total weight at most $\frac{c}{2}$, the solution **satisfies** the half-full property, as guaranteed by Proposition 3.3 since FF is an Any Fit algorithm.

The Next Fit may produce solutions with many bins filled to less than half capacity and therefore does not satisfy the half-full property.

Example 3.12: Half-full property

We consider again the instance of Example 3.3, with bin capacity $c = 10$ and item weights $w = (3, 8, 3, 8)$, on which NF opens a new bin for every item and returns:

$$bin = \begin{pmatrix} 1 & 2 & 3 & 4 \end{pmatrix}, \qquad nb = 4.$$

The packing patterns are:

$$P_1 = \{i \in \{1, 2, 3, 4\} : bin_i = 1\} = \{1\},$$

$$P_2 = \{i \in \{1,2,3,4\} : bin_i = 2\} = \{2\},$$
$$P_3 = \{i \in \{1,2,3,4\} : bin_i = 3\} = \{3\},$$
$$P_4 = \{i \in \{1,2,3,4\} : bin_i = 4\} = \{4\}.$$

The total weights in the bins are:

$$\sum_{i \in P_1} w_i = 3 \leq \frac{c}{2} = 5, \quad \sum_{i \in P_2} w_i = 8 > \frac{c}{2} = 5, \quad \sum_{i \in P_3} w_i = 3 \leq \frac{c}{2} = 5, \quad \sum_{i \in P_4} w_i = 8 > \frac{c}{2} = 5.$$

Since there are *two* bins (bins 1 and 3) with total weight at most $\frac{c}{2}$, the solution does **not** satisfy the half-full property.

3.5 Performance guarantees

- We now determine the worst case performance ratio of the four algorithms. Next Fit is not an Any Fit algorithm, so it needs an argument of its own. For the three Any Fit algorithms the bound $\mathscr{A}(I) \leq 2\, z(I)$ comes at once from the half-full property, without any assumption on the rule used to select the bin, and the question is how tight that bound is: for WF the factor 2 cannot be improved, whereas FF and BF obey a strictly better bound.

Proposition 3.4

Let $\mathscr{A}$ be a heuristic algorithm for the BPP that produces solutions satisfying the half-full property. Then, for every BPP instance I, we have

$$\frac{\mathscr{A}(I)}{z(I)} \leq 2.$$

Proof. Let $k := \mathscr{A}(I)$, and let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the packing patterns produced by $\mathscr{A}$, indexed in the order the bins are opened.

Since the solution satisfies the half-full property, there exists at most one bin $b \in \{1, 2, \ldots, k\}$ such that

$$\sum_{i \in P_b} w_i \leq \frac{c}{2}.$$

Therefore, at least $k - 1$ bins have total weight strictly greater than $\frac{c}{2}$. Summing over these bins, we obtain

$$\sum_{i \in \mathscr{I}} w_i \;>\; (k-1)\,\frac{c}{2}.$$

On the other hand, any feasible packing using $z(I)$ bins satisfies

$$\sum_{i \in \mathscr{I}} w_i \;\leq\; z(I)\, c.$$

Combining the two inequalities yields

$$(k-1)\,\frac{c}{2} \;<\; \sum_{i\in\mathscr{I}} w_i \;\le\; z(I)\,c \qquad\Longrightarrow\qquad k-1 \;<\; 2\,z(I) \qquad\Longrightarrow\qquad k \;<\; 2\,z(I)+1.$$

Since both k and $z(I)$ are positive integers, we have

$$k \;\le\; 2\,z(I).$$

Dividing by $z(I) \ge 1$ gives

$$\frac{k}{z(I)} = \frac{\mathscr{A}(I)}{z(I)} \;\le\; 2.$$

□

- Since every Any Fit algorithm produces solutions satisfying the half-full property, it follows immediately that

$$\frac{\mathscr{A}(I)}{z(I)} \le 2 \qquad \text{for every Any Fit algorithm } \mathscr{A}$$

and for every BPP instance I. In particular

$$\frac{FF(I)}{z(I)} \le 2, \qquad \frac{BF(I)}{z(I)} \le 2 \qquad \text{and} \qquad \frac{WF(I)}{z(I)} \le 2,$$

and therefore FF, BF and WF are 2-approximation algorithms.

3.5.1 The worst case performance ratio of NF

Proposition 3.5

For every BPP instance I, we have:

$$\frac{NF(I)}{z(I)} \le 2. \tag{3.1}$$

Proof. Let $k := NF(I)$, and let $P_1, P_2, \ldots, P_k$ be the packing patterns produced by NF, indexed in the order the bins are opened.

Fix any $b \in \{1, 2, \ldots, k-1\}$. The packing pattern P_b corresponds to a bin that is closed when the next item (call it f) does not fit into it, hence

$$\sum_{i\in P_b} w_i + w_f > c.$$

That same item f is the first item placed into the next bin, whose packing pattern is P_{b+1}, so

$$w_f \le \sum_{i\in P_{b+1}} w_i.$$

Therefore,

$$\sum_{i \in P_b} w_i + \sum_{i \in P_{b+1}} w_i \; > \; c.$$

Now consider the disjoint consecutive pairs $(P_1, P_2), (P_3, P_4), \ldots$. Summing the inequality above over the $\lfloor k/2 \rfloor$ pairs yields

$$\underbrace{\sum_{b=1}^{k} \sum_{i \in P_b} w_i}_{= \sum_{i \in \mathscr{I}} w_i \; \leq \; z(I)\, c} \quad > \left\lfloor \frac{k}{2} \right\rfloor c \quad \Longrightarrow \quad z(I) \; > \; \left\lfloor \frac{k}{2} \right\rfloor.$$

Since $z(I)$ is an integer, it follows that

$$z(I) \geq \left\lfloor \frac{k}{2} \right\rfloor + 1.$$

Then

$$\begin{cases} \text{if } k \text{ is even, then } \left\lfloor \frac{k}{2} \right\rfloor = \frac{k}{2} \implies z(I) \geq \frac{k}{2} + 1 \implies k \leq 2z(I) - 2 < 2z(I), \\ \\ \text{if } k \text{ is odd, then } \left\lfloor \frac{k}{2} \right\rfloor = \frac{k-1}{2} \implies z(I) \geq \frac{k+1}{2} \implies k \leq 2z(I) - 1 < 2z(I). \end{cases}$$

Therefore $k < 2z(I)$, and dividing by $z(I) \geq 1$ yields

$$\frac{k}{z(I)} = \frac{NF(I)}{z(I)} \leq 2.$$

□

Proposition 3.6

There are BPP instances I_n such that:

$$\frac{NF(I_n)}{z(I_n)} \; \to \; 2 \qquad \text{as} \qquad n \to +\infty. \tag{3.2}$$

Proof. Fix an integer $n \geq 8$ divisible by 4 and consider the BPP instance I_n with bin capacity $c = n$ and item set $\mathscr{I} = \{1, 2, \ldots, n\}$, where the item weights are

$$w_i = \begin{cases} \frac{n}{2}, & \text{if } i \text{ is odd,} \\ \\ 2, & \text{if } i \text{ is even.} \end{cases}$$

Items are processed by NF in the natural order $1, 2, \ldots, n$. For every odd i,

$$w_i + w_{i+1} = \frac{n}{2} + 2 < n,$$

so NF packs items i and $i+1$ into the same bin. The remaining space is

$$n - \left(\frac{n}{2} + 2\right) = \frac{n}{2} - 2 < \frac{n}{2},$$

hence the next item of weight $\frac{n}{2}$ does not fit and NF opens a new bin. Therefore,

$$NF(I_n) = \frac{n}{2}.$$

Pack the $\frac{n}{2}$ items of weight $\frac{n}{2}$ two per bin, obtaining $\frac{n}{4}$ full bins, and pack all $\frac{n}{2}$ small items together (since $\frac{n}{2} \cdot 2 = n$) in one additional bin; thus $z(I_n) \leq \frac{n}{4} + 1$. On the other hand,

$$z(I_n) \geq \left\lceil \frac{\sum_{i \in \mathscr{I}} w_i}{c} \right\rceil = \left\lceil \frac{\frac{n}{2}\,\frac{n}{2} + \frac{n}{2}\,2}{n} \right\rceil = \left\lceil \frac{n}{4} + 1 \right\rceil = \frac{n}{4} + 1,$$

Hence

$$z(I_n) = \frac{n}{4} + 1.$$

Consequently,

$$\lim_{n\to\infty} \frac{NF(I_n)}{z(I_n)} = \lim_{n\to\infty} \frac{\frac{n}{2}}{\frac{n}{4}+1} = \lim_{n\to\infty} \frac{\frac{n}{2}}{\frac{n+4}{4}} = \lim_{n\to\infty} \frac{2n}{n+4} = \lim_{n\to\infty} \frac{2}{1+\frac{4}{n}} = 2.$$

□

From Proposition 3.5 and Proposition 3.6 it follows that the worst case performance ratio of NF is two, i.e., we have

$$r(NF) = 2$$

3.5.2 The worst case performance ratio of WF

Proposition 3.7

There are BPP instances I_n such that:

$$\frac{WF(I_n)}{z(I_n)} \to 2 \qquad \text{as} \qquad n \to +\infty. \tag{3.3}$$

Proof. Fix an integer $n \geq 8$ divisible by 4 and consider the BPP instance I_n with bin capacity $c = n$ and item set $\mathscr{I} = \{1, 2, \ldots, n\}$, where the item weights are

$$w_i = \begin{cases} \frac{n}{2}, & \text{if } i \text{ is odd,} \\ 2, & \text{if } i \text{ is even,} \end{cases}$$

that is, the same instance used above for NF. Items are processed by WF in the natural order $1, 2, \ldots, n$.

We show by induction that, after the first $2k$ items have been processed, WF has opened

exactly k bins, each with residual capacity $\frac{n}{2} - 2$.

For $k = 1$, item 1 (of weight $\frac{n}{2}$) is packed into bin 1, whose residual capacity becomes $\frac{n}{2}$; then item 2 (of weight 2) is packed into bin 1, the only open bin, whose residual capacity becomes $\frac{n}{2} - 2$.

Assume the claim holds for some k, and consider items $2k+1$ and $2k+2$. Item $2k+1$ has weight $\frac{n}{2}$, while every open bin has residual capacity $\frac{n}{2} - 2 < \frac{n}{2}$: the item fits nowhere, so WF opens bin $k+1$, whose residual capacity becomes $\frac{n}{2}$. Item $2k+2$ has weight 2 and fits into every open bin; since bin $k+1$ is the one with the *largest* residual capacity, WF packs the item there, and the residual capacity of bin $k+1$ becomes $\frac{n}{2} - 2$. This proves the claim for $k+1$.

It is precisely the Worst Fit rule that keeps every bin at residual capacity $\frac{n}{2} - 2$: the small items are always placed in the emptiest bin, so no bin ever receives two items of weight $\frac{n}{2}$. Taking $k = \frac{n}{2}$ we obtain

$$WF(I_n) = \frac{n}{2}.$$

Pack the $\frac{n}{2}$ items of weight $\frac{n}{2}$ two per bin, obtaining $\frac{n}{4}$ full bins, and pack all $\frac{n}{2}$ small items together (since $\frac{n}{2} \cdot 2 = n$) in one additional bin; thus $z(I_n) \leq \frac{n}{4} + 1$. On the other hand,

$$z(I_n) \geq \left\lceil \frac{\sum_{i \in \mathscr{I}} w_i}{c} \right\rceil = \left\lceil \frac{\frac{n}{2}\,\frac{n}{2} + \frac{n}{2}\,2}{n} \right\rceil = \left\lceil \frac{n}{4} + 1 \right\rceil = \frac{n}{4} + 1,$$

hence

$$z(I_n) = \frac{n}{4} + 1.$$

Consequently,

$$\lim_{n\to\infty} \frac{WF(I_n)}{z(I_n)} = \lim_{n\to\infty} \frac{\frac{n}{2}}{\frac{n}{4}+1} = \lim_{n\to\infty} \frac{2n}{n+4} = \lim_{n\to\infty} \frac{2}{1+\frac{4}{n}} = 2.$$

□

From Proposition 3.4 and Proposition 3.7 it follows that the worst case performance ratio of WF is two, i.e., we have

$$r(WF) = 2$$

For WF the factor 2 therefore cannot be improved.

- We have $r(WF) = r(NF) = 2$: in the worst case WF is no better than NF. The two algorithms are however far from equivalent. WF never opens a new bin while the item still fits into one of the bins opened so far, whereas NF closes a bin as soon as an item does not fit and never reconsiders it. As a consequence NF may leave arbitrarily many bins filled to less than half of their capacity, whereas WF, as shown in the section on the half-full property, can leave at most one.

- The value $r(WF) = 2$ is due to

  D.S. Johnson, *Fast algorithms for bin packing*, Journal of Computer and System Sciences, 8(3):272–314, 1974.

3.5.3 The worst case performance ratios of FF and BF

Proposition 3.8

For every BPP instance I, we have:

$$FF(I) \leq \frac{17}{10} z(I) \quad \text{and} \quad BF(I) \leq \frac{17}{10} z(I). \tag{3.4}$$

Proof. Proof omitted. The two bounds were first proved in the weaker form

$$FF(I) \leq \frac{17}{10} z(I) + 2 \qquad \text{and} \qquad BF(I) \leq \frac{17}{10} z(I) + 2$$

in

D.S. Johnson, A. Demers, J.D. Ullman, M.R. Garey and R.L. Graham, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM Journal on Computing, 3(4):299–325, 1974.

The additive constant has later been removed, in the form stated above, respectively in

G. Dósa and J. Sgall, *First Fit bin packing: a tight analysis*, STACS 2013, pp. 538–549,

G. Dósa and J. Sgall, *Optimal analysis of Best Fit bin packing*, ICALP 2014, pp. 429–441. □

Proposition 3.9

There are BPP instances I_n such that:

$$\frac{FF(I_n)}{z(I_n)} \to \frac{17}{10} \quad \text{and} \quad \frac{BF(I_n)}{z(I_n)} \to \frac{17}{10} \qquad \text{as} \qquad n \to +\infty. \tag{3.5}$$

Proof. Proof omitted. Families of instances on which the ratio of FF and BF approaches $\frac{17}{10}$ are given in the paper by Johnson, Demers, Ullman, Garey and Graham cited above, and in

D.S. Johnson, *Near-optimal bin packing algorithms*, PhD thesis, Massachusetts Institute of Technology, 1973. □

From Proposition 3.8 and Proposition 3.9 it follows that the worst case performance ratio of FF and BF is seventeen tenths, i.e., we have

$$r(FF) = r(BF) = \frac{17}{10}$$

Unlike WF, FF and BF stay strictly below the factor 2 that the Any Fit condition alone guarantees.

3.5.4 What the Any Fit condition does and does not give

- **What the Any Fit condition gives.** The Any Fit condition alone guarantees the half-full property, and therefore the bound

  $$\mathscr{A}(I) \leq 2\, z(I)$$

  for every algorithm $\mathscr{A}$ of the class. This bound cannot be improved for the class as a whole: WF is an Any Fit algorithm and, as shown above, $r(WF) = 2$.

- **What the Any Fit condition leaves free.** The rule used to select one of the bins in which the item fits is what distinguishes the members of the family. On the instances of Proposition 3.9 the ratio of FF and BF tends to $\frac{17}{10}$, while on those of Proposition 3.7 the ratio of WF tends to 2: the selection rule alone accounts for the whole gap between $\frac{17}{10}$ and 2.

- It is known that no algorithm of the class can go below $\frac{17}{10}$, so $\frac{17}{10}$ and 2 are exactly the best and the worst behaviour that the Any Fit condition allows, and both are attained, the first by FF and BF, the second by WF.

- The Any Fit class and the bounds above are due to the paper of Johnson already cited in Section 3.5.2: in particular, the lower bound $\frac{17}{10}$ for every Any Fit algorithm is due to that paper (D.S. Johnson, 1974).

3.6 Typical mistakes

Warning

- **Believing that Next Fit is an Any Fit algorithm.** NF never reconsiders a closed bin, so it may open a new bin for an item that would fit into an older one (Example 3.3).
- **Choosing the wrong bin in BF and WF.** BF packs the item into the bin with the *smallest* residual capacity among the bins in which it fits, WF into the one with the *largest* residual capacity; in case of ties, both take the bin of smallest index.
- **Reading the half-full property as "no two bins can be merged".** Two bins of total weight 6 and 1 with $c = 10$ satisfy the property and can be merged; the property is what proves the guarantee 2.
- **Treating $r = 2$ as a value that is attained.** The worst-case ratios of NF and WF are suprema: the families of instances approach 2 as n grows, and no instance reaches it.
- **Changing the order of the items.** The four algorithms process the items in the order in which they are given; sorting them first is a different algorithm.

3.7 In summary

Summary

- NF, FF, BF and WF place each item as it arrives; NF keeps one open bin, the other three keep all the bins opened so far and open a new bin only when the item fits nowhere (Any Fit class).
- In an Any Fit solution the items of two bins never fit together, so all the bins but at most one are more than half full (half-full property), and $\mathscr{A}(I) \leq 2\, z(I)$.
- $r(NF) = r(WF) = 2$ and $r(FF) = r(BF) = \frac{17}{10}$: the selection rule alone separates the best and the worst behaviour of the Any Fit class.

Dashboard of the BPP instance $c = 10$, $w = (6, 5, 4, 7, 3)$. The bounds on $z(I)$ found so far:

Source	Lower bound	Upper bound	From
Counting bound	3	–	$\lceil \frac{25}{10} \rceil$
This chapter, NF	–	3	$bin = (1, 2, 2, 3, 3)$
This chapter, FF, BF, WF	–	3	3 bins each, with three different assignments
Optimal value	3	3	lower and upper bound coincide

3.8 C++ implementation

- In this C++ implementation section we use 0-based indexing (items and bins are numbered starting from 0). The algorithmic logic is unchanged.
- The step-by-step executions reported in this section all refer to the same BPP instance, namely a bin of capacity $c = 10$ and a set $\mathscr{I} = \{1, 2, 3, 4, 5\}$ of 5 items ($n = 5$), with the following item weights:

	$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
w_i	6	5	4	7	3

  With the 0-based indexing of this section the items are numbered $0, 1, 2, 3, 4$ and their weights are $w[0] = 6$, $w[1] = 5$, $w[2] = 4$, $w[3] = 7$ and $w[4] = 3$. This is the instance used in the examples of the previous sections, so the solutions found here are the ones already computed there.
- This is a C++ implementation of **Next Fit**:

```cpp
// items are processed in the input order
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i
// it is assumed that n >= 1

int NextFit(int n, int c, int *w, int *bin)
{
    int nb = 1;   // current bin (number of open bins)
    int rc  = c;   // residual capacity of the current bin

    for (int i = 0; i < n; i++)
    {
        // if item i does not fit into the current bin, open a new bin
        if (w[i] > rc)
        {
            nb++;  // move to the next bin
            rc = c; // reset the residual capacity
        }

        bin[i] = nb - 1; // pack item i into the current bin
        rc -= w[i];       // update the residual capacity
    }

    return nb;
}
```

This is the step-by-step execution of the C++ implementation of NF for this instance.

Initialization

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	-	-	-	-	-

$nb = 1$ $rc = 10$

1. Iteration 1 – Item 0 is packed into bin 0

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	-	-	-	-

$nb = 1$ $rc = 4$

2. Iteration 2 – Item 1 is packed into bin 1

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	-	-	-

$nb = 2$ $rc = 5$

3. Iteration 3 – Item 2 is packed into bin 1

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	-	-

$nb = 2$ $rc = 1$

4. Iteration 4 – Item 3 is packed into bin 2

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	2	-

$nb = 3$ $rc = 3$

5. Iteration 5 – Item 4 is packed into bin 2

$bin[i]$	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	2	2

$nb = 3$ $rc = 0$

- This is a `C++` implementation of **First Fit**:

```
// items are processed in the input order
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i
// it is assumed that n >= 1

int FirstFit(int n, int c, int *w, int *bin)
{
    int nb = 1;          // number of currently open bins
    int *rc = new int[n]; // residual capacities of bins (at most n bins)
    rc[0] = c;            // residual capacity of the first bin

    for (int i = 0; i < n; i++)
    {
```

```
14          int s = -1;       // selected bin for item i (-1 means: not found yet)
15
16          // scan bins in the order they were opened and pick the first that fits
17          for (int b = 0; b < nb; b++)
18          {
19              if (w[i] <= rc[b])
20              {
21                  s = b;  // selected bin
22                  break;
23              }
24          }
25
26          // if no open bin can accommodate item i, open a new bin
27          if (s == -1)
28          {
29              rc[nb] = c;   // initialize residual capacity of the new bin
30              s = nb;      // select the new bin
31              nb++;         // increase the number of open bins
32          }
33
34          bin[i] = s;      // pack item i into the selected bin
35          rc[s] -= w[i];   // update the residual capacity
36      }
37
38      delete[] rc;
39      return nb;
40  }
```

This is the step-by-step execution of the C++ implementation of FF for this instance.

Initialization

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	-	-	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	10	-	-	-	-

$nb = 1$

1. Iteration 1 – Item 0 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	-	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	-	-	-	-

$nb = 1$

2. Iteration 2 – Item 1 is packed into bin 1

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	5	-	-	-

$nb = 2$

3. Iteration 3 – Item 2 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	5	-	-	-

$nb = 2$

4. Iteration 4 – Item 3 is packed into bin 2

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	2	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	5	3	-	-

$nb = 3$

5. Iteration 5 – Item 4 is packed into bin 1

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	2	1
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	2	3	-	-

$nb = 3$

- This is a `C++` implementation of **Best Fit**:

```
// items are processed in the input order
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i
// it is assumed that n >= 1

int BestFit(int n, int c, int *w, int *bin)
{
    int nb = 1;          // number of currently open bins
```

```
    int *rc = new int[n]; // residual capacities of bins (at most n bins)
    rc[0] = c;            // residual capacity of the first bin

    for (int i = 0; i < n; i++)
    {
        int s = -1;       // selected bin for item i (-1 means: not found yet)
        int minrc = c + 1; // smallest residual capacity found so far

        // among the bins where item i fits, pick the one with the smallest
    residual capacity
        for (int b = 0; b < nb; b++)
        {
            if (w[i] <= rc[b] && rc[b] < minrc)
            {
                minrc = rc[b];
                s = b;  // selected bin
            }
        }

        // if no open bin can accommodate item i, open a new bin
        if (s == -1)
        {
            rc[nb] = c;   // initialize residual capacity of the new bin
            s = nb;      // select the new bin
            nb++;         // increase the number of open bins
        }

        bin[i] = s;      // pack item i into the selected bin
        rc[s] -= w[i];   // update the residual capacity
    }

    delete[] rc;
    return nb;
}
```

This is the step-by-step execution of the C++ implementation of BF for this instance.

Initialization

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	-	-	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	10	-	-	-	-

$nb = 1$

1. Iteration 1 – Item 0 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	-	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	-	-	-	-

$nb = 1$

2. Iteration 2 – Item 1 is packed into bin 1

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	5	-	-	-

$nb = 2$

3. Iteration 3 – Item 2 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	5	-	-	-

$nb = 2$

4. Iteration 4 – Item 3 is packed into bin 2

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	2	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	5	3	-	-

$nb = 3$

5. Iteration 5 – Item 4 is packed into bin 2

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	0	2	2
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	0	5	0	-	-

$nb = 3$

- This is a `C++` implementation of **Worst Fit**:

```cpp
// items are processed in the input order
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i
// it is assumed that n >= 1

int WorstFit(int n, int c, int *w, int *bin)
{
    int nb = 1;           // number of currently open bins
    int *rc = new int[n]; // residual capacities of bins (at most n bins)
    rc[0] = c;            // residual capacity of the first bin

    for (int i = 0; i < n; i++)
    {
        int s = -1;       // selected bin for item i (-1 means: not found yet)
        int maxrc = -1;   // largest residual capacity found so far

        // among the bins where item i fits, pick the one with the largest residual capacity
        for (int b = 0; b < nb; b++)
        {
            if (w[i] <= rc[b] && rc[b] > maxrc)
            {
                maxrc = rc[b];
                s = b;  // selected bin
            }
        }

        // if no open bin can accommodate item i, open a new bin
        if (s == -1)
        {
            rc[nb] = c;   // initialize residual capacity of the new bin
            s = nb;      // select the new bin
            nb++;         // increase the number of open bins
        }

        bin[i] = s;      // pack item i into the selected bin
        rc[s] -= w[i];   // update the residual capacity
    }

    delete[] rc;
    return nb;
}
```

This is the step-by-step execution of the C++ implementation of WF for this instance.

Initialization

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	-	-	-	-	-

rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	10	-	-	-	-

$nb = 1$

1. Iteration 1 – Item 0 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	-	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	-	-	-	-

$nb = 1$

2. Iteration 2 – Item 1 is packed into bin 1

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	-	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	5	-	-	-

$nb = 2$

3. Iteration 3 – Item 2 is packed into bin 1

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	-	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	1	-	-	-

$nb = 2$

4. Iteration 4 – Item 3 is packed into bin 2

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	2	-
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	4	1	3	-	-

$nb = 3$

5. Iteration 5 – Item 4 is packed into bin 0

bin[i]	$i=0$	$i=1$	$i=2$	$i=3$	$i=4$
	0	1	1	2	0
rc[b]	$b=0$	$b=1$	$b=2$	$b=3$	$b=4$
	1	1	3	-	-

$nb = 3$

- All the functions of this section are in the folder `BPP_HEUR_CODE/src`. The command `make test` compiles them and runs every example of this chapter and every instance of the exercises, checking the number of bins and the item-bin assignment against the expected ones.

Coloring problems

Chapter 4

Vertex coloring: limits of approximation

Prerequisites

- From this module, the chapter «Principles of approximation algorithms»: performance ratio, α-approximation algorithm, additive performance guarantee.
- From the module *Combinatorial Optimization Problems*, the chapter «Coloring problems»: the definition of the VCP, the chromatic number $\chi(G)$, the clique bound $\omega(G) \leq \chi(G)$ and the Petersen graph.
- From the module *Network Optimization*, the chapter «Graphs»: the breadth-first search.
- From the module *Combinatorial Optimization Problems*, the chapter «Combinatorial optimization: definitions»: polynomial time, the classes P and NP, NP-completeness.

Objectives

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

- prove that no polynomial algorithm has a performance ratio below $\frac{4}{3}$ for the VCP, unless $P = NP$;
- state the result of Zuckerman and explain why the VCP has no constant guarantee;
- color bipartite graphs and interval graphs optimally, and bound the colors of the greedy coloring by $\Delta(G) + 1$;
- prove that planar graphs can be colored with 5 colors in polynomial time, and derive an additive guarantee for them.

The problem of this chapter is the vertex coloring problem (VCP): its definition and its assumptions are on its page.

4.1 Notation

- As on the problem page, $G = (V, E)$ is a simple graph with n vertices and m edges, $N(v)$ is the set of the neighbours of v, $\deg(v) = |N(v)|$ its degree, and $\Delta(G) = \max_{v \in V} \deg(v)$ the **maximum degree**. The optimal value of the VCP is the chromatic number, $z(I) = \chi(G)$.
- **Recall.** As in the chapter «Principles of approximation algorithms», $\mathscr{A}(I)$ is the number of colors of the coloring returned by the algorithm $\mathscr{A}$. The VCP is a minimization problem, hence
$$\mathscr{A}(I) \geq \chi(G) \qquad \text{and} \qquad \frac{\mathscr{A}(I)}{\chi(G)} \geq 1$$
for every instance I.
- Giving each vertex its own color is always feasible, so $\chi(G) \leq n$, and the ratio n is trivial. This chapter shows that a polynomial algorithm cannot do much better on all graphs, and then the classes of graphs on which it can.

4.2 The VCP is not approximable

4.2.1 No ratio below four thirds

- The proof follows the scheme of all the negative results of approximation: an algorithm with a too good guarantee would decide in polynomial time an NP-complete problem.
- The NP-complete problem is **3-colorability**: given a graph G, decide whether $\chi(G) \leq 3$. It remains NP-complete on planar graphs of maximum degree 4 (Garey, Johnson and Stockmeyer, 1976).

Theorem 4.1

Let $\mathscr{A}$ be a polynomial algorithm for the VCP. If $P \neq NP$, there is a graph G such that
$$\frac{\mathscr{A}(I)}{\chi(G)} \geq \frac{4}{3}.$$

Proof. Suppose that $\frac{\mathscr{A}(I)}{\chi(G)} < \frac{4}{3}$ for every graph G, and run $\mathscr{A}$ on a graph G.

1. If $\chi(G) \leq 3$, then $\mathscr{A}(I) < \frac{4}{3}\chi(G) \leq 4$. The number of colors is an integer, hence $\mathscr{A}(I) \leq 3$.
2. If $\chi(G) \geq 4$, then $\mathscr{A}(I) \geq \chi(G) \geq 4$, because $\mathscr{A}$ returns a feasible coloring.

Hence $\chi(G) \leq 3$ if and only if $\mathscr{A}(I) \leq 3$. Running $\mathscr{A}$ and counting its colors takes polynomial time, so 3-colorability would be in P, and $P = NP$. □

- The proof uses only the integrality of $\chi(G)$: a ratio below $\frac{4}{3}$ cannot confuse 3 colors with 4. With 2 colors the same argument says nothing, because deciding whether $\chi(G) \leq 2$ is easy (Section 4.3.1).
- The planar graphs of maximum degree 4 are a special case of the VCP: the theorem holds on them too.

4.2.2 Much stronger results

- The theorem excludes only the ratios below $\frac{4}{3}$. Much more is known, and we state it without proof.

Theorem 4.2: Zuckerman, 2007

For every $\varepsilon > 0$, if $P \neq NP$, no polynomial algorithm $\mathscr{A}$ for the VCP satisfies, for every graph G with n vertices,

$$\frac{\mathscr{A}(I)}{\chi(G)} \leq n^{1-\varepsilon}.$$

- The ratio n is trivial, so the chromatic number is essentially as hard to approximate as possible. The proof uses probabilistically checkable proofs and extractors.
- Earlier, Garey and Johnson (1976) had raised the bound $\frac{4}{3}$ to 2, and Feige and Kilian (1998) had proved the bound $n^{1-\varepsilon}$ under a stronger assumption than $P \neq NP$.

Corollary 4.3

If $P \neq NP$, the VCP has no α-approximation algorithm, for any constant $\alpha \geq 1$.

Proof. Let $\mathscr{A}$ be a polynomial α-approximation algorithm, and take $\varepsilon = \frac{1}{2}$. The graphs with $n > \alpha^2$ vertices satisfy $\frac{\mathscr{A}(I)}{\chi(G)} \leq \alpha < n^{1/2}$. The graphs with $n \leq \alpha^2$ vertices are solved exactly by enumeration, in a time bounded by a constant that depends only on α, with ratio $1 \leq n^{1/2}$. The resulting algorithm is polynomial and has ratio at most $n^{1/2}$ on every graph, against Theorem 4.2. □

The VCP is not approximable: unless $P = NP$, no polynomial algorithm has a constant performance guarantee, not even a guarantee of the form $n^{1-\varepsilon}$. Positive results exist only on special classes of graphs.

4.3 Special cases

- On the four classes of this section the VCP is solved exactly, or with a guarantee, in polynomial time. The classes are recognized from the graph, or given with it (the intervals of an interval graph).

4.3.1 Bipartite graphs

Definition 4.4: bipartite graph

A graph $G = (V, E)$ is **bipartite** if $\chi(G) \leq 2$, i.e., if V can be split into two stable sets.

Proposition 4.5

Deciding whether $\chi(G) \leq 2$, and finding a coloring with 2 colors when it exists, takes $O(n + m)$ time.

Proof. Run a breadth-first search from a vertex of each connected component, and let $\ell(v)$ be the level of v, its distance from the root of its component. Color v with color 1 if $\ell(v)$ is even and with color 2 otherwise.

1. If no edge joins two vertices of levels with the same parity, the coloring is feasible.
2. Otherwise, let $\{u, v\}$ be such an edge. The paths of the search tree from the root to u and to v, with the edge $\{u, v\}$, form a closed walk of odd length $\ell(u)+\ell(v)+1$. A closed walk of odd length contains a cycle of odd length, which needs 3 colors: $\chi(G) \geq 3$.

The search and the check of the edges take $O(n + m)$ time. □

- The proof shows the classical characterization: a graph is bipartite if and only if it has no cycle of odd length (Kőnig, 1936).
- On a graph with at least one edge, the search gives $\chi(G) = 2$, or the certificate $\chi(G) \geq 3$ given by an odd cycle.

4.3.2 Graphs of bounded maximum degree

- The **greedy coloring** GC examines the vertices in a given order and gives each vertex the smallest color, among $1, 2, 3, \ldots$, not used by its neighbours already colored. It runs in $O(n + m)$ time.

Proposition 4.6

For every graph G and every order of the vertices,

$$GC(I) \leq \Delta(G) + 1.$$

Proof. When the vertex v is colored, its neighbours already colored use at most $\deg(v) \leq \Delta(G)$ colors. One of the colors $1, 2, \ldots, \Delta(G) + 1$ is therefore free, and the smallest free color is at most $\Delta(G) + 1$. □

- The bound compares $GC(I)$ with $\Delta(G) + 1$, not with $\chi(G)$: on all graphs it is not a performance guarantee, because $\Delta(G)$ can be $n - 1$ while $\chi(G) = 2$ (a star).

- On the graphs of maximum degree at most a *fixed* constant Δ, it is one. Every graph with an edge has $\chi(G) \geq 2$, hence

$$\frac{GC(I)}{\chi(G)} \leq \frac{\Delta+1}{2}.$$

- The bound $\Delta(G)+1$ is attained by complete graphs and by cycles of odd length, whatever the order.

- **Remark (Brooks, 1941).** A connected graph that is neither complete nor a cycle of odd length satisfies $\chi(G) \leq \Delta(G)$, and such a coloring can be found in linear time.

4.3.3 Planar graphs

- A graph is **planar** if it can be drawn in the plane without crossing edges. Whether a graph is planar, and a drawing when it is, can be found in linear time (Hopcroft and Tarjan, 1974).

Lemma 4.7

Every planar graph with $n \geq 3$ vertices has $m \leq 3n-6$ edges, and a vertex of degree at most 5.

Proof. Adding edges to a drawing without creating crossings does not decrease m, so we may assume that no such edge can be added: the drawing is connected and every face is a triangle. Euler's formula for a connected planar drawing with f faces gives $n-m+f=2$. Every face has 3 edges and every edge borders 2 faces, so $3f = 2m$, and

$$2 = n - m + f \;=\; n - m + \frac{2m}{3} \qquad \Longrightarrow \qquad m \leq 3n-6.$$

The degrees sum to $2m \leq 6n-12 < 6n$, so some vertex has degree at most 5. □

Theorem 4.8: Heawood, 1890

Every planar graph can be colored with 5 colors, and such a coloring can be found in polynomial time.

Proof. By induction on n; graphs with $n \leq 5$ are trivial. Let v be a vertex of degree at most 5 (Lemma 4.7), and color $G-v$, which is planar, with 5 colors.

1. If the neighbours of v use at most 4 colors, v takes the fifth one.
2. Otherwise v has 5 neighbours $v_1, \ldots, v_5$, in this order around v in the drawing, with colors $1, \ldots, 5$. Let H_{13} be the graph induced by the vertices of colors 1 and 3.
3. If v_1 and v_3 are in different components of H_{13}, exchange the colors 1 and 3 in the component of v_1: the coloring stays feasible, and color 1 becomes free for v.
4. Otherwise a path of colors 1 and 3 joins v_1 to v_3; with v, it closes a cycle that

separates v_2 from v_4 in the drawing. A path from v_2 to v_4 must cross that cycle, so v_2 and v_4 are in different components of H_{24}: exchanging 2 and 4 in the component of v_2 frees color 2 for v.

Each step uses a search in $O(n+m) = O(n)$ time, and there are n steps: $O(n^2)$ time in total. □

- **Remark (the four color theorem).** Every planar graph can be colored with 4 colors (Appel and Haken, 1977). The proof uses a computer; Robertson, Sanders, Seymour and Thomas (1997) simplified it and gave an algorithm that finds such a coloring in $O(n^2)$ time.

- The algorithm PL for planar graphs combines the previous results: if G has no edge, it uses 1 color; if G is bipartite, it uses 2 colors (Proposition 4.5); otherwise it uses the 5 colors of Theorem 4.8.

Corollary 4.9

PL is a polynomial 2-additive approximation algorithm for the VCP on planar graphs:

$$PL(I) - \chi(G) \leq 2.$$

With the four color algorithm in place of the five color one, the additive error is at most 1.

Proof. In the first two cases PL is exact. In the third case G has a cycle of odd length, so $\chi(G) \geq 3$, while $PL(I) \leq 5$: the additive error is at most $5 - 3 = 2$. With 4 colors it is at most $4 - 3 = 1$. □

- The additive error 1 cannot be improved to 0: deciding whether a planar graph has $\chi(G) \leq 3$ is NP-complete. For the same reason, the ratio $\frac{4}{3}$ of the four color version is the best possible on planar graphs (Theorem 4.1).

4.3.4 Interval graphs

Definition 4.10: interval graph

A graph $G = (V, E)$ is an **interval graph** if every vertex v is associated with a closed interval $[a_v, b_v]$ of the real line, and $\{u, v\} \in E$ if and only if the two intervals intersect.

- Interval graphs describe requests over time that conflict when they overlap: lectures that need a room, reservations that need a resource. A color is a room.

Proposition 4.11

On an interval graph, the greedy coloring with the vertices in order of non-decreasing left endpoint a_v uses $\omega(G)$ colors. It is therefore optimal: $GC(I) = \omega(G) = \chi(G)$.

Proof. Let v be a vertex that receives color k. The colors $1, \dots, k-1$ are used by neighbours u of v colored before v, so with $a_u \le a_v$.

1. Since $[a_u, b_u]$ intersects $[a_v, b_v]$ and $a_u \le a_v$, we have $b_u \ge a_v$: the point a_v lies in the interval of each of these neighbours, and in the interval of v.
2. The k intervals that contain a_v pairwise intersect, so their vertices form a clique and $k \le \omega(G)$.

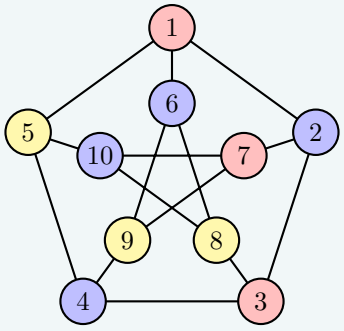
Hence $GC(I) \le \omega(G) \le \chi(G) \le GC(I)$, by the clique bound. □

- Sorting the intervals takes $O(n \log n)$ time, and the coloring $O(n+m)$.
- The order matters: with another order the greedy coloring may use more than $\omega(G)$ colors on an interval graph (Example 4.2).

4.4 Examples

Example 4.1: The special cases on the Petersen graph

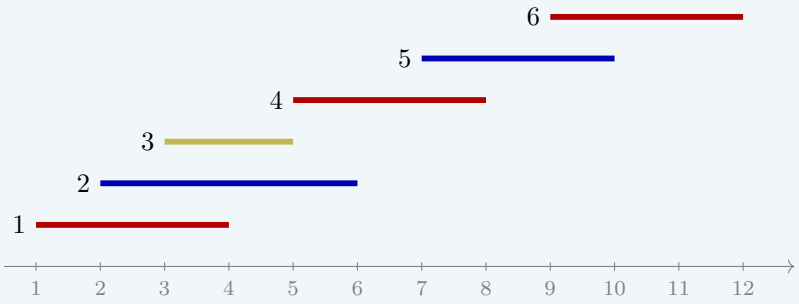
The reference instance of the VCP is the Petersen graph, with $n = 10$, $m = 15$ and $\Delta(G) = 3$.



- **Is it bipartite?** The search from vertex 1 gives the levels 0 to vertex 1, 1 to the vertices $2, 5, 6$, and 2 to the other six. The edge $\{3, 4\}$ joins two vertices of level 2: with the paths $1, 2, 3$ and $1, 5, 4$ it closes the odd cycle $1, 2, 3, 4, 5$, so $\chi(G) \ge 3$.
- **Greedy coloring.** In the order $1, 2, \dots, 10$, GC gives the colors $1, 2, 1, 2, 3, 2, 1, 3, 3, 2$ (red, blue, yellow in the figure): $GC(I) = 3 \le \Delta(G) + 1 = 4$.
- Together, $3 \le \chi(G) \le GC(I) = 3$: the greedy coloring is optimal on this instance.

Example 4.2: The greedy coloring on an interval graph

Six intervals, with the edges between intersecting intervals:



$$[a_v, b_v]: \ [1,4],\ [2,6],\ [3,5],\ [5,8],\ [7,10],\ [9,12],$$

$$E = \big\{\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{3,4\},\{4,5\},\{5,6\}\big\}.$$

- The left endpoints are already in increasing order. GC gives the colors $1,2,3,1,2,1$ (red, blue, yellow in the figure): $GC(I) = 3$.
- The intervals $1,2,3$ contain the point 3, so $\{1,2,3\}$ is a clique and $\omega(G) = 3$: the coloring is optimal, as Proposition 4.11 guarantees.
- With the order $1,2,3,5,4,6$ the colors are $1,2,3$, then 1 to vertex 5 and 4 to vertex 4, whose neighbours $2,3,5$ use the colors $1,2,3$: 4 colors, one more than $\chi(G)$.

4.5 Sources

- M.R. Garey, D.S. Johnson, L. Stockmeyer, Some simplified NP-complete graph problems, *Theoretical Computer Science* 1(3):237–267, 1976.
- M.R. Garey, D.S. Johnson, The complexity of near-optimal graph coloring, *Journal of the ACM* 23(1):43–49, 1976.
- U. Feige, J. Kilian, Zero knowledge and the chromatic number, *Journal of Computer and System Sciences* 57(2):187–199, 1998.
- D. Zuckerman, Linear degree extractors and the inapproximability of max clique and chromatic number, *Theory of Computing* 3:103–128, 2007.
- R.L. Brooks, On colouring the nodes of a network, *Proceedings of the Cambridge Philosophical Society* 37(2):194–197, 1941.
- P.J. Heawood, Map-colour theorem, *Quarterly Journal of Pure and Applied Mathematics* 24, 1890.
- J. Hopcroft, R. Tarjan, Efficient planarity testing, *Journal of the ACM* 21(4):549–568, 1974.
- K. Appel, W. Haken, Every planar map is four colorable, *Illinois Journal of Mathematics* 21, 1977.

- N. Robertson, D. Sanders, P. Seymour, R. Thomas, The four-colour theorem, *Journal of Combinatorial Theory, Series B* 70(1):2–44, 1997.

4.6 Typical mistakes

Warning

- **Calling the bound $\Delta(G)+1$ a performance guarantee.** It compares the greedy coloring with $\Delta(G)+1$, not with $\chi(G)$; it becomes a constant ratio only on graphs of bounded maximum degree.
- **Reading the theorem of $\frac{4}{3}$ as an upper bound.** It says that ratios below $\frac{4}{3}$ are impossible, not that a ratio of $\frac{4}{3}$ is achievable: by the result of Zuckerman, no constant ratio is.
- **Using the four color theorem as an exact algorithm.** It colors every planar graph with at most 4 colors, but deciding whether 3 colors suffice remains NP-complete on planar graphs.
- **Sorting the intervals by right endpoint, or by length.** Proposition 4.11 uses the order of the left endpoints: the colors used before v all contain the point a_v.
- **Forgetting the odd cycle.** A breadth-first search that finds a conflict proves $\chi(G) \geq 3$ only through the odd cycle it closes; the cycle is the certificate.

4.7 In summary

Summary

- Unless $P = NP$, no polynomial algorithm colors every graph with fewer than $\frac{4}{3}\chi(G)$ colors: it would decide 3-colorability. By the result of Zuckerman, not even the ratio $n^{1-\varepsilon}$ is possible: the VCP is not approximable.
- Bipartite graphs and interval graphs are colored optimally in polynomial time; on graphs of bounded degree the greedy coloring has the ratio $\frac{\Delta+1}{2}$; on planar graphs the additive error is at most 2, or 1 with the four color theorem.

graphs	algorithm	result
all graphs	any polynomial	no constant ratio, unless $P = NP$
bipartite	breadth-first search	exact, $O(n+m)$
maximum degree $\leq \Delta$	greedy coloring	at most $\Delta+1$ colors
planar	PL	additive error ≤ 2 (or ≤ 1)
interval	greedy by left endpoint	exact, $\omega(G)$ colors

Dashboard of the reference instance.

Instance	Lower bound	Heuristic value	Optimal value
Petersen graph	3, odd cycle	GC 3	3

Clique problems

Chapter 5

Maximum clique: limits of approximation

Prerequisites

- From this module, the chapter «Principles of approximation algorithms»: performance ratio for a maximization problem, α-approximation algorithm, PTAS.
- From the module *Combinatorial Optimization Problems*, the chapter «Clique problems»: the definitions of the MCP and of the MSSP, the clique number $\omega(G)$, the complement graph and $\omega(G) = \alpha(\overline{G})$.
- From this module, the chapter «Vertex coloring: limits of approximation»: planar graphs and the bound $m \leq 3n - 6$.

Objectives

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

- build the strong product of two graphs and prove $\omega(G \boxtimes H) = \omega(G)\,\omega(H)$;
- prove that a constant-ratio algorithm for the MCP would give one for every constant ratio;
- state the results of Håstad and Zuckerman, and conclude that the MCP and the MSSP are not approximable;
- solve the MCP exactly on planar graphs and on graphs of bounded degree, and approximate the MSSP on graphs of bounded degree.

The problem of this chapter is the maximum clique problem (MCP): its definition and its assumptions are on its page.

5.1 Notation

- As on the problem page, $G = (V, E)$ is a simple graph with n vertices and m edges, $N(v)$ is the set of the neighbours of v and $\Delta(G)$ the maximum degree. The optimal value of the MCP is the clique number, $z(I) = \omega(G)$, and that of the MSSP is the stability number $\alpha(G)$.
- **Recall.** As in the chapter «Principles of approximation algorithms», $\mathscr{A}(I)$ is the number of vertices of the clique returned by the algorithm $\mathscr{A}$. The MCP is a maximization problem, hence
$$\mathscr{A}(I) \leq \omega(G) \qquad \text{and} \qquad \frac{\omega(G)}{\mathscr{A}(I)} \geq 1$$
for every instance I.
- A single vertex is a clique, so the ratio n is trivial. This chapter shows that a polynomial algorithm cannot do much better on all graphs, and then the classes of graphs on which it can.

5.2 The MCP is not approximable

5.2.1 The strong product

Definition 5.1: strong product

The **strong product** of two graphs $G = (V, E)$ and $H = (W, F)$ is the graph $G \boxtimes H$ with vertex set $V \times W$, in which two distinct vertices (u, x) and (v, y) are adjacent if and only if
$$\big(u = v \ \text{ or } \ \{u, v\} \in E\big) \quad \text{and} \quad \big(x = y \ \text{ or } \ \{x, y\} \in F\big).$$

- $G \boxtimes H$ has $|V|\,|W|$ vertices and is built in time polynomial in the size of G and H. In particular $G \boxtimes G$ has n^2 vertices.
- For a set $K \subseteq V \times W$, the **projection** is $p(K) = \{u \in V : (u, x) \in K \text{ for some } x\}$, and the **fibre** of $u \in p(K)$ is $K_u = \{x \in W : (u, x) \in K\}$.

Proposition 5.2

For every two graphs G and H:

1. if $K \subseteq V \times W$ is a clique of $G \boxtimes H$, then $p(K)$ is a clique of G, every fibre K_u is a clique of H, and $|K| \leq |p(K)| \cdot \max_u |K_u|$;
2. $\omega(G \boxtimes H) = \omega(G)\,\omega(H)$.

Proof. Part 1. Let $u \neq v$ be in $p(K)$, with $(u, x), (v, y) \in K$. These two vertices are adjacent and $u \neq v$, so $\{u, v\} \in E$: $p(K)$ is a clique. Let $x \neq y$ be in K_u: (u, x) and (u, y) are adjacent, so $\{x, y\} \in F$, and K_u is a clique. The fibres split K, hence
$$|K| \;=\; \sum_{u \in p(K)} |K_u| \;\leq\; |p(K)| \cdot \max_u |K_u|.$$

Part 2. If K_G and K_H are maximum cliques of G and H, every two distinct vertices of $K_G \times K_H$ satisfy the definition, so $\omega(G \boxtimes H) \geq \omega(G)\,\omega(H)$. Conversely, by Part 1 every clique K of $G \boxtimes H$ has $|K| \leq |p(K)| \cdot \max_u |K_u| \leq \omega(G)\,\omega(H)$. □

- The *lexicographic* product also satisfies $\omega(G[H]) = \omega(G)\,\omega(H)$; we use the strong product, whose definition is symmetric in the two graphs.

5.2.2 A constant ratio would give every constant ratio

- Part 1 of Proposition 5.2 is constructive: from a clique K of $G \boxtimes G$, either $p(K)$ or the largest fibre is a clique of G with at least $\sqrt{|K|}$ vertices, since the product of their sizes is at least $|K|$. Both are computed in polynomial time.

Theorem 5.3

If the MCP has a polynomial α-approximation algorithm for some constant $\alpha \geq 1$, then it has a polynomial β-approximation algorithm for every constant $\beta > 1$.

Proof. Let $\mathscr{A}$ be a polynomial α-approximation algorithm, and define the algorithm $\mathscr{B}$:

1. build $G \boxtimes G$ and run $\mathscr{A}$ on it, obtaining a clique K;
2. return the larger of $p(K)$ and of the largest fibre of K, a clique of G with at least $\sqrt{|K|}$ vertices.

By Proposition 5.2, $\omega(G \boxtimes G) = \omega(G)^2$, so $|K| \geq \frac{\omega(G)^2}{\alpha}$ and

$$\mathscr{B}(I) \;\geq\; \sqrt{|K|} \;\geq\; \frac{\omega(G)}{\sqrt{\alpha}}\,:$$

$\mathscr{B}$ is a polynomial $\sqrt{\alpha}$-approximation algorithm. Applying the construction t times gives the ratio $\alpha^{1/2^t}$, with $\mathscr{A}$ run on a graph with n^{2^t} vertices. Choose the constant t with $2^t \geq \frac{\ln \alpha}{\ln \beta}$: then $\alpha^{1/2^t} \leq \beta$, and n^{2^t} is a polynomial. □

- In other words, the MCP has a constant-ratio algorithm if and only if it has a PTAS: the strong product *amplifies* any constant ratio. The running time grows like a power n^{2^t}, which is polynomial only because t is a constant.

5.2.3 The results of Håstad and Zuckerman

- The negative side is known, and we state it without proof: it uses probabilistically checkable proofs.

Theorem 5.4: Håstad, 1999; Zuckerman, 2007

For every $\varepsilon > 0$, if $P \neq NP$, no polynomial algorithm $\mathscr{A}$ for the MCP satisfies, for every graph G with n vertices,

$$\frac{\omega(G)}{\mathscr{A}(I)} \leq n^{1-\varepsilon}.$$

- Håstad (1999) proved the theorem under the assumption that NP problems cannot be solved in randomized polynomial time ($NP \neq ZPP$); Zuckerman (2007) proved it under $P \neq NP$.

- As for the VCP, the theorem excludes every constant ratio: an α-approximation algorithm has ratio at most $\alpha < n^{1/2}$ on the graphs with $n > \alpha^2$ vertices, and the smaller graphs are solved by enumeration in constant time. Combined with Theorem 5.3: unless $P = NP$, the MCP has neither a constant ratio nor a PTAS.

The MCP is not approximable: unless $P = NP$, no polynomial algorithm has a constant performance guarantee, not even a guarantee of the form $n^{1-\varepsilon}$. Positive results exist only on special classes of graphs.

5.3 Special cases

5.3.1 Planar graphs

Proposition 5.5

Every planar graph has $\omega(G) \leq 4$, and on planar graphs the MCP is solved exactly in $O(n^4)$ time.

Proof. A subgraph of a planar graph is planar. The complete graph with 5 vertices has 10 edges, more than $3 \cdot 5 - 6 = 9$, so it is not planar, and a planar graph has no clique with 5 vertices. The subsets of at most 4 vertices are $O(n^4)$, and each one is checked in constant time. □

5.3.2 Graphs of bounded maximum degree

Proposition 5.6

Every graph has $\omega(G) \leq \Delta(G) + 1$, and the MCP is solved exactly in $O(n\, 2^{\Delta}\, \Delta^2)$ time, with $\Delta = \Delta(G)$.

Proof. A clique that contains the vertex v is contained in $\{v\} \cup N(v)$, which has at most $\Delta + 1$ vertices. For every vertex v, enumerate the $2^{\deg(v)} \leq 2^{\Delta}$ subsets $S \subseteq N(v)$ and keep the largest $\{v\} \cup S$ that is a clique; checking the pairs of S takes $O(\Delta^2)$ time. Every maximum clique is found from any of its vertices. □

- The time is polynomial for every fixed Δ, and even when $\Delta \leq \log_2 n$, since then $2^{\Delta} \leq n$.

5.3.3 The maximum stable set problem

- By $\omega(G) = \alpha(\overline{G})$ (chapter «Clique problems» of the module *Combinatorial Optimization Problems*), a stable set of G is a clique of $\overline{G}$, which is built in $O(n^2)$ time and has the same n.
- An algorithm with ratio $\rho(n)$ for one of the two problems, applied to the complement graph, has the same ratio $\rho(n)$ for the other one.

Corollary 5.7

For every $\varepsilon > 0$, if $P \neq NP$, no polynomial algorithm for the MSSP has ratio $\frac{\alpha(G)}{\mathscr{A}(I)} \leq n^{1-\varepsilon}$ on every graph. In particular, the MSSP has no constant ratio.

- The special cases do *not* transfer: the complement of a planar graph, or of a graph of bounded degree, is dense. On those classes the MSSP remains NP-hard, but it is approximable.

Proposition 5.8

On every graph, a **maximal** stable set S (no vertex can be added to it) has $|S| \geq \frac{n}{\Delta(G)+1}$. Hence the greedy algorithm that adds vertices while possible satisfies

$$\frac{\alpha(G)}{|S|} \;\leq\; \Delta(G) + 1.$$

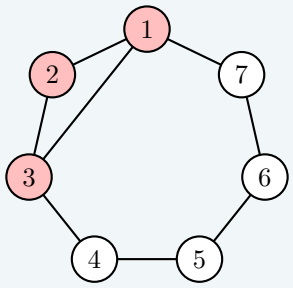
Proof. Every vertex outside S has a neighbour in S, otherwise it could be added. Hence $V = \bigcup_{v \in S} \big(\{v\} \cup N(v)\big)$, and $n \leq |S|\,(\Delta(G) + 1)$. Since $\alpha(G) \leq n$, the ratio is at most $\frac{n}{|S|} \leq \Delta(G) + 1$. □

- On graphs of maximum degree at most a fixed Δ this is a constant ratio. On planar graphs the MSSP even has a PTAS (Baker, 1994).

5.4 Example

Example 5.1: Strong product and enumeration on the MCP reference instance

The reference instance of the MCP is the cycle $1, 2, \ldots, 7, 1$ with the chord $\{1, 3\}$: $n = 7$, $m = 8$, $\Delta(G) = 3$ and $\omega(G) = 3$, with the unique maximum clique $\{1, 2, 3\}$ (in red).



- **Strong product.** $G \boxtimes G$ has $7^2 = 49$ vertices and 240 edges, and $\omega(G \boxtimes G) = 3^2 = 9$, attained by $\{1, 2, 3\} \times \{1, 2, 3\}$.
- Suppose an algorithm returns on $G \boxtimes G$ the clique $K = \{1, 2, 3\} \times \{4, 5\}$, with $|K| = 6$. Its projection $p(K) = \{1, 2, 3\}$ and its fibres $K_u = \{4, 5\}$ are cliques of G; the larger one has $3 \geq \sqrt{6}$ vertices.
- **Bounded degree.** The enumeration of Proposition 5.6 examines $\sum_v 2^{\deg(v)} = 2^3 + 2^2 + 2^3 + 4 \cdot 2^2 = 36$ subsets, and finds $\{1, 2, 3\}$ from the vertices 1, 2 and 3.
- **Planarity.** The graph is planar, as the figure shows, and $\omega(G) = 3 \leq 4$.

5.5 Sources

- M.R. Garey, D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, 1979 (the self-improvement of the MCP by graph products).
- J. Håstad, Clique is hard to approximate within $n^{1-\varepsilon}$, *Acta Mathematica* 182(1):105–142, 1999.
- D. Zuckerman, Linear degree extractors and the inapproximability of max clique and chromatic number, *Theory of Computing* 3:103–128, 2007.
- B.S. Baker, Approximation algorithms for NP-complete problems on planar graphs, *Journal of the ACM* 41(1):153–180, 1994.

5.6 Typical mistakes

Warning

- **Inverting the ratio.** The MCP is a maximization problem: the ratio is $\frac{\omega(G)}{\mathscr{A}(I)}$, at least 1.
- **Iterating the product a number of times that grows with n.** The graph after t products has n^{2^t} vertices: polynomial only for a constant t, so the amplification gives every constant ratio, not the ratio 1.
- **Transferring the special cases to the MSSP.** The complement of a planar graph is not planar: the MCP is easy on planar graphs, the MSSP is NP-hard there.
- **Confusing the bound $\Delta(G)+1$ of the MCP with that of the MSSP.** For the MCP, $\omega(G) \leq \Delta(G)+1$ bounds the optimum and makes enumeration fast; for the MSSP, $\Delta(G)+1$ bounds the ratio of a maximal stable set.

5.7 In summary

Summary

- The strong product squares the clique number, and a clique of the product gives back a clique of G of at least the square root of its size: a constant ratio for the MCP would give every constant ratio.
- By the results of Håstad and Zuckerman, unless $P = NP$ not even the ratio $n^{1-\varepsilon}$ is possible: the MCP is not approximable, and neither is the MSSP, through the complement graph.
- On planar graphs and on graphs of bounded degree the MCP is solved exactly by enumeration; on graphs of bounded degree a maximal stable set has ratio at most $\Delta(G)+1$ for the MSSP.

problem and graphs	algorithm	result
MCP, all graphs	any polynomial	no constant ratio, unless $P = NP$
MCP, planar	enumeration	exact, $\omega(G) \leq 4$
MCP, maximum degree $\leq \Delta$	enumeration	exact, $O(n\,2^{\Delta}\Delta^2)$
MSSP, all graphs	any polynomial	no constant ratio, unless $P = NP$
MSSP, maximum degree $\leq \Delta$	maximal stable set	ratio $\leq \Delta + 1$

Dashboard of the reference instance.

Instance	Upper bound	Value found	Optimal value
cycle with a chord	$\Delta(G)+1 = 4$	3, enumeration	3

Routing problems

Chapter 6

The TSP: the general case and the metric case

Prerequisites

- From this module, the chapter «Principles of approximation algorithms»: performance ratio, relative performance guarantee, tight guarantee and worst-case performance ratio.
- From the module *Combinatorial Optimization Problems*, the chapter «Routing problems»: the definition and the reference instance of the TSP.
- From the module *Network Optimization*, the chapter «Minimum spanning trees»: Kruskal's algorithm.
- From the module *Combinatorial Optimization Problems*, the chapter «Combinatorial optimization: definitions»: polynomial time, the classes P and NP, NP-completeness.

Objectives

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

- prove that no constant guarantee exists for the general TSP, unless $P = NP$;
- explain why the triangle inequality is what makes the TSP approximable;
- execute the double tree and the Christofides algorithms by hand on a metric instance;
- prove the guarantees 2 of the double tree and $\frac{3}{2}$ of the Christofides algorithm;
- build the families of instances that make these guarantees tight.

The problem of this chapter is the travelling salesman problem (TSP): its definition and its assumptions are on its page.

6.1 The general TSP is not approximable

- As on the problem page, an instance has a set $\mathscr{V} = \{1, 2, \ldots, n\}$ of cities, $n \geq 3$, and symmetric distances $d_{ij} = d_{ji} \in \mathbb{Z}_{\geq 1}$; a tour σ is a permutation of $\mathscr{V}$, of length $L(\sigma) = \sum_{k=1}^{n-1} d_{\sigma_k \sigma_{k+1}} + d_{\sigma_n \sigma_1}$, and the length is minimized.
- **Recall.** As in the chapter «Principles of approximation algorithms», $z(I)$ is the optimal value of the instance I and $\mathscr{A}(I)$ the value of the solution returned by the algorithm $\mathscr{A}$. The TSP is a minimization problem, hence

$$\mathscr{A}(I) \geq z(I) \qquad \text{and} \qquad \frac{\mathscr{A}(I)}{z(I)} \geq 1$$

  for every instance I, and the worst case performance ratio is $r(\mathscr{A}) = \sup_I \mathscr{A}(I)/z(I)$.
- For the general TSP no polynomial algorithm has a constant guarantee, unless $P = NP$. The proof follows the scheme of all such results: an approximation algorithm with a too good guarantee would decide in polynomial time an NP-complete problem.

Proposition 6.1

Let $\alpha \geq 1$ be a constant. If $P \neq NP$, there is no polynomial algorithm $\mathscr{A}$ for the TSP such that, for every TSP instance I,

$$\frac{\mathscr{A}(I)}{z(I)} \;\leq\; \alpha.$$

Proof. The **Hamiltonian cycle problem** asks whether a graph $G = (V, E)$ has a cycle through all its vertices; it is NP-complete. Suppose that $\mathscr{A}$ exists, and, given a graph G with $n \geq 3$ vertices, build the TSP instance I_G on the cities $\mathscr{V} = V$ with

$$d_{ij} = \begin{cases} 1 & \text{if } \{i, j\} \in E, \\ \lceil \alpha n \rceil + 1 & \text{otherwise.} \end{cases}$$

If G has a Hamiltonian cycle, that cycle is a tour of length n, so $z(I_G) = n$ and $\mathscr{A}(I_G) \leq \alpha n$. If G has no Hamiltonian cycle, every tour uses at least one pair $\{i, j\} \notin E$, so

$$\mathscr{A}(I_G) \;\geq\; z(I_G) \;\geq\; (n-1) + \lceil \alpha n \rceil + 1 \;>\; \alpha n.$$

Hence G is Hamiltonian if and only if $\mathscr{A}(I_G) \leq \alpha n$. Building I_G and running $\mathscr{A}$ takes polynomial time, so the Hamiltonian cycle problem would be in P, and $P = NP$. □

- The instance I_G violates the **triangle inequality** $d_{ij} \leq d_{ik} + d_{kj}$ as soon as G has a path i, k, j with $\{i, j\} \notin E$: $d_{ij} = \lceil \alpha n \rceil + 1 > 2 = d_{ik} + d_{kj}$. The proof needs such huge distances, and it says nothing on the instances that satisfy the inequality.
- The result and its proof are due to Sahni and Gonzalez (1976), and the NP-completeness of the Hamiltonian cycle problem to Karp (1972).

The general TSP is not approximable: unless $P = NP$, no polynomial algorithm has a constant performance guarantee. The *metric* TSP, the special case with the triangle inequality, is approximable, and the rest of the chapter is devoted to it.

6.2 A special case: the metric TSP

Definition 6.2: metric TSP

An instance of the TSP, with cities $\mathcal{V} = \{1, 2, \ldots, n\}$ and symmetric distances $d_{ij} = d_{ji} \in \mathbb{Z}_{\geq 1}$, is **metric** if the distances satisfy the triangle inequality

$$d_{ij} \leq d_{ik} + d_{kj} \qquad \text{for all distinct } i, j, k \in \mathcal{V}. \tag{6.1}$$

- The **metric TSP** is the TSP restricted to the metric instances. The instances I_G of Proposition 6.1 are not metric: the triangle inequality is exactly what makes the TSP approximable, and every guarantee of the next sections uses it.
- The reference instance has 5 cities, it is metric, and its unique optimal tour is $(1, 3, 5, 2, 4)$, with $z(I) = 18$. Its distances are:

d_{ij}	1	2	3	4	5
1	—	5	2	2	9
2	5	—	4	3	4
3	2	4	—	4	7
4	2	3	4	—	7
5	9	4	7	7	—

6.3 The double tree and the Christofides algorithms

- The two algorithms of this section build a tour from a **minimum spanning tree** (MST) of the complete graph on $\mathcal{V}$, with edge weights d_{ij}. We compute it with Kruskal's algorithm: the edges are examined by non-decreasing distance (ties by the pair of indices) and an edge is taken if it does not close a cycle with the edges already taken. The weight of a tree T is $d(T) = \sum_{\{i,j\} \in T} d_{ij}$.
- Both algorithms first build a closed walk that passes through every city, possibly more than once, and then turn it into a tour by **shortcutting**: the walk is followed and every city already visited is skipped, i.e., a city is kept only at its first occurrence.
- A closed walk that uses every edge of a (multi)graph exactly once is an **Eulerian circuit**. In the examples we build it with **Fleury's rule**: start from city 1 and, at each step, leave the current city along an unused edge towards the neighbour of smallest index, unless that edge is a *bridge* of the unused edges (its removal disconnects them) and another edge is available.

6.3.1 The double tree algorithm

The *Double Tree Algorithm* (denoted by DT) is a constructive heuristic for the metric TSP. It computes a minimum spanning tree T, doubles each of its edges, follows an Eulerian circuit of the doubled tree, and shortcuts it into a tour.

Algorithm

Algorithm: Double Tree

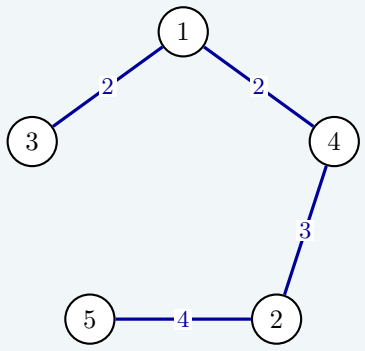
Input: metric TSP instance with n cities and distances $d[1..n][1..n]$
Output: a tour σ and its length $L(\sigma)$

```
1 Function DoubleTree(n, d)
2     T := a minimum spanning tree of the complete graph (Kruskal)
3     D := the multigraph with two copies of each edge of T
4     W := an Eulerian circuit of D from city 1 (Fleury)
5     σ := the cities of W in order of first occurrence          // shortcutting
6     return σ, L(σ)
```

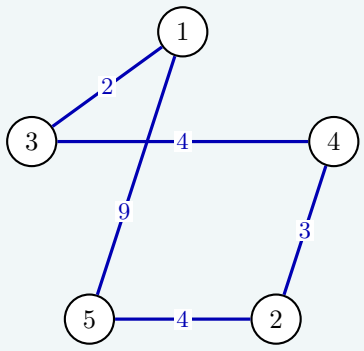
- In the doubled tree, Fleury's rule never goes back to the parent while a child has still to be visited, because the last copy of the edge towards the parent is a bridge. The Eulerian circuit is therefore a depth-first visit of T from city 1, with the children in increasing order, and the tour σ is the order in which the visit meets the cities for the first time (the *preorder*).
- With Kruskal's algorithm and a depth-first visit, DT runs in $O(n^2 \log n)$ time.

Example 6.1: DT on the TSP reference instance

Kruskal's algorithm examines $\{1,3\}$ (2, taken), $\{1,4\}$ (2, taken), $\{2,4\}$ (3, taken), $\{2,3\}$ (4, it closes the cycle $2,3,1,4$: rejected), $\{2,5\}$ (4, taken). The tree is $T = \{\{1,3\},\{1,4\},\{2,4\},\{2,5\}\}$, with $d(T) = 2+2+3+4 = 11$, and it is the unique MST of the instance.



MST T, $d(T) = 11$



tour of DT, $L(\sigma) = 22$

The Eulerian circuit of the doubled tree from city 1 is

$$W = (1,3,1,4,2,5,2,4,1), \qquad 2+2+2+3+4+4+3+2 = 22 = 2\,d(T).$$

Keeping each city at its first occurrence gives $\sigma = (1, 3, 4, 2, 5)$, of length

$$L(\sigma) = d_{13} + d_{34} + d_{42} + d_{25} + d_{51} = 2 + 4 + 3 + 4 + 9 = 22.$$

The shortcuts replace $3 \to 1 \to 4$ $(2 + 2)$ by $3 \to 4$ (4) and $5 \to 2 \to 4 \to 1$ $(4 + 3 + 2)$ by $5 \to 1$ (9): on this instance both triangle inequalities used hold with equality, and shortcutting saves nothing. Hence $DT(I) = 22$ and $\frac{DT(I)}{z(I)} = \frac{22}{18} = \frac{11}{9}$.

6.3.2 The Christofides algorithm

The *Christofides Algorithm* (denoted by CH) is a constructive heuristic for the metric TSP. It computes a minimum spanning tree T and, instead of doubling it, adds to T a minimum weight perfect matching M on the cities of odd degree in T; then it follows an Eulerian circuit of $T \cup M$ and shortcuts it into a tour.

Algorithm

Algorithm: Christofides

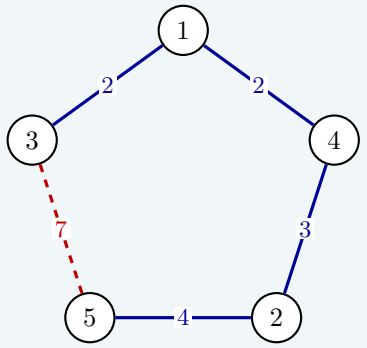
Input: metric TSP instance with n cities and distances $d[1..n][1..n]$
Output: a tour σ and its length $L(\sigma)$

Function *Christofides*(n, d)
  $T :=$ a minimum spanning tree of the complete graph (Kruskal)
  $O := \{ v \in \mathcal{V} : v$ has odd degree in $T \}$ `// |O| is even`
  $M :=$ a minimum weight perfect matching of the cities in O
  $D := T \cup M$ `// multigraph, all degrees even`
  $W :=$ an Eulerian circuit of D from city 1 (Fleury)
  $\sigma :=$ the cities of W in order of first occurrence `// shortcutting`
  return σ, $L(\sigma)$

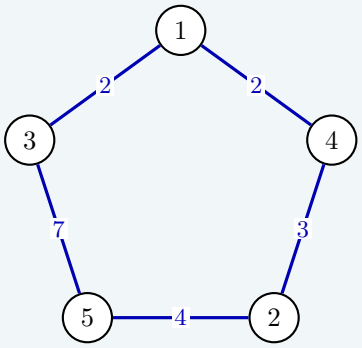
- **Why $|O|$ is even.** The degrees of a graph sum up to twice the number of edges, an even number, so the number of vertices of odd degree is even, and a perfect matching of O exists (the graph is complete).
- **Why $T \cup M$ has even degrees.** Each city of O gets exactly one more edge from M, and the other cities none. If an edge belongs to both T and M, D contains two copies of it.
- A minimum weight perfect matching can be computed in polynomial time, $O(n^3)$, with Edmonds' blossom algorithm; in the examples and exercises $|O| \leq 4$ and the matchings are enumerated ($|O| = 4$ gives 3 perfect matchings).

Example 6.2: CH on the TSP reference instance

The MST is the one of Example 6.1, with degrees $\deg(1) = 2$, $\deg(2) = 2$, $\deg(3) = 1$, $\deg(4) = 2$, $\deg(5) = 1$. Hence $O = \{3, 5\}$ and the only perfect matching is $M = \{\{3, 5\}\}$, with $d(M) = 7$.



T (solid) and M (dashed)



tour of CH, $L(\sigma) = 18$

$T \cup M$ is a cycle through all the cities. Fleury's rule from city 1 (neighbours 3 and 4) goes to 3 and gives $W = (1, 3, 5, 2, 4, 1)$, of length $d(T) + d(M) = 11 + 7 = 18$, in which no city is repeated: no shortcut is needed and $\sigma = (1, 3, 5, 2, 4)$, with $CH(I) = 18 = z(I)$. On the reference instance CH finds the optimal tour, while DT is 4 units above it.

6.4 Structural properties

- The guarantees of the next section compare the length of a heuristic tour with a lower bound on $z(I)$. We need three properties: shortcutting does not increase the length, an MST is shorter than any tour, and so is (twice) a minimum perfect matching of an even set of cities.

Observation 6.3

In a metric TSP instance, let W be a closed walk that visits every city at least once, and let σ be the tour obtained from W by shortcutting. Then $L(\sigma) \leq$ length of W.

Proof. Each edge $\{\sigma_k, \sigma_{k+1}\}$ of the tour (and the closing edge $\{\sigma_n, \sigma_1\}$) replaces a stretch $\sigma_k = u_0, u_1, \ldots, u_q = \sigma_{k+1}$ of W whose inner cities are skipped. Applying the triangle inequality (6.1) $q - 1$ times,

$$d_{u_0 u_q} \leq d_{u_0 u_1} + d_{u_1 u_q} \leq \ldots \leq \sum_{p=0}^{q-1} d_{u_p u_{p+1}}.$$

The stretches are disjoint parts of W, so summing over all the edges of σ gives $L(\sigma) \leq$ length of W. □

Observation 6.4

For every TSP instance (metric or not) and every minimum spanning tree T,

$$d(T) \leq z(I).$$

Proof. Removing any edge from an optimal tour leaves a path through all the cities, which is a spanning tree. Its weight is $z(I)$ minus the removed distance, at most $z(I)$ since distances are positive, and the MST weighs no more than any spanning tree. □

Observation 6.5

In a metric TSP instance, let $O \subseteq \mathcal{V}$ with $|O|$ even and $|O| \geq 2$, and let M be a minimum weight perfect matching of O. Then

$$d(M) \leq \frac{z(I)}{2}.$$

Proof. Let $\tilde{\sigma}$ be an optimal tour and shortcut it to the cities of O: we obtain a cycle $o_1, o_2, \ldots, o_{|O|}, o_1$ through the cities of O which, by Observation 6.3, has length at most $z(I)$. Since $|O|$ is even, its edges split into two perfect matchings of O, taking alternately one edge each:

$$M_1 = \{\{o_1, o_2\}, \{o_3, o_4\}, \ldots\}, \qquad M_2 = \{\{o_2, o_3\}, \{o_4, o_5\}, \ldots, \{o_{|O|}, o_1\}\}.$$

Hence $d(M_1) + d(M_2) \leq z(I)$, one of the two weighs at most $\frac{z(I)}{2}$, and $d(M) \leq \min\{d(M_1), d(M_2)\} \leq \frac{z(I)}{2}$. (For $|O| = 2$ the cycle is the edge $\{o_1, o_2\}$ travelled twice and $M_1 = M_2$.) □

- The last property we need is classical: a connected multigraph in which every vertex has even degree has an Eulerian circuit (a classical result of Euler), and Fleury's rule finds one. The doubled tree of DT and the multigraph $T \cup M$ of CH are connected (they contain T) and have even degrees.
- On the reference instance: $d(T) = 11 \leq 18$, and with $O = \{3, 5\}$ the optimal tour shortcut to O is $3, 5, 3$, of length $14 \leq 18$, so $d(M) = 7 \leq 9$.

6.5 Performance guarantees

6.5.1 The worst case performance ratio of DT

Proposition 6.6

For every metric TSP instance I,

$$\frac{DT(I)}{z(I)} \leq 2.$$

Proof. The Eulerian circuit W of the doubled tree uses every edge of T twice, so its length is $2\,d(T)$, and it visits every city because T is spanning. By Observation 6.3 and Observation 6.4,

$$DT(I) = L(\sigma) \leq 2\,d(T) \leq 2\,z(I).$$

□

Proposition 6.7

There are metric TSP instances I_n, $n \geq 5$ odd, such that

$$\frac{DT(I_n)}{z(I_n)} = \frac{2n}{n+2} \to 2 \qquad \text{as} \qquad n \to +\infty.$$

Proof. Let $n = 2k+1$ with $k \geq 2$, $a := n$ and $b := n+1$. City 1 is a centre and the cities $2, \ldots, n$ are leaves; the leaves are joined by the path

$$P = (3, 5, \ldots, 2k+1,\ 2, 4, \ldots, 2k),$$

whose consecutive cities never have consecutive indices (the differences are 2, and $2k-1 \geq 3$ for the edge $\{2k+1, 2\}$). The distances are

$$d_{ij} = \begin{cases} a & \text{if } i = 1 \text{ or } j = 1, \\ b & \text{if } \{i, j\} \text{ is an edge of } P, \\ 2a & \text{otherwise.} \end{cases}$$

All distances lie between a and $2a$, so $d_{ij} \leq 2a \leq d_{ik} + d_{kj}$: the instance is metric.

DT. The only distances equal to a are those from the centre, and a spanning tree has $n-1$ edges, each of weight at least a: the star centred at 1 is the unique MST, with $d(T) = (n-1)a$. The depth-first visit from city 1 meets the leaves in increasing order, $\sigma = (1, 2, 3, \ldots, n)$, and no pair of consecutive leaves is an edge of P:

$$DT(I_n) = a + (n-2)\,2a + a = 2(n-1)a = 2n(n-1).$$

Optimum. A tour has exactly two edges at city 1 (length a each) and $n-2$ edges between leaves (length at least b each), so $z(I_n) \geq 2a + (n-2)b$. The tour $(1, 3, 5, \ldots, 2k+1, 2, 4, \ldots, 2k)$ follows P and attains this value, hence

$$z(I_n) = 2a + (n-2)b = 2n + (n-2)(n+1) = (n+2)(n-1).$$

Therefore

$$\frac{DT(I_n)}{z(I_n)} = \frac{2n(n-1)}{(n+2)(n-1)} = \frac{2n}{n+2} = \frac{2}{1+\frac{2}{n}} \to 2.$$

□

From Proposition 6.6 and Proposition 6.7 it follows that DT is a 2-approximation algorithm for the metric TSP, and that its worst case performance ratio is

$$r(DT) = 2$$

- For $n = 5$: $a = 5$, $b = 6$, $P = (3, 5, 2, 4)$, $DT(I_5) = 40$, $z(I_5) = 28$ and the ratio is $\frac{10}{7}$; for $n = 7$ and $n = 9$ the ratios are $\frac{14}{9}$ and $\frac{18}{11}$.
- On the reference instance $DT(I) = 22 = 2\,d(T)$: the upper bound of the proof is attained, but $z(I) = 18 > d(T) = 11$, and the ratio is only $\frac{11}{9}$.
- The analysis of DT is due to Rosenkrantz, Stearns and Lewis (1977).

6.5.2 The worst case performance ratio of CH

Proposition 6.8

For every metric TSP instance I,

$$\frac{CH(I)}{z(I)} \leq \frac{3}{2}.$$

Proof. The Eulerian circuit W of $T \cup M$ uses every edge once, so its length is $d(T)+d(M)$, and it visits every city. By Observation 6.3, Observation 6.4 and Observation 6.5 (with the set O of odd-degree cities, $|O| \geq 2$ because a tree with at least two vertices has at least two leaves),

$$CH(I) = L(\sigma) \leq d(T) + d(M) \leq z(I) + \frac{z(I)}{2} = \frac{3}{2}\, z(I).$$

□

Proposition 6.9

There are metric TSP instances I_n, $n \geq 5$ odd, such that

$$\frac{CH(I_n)}{z(I_n)} \to \frac{3}{2} \qquad \text{as} \qquad n \to +\infty.$$

Proof. Let $n \geq 5$ be odd, $a := n$ and $b := n + 1$, so that $a < b < 2a$. Consider the graph H on the cities $1, \ldots, n$ with the edges $\{i, i+1\}$ of length a and $\{i, i+2\}$ of length b (a strip of triangles), and let d_{ij} be the length of a shortest path from i to j in H; shortest path distances always satisfy the triangle inequality. A path from i to $j > i$ advances by

1 (cost a) or by 2 (cost b) at each edge, and since $b < 2a$ it is best to advance by 2 as much as possible:

$$d_{ij} = \left\lfloor \frac{|i-j|}{2} \right\rfloor b + \big(|i-j| \bmod 2\big)\, a.$$

CH. The distances equal to a are exactly those between consecutive cities, so, as in the previous proof, the path $T = \{\{1,2\},\{2,3\},\dots,\{n-1,n\}\}$ is the unique MST, with $d(T) = (n-1)a$.

Its odd-degree cities are $O = \{1, n\}$, so $M = \{\{1, n\}\}$, with $d(M) = \frac{n-1}{2} b$. $T \cup M$ is the cycle $1, 2, \dots, n, 1$, Fleury's rule follows it from city 1 towards city 2, and no shortcut is needed:

$$CH(I_n) = (n-1)a + \frac{n-1}{2}\, b = (n-1)\,\frac{3n+1}{2}.$$

An upper bound on the optimum. The tour $(1, 3, 5, \dots, n, n-1, n-3, \dots, 2)$ uses $n-2$ edges of length b and the two edges $\{n, n-1\}$ and $\{2, 1\}$ of length a, so

$$z(I_n) \le (n-2)b + 2a = (n-2)(n+1) + 2n = (n+2)(n-1).$$

Together with Proposition 6.8,

$$\frac{3}{2} \ge \frac{CH(I_n)}{z(I_n)} \ge \frac{(n-1)(3n+1)}{2(n+2)(n-1)} = \frac{3n+1}{2n+4} = \frac{3+\frac{1}{n}}{2+\frac{4}{n}} \to \frac{3}{2}.$$

□

From Proposition 6.8 and Proposition 6.9 it follows that CH is a $\frac{3}{2}$-approximation algorithm for the metric TSP, and that its worst case performance ratio is

$$r(CH) = \frac{3}{2}$$

- For $n = 5, 7, 9$ the enumeration of all the tours shows that the upper bound on $z(I_n)$ is the optimum (28, 54, 88), and the ratios are $\frac{32}{28} = \frac{8}{7}$, $\frac{66}{54} = \frac{11}{9}$, $\frac{112}{88} = \frac{14}{11}$.
- The algorithm was found independently by Christofides (1976) and Serdyukov (1978). For more than forty years no polynomial algorithm with a better constant was known for the metric TSP; the first improvement, by an extremely small constant, is a randomized algorithm by Karlin, Klein and Oveis Gharan (2021).

6.6 Sources

- D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis II, An analysis of several heuristics for the traveling salesman problem, *SIAM Journal on Computing* 6(3):563–581, 1977.
- N. Christofides, Worst-case analysis of a new heuristic for the travelling salesman problem, Report 388, Carnegie-Mellon University, 1976.
- A.I. Serdyukov, On some extremal walks in graphs, *Upravlyaemye Sistemy* 17:76–79, 1978.
- S. Sahni, T. Gonzalez, P-complete approximation problems, *Journal of the ACM* 23(3):555–565, 1976.
- R.M. Karp, Reducibility among combinatorial problems, in *Complexity of Computer Computations*, Plenum Press, 85–103, 1972.

6.7 Typical mistakes

Warning

- **Applying DT or CH to a non-metric instance.** The guarantees 2 and $\frac{3}{2}$ use the triangle inequality, in the shortcutting step; without it no constant guarantee exists.
- **Shortcutting a city that is visited for the first time.** Following the Eulerian circuit, a city is skipped only if it has already been visited; the first occurrence is kept.
- **Matching the wrong cities in CH.** The perfect matching is computed on the cities of *odd* degree in the spanning tree, whose number is even, and has minimum weight.

6.8 In summary

Summary

- The general TSP is not approximable: an algorithm with a constant guarantee would decide the Hamiltonian cycle problem in polynomial time.
- The metric TSP is the special case that approximates. The guarantees of DT and CH are proved with lower bounds on $z(I)$: the minimum spanning tree and half of the minimum perfect matching; shortcutting, which uses the triangle inequality, turns the Eulerian circuit into a tour that is not longer.
- Both ratios are tight, with families of instances built in this chapter.

problem	algorithm	worst case ratio
general TSP	any polynomial	no constant, unless $P = NP$
metric TSP	DT	2 (tight)
metric TSP	CH	$\frac{3}{2}$ (tight)

Dashboard of the reference instance. The bounds on $z(I)$ found so far:

Instance	Lower bound	Heuristic values	Optimal value
metric TSP, 5 cities	11, minimum spanning tree	DT 22, CH 18	18

Scheduling problems

Chapter 7

Approximation for $P||C_{\max}$

Prerequisites

- From this module, the chapter «Principles of approximation algorithms»: performance ratio, relative performance guarantee, tight guarantee and worst-case performance ratio.
- From the module *Combinatorial Optimization Problems*, the chapter «Scheduling problems»: the definition and the reference instance of $P\,||\,C_{\max}$, and its elementary lower bound.
- From the module *Heuristic Algorithms*, the chapter «Constructive heuristics»: the algorithms $LIST$ and LPT as constructive rules.

Objectives

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

- execute $LIST$ and LPT by hand and draw the resulting schedule;
- prove the guarantees $2 - \frac{1}{m}$ of $LIST$ and $\frac{4}{3} - \frac{1}{3m}$ of LPT;
- build the families of instances that make these guarantees tight;
- explain why sorting the jobs by non-increasing processing time improves the guarantee.

The problem of this chapter is the scheduling problem on identical parallel machines ($P\,||\,C_{\max}$): its definition and its assumptions are on its page.

7.1 Notation

- As on the problem page, there are m identical machines and a set $\mathscr{J} = \{1, 2, \ldots, n\}$ of jobs with processing times t_j. A solution assigns each job $j \in \mathscr{J}$ to a machine $mach_j = h$; the set of jobs of machine h is J_h, its load is $L_h = \sum_{j \in J_h} t_j$, and the makespan is $C_{\max} = \max_h L_h$.
- **Recall.** As in the chapter «Principles of approximation algorithms», $z(I)$ is the optimal value of the instance I and $\mathscr{A}(I)$ the value of the solution returned by the algorithm $\mathscr{A}$. $P\,||\,C_{\max}$ is a minimization problem, hence

$$\mathscr{A}(I) \geq z(I) \qquad \text{and} \qquad \frac{\mathscr{A}(I)}{z(I)} \geq 1$$

  for every instance I, and the worst case performance ratio is $r(\mathscr{A}) = \sup_I \mathscr{A}(I)/z(I)$.
- We use the elementary lower bound of the chapter «Scheduling problems» of the module *Combinatorial Optimization Problems*:

$$z(I) \;\geq\; \max\left\{ \max_{j \in \mathscr{J}} t_j,\;\; \frac{\sum_{j \in \mathscr{J}} t_j}{m} \right\}. \tag{7.1}$$

- The reference instance has $m = 3$ and $t = (5, 5, 4, 4, 3, 3, 3)$; the bound gives $\max\{5, 27/3\} = 9$, and $z(I) = 9$ with $J_1 = \{1, 3\}$, $J_2 = \{2, 4\}$, $J_3 = \{5, 6, 7\}$.

7.2 The algorithms *LIST* and *LPT*

7.2.1 List scheduling

The *List Scheduling Algorithm* (denoted by $LIST$) is a constructive heuristic for $P\,||\,C_{\max}$. It examines the jobs in the order of a given list λ and assigns each job to a machine with the smallest current load (in case of ties, the machine with the smallest index).

Algorithm

Algorithm: List Scheduling

Input: $P \,||\, C_{\max}$ instance with n jobs, times $t[1..n]$, m machines; list $\lambda[1..n]$
Output: Job-machine assignment $mach[1..n]$ and makespan $C_{\max}$

```
1  Function ListScheduling(n, m, t, λ)
2      mach := new array[1..n]                          // job-machine assignment
3      for h := 1 to m do
4          L[h] ← 0                                     // loads of the machines
5      end
6      for k := 1 to n do
7          j := λ[k]                                    // k-th job of the list
8          s := a machine with minimum L[s] (smallest index)   // least loaded
               machine
9          mach[j] ← s                                  // assign job j to machine s
10         L[s] ← L[s] + t[j]
11     end
12     return mach, C_max = max_h L[h]
```

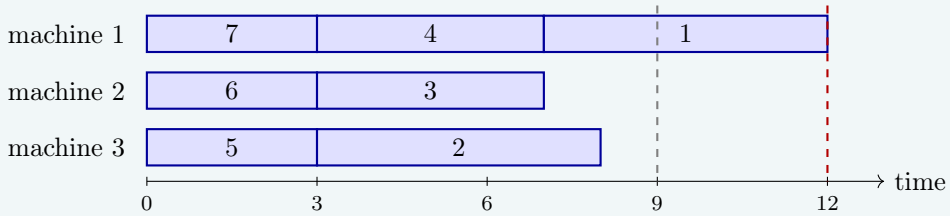
- *LIST* is the same algorithm of the chapter «Constructive heuristics» of the module *Heuristic Algorithms*, where it is introduced as a greedy rule. Here we are interested in its worst case: how bad can a list be?
- Each machine processes its jobs without idle times, so the job j assigned at iteration k **starts** at time S_j, equal to the load of its machine just before the assignment, and ends at $S_j + t_j$.

Example 7.1: *LIST* on the reference instance, shortest jobs first

Consider the list $\lambda = (7, 6, 5, 4, 3, 2, 1)$. The loads before each assignment and the chosen machine are:

job j	t_j	loads (L_1, L_2, L_3) before	machine
7	3	$(0, 0, 0)$	1
6	3	$(3, 0, 0)$	2
5	3	$(3, 3, 0)$	3
4	4	$(3, 3, 3)$	1
3	4	$(7, 3, 3)$	2
2	5	$(7, 7, 3)$	3
1	5	$(7, 7, 8)$	1

The final loads are $(12, 7, 8)$, with $J_1 = \{7, 4, 1\}$, $J_2 = \{6, 3\}$, $J_3 = \{5, 2\}$, and $LIST(I) = 12$:



The last job, job 1, starts at $S_1 = 7$, the smallest load at that moment: all the machines are busy until time 7 at least. This is the observation on which the guarantee of $LIST$ is built.

7.2.2 Longest processing time first

The *Longest Processing Time first Algorithm* (denoted by LPT) is a constructive heuristic for $P\,||\,C_{\max}$. It sorts the jobs by non-increasing processing time and applies $LIST$ to this list: each job, from the longest to the shortest, is assigned to a least loaded machine.

Algorithm

Algorithm: Longest Processing Time first

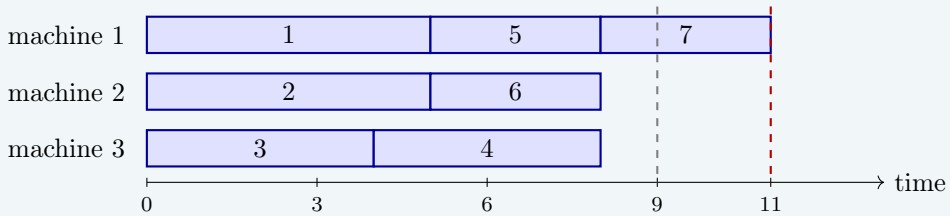
Input: $P\,||\,C_{\max}$ instance with n jobs, times $t[1..n]$, m machines
Output: Job-machine assignment $mach[1..n]$ and makespan $C_{\max}$

```
1 Function LPT(n, m, t)
2 |  λ := the jobs sorted by non-increasing t[j] (ties by index)   // longest jobs first
3 |  return ListScheduling(n, m, t, λ)
```

- In the proofs it is convenient to renumber the jobs in the order of λ, so that $t_1 \geq t_2 \geq \cdots \geq t_n$ and LPT assigns the jobs in the order $1, 2, \ldots, n$.

Example 7.2: LPT on the reference instance

The jobs are already sorted, $\lambda = (1, 2, 3, 4, 5, 6, 7)$. Jobs 1, 2, 3 go to the three empty machines, with loads $(5, 5, 4)$; job 4 goes to machine 3, the least loaded: $(5, 5, 8)$; job 5 goes to machine 1: $(8, 5, 8)$; job 6 to machine 2: $(8, 8, 8)$; job 7 finds three machines with load 8 and goes to machine 1: $(11, 8, 8)$. Hence $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$ and $LPT(I) = 11$:



With $z(I) = 9$, the ratio is $\frac{11}{9}$: sorting improves the makespan from 12 to 11, but the last job, short as it is, unbalances three perfectly balanced machines.

7.3 A property of list scheduling

- The guarantees of the next section compare the value of a heuristic solution with the lower bound (7.1) on $z(I)$. Besides the bound, they use one property of the schedules built by $LIST$: the job that ends last cannot start too late.

Observation 7.1

Let ℓ be a job that ends last in a schedule built by $LIST$ (with any list). Then its starting time satisfies

$$S_\ell \le \frac{1}{m} \sum_{j \in \mathscr{J} \setminus \{\ell\}} t_j.$$

Proof. When ℓ is assigned, S_ℓ is the smallest of the m loads, hence it is not larger than their average. The loads at that moment contain only jobs assigned before ℓ, so their sum is at most $\sum_{j \neq \ell} t_j$, and $S_\ell \le \frac{1}{m} \sum_{j \neq \ell} t_j$. □

7.4 Performance guarantees

7.4.1 The worst case performance ratio of LIST

Proposition 7.2

For every $P \,||\, C_{\max}$ instance I and every list λ,

$$\frac{LIST(I)}{z(I)} \le 2 - \frac{1}{m}.$$

Proof. Let ℓ be a job that ends last, so that $LIST(I) = S_\ell + t_\ell$. By Observation 7.1,

$$LIST(I) \le \frac{1}{m} \sum_{j \neq \ell} t_j + t_\ell = \underbrace{\frac{1}{m} \sum_{j \in \mathscr{J}} t_j}_{\le z(I)} + \left(1 - \frac{1}{m}\right) \underbrace{t_\ell}_{\le z(I)} \le \left(2 - \frac{1}{m}\right) z(I),$$

where both bounds come from (7.1). Dividing by $z(I)$ gives the claim. □

Proposition 7.3

For every $m \ge 2$ there is a $P \,||\, C_{\max}$ instance I_m with m machines and a list λ such that

$$\frac{LIST(I_m)}{z(I_m)} = 2 - \frac{1}{m}.$$

Proof. I_m has $n = m(m-1) + 1$ jobs: jobs $1, \dots, m(m-1)$ with $t_j = 1$ and job n with $t_n = m$; the list is $\lambda = (1, 2, \dots, n)$. We have $n > m$ for $m \ge 2$.

The unit jobs are assigned in rounds of m: in each round all the machines have the same load, the jobs go to machines $1, 2, \ldots, m$ in this order and every load grows by 1. After the $m-1$ rounds all the loads are $m-1$, and job n starts at $m-1$ on machine 1:

$$LIST(I_m) = (m-1) + m = 2m - 1.$$

Putting job n alone on machine 1 and the $m(m-1)$ unit jobs m per machine on machines $2, \ldots, m$ gives makespan m, and (7.1) gives $z(I_m) \geq t_n = m$. Hence $z(I_m) = m$ and

$$\frac{LIST(I_m)}{z(I_m)} = \frac{2m-1}{m} = 2 - \frac{1}{m}.$$

□

From Proposition 7.2 and Proposition 7.3 it follows that $LIST$ is a $\left(2 - \frac{1}{m}\right)$-approximation algorithm, whatever the list, and that the worst case performance ratio of $LIST$ on m machines is

$$r(LIST) = 2 - \frac{1}{m}$$

- For $m = 3$ the family has $t = (1, 1, 1, 1, 1, 1, 3)$, $LIST(I_3) = 5$ and $z(I_3) = 3$. The bad job is the long one, put last: LPT would put it first and find the optimum.
- On Example 7.1, $LIST(I) = 12 \leq \frac{5}{3} \cdot 9 = 15$.
- The bound and the family are due to Graham (1966).

7.4.2 The worst case performance ratio of LPT

Proposition 7.4

For every $P \,||\, C_{\max}$ instance I,

$$\frac{LPT(I)}{z(I)} \leq \frac{4}{3} - \frac{1}{3m}.$$

Proof. Number the jobs so that $t_1 \geq t_2 \geq \cdots \geq t_n$, the order in which LPT assigns them, and let ℓ be a job that ends last, so that $LPT(I) = S_\ell + t_\ell$. The jobs after ℓ play no role: let I' be the instance with the jobs $1, \ldots, \ell$ only. LPT assigns them on I' exactly as on I, so $LPT(I') = LPT(I)$, and $z(I') \leq z(I)$ since a schedule of I restricted to the jobs of I' is a schedule of I'. It is therefore enough to prove $LPT(I') \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I')$. In I' the job ℓ is the shortest one. We distinguish two cases.

Case 1: $t_\ell \leq \frac{z(I')}{3}$**.** As in the proof of Proposition 7.2, with Observation 7.1 applied to I',

$$LPT(I') \leq \frac{1}{m}\sum_{j=1}^{\ell} t_j + \left(1 - \frac{1}{m}\right) t_\ell \leq z(I') + \left(1 - \frac{1}{m}\right)\frac{z(I')}{3} = \left(\frac{4}{3} - \frac{1}{3m}\right) z(I').$$

Case 2: $t_\ell > \frac{z(I')}{3}$**.** Every job of I' is longer than $\frac{z(I')}{3}$, so in an optimal schedule of I'

no machine has three jobs, and $\ell \le 2m$. If $\ell \le m$, job ℓ starts on an empty machine and $LPT(I') = t_\ell \le z(I')$. Otherwise $\ell = m + r$ with $1 \le r \le m$; let $k := m - r + 1$, so that $1 \le k \le m$. We show that $z(I') \ge t_k + t_\ell \ge LPT(I')$.

Lower bound. Consider an optimal schedule of I'. If two of the jobs $1, \dots, k$ share a machine, its load is at least $2t_k \ge t_k + t_\ell$. Otherwise they lie on k different machines; the other $m - k = r - 1$ machines receive at most $2(r-1)$ jobs, while the jobs $k+1, \dots, \ell$ are $\ell - k = 2r - 1$. So one of them shares a machine with one of the jobs $1, \dots, k$, and that load is at least $t_k + t_\ell$. In both cases $z(I') \ge t_k + t_\ell$.

Upper bound. The first m jobs go to m different machines, since an empty machine has load 0 and every job has positive length. Before job ℓ, the m machines hold $\ell - 1 = m + r - 1$ jobs, each machine at least one, so at most $r - 1$ machines hold two jobs or more and at least $m - r + 1 = k$ machines hold exactly one job, one of the jobs $1, \dots, m$. These are k different jobs among $1, \dots, m$, so the largest index among them is at least k, and its length is at most t_k. Hence the smallest load before job ℓ is $S_\ell \le t_k$, and $LPT(I') = S_\ell + t_\ell \le t_k + t_\ell$.

Therefore $LPT(I') \le z(I')$ in Case 2, which is even better than the claim. □

Proposition 7.5

For every $m \ge 2$ there is a $P \,||\, C_{\max}$ instance I_m with m machines such that

$$\frac{LPT(I_m)}{z(I_m)} = \frac{4}{3} - \frac{1}{3m}.$$

Proof. I_m has $n = 2m + 1$ jobs, already sorted: two jobs of each length $2m - 1, 2m - 2, \dots, m + 1$, and three jobs of length m,

$$t = (2m-1,\ 2m-1,\ 2m-2,\ 2m-2,\ \dots,\ m+1,\ m+1,\ m,\ m,\ m).$$

For $j \le 2m$ the length is $t_j = 2m - \lceil j/2 \rceil$, and $t_{2m+1} = m$.

Optimum. The total length is $2\sum_{v=m+1}^{2m-1} v + 3m = 3m^2$, so (7.1) gives $z(I_m) \ge 3m$. The schedule that pairs a job of length v with a job of length $3m - v$, for $v = m+1, \dots, 2m-1$ (both lengths lie in $\{m+1, \dots, 2m-1\}$, and each length is used twice), fills $m-1$ machines with load $3m$, and puts the three jobs of length m on the last machine, again with load $3m$. Hence $z(I_m) = 3m$.

LPT. Jobs $1, \dots, m$ go to machines $1, \dots, m$. Then job $m + 1$ goes to a machine with the smallest single job, t_m, and its load becomes $t_m + t_{m+1}$. In general, we claim that job $m + q$, for $q = 1, \dots, m$, goes to a machine whose only job has length t_{m+1-q}, and the load becomes

$$t_{m+1-q} + t_{m+q} = 3m - 1.$$

The identity. With $j = m + 1 - q$, so that $2m + 1 - j = m + q$: if $j = 2i - 1$ is odd then $\lceil j/2 \rceil = i$ and $\lceil (2m + 1 - j)/2 \rceil = m - i + 1$, and if $j = 2i$ is even the two ceilings are again i and $m - i + 1$. In both cases $t_j + t_{2m+1-j} = (2m - i) + (m + i - 1) = 3m - 1$.

The claim. It follows by induction. A machine with two jobs has load $3m - 1 > 2m - 1 \ge$ any single job, so it is never the least loaded while machines with one job remain. Among

the machines with one job, the least loaded holds the shortest remaining job, of length t_{m+1-q} (with equal lengths the loads are the same).

After $2m$ jobs all the loads are $3m-1$, and the last job, of length m, gives

$$LPT(I_m) = 3m - 1 + m = 4m - 1, \qquad \frac{LPT(I_m)}{z(I_m)} = \frac{4m-1}{3m} = \frac{4}{3} - \frac{1}{3m}.$$

□

From Proposition 7.4 and Proposition 7.5 it follows that LPT is a $\left(\frac{4}{3} - \frac{1}{3m}\right)$-approximation algorithm, and that its worst case performance ratio on m machines is

$$r(LPT) = \frac{4}{3} - \frac{1}{3m}$$

Sorting the list lowers the guarantee of $LIST$ from $2 - \frac{1}{m}$ to $\frac{4}{3} - \frac{1}{3m}$.

- For $m = 3$ the family is $t = (5, 5, 4, 4, 3, 3, 3)$, the reference instance: the loads after six jobs are $(8, 8, 8) = 3m - 1$, the seventh job gives $LPT(I) = 11 = 4m - 1$, and $z(I) = 9 = 3m$ (Example 7.2). The reference instance is a worst case for LPT: $\frac{11}{9} = \frac{4}{3} - \frac{1}{9}$.
- On the tight instances the last job has $t_\ell = m = \frac{z(I_m)}{3}$: they lie exactly on the border between the two cases of the proof, and every inequality of Case 1 holds with equality.
- The bound and the family are due to Graham (1969).

7.5 Sources

- R.L. Graham, Bounds for certain multiprocessing anomalies, *Bell System Technical Journal* 45(9):1563–1581, 1966.
- R.L. Graham, Bounds on multiprocessing timing anomalies, *SIAM Journal on Applied Mathematics* 17(2):416–429, 1969.

7.6 Typical mistakes

Warning

- **Using the guarantee of LPT for an unsorted list.** $LIST$ has the guarantee $2 - \frac{1}{m}$ for every list; the better guarantee $\frac{4}{3} - \frac{1}{3m}$ needs the jobs sorted by non-increasing processing time.
- **Computing the ratio with the lower bound.** The bound (7.1) can be smaller than $z(I)$: the ratio $\mathscr{A}(I)/\text{LB}$ is only an upper bound on the performance ratio $\mathscr{A}(I)/z(I)$, and the two coincide only when the bound is tight.
- **Breaking ties with another rule.** $LIST$ assigns each job to a least loaded machine of smallest index; another tie rule can give a different schedule, and the solutions of the exercises follow the rule of the List Scheduling algorithm.

7.7 In summary

Summary

- The guarantees of $LIST$ and LPT are proved with the load bound (7.1) on $z(I)$ and with Observation 7.1: the job that ends last starts no later than the average load of the other jobs.
- Both ratios are tight, with Graham's families of instances; for $m = 3$ the reference instance is the tight instance of LPT.
- Sorting the list lowers the guarantee from $2 - \frac{1}{m}$ to $\frac{4}{3} - \frac{1}{3m}$.

problem	algorithm	worst case ratio
$P \,\|\|\, C_{\max}$	$LIST$	$2 - \frac{1}{m}$ (tight)
$P \,\|\|\, C_{\max}$	LPT	$\frac{4}{3} - \frac{1}{3m}$ (tight)

Dashboard of the reference instance. The bounds on $z(I)$ found so far:

Instance	Lower bound	Heuristic values	Optimal value
$P \,\|\|\, C_{\max}$, $m = 3$, $t = (5, 5, 4, 4, 3, 3, 3)$	9, load bound	$LIST$ 12 (list $7, 6, \ldots, 1$), LPT 11	9