

SAPIENZA
UNIVERSITÀ DI ROMA

Heuristic Algorithms

Lecture notes

Heuristic algorithms for combinatorial optimization problems: primal and dual bounds and the evaluation of heuristics, constructive heuristics, local search, metaheuristics and MIP-based heuristics, applied to knapsack, packing, routing, scheduling and coloring problems.

Fabio Furini

fabiofurini.github.io

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

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

Academic year 2026–2027

Contents

1 Heuristic algorithms: definitions **3**

1.1 Heuristic algorithms . . . 4
1.2 The problems and the assumptions . . . 4
1.3 Primal and dual bounds . . . 6
1.4 Classification of heuristic algorithms . . . 8
1.5 Solution representation and search space . . . 9
1.6 Neighbourhoods and local optima . . . 10
1.7 Intensification and diversification . . . 11
1.8 How to evaluate a heuristic algorithm . . . 12
1.9 Typical mistakes . . . 13
1.10 In summary . . . 13

2 Constructive heuristics **14**

2.1 Constructive heuristics . . . 15
2.2 The problems and the assumptions . . . 15
2.3 The greedy paradigm . . . 16
2.4 The 0/1 knapsack problem: which order? . . . 18
2.5 The bin packing problem: first sort, then pack . . . 20
2.6 Scheduling on parallel machines: longest jobs first . . . 24
2.7 The travelling salesman problem: append or insert? . . . 28
2.8 Vertex coloring: greedy and DSATUR . . . 35
2.9 Randomized construction . . . 39
2.10 Look-ahead and regret . . . 41
2.11 What constructive heuristics give, and what they do not . . . 44
2.12 Typical mistakes . . . 45
2.13 In summary . . . 46
2.14 `C++` implementation . . . 46

3 Local search **58**

3.1 Local search . . . 58
3.2 The problems, the assumptions and the starting solutions . . . 59
3.3 Local search: the scheme . . . 60
3.4 Neighbourhoods for the four problems . . . 61

3.5 First improvement and best improvement . . . 64
3.6 Evaluating a move incrementally . . . 67
3.7 TSP: 2-opt and Or-opt . . . 69
3.8 $P \,||\, C_{\max}$: move and swap . . . 74
3.9 Flat landscapes and auxiliary objectives . . . 77
3.10 KP: feasibility and penalties . . . 83
3.11 Variable neighbourhood descent . . . 87
3.12 Multi-start local search . . . 89
3.13 Typical mistakes . . . 92
3.14 In summary . . . 92
3.15 `C++` implementation . . . 93

4 Metaheuristics 103
4.1 Metaheuristics . . . 104
4.2 The problems and the assumptions . . . 104
4.3 How to escape from a local optimum . . . 108
4.4 Simulated annealing . . . 110
4.5 Tabu search . . . 116
4.6 Iterated local search . . . 119
4.7 Variable neighbourhood search . . . 123
4.8 GRASP . . . 125
4.9 Large neighbourhood search . . . 128
4.10 Population-based methods: genetic and memetic algorithms . . . 133
4.11 Comparison and conclusions . . . 135
4.12 Typical mistakes . . . 137
4.13 In summary . . . 138
4.14 `C++` implementation . . . 138
References . . . 158

5 MIP heuristics for the classical problems 160
5.1 MIP-based heuristics . . . 161
5.2 The problems and their MIP models . . . 161
5.3 Rounding . . . 164
5.4 Diving . . . 165
5.5 The feasibility pump . . . 167
5.6 RINS, local branching and proximity search . . . 168
5.7 Relax-and-fix and fix-and-optimize on lot sizing . . . 172
5.8 Heuristics inside the solver . . . 174
5.9 What the MIP-based heuristics give . . . 175
5.10 Typical mistakes . . . 176
5.11 In summary . . . 176
5.12 `Python` implementation . . . 177

Chapter 1

Heuristic algorithms: definitions

Prerequisites

- From the module *Combinatorial Optimization Problems*, the 0/1 knapsack problem, the bin packing problem, the travelling salesman problem and $P\,||\,C_{\max}$: instances, feasible solutions and optimal values of the four problems used as examples.
- From the module *Integer linear programming*, Chapter 2 «Primal and dual bounds»: the LP relaxation as a source of dual bounds. A short recap is given here for the 0/1 knapsack problem.

Objectives

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

- define a heuristic algorithm, and compute its gap with respect to a dual bound;
- tell primal bounds from dual bounds on the four reference instances;
- classify heuristic algorithms, and describe the solution representation and the search space;
- define neighbourhoods and local optima, and explain intensification and diversification;
- evaluate a heuristic algorithm by quality, running time and repeated runs.

1.1 Heuristic algorithms

Definition 1.1: Heuristic algorithm

Given an **optimization problem** $\mathscr{P}$, let $z(I)$ be the **optimal value** of $\mathscr{P}$ on an **instance** I. A **heuristic algorithm** for $\mathscr{P}$ is an algorithm that, for every instance I of $\mathscr{P}$, computes a feasible solution of I, without guaranteeing that it is optimal. We denote by $\mathscr{A}(I)$ the value of the solution computed by the heuristic algorithm $\mathscr{A}$ on the instance I.

- Since the solution computed by $\mathscr{A}$ is feasible, for every instance I

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

- The four problems used as examples — the 0/1 knapsack problem, the bin packing problem, the travelling salesman problem and the scheduling problem $P\,||\,C_{\max}$ — are all **NP-hard**: an exact algorithm may require a running time that grows exponentially with the size of the instance.

- **Approximation algorithms** are heuristic algorithms with a *proved* worst-case guarantee on the ratio between $\mathscr{A}(I)$ and $z(I)$. Most of the heuristic algorithms used in practice have no such guarantee: their quality is assessed on the instances at hand, as described in Section 1.8.

A heuristic algorithm trades the **quality** of the solution for the **running time**: the point of a heuristic is to compute, in a short time, a solution whose value is close to $z(I)$.

1.2 The problems and the assumptions

- The heuristic algorithms of these notes are illustrated on four classical combinatorial optimization problems: the 0/1 knapsack problem (KP), the bin packing problem (BPP), the travelling salesman problem (TSP) and $P\,||\,C_{\max}$. Their definitions and assumptions are on their pages; the table recalls the notation.

problem	data	solution	objective
KP	items $\mathscr{I}$, profits p_j, weights w_j, capacity c	subset of items	max total profit
BPP	items $\mathscr{I}$, weights w_j, capacity c	packing patterns	min number of bins
TSP	cities $\mathscr{V}$, distances d_{ij}	tour σ	min length $L(\sigma)$
$P\,\|\|\,C_{\max}$	jobs $\mathscr{J}$, times t_j, machines m	sets $J_1, \ldots, J_m$	min makespan

- We assume that every instance satisfies the assumptions of its problem: $w_j \leq c$ for all items and $\sum_j w_j > c$ for the KP and the BPP; $n \geq 3$ and symmetric distances for the TSP; $n > m$ for $P \,||\, C_{\max}$.

- The following four instances are used throughout the notes on heuristic algorithms. Their optimal values are known, so the quality of every heuristic solution can be measured exactly.

Example 1.1: KP reference instance

A knapsack of capacity $c = 9$ and 4 items with profits and weights

	$j=1$	$j=2$	$j=3$	$j=4$
p_j	10	7	6	4
w_j	5	4	3	3

The optimal solution is $\tilde{S} = \{1, 2\}$, of weight 9 and profit $z(I) = 17$.

Example 1.2: BPP reference instance

Bins of capacity $c = 7$ and 6 items with weights

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$
w_j	4	4	5	3	2	3

The total weight is $21 = 3c$, so at least 3 bins are needed, and the packing patterns $\{1, 4\}$, $\{2, 6\}$, $\{3, 5\}$ use exactly 3 bins, all of them full: $z(I) = 3$.

Example 1.3: TSP reference instance

5 cities with the symmetric distances

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	—

The instance is metric. Among its 12 distinct tours, the unique optimal one is $\sigma = (1, 3, 5, 2, 4)$, of length $z(I) = 18$.

Example 1.4: $P\,||\,C_{\max}$ reference instance

$m = 3$ identical machines and 7 jobs with processing times

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
t_j	5	5	4	4	3	3	3

The total processing time is 27, so the makespan is at least $\lceil 27/3 \rceil = 9$, and the sets $J_1 = \{1, 3\}$, $J_2 = \{2, 4\}$, $J_3 = \{5, 6, 7\}$ attain it: $z(I) = 9$.

1.3 Primal and dual bounds

- The value of *any* feasible solution bounds the optimal value from one side, and is called a **primal bound**. A heuristic algorithm is a method for computing primal bounds.

- A bound on the optimal value from the *other* side, obtained without computing a feasible solution, is called a **dual bound**. Dual bounds come from relaxations of the problem, such as the LP relaxation, or from simple properties of the problem.

- For a minimization problem a primal bound is an **upper bound** UB and a dual bound is a **lower bound** LB; for a maximization problem the roles are exchanged. In both cases the optimal value is sandwiched between the two bounds:

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

- The **gap** of a heuristic value $\mathscr{A}(I)$ is the relative gap $\frac{\mathrm{UB}-\mathrm{LB}}{\mathrm{LB}}$ defined in the module *Combinatorial Optimization Problems*, «Primal bounds, dual bounds and gap»: the difference of the two bounds divided by the **lower** one. With the heuristic value in place of the primal bound it reads

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

- The **relative error** of $\mathscr{A}$ on I is $\frac{\mathscr{A}(I)-z(I)}{z(I)}$ for a minimization problem and $\frac{z(I)-\mathscr{A}(I)}{z(I)}$ for a maximization problem. It cannot be computed when $z(I)$ is unknown, which is the typical situation; the gap, instead, uses only computed quantities.

Proposition 1.2

For every instance I, the relative error of $\mathscr{A}$ on I is at most $gap(I)$.

Proof. For a minimization problem, $\mathrm{LB}(I) \le z(I)$, hence $\frac{\mathscr{A}(I)}{z(I)} \le \frac{\mathscr{A}(I)}{\mathrm{LB}(I)}$ and

$$\frac{\mathscr{A}(I) - z(I)}{z(I)} = \frac{\mathscr{A}(I)}{z(I)} - 1 \le \frac{\mathscr{A}(I)}{\mathrm{LB}(I)} - 1 = gap(I).$$

For a maximization problem, $\mathscr{A}(I) \le z(I) \le \mathrm{UB}(I)$, hence

$$\frac{z(I) - \mathscr{A}(I)}{z(I)} \le \frac{z(I) - \mathscr{A}(I)}{\mathscr{A}(I)} \le \frac{\mathrm{UB}(I) - \mathscr{A}(I)}{\mathscr{A}(I)} = gap(I).$$

□

- The gap is therefore a **certificate**: if $gap(I) \le 5\%$, the solution is within 5% of the optimum, even if the optimum is unknown. If $gap(I) = 0$, the solution is optimal.

Example 1.5: Bounds and gaps on the reference instances

KP. The solution $\{1, 3\}$ has weight $8 \le 9$ and profit $\mathscr{A}(I) = 16$. The LP relaxation, with the items sorted by non-increasing efficiency $(1, 3, 2, 4)$, packs items 1 and 3 and a fraction $\frac{1}{4}$ of item 2, so $z(I) \le 10 + 6 + 7 \cdot \frac{1}{4} = 17.75$ and, the profits being integer, $\mathrm{UB}(I) = 17$. Hence $gap(I) = \frac{17-16}{16} = 6.25\%$, while the relative error is $\frac{17-16}{17} \approx 5.9\%$.

BPP. The packing patterns $\{1\}$, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{6\}$ are feasible and use $\mathscr{A}(I) = 5$ bins, while $\mathrm{LB}(I) = \lceil \frac{21}{7} \rceil = 3$. Hence $gap(I) = \frac{5-3}{3} \approx 66.7\%$.

TSP. The tour $(1, 3, 2, 4, 5)$ has length $\mathscr{A}(I) = 2 + 4 + 3 + 7 + 9 = 25$. A dual bound is given by Proposition 1.3 below: $\mathrm{LB}(I) = 17$. Hence $gap(I) = \frac{25-17}{17} \approx 47.1\%$, while the relative error is $\frac{25-18}{18} \approx 38.9\%$.

$P\,||\,C_{\max}$. The solution with $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$ has makespan $\mathscr{A}(I) = 11$, while $\mathrm{LB}(I) = \max\{5, \lceil 27/3 \rceil\} = 9$. Hence $gap(I) = \frac{11-9}{9} \approx 22.2\%$.

On the BPP and $P\,||\,C_{\max}$ the dual bound equals $z(I)$, so the gap equals the relative error. On the KP the dual bound also equals $z(I)$, but the gap divides by the smaller value $\mathscr{A}(I)$ and is slightly larger. On the TSP the dual bound is weaker, and the gap overestimates the relative error.

Proposition 1.3

For every TSP instance I, let a_i and b_i be the two smallest distances from city i to the other cities. Then

$$z(I) \ge \left\lceil \frac{1}{2} \sum_{i \in V} (a_i + b_i) \right\rceil.$$

Proof. In every tour each city is the endpoint of exactly two edges, of length at least a_i and b_i. Summing the lengths of the two edges incident to each city counts every edge of the tour twice, hence

$$2\,L(\sigma) \;=\; \sum_{i \in V} \big(\text{lengths of the two edges incident to } i\big) \;\geq\; \sum_{i \in V} (a_i + b_i)$$

for every tour σ, and in particular for an optimal one. The distances being integer, $z(I)$ is an integer, and the bound can be rounded up. □

- On the reference instance the two smallest distances are $(2, 2)$ for city 1, $(3, 4)$ for city 2, $(2, 4)$ for city 3, $(2, 3)$ for city 4 and $(4, 7)$ for city 5, so $\mathrm{LB}(I) = \left\lceil \frac{4+7+6+5+11}{2} \right\rceil = \lceil 16.5 \rceil = 17$.

A heuristic algorithm gives only one side of the gap. Without a dual bound on the other side, nothing can be said about the quality of the solution, however good it may look.

1.4 Classification of heuristic algorithms

- Heuristic algorithms are classified according to how they produce and use solutions, in the four classes below.

class	idea
constructive heuristics	build one solution step by step, extending a partial solution with one element at a time; a choice, once made, is never revised
local search	start from a feasible solution and repeatedly replace it by a better solution obtained by a small modification, until no small modification improves it
metaheuristics	general strategies that guide a local search beyond the point where it stops, exploring other regions of the solutions
MIP-based heuristics	use a mixed-integer programming formulation, its relaxation and a MIP solver to guide the search or to explore large sets of solutions

- The classes are not alternatives: a local search needs a starting solution, typically computed by a constructive heuristic; a metaheuristic contains one or more local searches; a MIP-based heuristic may start from, and improve, the solution of any of the others.

1.5 Solution representation and search space

- A heuristic algorithm manipulates solutions, which must be stored in a data structure: the **representation** of the solutions. The set of all the objects that the representation can describe is the **search space**.

- The representation is a design choice, and it determines which modifications of a solution are easy to perform. For the four problems we use the following representations.

problem	representation	size of the search space	feasible?
KP	binary vector $x \in \{0,1\}^n$	2^n	not always: the capacity can be exceeded
BPP	vector bin, with bin_j the bin of item j	at most n^n	not always: a bin can be overfull
TSP	permutation σ of the cities	$\frac{(n-1)!}{2}$ distinct tours	always
$P \mid\mid C_{\max}$	vector $mach$, with $mach_j$ the machine of job j	m^n	always

- When the representation can describe infeasible objects, a heuristic must either restrict itself to feasible ones or accept infeasible ones and penalize them.

- The search spaces grow very quickly: a TSP with $n = 20$ cities has $\frac{19!}{2} \approx 6 \cdot 10^{16}$ distinct tours. A heuristic explores only a tiny part of the search space, and the question is *which* part.

1.6 Neighbourhoods and local optima

- Let $\mathscr{F}(I)$ be the set of the feasible solutions of the instance I, and let $f(S)$ be the value of the solution $S \in \mathscr{F}(I)$, so that $z(I)$ is the minimum (or maximum) of f over $\mathscr{F}(I)$.

Definition 1.4: Neighbourhood

A **neighbourhood** is a function N that associates with every feasible solution $S \in \mathscr{F}(I)$ a set $N(S) \subseteq \mathscr{F}(I)$ of feasible solutions, the **neighbours** of S. A **move** is the transformation of S into one of its neighbours.

- A neighbourhood is usually defined by a family of small modifications of the representation: change one component of the vector x, move one job to another machine, replace two edges of a tour.

Definition 1.5: Local optimum

A feasible solution $S \in \mathscr{F}(I)$ is a **local optimum** with respect to the neighbourhood N if no neighbour of S is better than S:

$$\begin{cases} f(S) \leq f(S') & \text{for all } S' \in N(S) \quad \text{if } \mathscr{P} \text{ is a minimization problem,} \\ f(S) \geq f(S') & \text{for all } S' \in N(S) \quad \text{if } \mathscr{P} \text{ is a maximization problem.} \end{cases}$$

An optimal solution is also called a **global optimum**.

- A global optimum is a local optimum for every neighbourhood, but a local optimum need not be a global optimum. Moreover, **being a local optimum depends on the neighbourhood**: the same solution can be a local optimum for one neighbourhood and not for another.

Example 1.6: Two neighbourhoods for the KP

On the KP reference instance ($c = 9$), consider the solution $x = (1, 0, 1, 0)$, i.e., the items $\{1, 3\}$, of weight 8 and profit 16.

The **flip** neighbourhood changes one component of x, i.e., adds or removes one item:

neighbour	weight	profit	feasible
$(0, 0, 1, 0)$	3	6	yes
$(1, 1, 1, 0)$	12	—	no
$(1, 0, 0, 0)$	5	10	yes
$(1, 0, 1, 1)$	11	—	no

The feasible neighbours have profit 6 and 10, smaller than 16: x is a **local optimum** with respect to the flip neighbourhood.

The **swap** neighbourhood removes one selected item and adds one unselected item:

neighbour	weight	profit	feasible
$(0,1,1,0)$: out 1, in 2	7	13	yes
$(0,0,1,1)$: out 1, in 4	6	10	yes
$(1,1,0,0)$: out 3, in 2	9	17	yes
$(1,0,0,1)$: out 3, in 4	8	14	yes

The neighbour $(1,1,0,0)$ has profit $17 > 16$: x is **not** a local optimum with respect to the swap neighbourhood, and the move leads to the global optimum.

Definition 1.6: Exact neighbourhood

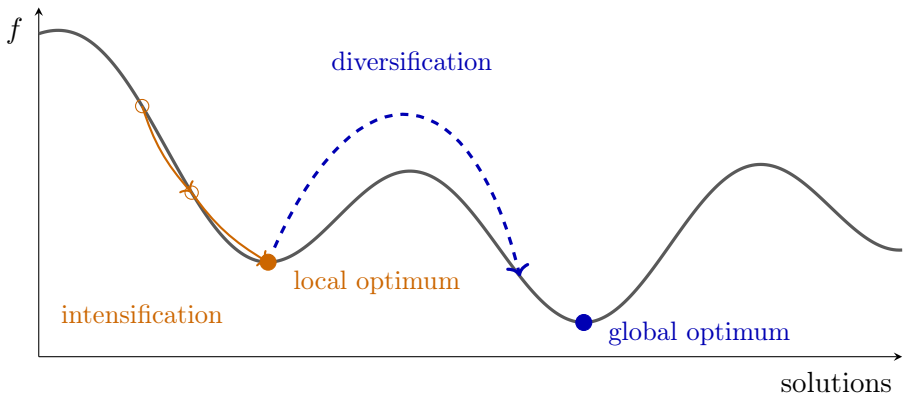
A neighbourhood N is **exact** if every local optimum with respect to N is a global optimum.

- Exact neighbourhoods are rare, and they exist for problems that can be solved in polynomial time: the neighbourhood of the simplex method, which moves between adjacent vertices of the polyhedron, is exact for linear programming; the neighbourhood that exchanges one edge in and one edge out is exact for the minimum spanning tree problem.

- For NP-hard problems, small neighbourhoods are not exact, and the neighbourhoods that are exact, such as the one containing all the feasible solutions, are too large to be explored. The design of a neighbourhood is a compromise between its **size**, which determines the time needed to explore it, and the **quality** of its local optima.

1.7 Intensification and diversification

- The values of the solutions, seen as a function on the search space, form the **landscape** explored by a heuristic. For a minimization problem the local optima are the bottoms of the valleys, and the global optimum is the bottom of the deepest valley.



- **Intensification** is the search for better solutions in the region where good solutions have already been found: moving downhill, toward the bottom of the current valley. Local search is pure intensification, and stops at the first local optimum it reaches.

- **Diversification** is the move toward regions of the search space that have not been explored yet: leaving the current valley, possibly through worse solutions, to reach another one.

- Too much intensification traps the search in the first valley; too much diversification makes it wander without ever exploring a valley to its bottom. Every metaheuristic is a way of balancing the two.

1.8 How to evaluate a heuristic algorithm

- A heuristic algorithm is evaluated by the **quality** of its solutions and by its **running time**, which are conflicting goals.

- **Quality** is measured by the gap with respect to the best available dual bound, or by the relative error when the optimal value is known, as on the reference instances of these notes.

- **Worst-case analysis**, as for approximation algorithms, gives guarantees valid for all the instances, but they are often far from the typical behaviour. **Experimental analysis** measures the gaps and the running times on a set of benchmark instances, which must be large and varied enough to avoid conclusions that hold only for a few instances.

- A single instance proves nothing: on the TSP reference instance the same rule, applied from different starting cities, gives tours of length between 18 and 25. Comparisons between heuristics must use the same instances and report the distribution of the results, not only the best one.

- For heuristics that use random choices, the results depend on the random numbers: each instance must be solved several times, with different **seeds** of the random number generator, reporting the average and the spread of the results.

A heuristic algorithm is good if, on the instances it is meant for, it finds solutions with a small gap in the time available, consistently.

1.9 Typical mistakes

Warning

- **Judging a heuristic from one run or one instance.** The same rule, from different starting cities, gives tours of length between 18 and 25 on the TSP reference instance.
- **Confusing the gap with the relative error.** The gap is measured against a dual bound, so it is never smaller than the relative error; in a minimization problem they coincide only when the dual bound equals $z(I)$.
- **Using the wrong denominator.** The gap always divides by the lower bound: for a maximization problem it is $\frac{\mathrm{UB}(I)-\mathscr{A}(I)}{\mathscr{A}(I)}$, for a minimization problem $\frac{\mathscr{A}(I)-\mathrm{LB}(I)}{\mathrm{LB}(I)}$.
- **Forgetting to round the dual bound.** With integer data the bound 17.75 of the KP is rounded down to 17, and the bound 16.5 of the TSP is rounded up to 17.
- **Trusting a good-looking solution.** Without a dual bound on the other side, nothing can be said about the quality of a solution.

1.10 In summary

Summary

- A heuristic algorithm returns a feasible solution without guaranteeing its optimality: its value is a *primal* bound.
- The gap compares the heuristic value with the best dual bound, and it is an upper bound on the relative error.
- Heuristic algorithms are constructive, local search, metaheuristic or MIP-based; a neighbourhood defines the local optima, and the search balances intensification and diversification.
- Quality is assessed on many instances and, for random methods, on many seeds.

Dashboard of the reference instances. Heuristic solutions of this chapter and their dual bounds:

problem	primal bound	dual bound	gap	$z(I)$
KP (max)	16	$\mathrm{UB}(I) = 17$	6.25%	17
BPP (min)	5	$\mathrm{LB}(I) = 3$	66.7%	3
TSP (min)	25	$\mathrm{LB}(I) = 17$	47.1%	18
$P \,\|\|\, C_{\max}$ (min)	11	$\mathrm{LB}(I) = 9$	22.2%	9

Chapter 2

Constructive heuristics

Prerequisites

- From this module, the chapter «Heuristic algorithms: definitions»: primal and dual bounds, the gap, and the four reference instances.
- From the module *Combinatorial Optimization Problems*, the problems: the KP, the BPP, $P\,||\,C_{\max}$, the TSP and the VCP.

Objectives

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

- apply the greedy algorithm to the KP with the three orders (profit, weight, efficiency);
- execute FF, FFD and BFD for the BPP, and LIST and LPT for $P\,||\,C_{\max}$;
- build a TSP tour by nearest neighbour and by the insertion heuristics;
- color a graph with the greedy coloring and with $DSATUR$;
- apply randomized construction, regret insertion and the pilot method;
- compare the results with the dual bounds, and explain when a constructive heuristic fails.

2.1 Constructive heuristics

Definition 2.1: Constructive heuristic

A **constructive heuristic** is a heuristic algorithm that builds one feasible solution step by step: it starts from the empty **partial solution** and extends it with one element at a time, until the solution is complete.

- The elements depend on the problem: the items selected for the knapsack, the assignments of items to bins and of jobs to machines, the edges of a tour.
- The quality of the result depends on how the element to add is chosen at every step (Section 2.3).

2.2 The problems and the assumptions

- The heuristics are illustrated on the four problems of the chapter «Heuristic algorithms: definitions»: the 0/1 knapsack problem (KP), the bin packing problem (BPP), the travelling salesman problem (TSP) and $P\,||\,C_{\max}$; the greedy coloring and $DSATUR$ are applied to the vertex coloring problem (VCP). Their definitions and assumptions are on their pages; we recall the notation.

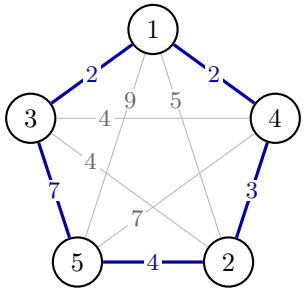
- As in the other notes, $\mathscr{A}(I)$ denotes the value of the solution computed by the heuristic $\mathscr{A}$ on the instance I, and $z(I)$ the optimal value. For $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ is the **load** of machine h, and the vector $mach$ stores the machine $mach_j$ of every job j; for the BPP, the vector bin stores the bin bin_j of every item j, and rc denotes a residual capacity.

- The examples use the four **reference instances** of the chapter «Heuristic algorithms: definitions», whose optimal values are known:

problem	reference instance	optimum
KP	$p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$	$\{1, 2\}$, $z(I) = 17$
BPP	$w = (4, 4, 5, 3, 2, 3)$, $c = 7$	$\{1, 4\}, \{2, 6\}, \{3, 5\}$, $z(I) = 3$
$P\,||\,C_{\max}$	$t = (5, 5, 4, 4, 3, 3, 3)$, $m = 3$	$\{1, 3\}, \{2, 4\}, \{5, 6, 7\}$, $z(I) = 9$
TSP	5 cities, distances below	$\sigma = (1, 3, 5, 2, 4)$, $z(I) = 18$

- In the figures, the five cities of the TSP reference instance are always drawn in the same positions, on a pentagon, in the order $(1, 3, 5, 2, 4)$ of the optimal tour: the optimal tour is then the boundary of the pentagon, and every tour that uses a diagonal is visibly different from it. The figure shows the distances of the instance and, in bold, its optimal tour.

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	—



2.3 The greedy paradigm

- A solution of each of the four problems is made of **elements**: the items selected for the knapsack, the assignments of items to bins and of jobs to machines, the edges of a tour. A constructive heuristic starts from the empty **partial solution** and adds one element at a time.

- At every step three questions must be answered: **which elements can be added** (the candidates), **which one is added** (the selection criterion), and **whether the result can still be completed to a feasible solution** (the feasibility test). A **greedy** heuristic answers the second question in the simplest way: it adds the candidate that looks best *now*, according to a criterion that measures the immediate benefit of the element.

Definition 2.2: Greedy algorithm

A **greedy algorithm** is a constructive heuristic that builds a solution by repeatedly selecting, among the candidate elements, the one that is best according to a *selection criterion*; the element is added to the partial solution if the feasibility test allows it, and is discarded otherwise.

In a greedy algorithm each decision is taken once and for all: *a choice, once made, is never revised.*

- The pseudo-code below is a template rather than an algorithm: the set of candidates, the criterion and the test are the three parts that every specific heuristic of these notes fills in.

Algorithm

Algorithm: Greedy template

Input: an instance I, the set C of the elements of the solutions
Output: a feasible solution S

```
1  Function Greedy(I)
2      S := ∅                                          // empty partial solution
3      while C ≠ ∅ and S is not complete do
4          e := the best element of C according to the selection criterion
5          C ← C \ {e}                                 // e is examined only once
6          if S ∪ {e} passes the feasibility test then
7              S ← S ∪ {e}                             // the choice is final
8          end
9      end
10     return S
```

- **The order decides everything.** The template examines the elements in the order induced by the selection criterion, and an element rejected or accepted early cannot be reconsidered later. Two greedy heuristics with the same candidates and the same feasibility test, but with different criteria, can produce very different solutions: changing the criterion is the main design decision.

- **Static and dynamic criteria.** A criterion is *static* if the value of an element does not depend on the partial solution: the elements can then be sorted once at the beginning, and the heuristic scans the sorted list (efficiency of the KP items, weight of the BPP items, processing time of the jobs). A criterion is *dynamic* if the value of an element changes as the solution grows, and it must be recomputed at every step (distance from the last city of a partial tour, residual capacity of the bins, load of the machines).

- **Myopia.** The criterion looks only at the immediate effect of a choice, not at its consequences on the choices still to be made. This is what makes greedy heuristics fast, and also what makes them fail: a choice that is good now can force bad choices later.

- **Running time.** With a static criterion the running time is typically dominated by the sorting, $O(n \log n)$, plus the cost of the feasibility tests; with a dynamic criterion it is the number of steps times the cost of re-evaluating the candidates, typically $O(n^2)$ or $O(n^3)$.

- **When greedy is exact.** For some problems a greedy algorithm always finds an optimal solution: Kruskal's algorithm for the minimum spanning tree problem examines the edges by non-decreasing cost and accepts an edge if it does not close a cycle; the fractional knapsack problem is solved optimally by taking the items by non-increasing efficiency. For the four NP-hard problems of these notes no greedy algorithm is exact, and the question becomes *which* criterion gives good solutions.

- The following table shows how the template is filled in for the four problems; the sections that follow study each of them.

problem	element	selection criterion	feasibility test
KP	an item	profit, weight or efficiency (static)	the item fits into the residual capacity
BPP	an item and a bin	item: input order or weight (static); bin: first, best, worst (dynamic)	the item fits into the bin
$P \mid\mid C_{\max}$	a job and a machine	job: input order or processing time (static); machine: least loaded (dynamic)	always feasible
TSP	a city and its position	nearest, farthest, cheapest (dynamic)	always feasible

2.4 The 0/1 knapsack problem: which order?

- For the KP the elements are the items, and the feasibility test is the capacity constraint: an item is selected if its weight does not exceed the residual capacity rc. Since the test is fixed, a greedy heuristic for the KP is completely determined by the **order** in which the items are examined. Three orders are natural:

by **profit** (GR_p):	non-increasing p_j	most valuable items first
by **weight** (GR_w):	non-decreasing w_j	as many items as possible
by **efficiency** (GR):	non-increasing $\frac{p_j}{w_j}$	best value per unit of weight

Ties are broken in favour of the item with the smallest index.

- The greedy by efficiency is denoted by GR; here we compare it with the other two orders.

The *greedy algorithm with order* ρ is a constructive heuristic for the KP. It examines the items in the order $\rho = (\rho_1, \ldots, \rho_n)$, a permutation of $\mathscr{I}$, starting from the empty knapsack, and selects each item if and only if it fits into the residual capacity.

Algorithm

Algorithm: Greedy with order

Input: KP instance with n items, profits $p[1..n]$, weights $w[1..n]$, capacity c; order $\rho[1..n]$

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

```
1  Function GreedyOrderKP(n, c, w, p, ρ)
2      for j := 1 to n do
3          x[j] ← 0                          // item j is initially not selected
4      end
5      rc := c                                // residual capacity
6      for k := 1 to n do
7          j := ρ[k]                          // k-th item in the order
8          if w[j] ≤ rc then
9              x[j] ← 1                       // select item j
10             rc ← rc − w[j]                 // update residual capacity
11         end
12     end
13     return x
```

- The three heuristics differ only in the order ρ, computed once by sorting the items in $O(n \log n)$ time. Each criterion captures one aspect of a good solution and ignores the other: the profit ignores how much capacity an item consumes, the weight ignores how much an item is worth, and the efficiency balances the two, but ignores how well the items fill the knapsack.

Example 2.1: Three orders on the KP reference instance

On the KP reference instance ($c = 9$) the three orders are

criterion	values	order ρ
profit	$p = (10, 7, 6, 4)$	$(1, 2, 3, 4)$
weight	$w = (5, 4, 3, 3)$	$(3, 4, 2, 1)$
efficiency	$\frac{p}{w} = \left(\frac{10}{5}, \frac{7}{4}, \frac{6}{3}, \frac{4}{3}\right) = (2,\ 1.75,\ 2,\ 1.33)$	$(1, 3, 2, 4)$

(items 3 and 4 have the same weight, items 1 and 3 the same efficiency, and the ties are broken by the index). The executions, with the residual capacity after each step, are:

	step 1	step 2	step 3	step 4
GR_p	item 1: in, $rc = 4$	item 2: in, $rc = 0$	item 3: out	item 4: out
GR_w	item 3: in, $rc = 6$	item 4: in, $rc = 3$	item 2: out	item 1: out
GR	item 1: in, $rc = 4$	item 3: in, $rc = 1$	item 2: out	item 4: out

Hence

$$GR_p(I) = p_1 + p_2 = 17, \qquad GR_w(I) = p_3 + p_4 = 10, \qquad GR(I) = p_1 + p_3 = 16.$$

The greedy by profit finds the optimal solution $\{1, 2\}$, which fills the knapsack exactly; the greedy by efficiency, after items 1 and 3, is left with $rc = 1$, too little for any other item; the greedy by weight takes the two light items, which are also the least valuable ones.

- On this instance the "naive" criterion wins. This is not a general rule: no order is the best on every instance, and **each of the three orders can be arbitrarily bad**. Let $c \geq 3$:
  - **profit**: one item with $p_1 = 2$, $w_1 = c$ and c items with $p_j = 1$, $w_j = 1$. GR_p takes item 1, which fills the knapsack: $GR_p(I) = 2$, while the c light items together give $z(I) = c$;
  - **weight**: two items with $p_1 = 1$, $w_1 = 1$ and $p_2 = c$, $w_2 = c$. GR_w takes item 1 and item 2 no longer fits: $GR_w(I) = 1$, while $z(I) = c$;
  - **efficiency**: two items with $p_1 = 2$, $w_1 = 1$ and $p_2 = c$, $w_2 = c$: here $GR(I) = 2$ and $z(I) = c$.

  In all three cases the ratio $\frac{z(I)}{\mathscr{A}(I)}$ grows with c.

- A remedy is to **combine** two solutions: the Ext-Greedy algorithm GR^e returns the better of the solution of GR and of the best single item, and this simple combination guarantees $\frac{z(I)}{GR^e(I)} \leq 2$. Running several greedy heuristics and keeping the best solution is a general and cheap way to protect against the weakness of each single criterion.

2.5 The bin packing problem: first sort, then pack

- Four classical greedy heuristics for the BPP process the items **in the input order** and differ in the rule used to choose the bin: Next Fit (NF) keeps a single open bin, while First Fit (FF), Best Fit (BF) and Worst Fit (WF) keep all the bins and choose, among those in which the item fits, the first opened, the fullest or the emptiest one. On the BPP reference instance ($c = 7$, $w = (4, 4, 5, 3, 2, 3)$, $z(I) = 3$) they give

$$NF(I) = 5, \qquad FF(I) = 4, \qquad BF(I) = 3, \qquad WF(I) = 4.$$

- In those heuristics the *bin* is chosen by a criterion, but the *item* is not: the items arrive in the order of the input, which may be a bad one. When the whole instance is known in advance — the *offline* setting — nothing forces us to accept that order. The idea of this section is to choose the order of the items too: **first sort, then pack**.

- **Why the large items first.** The difficult items of a BPP instance are the large ones: two items larger than $\frac{c}{2}$ never share a bin, and a large item fits only into a bin that is still almost empty. Small items, on the contrary, can fill the gaps left in any bin. If the small items come first, they are spread over the bins and occupy the space that the large items will need later; if the large items come first, each of them gets its own bin, and the small items are then used to fill the gaps.

2.5.1 First Fit Decreasing

The *First Fit Decreasing Algorithm* (denoted by FFD) is a constructive heuristic for the BPP. It first sorts the items by non-increasing weight, and then packs them in this order by First Fit: each item is packed into the first bin, in the order in which the bins were opened, in which it fits; if it fits into no bin, a new bin is opened.

Algorithm

Algorithm: First Fit Decreasing

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 FirstFitDecreasing(n, w, c)
2      ρ := the items sorted by non-increasing weight          // first sort
3      bin := new array[1..n]                                  // item-bin assignment
4      rc := new array[1..n]                                   // residual capacities
5      nb := 0                                                 // number of bins
6      for k := 1 to n do
7          j := ρ[k]                                           // then pack: k-th largest item
8          s := −1                                             // selected bin
9          for b := 1 to nb do
10             if w[j] ≤ rc[b] then
11                 s ← b
12                 break
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[j] ← s                                          // assign item
21         rc[s] ← rc[s] − w[j]
22     end
23     return bin, nb
```

- The packing loop is exactly the loop of FF: the only difference is the line that computes the order ρ. Sorting costs $O(n \log n)$, and FFD runs in the same time as FF, $O(n^2)$ with the simple scan of the bins.

Example 2.2: FF and FFD on the BPP reference instance

FF, in the input order: items 1, 2 and 3 (weights $4, 4, 5$) open one bin each, because no two of them fit together; item 4 (weight 3) goes into bin 1 ($rc[1] = 0$); item 5 (weight 2) goes into bin 2 ($rc[2] = 1$); item 6 (weight 3) fits into no bin, whose residual capacities are

$0, 1, 2$, and opens bin 4. Hence $FF(I) = 4$.

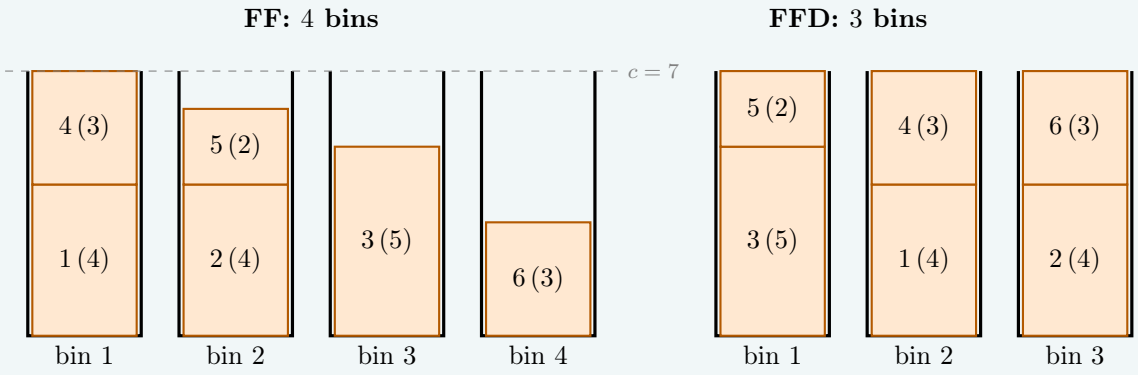
FFD. The weights sorted by non-increasing value are $5, 4, 4, 3, 3, 2$, i.e., $\rho = (3, 1, 2, 4, 6, 5)$ (ties by index). Then:

1. Item 3 (weight 5): no bin is open, FFD opens bin 1; $rc[1] = 7 - 5 = 2$.
2. Item 1 (weight 4): $4 > rc[1] = 2$, FFD opens bin 2; $rc[2] = 7 - 4 = 3$.
3. Item 2 (weight 4): $4 > rc[1] = 2$, $4 > rc[2] = 3$, FFD opens bin 3; $rc[3] = 7 - 4 = 3$.
4. Item 4 (weight 3): $3 > rc[1] = 2$, $3 \leq rc[2] = 3$: bin 2; $rc[2] = 0$.
5. Item 6 (weight 3): $3 > rc[1] = 2$, $3 > rc[2] = 0$, $3 \leq rc[3] = 3$: bin 3; $rc[3] = 0$.
6. Item 5 (weight 2): $2 \leq rc[1] = 2$: bin 1; $rc[1] = 0$.

Hence $bin = (2, 3, 1, 2, 1, 3)$, the packing patterns are

$$P_1 = \{3, 5\}, \qquad P_2 = \{1, 4\}, \qquad P_3 = \{2, 6\},$$

and $FFD(I) = 3 = \lceil \frac{21}{7} \rceil$: the solution is optimal, and all three bins are full.



In FF the small item 5 arrives before the large item 6 and takes the space that item 6 would need in bin 2; in FFD item 5 comes last and fills the gap of bin 1.

2.5.2 Best Fit Decreasing

The *Best Fit Decreasing Algorithm* (denoted by BFD) is a constructive heuristic for the BPP. It first sorts the items by non-increasing weight, and then packs them in this order by Best Fit: each item is packed into the bin with the smallest residual capacity among those in which it fits (in case of ties, the bin opened first); if it fits into no bin, a new bin is opened.

Algorithm

Algorithm: Best Fit Decreasing

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 BestFitDecreasing(n, w, c)
2      ρ := the items sorted by non-increasing weight          // first sort
3      bin := new array[1..n]                                  // item-bin assignment
4      rc := new array[1..n]                                   // residual capacities
5      nb := 0                                                 // number of bins
6      for k := 1 to n do
7          j := ρ[k]                                           // then pack: k-th largest item
8          s := −1                                             // selected bin
9          for b := 1 to nb do
10             if w[j] ≤ rc[b] and (s = −1 or rc[b] < rc[s]) then
11                 s ← b                                       // fullest bin found so far
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[j] ← s                                          // assign item
20         rc[s] ← rc[s] − w[j]
21     end
22     return bin, nb
```

- Again, the packing loop is that of BF, preceded by the sorting.

Example 2.3: BFD on the BPP reference instance

The order is $\rho = (3, 1, 2, 4, 6, 5)$, as for FFD. Items 3, 1 and 2 open bins 1, 2, 3, with residual capacities $2, 3, 3$. Item 4 (weight 3) fits into bins 2 and 3, both with residual capacity 3: the tie is broken in favour of bin 2, and $rc[2] = 0$. Item 6 (weight 3) fits only into bin 3, and $rc[3] = 0$. Item 5 (weight 2) fits only into bin 1, and $rc[1] = 0$. BFD returns the same packing patterns as FFD, $P_1 = \{3, 5\}$, $P_2 = \{1, 4\}$, $P_3 = \{2, 6\}$, and $BFD(I) = 3$.

2.5.3 What sorting gives, and what it does not

- **A better guarantee.** Sorting the items improves the worst case. FF and BF have worst-case performance ratio $\frac{17}{10}$. For FFD it is known that

$$FFD(I) \leq \frac{11}{9}\, z(I) + \frac{6}{9} \qquad \text{for every BPP instance } I,$$

and that the bound is tight. A bound of the form $\frac{11}{9}z(I) + 4$ was first proved for both FFD and BFD by 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 (1974) 299–325); the tight additive constant is due to G. Dósa (*The tight bound of First Fit Decreasing bin-packing algorithm is* $FFD(I) \leq 11/9\,OPT(I) + 6/9$, ESCAPE 2007, LNCS 4614, pp. 1–11).

- **Not a better solution on every instance.** A worst-case guarantee compares the algorithm with the optimum, not with another algorithm: on some instances the input order is luckier than the sorted one.

Example 2.4: Sorting can hurt

Let $c = 7$ and $w = (2, 2, 3, 2, 2, 3)$, with total weight 14, so that $z(I) \geq \lceil \frac{14}{7} \rceil = 2$.

FF packs items $1, 2, 3$ into bin 1 $(2+2+3=7)$ and items $4, 5, 6$ into bin 2 $(2+2+3=7)$: $FF(I) = 2$, which is optimal.

FFD examines the items in the order $\rho = (3, 6, 1, 2, 4, 5)$. Items 3 and 6 go into bin 1 ($rc[1] = 1$); items 1, 2, 4 do not fit into bin 1 and go into bin 2 ($rc[2] = 1$); item 5 fits into neither bin and opens bin 3: $FFD(I) = 3$.

The two items of weight 3, packed together by FFD, waste one unit of capacity in bin 1; the input order happened to pair each of them with two items of weight 2, filling both bins exactly.

2.6 Scheduling on parallel machines: longest jobs first

- For $P \,||\, C_{\max}$ the elements are the assignments of jobs to machines, and every assignment is feasible. A natural greedy rule assigns each job to the machine that is currently **least loaded**, so that the loads grow as evenly as possible. As for the BPP, the rule chooses the machine, and the order of the jobs is a second, independent, choice.

2.6.1 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 ρ 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 $\rho[1..n]$
Output: Job-machine assignment $mach[1..n]$ and makespan $C_{\max}$

```
1  Function ListScheduling(n, m, t, ρ)
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 := ρ[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 scheduling counterpart of an *online* rule: it does not need to know the future jobs, and it is used, for example, when jobs arrive one at a time. With a linear scan of the machines it runs in $O(nm)$ time, and in $O(n \log m)$ time with a heap of the loads.

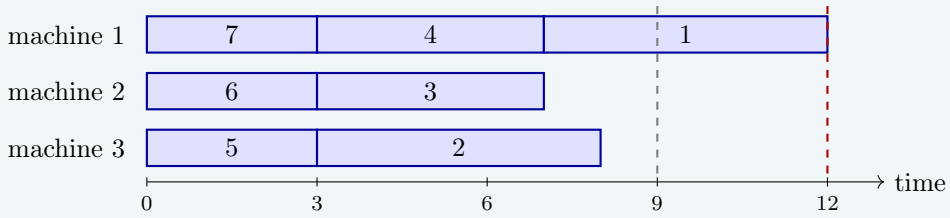
- On the reference instance, $t = (5, 5, 4, 4, 3, 3, 3)$, the jobs are listed by non-increasing processing time, which is a favourable order. To see the effect of the order, the example examines the same jobs in the reverse list.

Example 2.5: $LIST$ on the reference instance, shortest jobs first

Consider the list $\rho = (7, 6, 5, 4, 3, 2, 1)$, i.e., the jobs by non-decreasing processing time. 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 of the list is one of the longest: when it arrives, the least loaded machine already has load 7, and the makespan jumps to 12, three units above the lower bound 9 (dashed grey line).

Proposition 2.3

For every $P \,||\, C_{\max}$ instance I and every list ρ,

$$LIST(I) \;\leq\; \left(2 - \frac{1}{m}\right) z(I).$$

Proof. Proof omitted. The result is due to

R.L. Graham, *Bounds for certain multiprocessing anomalies*, Bell System Technical Journal 45 (1966) 1563–1581. □

$LIST$ is a $\left(2 - \frac{1}{m}\right)$-approximation algorithm for $P \,||\, C_{\max}$, whatever the order of the list.

- On the example, $LIST(I) = 12 \leq \frac{5}{3} \cdot 9 = 15$.

2.6.2 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 $C_{\max}$

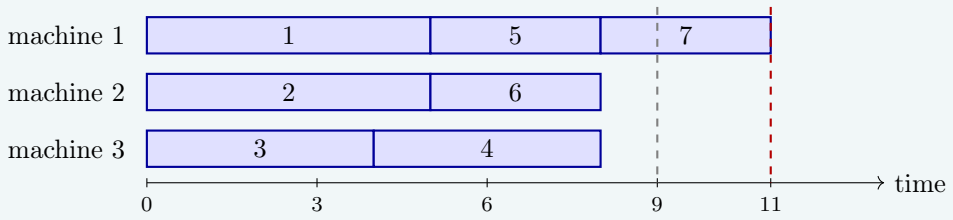
Function $LPT(n, m, t)$
  $\rho :=$ the jobs sorted by non-increasing $t[j]$ (ties by index) `// longest jobs first`
  return $ListScheduling(n, m, t, \rho)$

- **Why the longest jobs first.** The makespan is decided by the job that finishes last. If the jobs are short at the end of the list, the last assignments can only add a little to the least loaded machine, and they act as "sand" that levels the loads. The parallel with FFD is exact: the large elements first, the small ones to fill the gaps.

- LPT runs in $O(n \log n + n \log m)$ time.

Example 2.6: LPT on the reference instance

The jobs are already sorted, $\rho = (1, 2, 3, 4, 5, 6, 7)$. 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 $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, $mach = (1, 2, 3, 3, 1, 2, 1)$, and $LPT(I) = 11$:



The elementary lower bound of the chapter «Heuristic algorithms: definitions» gives $z(I) \geq \max\left\{\max_j t_j, \left\lceil \frac{\sum_j t_j}{m} \right\rceil\right\} = \max\{5, 9\} = 9$, and $z(I) = 9$. Sorting improves the makespan from 12 to 11, but LPT is not optimal: after six jobs the loads are perfectly balanced, $(8, 8, 8)$, and the seventh job unbalances them by 3.

Proposition 2.4

For every $P \,||\, C_{\max}$ instance I,

$$LPT(I) \;\leq\; \left(\frac{4}{3} - \frac{1}{3m}\right) z(I).$$

Proof. Proof omitted. The result is due to

R.L. Graham, *Bounds on multiprocessing timing anomalies*, SIAM Journal on Applied Mathematics 17 (1969) 416–429. □

LPT is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm for $P \,||\, C_{\max}$: sorting the list lowers the guarantee of $LIST$ from $2 - \frac{1}{m}$ to $\frac{4}{3} - \frac{1}{3m}$.

- For $m = 3$ the bound is $\frac{4}{3} - \frac{1}{9} = \frac{11}{9}$, and on the reference instance

$$\frac{LPT(I)}{z(I)} = \frac{11}{9}.$$

The reference instance attains Graham's bound exactly: it belongs to the family of instances, given by Graham, with $2m + 1$ jobs of processing times $2m - 1, 2m - 1, 2m - 2, 2m - 2, \ldots, m + 1, m + 1, m, m, m$, on which LPT gives $4m - 1$ and the optimum is $3m$. For $m = 3$ these are the times $(5, 5, 4, 4, 3, 3, 3)$. The bound of Proposition 2.4 therefore cannot be improved.

2.7 The travelling salesman problem: append or insert?

- For the TSP every permutation of the cities is a feasible tour, so the feasibility test is trivial, and the whole design is in the selection criterion. Two families of constructive heuristics are natural, and they differ in **where** the new city is placed:
  - **append**: the partial solution is a path, and the new city is added at its end;
  - **insert**: the partial solution is a closed subtour, and the new city is inserted between two consecutive cities, wherever it costs least.

2.7.1 Nearest neighbour

The *Nearest Neighbour Algorithm* (denoted by NN) is a constructive heuristic for the TSP. Starting from a given city s, it repeatedly moves from the last visited city to the *nearest* city not yet visited (in case of ties, the one with the smallest index); when all the cities have been visited, it goes back to s.

Algorithm

Algorithm: Nearest Neighbour

Input: TSP instance with n cities, distances $d[1..n][1..n]$; starting city s
Output: Tour $\sigma[1..n]$

1 **Function** *NearestNeighbour*(n, d, s)
2 $\quad \sigma[1] \leftarrow s$ `// the path starts at` s
3 $\quad visited := \{s\}$
4 $\quad$ **for** $len := 2$ **to** n **do**
5 $\quad\quad last := \sigma[len-1]$ `// last city of the path`
6 $\quad\quad k :=$ the city $j \notin visited$ with minimum $d[last][j]$ (smallest index)
7 $\quad\quad \sigma[len] \leftarrow k$ `// append` k `at the end`
8 $\quad\quad visited \leftarrow visited \cup \{k\}$
9 $\quad$ **end**
10 $\quad$ **return** σ `// the tour closes with the edge` $(\sigma[n], s)$

- The criterion is dynamic: the distances that matter are those from the last city, which changes at every step. NN runs in $O(n^2)$ time.

- **The weakness of** NN is visible in its last steps: the cities that are left are those that were never the nearest to anything, often far from each other and from s, and the closing edge back to s is not chosen at all, it is forced.

Example 2.7: NN on the TSP reference instance

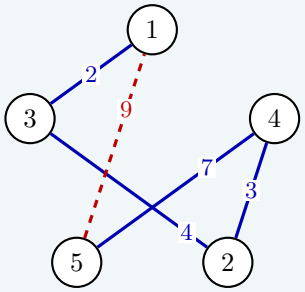
Starting from city 1:

1. from city 1 the distances to $2, 3, 4, 5$ are $5, 2, 2, 9$: cities 3 and 4 are the nearest, and the tie is broken in favour of city 3;
2. from city 3 the distances to $2, 4, 5$ are $4, 4, 7$: city 2 (tie with 4);
3. from city 2 the distances to $4, 5$ are $3, 4$: city 4;
4. from city 4 only city 5 is left, at distance 7;
5. the tour closes with the edge $(5, 1)$, of length 9.

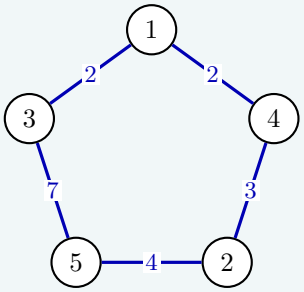
Hence $\sigma = (1, 3, 2, 4, 5)$ and

$$NN(I) = d_{13} + d_{32} + d_{24} + d_{45} + d_{51} = 2 + 4 + 3 + 7 + 9 = 25,$$

39% more than $z(I) = 18$; this is the tour used in the chapter «Heuristic algorithms: definitions» to illustrate the gap. The four edges chosen by the criterion have length $2+4+3+7 = 16$, and the forced closing edge alone costs 9, the largest distance of the instance.



NN from city 1: $L = 25$
(dashed: the closing edge)



optimal tour: $L = 18$

2.7.2 Multi-start nearest neighbour

- The result of NN depends on the starting city, which is an arbitrary choice: the tour is a cycle and has no natural first city. Since NN is fast, it can simply be run from every city.

The *multi-start nearest neighbour algorithm* runs NN once from every city $s \in \mathcal{V}$ and returns the shortest of the n tours obtained (in case of ties, the first one found).

Algorithm

Algorithm: Multi-start Nearest Neighbour

Input: TSP instance with n cities, distances $d[1..n][1..n]$
Output: Tour $\sigma^{best}[1..n]$

```
1 Function MultiStartNN(n, d)
2     L^best := +∞
3     for s := 1 to n do
4         σ := NearestNeighbour(n, d, s)
5         if L(σ) < L^best then
6             σ^best ← σ, L^best ← L(σ)          // new best tour
7         end
8     end
9     return σ^best
```

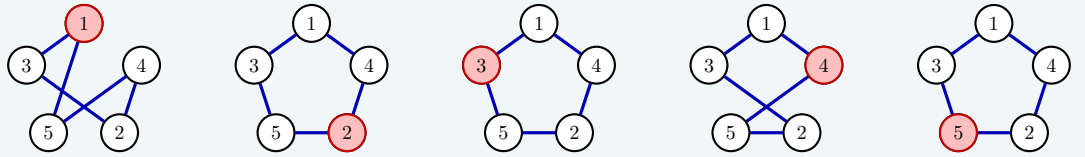
- The running time is n times that of NN, i.e., $O(n^3)$. Multi-start is the simplest way to turn an arbitrary choice into a search: each start gives a different solution, and the best one is kept.

Example 2.8: Multi-start NN on the TSP reference instance

The five runs of NN give

start s	1	2	3	4	5
tour	$(1,3,2,4,5)$	$(2,4,1,3,5)$	$(3,1,4,2,5)$	$(4,1,3,2,5)$	$(5,2,4,1,3)$
L	25	18	18	19	18

The tours from cities 2, 3 and 5 are the optimal tour $(1,3,5,2,4)$, traversed from different cities and in different directions; the multi-start returns the tour from city 2, of length $18 = z(I)$.



start 1: $L = 25$ start 2: $L = 18$ start 3: $L = 18$ start 4: $L = 19$ start 5: $L = 18$

2.7.3 Insertion heuristics

- NN can only add a city at the end of a path. An **insertion heuristic** keeps a closed *subtour* T through some of the cities, and inserts the new city wherever it is cheapest. Inserting city k between two consecutive cities i and j of T replaces the edge (i,j) with the edges (i,k) and (k,j), and increases the length by the **insertion cost**

$$\Delta(i,k,j) \;=\; d_{ik} + d_{kj} - d_{ij} \qquad \text{for every edge } (i,j) \text{ of the subtour } T.$$

On metric instances $\Delta(i,k,j) \geq 0$, by the triangle inequality.

- Each step makes two decisions: **which** city is inserted (the selection rule) and **where** (always the edge of minimum insertion cost). The three classical heuristics differ only in the selection rule. Let $d(q,T) = \min_{i \in T} d_{qi}$ be the distance of city q from the subtour.

The *insertion heuristics* for the TSP start from the subtour $T = (s)$ and, until all the cities are in T, select a city $k \notin T$ and insert it into the edge of T of minimum insertion cost:

nearest insertion (NI): k minimizes $d(q,T)$, the distance from the subtour

farthest insertion (FI): k maximizes $d(q,T)$

cheapest insertion (CI): k minimizes $\min\limits_{(i,j) \in T} \Delta(i,q,j)$, the cost of its insertion

Ties are broken in favour of the smallest index, for the city and for the edge.

Algorithm

Algorithm: Insertion (NI, FI, CI)

Input: TSP instance with n cities, distances $d[1..n][1..n]$; starting city s; $rule \in \{NI, FI, CI\}$

Output: Tour $\sigma[1..n]$

```
1 Function Insertion(n, d, s, rule)
2     σ := (s)                                          // subtour with one city
3     for len := 1 to n − 1 do
4         k := the city q ∉ σ selected by rule           // which city
5         a := the position a ∈ {1, ..., len} minimizing Δ(σ[a], k, σ[a + 1]), with
              σ[len + 1] = σ[1]                         // where
6         insert k between σ[a] and σ[a + 1]
7     end
8     return σ
```

- **The subtour with one city.** The subtour (s) is a closed tour with a single "edge" (s, s) of length 0, so inserting k costs $d_{sk} + d_{ks} - 0 = 2d_{sk}$, and the subtour (s, k) goes from s to k and back. No special case is needed.

- **Append or insert.** Every tour built by NN could also be built by an insertion heuristic, but not conversely: inserting anywhere keeps the subtour closed at every step, so the length of the closing edge is paid, and controlled, from the beginning.

- **The intuition behind FI.** Taking the *farthest* city looks absurd for a greedy algorithm, but it builds first the outline of the tour, through the cities that are far apart, and then inserts the remaining cities, which are "on the way" and cost little.

- With the distances $d(q, T)$ and the minimum insertion costs updated incrementally after each insertion, NI and FI run in $O(n^2)$ time; a simple implementation of CI runs in $O(n^3)$.

Example 2.9: NI and CI on the TSP reference instance

Starting from $T = (1)$. At each step we list, for every city $q \notin T$, its distance $d(q, T)$ and its insertion costs on the edges of T, in the order of T.

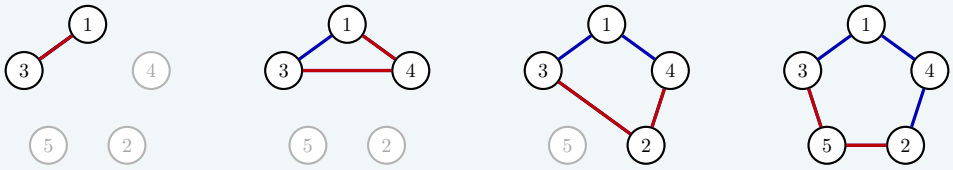
subtour T	$d(q, T)$	insertion costs	inserted
(1)	$2{:}\,5,\ 3{:}\,2,\ 4{:}\,2,\ 5{:}\,9$	$2{:}\,10,\ 3{:}\,4,\ 4{:}\,4,\ 5{:}\,18$	3 on $(1,1)$, cost 4
$(1,3)$	$2{:}\,4,\ 4{:}\,2,\ 5{:}\,7$	$2{:}\,(7,7),\ 4{:}\,(4,4),\ 5{:}\,(14,14)$	4 on $(1,3)$, cost 4
$(1,4,3)$	$2{:}\,3,\ 5{:}\,7$	$2{:}\,(6,3,7),\ 5{:}\,(14,10,14)$	2 on $(4,3)$, cost 3
$(1,4,2,3)$	$5{:}\,4$	$5{:}\,(14,8,7,14)$	5 on $(2,3)$, cost 7

NI selects the city with the smallest $d(q, T)$, CI the city with the smallest insertion cost:

on this instance the two rules select the same cities at every step (city 3 at the first step by the tie with city 4). Both return $\sigma = (1, 4, 2, 5, 3)$, and

$$NI(I) = CI(I) = 4 + 4 + 3 + 7 = 18 = z(I),$$

the optimal tour $(1, 3, 5, 2, 4)$ traversed in the opposite direction. The subtours grow as follows (new edges in red):



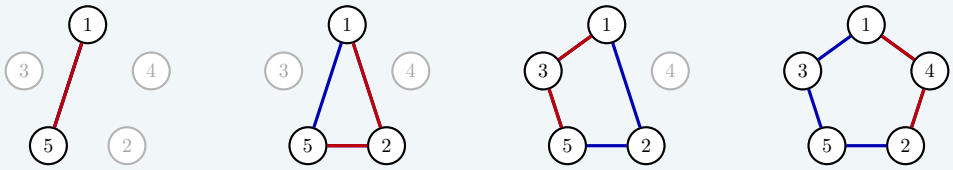
$L = 4$ $L = 8$ $L = 11$ $L = 18$

Example 2.10: FI on the TSP reference instance

Starting from $T = (1)$:

subtour T	$d(q, T)$	insertion costs of the selected city	inserted
(1)	2: 5, 3: 2, 4: 2, 5: 9	5: 18	5 on $(1, 1)$, cost 18
$(1, 5)$	2: 4, 3: 2, 4: 2	2: $(0, 0)$	2 on $(1, 5)$, cost 0
$(1, 2, 5)$	3: 2, 4: 2	3: $(1, 7, 0)$	3 on $(5, 1)$, cost 0
$(1, 2, 5, 3)$	4: 2	4: $(0, 6, 4, 4)$	4 on $(1, 2)$, cost 0

At the first step FI takes city 5, the farthest from city 1, and the subtour $(1, 5)$ already has length $2d_{15} = 18$. Cities 2, 3 and 4 lie "on the way" and are inserted at zero cost, because $d_{12} + d_{25} = d_{15}$, $d_{53} + d_{31} = d_{51}$ and $d_{14} + d_{42} = d_{12}$. Hence $\sigma = (1, 4, 2, 5, 3)$ and $FI(I) = 18 = z(I)$: the same tour as NI and CI, built in a completely different order.



$L = 18$ $L = 18$ $L = 18$ $L = 18$

- **What is known in the worst case.** On metric instances, NI and CI never return a tour longer than twice the optimal one, while the ratio of NN can grow like $\log n$ and only a logarithmic bound is known for FI (D.J. Rosenkrantz, R.E. Stearns, P.M. Lewis, *An analysis of several heuristics for the traveling salesman problem*, SIAM Journal on Computing 6 (1977) 563–581). In practice, on Euclidean instances, FI is usually the best of the insertion heuristics, and NN tours are typically about 25% longer than the optimum.

2.7.4 A comparison on a larger instance

- On the reference instance the three insertion heuristics, from every starting city, all find the optimal tour, and the differences between the rules do not show. To compare them we use a second TSP instance, with 6 cities in the plane and distances equal to the Euclidean distances rounded to the nearest integer. The instance is metric (checked on all the triples of cities).

Example 2.11: A TSP instance with 6 cities

The cities have coordinates

$$1:(2,1), \quad 2:(5,5), \quad 3:(4,4), \quad 4:(0,5), \quad 5:(7,0), \quad 6:(4,1),$$

and distances

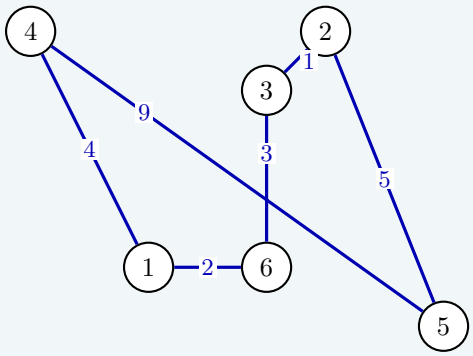
d_{ij}	1	2	3	4	5	6
1	—	5	4	4	5	2
2	5	—	1	5	5	4
3	4	1	—	4	5	3
4	4	5	4	—	9	6
5	5	5	5	9	—	3
6	2	4	3	6	3	—

The instance has $\frac{5!}{2} = 60$ distinct tours; enumerating them, the unique optimal tour is $(1,4,3,2,5,6)$, with $z(I) = 19$, and only 7 tours have length smaller than 24. The heuristics of this section give:

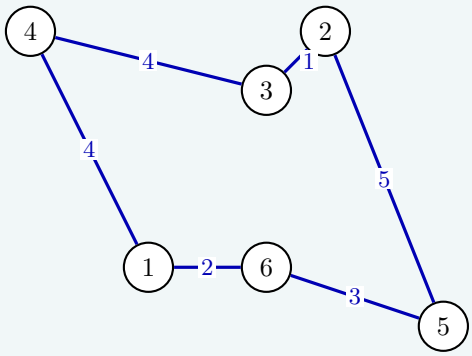
	starting city					
	1	2	3	4	5	6
NN	25	24	25	24	24	24

from city 1	L
NI	23
FI	19
CI	20

The best tour of the multi-start NN has length 24, 26% above the optimum, while FI finds the optimal tour $(1,6,5,2,3,4)$ (the optimal tour in the opposite direction):



multi-start NN (from city 2): $L = 24$



FI from city 1: $L = 19 = z(I)$

All six NN tours contain the edge $(4,5)$, of length 9, the longest of the instance: cities 4 and 5 are at opposite ends of the instance, and NN, from whatever city it starts, leaves at least one of them to the last steps, when the only way left is the long one.

2.8 Vertex coloring: greedy and DSATUR

- The problem of this section is the vertex coloring problem (VCP): its definition and its assumptions are on its page. The graph is $G = (V, E)$, and the optimal value is the chromatic number $\chi(G)$.

2.8.1 The greedy coloring

The *Greedy Coloring Algorithm* (denoted by GC) is a constructive heuristic for the VCP. The colors are the integers $1, 2, 3, \ldots$; the algorithm examines the vertices in the order of a given list λ and gives each vertex the smallest color not used by its neighbours already colored.

Algorithm

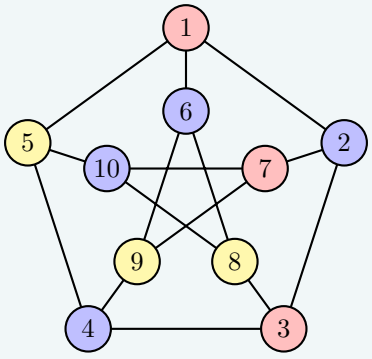
Algorithm: Greedy Coloring

Input: VCP instance $G = (V, E)$ with n vertices; list $\lambda[1..n]$
Output: Vertex-color assignment $col[1..n]$ and number of colors nc

Function *GreedyColoring*(n, E, λ)
  $col :=$ new array$[1..n]$, all entries 0 `// 0 = not yet colored`
  for $k := 1$ **to** n **do**
    $v := \lambda[k]$
    $U := \{\, col[u] : u \in N(v),\ col[u] \neq 0 \,\}$ `// colors of the colored neighbours`
    $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \notin U \,\}$ `// smallest free color`
  end
  return col, $nc = \max_v col[v]$

- The color classes $Q_c = \{v : col[v] = c\}$ are independent sets by construction, so GC always returns a feasible coloring. It runs in $O(n + |E|)$ time.
- As $LIST$, GC depends on the order of the list, and the order can make a large difference.

Example 2.12: GC **on the Petersen graph, two orders**



List $\lambda = (1, 2, \ldots, 10)$**.** The colors of the colored neighbours U and the color assigned are:

v	1	2	3	4	5	6	7	8	9	10
U	$\emptyset$	$\{1\}$	$\{2\}$	$\{1\}$	$\{1,2\}$	$\{1\}$	$\{2\}$	$\{1,2\}$	$\{1,2\}$	$\{1,3\}$
$col[v]$	1	2	1	2	3	2	1	3	3	2

GC uses 3 colors (red = 1, blue = 2, yellow = 3 in the figure). The outer 5-cycle $1, 2, 3, 4, 5$ is an odd cycle and needs 3 colors, so $\chi(G) = 3$ and this coloring is optimal.

List $\lambda = (1, 2, 3, 4, 5, 7, 8, 6, 9, 10)$**.** The first five vertices get the same colors as before; then

v	7	8	6	9	10
U	$\{2\}$	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$
$col[v]$	1	2	3	4	4

and GC uses 4 colors: vertex 8 takes color 2, which the first order gave to vertex 6, and vertices 9 and 10 then see all three colors among their neighbours.

2.8.2 DSATUR

The *DSATUR Algorithm* (Degree of SATURation, denoted by $DSATUR$) is a constructive heuristic for the VCP. It does not use a fixed list: at each step it colors, with the smallest free color, an uncolored vertex of maximum **saturation degree**, i.e., with the largest number of *different* colors among its neighbours; ties are broken by the largest degree in the subgraph of the uncolored vertices, and then by the smallest index.

Algorithm

Algorithm: DSATUR

Input: VCP instance $G = (V, E)$ with n vertices
Output: Vertex-color assignment $col[1..n]$ and number of colors nc

Function *DSATUR*(n, E)
  col := new array$[1..n]$, all entries 0
  for $k := 1$ **to** n **do**
    for *each uncolored* u **do**
      $sat[u] \leftarrow |\{\, col[w] : w \in N(u),\ col[w] \neq 0 \,\}|$ `// saturation degree`
      $du[u] \leftarrow |\{\, w \in N(u) : col[w] = 0 \,\}|$ `// degree among the uncolored`
    end
    $v :=$ the uncolored vertex maximizing $(sat[v], du[v])$ lexicographically (smallest index)
    $col[v] \leftarrow \min\{\, c \in \mathbb{Z}_{\geq 1} : c \neq col[w]\ \forall w \in N(v) \,\}$
  end
  return col, $nc = \max_v col[v]$

- **Why the saturation degree.** A vertex with many different colors around it has few colors left: if we postpone it, it may end up with none of the colors already in use. $DSATUR$ colors first the vertices that are most constrained, the same idea as "the hardest

items first" of FFD and LPT, but updated dynamically after each assignment.

- $DSATUR$ is GC with a list built on the fly, so every property of GC that holds for every list holds for $DSATUR$ too. A straightforward implementation runs in $O(n^2)$ time.

Example 2.13: $DSATUR$ on the Petersen graph

All the vertices have degree 3. The vertex chosen at each step, its saturation degree, its degree among the uncolored vertices and its color are:

step	1	2	3	4	5	6	7	8	9	10
v	1	2	3	4	5	6	8	7	9	10
$sat[v]$	0	1	1	1	2	1	2	1	2	2
$du[v]$	3	2	2	2	1	2	1	2	0	0
$col[v]$	1	2	1	2	3	2	3	1	3	2

At step 1 all the vertices tie and vertex 1 is chosen. At step 2 the vertices 2, 5, 6 have saturation degree 1 and two uncolored neighbours each: the smallest index, 2, is chosen. At step 7 vertex 8 (neighbours 3 and 6, colors 1 and 2) has saturation degree 2 and goes before vertex 7. $DSATUR$ uses $3 = \chi(G)$ colors, the same coloring as GC with the list $(1, 2, \ldots, 10)$.

2.8.3 What the greedy coloring guarantees

Proposition 2.5

For every graph G and every list λ,

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

Proof. When vertex v is colored, the set U of the colors of its colored neighbours has at most $\deg(v) \leq \Delta(G)$ elements, so one of the colors $1, 2, \ldots, \Delta(G) + 1$ is not in U, and the smallest free color is at most $\Delta(G) + 1$. □

- The bound holds for $DSATUR$ too, which is GC with a particular list, and it is attained by complete graphs ($\Delta = n - 1$, $\chi = n$) and odd cycles ($\Delta = 2$, $\chi = 3$), whatever the list.
- It is *not* a performance guarantee: it compares $GC(I)$ with $\Delta(G)+1$, not with $\chi(G)$, and $\Delta(G) + 1$ can be much larger than $\chi(G)$. The next proposition shows that a bad list can really force that many colors on a graph that needs only two.
- On the other hand, **some** list is always optimal: if $Q_1, \ldots, Q_{\chi(G)}$ are the color classes of an optimal coloring, GC with the vertices of Q_1 first, then those of Q_2, and so on, gives each vertex of Q_c a color at most c, so it uses at most $\chi(G)$ colors. The difficulty is entirely in finding the list.

Proposition 2.6

For every $k \geq 3$ there is a graph G_k with $n = 2k$ vertices, $\chi(G_k) = 2$ and $\Delta(G_k) = k - 1$, and a list λ, such that

$$GC(I_k) \;=\; k \;=\; \frac{n}{2}.$$

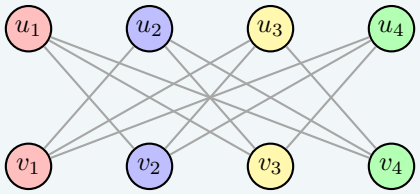
Proof. G_k is the **crown graph**: the vertices are $u_1, \ldots, u_k$ and $v_1, \ldots, v_k$, and u_i is adjacent to v_j if and only if $i \neq j$ (the complete bipartite graph $K_{k,k}$ without a perfect matching). It is bipartite, with an edge, so $\chi(G_k) = 2$; every vertex has degree $k - 1$. The list is $\lambda = (u_1, v_1, u_2, v_2, \ldots, u_k, v_k)$.

We show by induction on i that GC gives color i to both u_i and v_i. For $i = 1$ no neighbour is colored when u_1 is examined, and v_1 is not adjacent to u_1: both get color 1. When u_i is examined, its colored neighbours are $v_1, \ldots, v_{i-1}$, with colors $1, \ldots, i-1$ by induction, so u_i gets color i; when v_i is examined, its colored neighbours are $u_1, \ldots, u_{i-1}$ (u_i is not a neighbour), with colors $1, \ldots, i-1$, so v_i gets color i too. Hence $GC(I_k) = k$. □

- On the crown graphs $\frac{GC(I_k)}{\chi(G_k)} = \frac{k}{2} = \frac{n}{4}$, which grows without bound: with an arbitrary list, GC has no constant performance guarantee. The list $(u_1, \ldots, u_k, v_1, \ldots, v_k)$, the color classes first, gives 2 colors.
- The crown graphs satisfy the assumptions of the VCP for $k \geq 3$: no isolated vertices, and $N(u_i) = \{v_j : j \neq i\}$ is not contained in $N(u_{i'})$ nor in any $N(v_j) \subseteq \{u_1, \ldots, u_k\}$.

Example 2.14: *GC* and *DSATUR* on the crown graph G_4

Number the vertices $u_i = 2i - 1$ and $v_i = 2i$, so that $u_1, v_1, \ldots, u_4, v_4$ are $1, 2, \ldots, 8$, and $E = \{\{2i-1, 2j\} : i \neq j\}$, with 12 edges.



With the list $(1, 2, \ldots, 8) = (u_1, v_1, \ldots, u_4, v_4)$, GC uses 4 colors, one for each pair $\{u_i, v_i\}$ (shown in the figure); with the list $(1, 3, 5, 7, 2, 4, 6, 8)$ it uses 2 colors.

DSATUR chooses vertex 1 (u_1, color 1), then the vertices 4, 5, 2, 3, 6, 7, 8, with colors $2, 1, 2, 1, 2, 1, 2$: the vertices u_i get color 1 and the vertices v_i color 2, and *DSATUR* uses $2 = \chi(G_4)$ colors. After the first step, every vertex adjacent to a colored vertex has saturation degree 1, and the vertex chosen is always on the side opposite to the last one colored.

2.8.4 DSATUR as a heuristic

- On bipartite graphs $DSATUR$ always uses at most 2 colors, as on the crown graphs; this is proved in the paper that introduced it (Brélaz, 1979).
- On general graphs $DSATUR$ has no performance guarantee, since the VCP is not approximable. It is nevertheless one of the most used constructive heuristics for the VCP, because on most graphs it uses few colors more than $\chi(G)$, often none; it is also the base of exact branch-and-bound algorithms for the VCP, which branch on the vertex chosen by the $DSATUR$ rule.

2.9 Randomized construction

- A greedy heuristic is deterministic: run twice on the same instance, it returns the same solution. The multi-start NN obtains different solutions by changing the starting city, but the number of starts is limited to n. A more flexible way to obtain many different, and still good, solutions is to **randomize the choice**: at each step, instead of the best candidate, choose at random one of the *good* candidates.

- The good candidates form the **restricted candidate list** (RCL). For a minimization criterion with values $g(q)$ of the candidates q, let $g^{\min}$ and $g^{\max}$ be the smallest and the largest value; the *value-based* RCL with parameter $\alpha \in [0,1]$ is

$$RCL \;=\; \Big\{ q \;:\; g(q) \;\le\; g^{\min} + \alpha\,\big(g^{\max} - g^{\min}\big) \Big\}.$$

With $\alpha = 0$ the RCL contains only the best candidates, and the construction is greedy (with random tie-breaking); with $\alpha = 1$ it contains all the candidates, and the construction is purely random.

- An alternative is the *cardinality-based* RCL, made of the k best candidates for a fixed k. Randomized construction is the first ingredient of the metaheuristic GRASP (T.A. Feo, M.G.C. Resende, *Greedy randomized adaptive search procedures*, Journal of Global Optimization 6 (1995) 109–133).

- **Random numbers.** To make examples and exercises reproducible, the notes on heuristic algorithms use the *minimal standard* generator of Park and Miller, which produces a sequence of numbers $u_1, u_2, \ldots$ in $(0,1)$ from an integer **seed**:

$$seed \;\leftarrow\; (16807 \cdot seed) \bmod 2147483647, \qquad\qquad u \;=\; \frac{seed}{2147483647}.$$

- The modulus is $2^{31}-1$, a prime number; with 64-bit integers the product never overflows. The same seed gives the same sequence on every computer and in every programming language. A candidate is drawn from an RCL with $|RCL|$ elements, listed by increasing index, by taking the element in position $\lfloor u \cdot |RCL| \rfloor + 1$; a random number is used only when $|RCL| \ge 2$.

The *randomized nearest neighbour algorithm* (denoted by RNN_α) is a constructive heuristic for the TSP. Starting from city s, at each step it builds the RCL of the unvisited cities whose distance from the last city is at most $d^{\min} + \alpha\,(d^{\max} - d^{\min})$, and appends a city drawn uniformly at random from the RCL.

Algorithm

Algorithm: Randomized Nearest Neighbour

Input: TSP instance with n cities, distances $d[1..n][1..n]$; starting city s; $\alpha \in [0,1]$; *seed*

Output: Tour $\sigma[1..n]$

Function *RandomizedNN*$(n, d, s, \alpha, seed)$
  $\sigma[1] \leftarrow s$, $visited := \{s\}$
  for $len := 2$ **to** n **do**
    $last := \sigma[len-1]$
    $d^{\min} := \min_{j \notin visited} d[last][j]$, $d^{\max} := \max_{j \notin visited} d[last][j]$
    $RCL := \{\, j \notin visited : d[last][j] \le d^{\min} + \alpha\,(d^{\max} - d^{\min}) \,\}$ `// by index`
    if $|RCL| \ge 2$ **then**
      $u :=$ ParkMiller$(seed)$ `// random number in` $(0,1)$
      $k :=$ the element in position $\lfloor u \cdot |RCL| \rfloor + 1$ of RCL
    else
      $k :=$ the only element of RCL
    end
    $\sigma[len] \leftarrow k$, $visited \leftarrow visited \cup \{k\}$
  end
  return σ

- Repeating the construction with the same generator — the seed is *not* reset between two runs — gives different tours; the best one is kept. The parameter α balances quality and variety: a small α gives tours close to the NN tour, a large α gives very different but usually poor tours.

Example 2.15: $RNN_{0.3}$ on the TSP reference instance

We run RNN_α four times from city 1, with $\alpha = 0.3$ and initial seed 1. The generator gives

u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9
0.0000	0.1315	0.7556	0.4587	0.5328	0.2190	0.0470	0.6789	0.6793

($u_1 = \frac{16807}{2147483647} \approx 0.0000078$). From city 1 the distances to $2,3,4,5$ are $5,2,2,9$, so the threshold is $2 + 0.3 \cdot (9-2) = 4.1$ and $RCL = \{3,4\}$.

Run 1. u_1: position $\lfloor 0.0000 \cdot 2 \rfloor + 1 = 1$, city 3. From 3 the distances to $2,4,5$ are $4,4,7$, threshold $4 + 0.3 \cdot 3 = 4.9$, $RCL = \{2,4\}$; u_2: position 1, city 2. From 2 the distances to $4,5$ are $3,4$, threshold 3.3, $RCL = \{4\}$: city 4, then city 5. Tour $(1,3,2,4,5)$, $L = 25$: the NN tour.

Run 2. u_3: position $\lfloor 0.7556 \cdot 2 \rfloor + 1 = 2$, city 4. From 4 the distances to $2, 3, 5$ are $3, 4, 7$, threshold $3 + 0.3 \cdot 4 = 4.2$, $RCL = \{2, 3\}$; u_4: position 1, city 2. From 2 the distances to $3, 5$ are $4, 4$: $d^{\min} = d^{\max} = 4$ and $RCL = \{3, 5\}$; u_5: position $\lfloor 0.5328 \cdot 2 \rfloor + 1 = 2$, city 5; then city 3. Tour $(1, 4, 2, 5, 3)$, $L = 18 = z(I)$.

Run 3. u_6: city 3; u_7: city 2; then cities 4 and 5. Tour $(1, 3, 2, 4, 5)$, $L = 25$.

Run 4. u_8: city 4; $RCL = \{2, 3\}$ and u_9: position 2, city 3. From 3 the distances to $2, 5$ are $4, 7$, threshold 4.9, $RCL = \{2\}$: city 2, then city 5. Tour $(1, 4, 3, 2, 5)$, $L = 2+4+4+4+9 = 23$.

The four runs give $25, 18, 25, 23$: the second one finds the optimal tour, which NN from city 1 misses. The randomization has turned a single deterministic answer into a small search.

2.10 Look-ahead and regret

- A greedy criterion asks: *which choice is best now?* Its myopia can be reduced by asking two different questions:
  - **regret**: *what do I lose if I postpone this choice?* An element that has a single good option should be served before it loses it; an element with many equivalent options can wait;
  - **look-ahead**: *where does this choice lead?* Each candidate is evaluated by completing the partial solution with a fast heuristic, and the candidate with the best completed solution is chosen.

  Both ideas cost more time per step than the plain greedy criterion, and both keep the essential feature of a constructive heuristic: the choice made is final.

2.10.1 Regret insertion

- For an insertion heuristic, let $\Delta_q^1 \le \Delta_q^2$ be the two smallest insertion costs of city q over the edges of the current subtour (with $\Delta_q^2 = \Delta_q^1$ if the subtour has a single edge). The **regret** of q is

$$reg_q \;=\; \Delta_q^2 - \Delta_q^1 \;\ge\; 0 :$$

the extra cost paid for q if its best edge is used by another city before q is inserted.

The *regret-2 insertion algorithm* (denoted by REG_2) is a constructive heuristic for the TSP. Starting from the subtour (s), at each step it inserts the city with the *largest regret*, in case of ties the one with the smallest Δ_q^1 and then the smallest index, into its edge of minimum insertion cost.

Algorithm

Algorithm: Regret-2 Insertion

Input: TSP instance with n cities, distances $d[1..n][1..n]$; starting city s
Output: Tour $\sigma[1..n]$

Function *Regret2Insertion*(n, d, s)
  $\sigma := (s)$
  for $len := 1$ **to** $n - 1$ **do**
    foreach *city* $q \notin \sigma$ **do**
      $\Delta_q^1, \Delta_q^2 :=$ the two smallest values of $\Delta(\sigma[a], q, \sigma[a+1])$, $a = 1, \ldots, len$
      $reg_q := \Delta_q^2 - \Delta_q^1$ `// regret of` q
    end
    $k :=$ the city with maximum reg_q (ties: minimum Δ_q^1, then smallest index)
    insert k into its edge of minimum insertion cost
  end
  return σ

- Regret-k heuristics generalize the idea, summing the differences between the best option and the $k-1$ next ones; they are widely used for vehicle routing problems, where a customer may have a single feasible route left (J.-Y. Potvin, J.-M. Rousseau, *A parallel route building algorithm for the vehicle routing and scheduling problem with time windows*, European Journal of Operational Research 66 (1993) 331–340). The same idea applies to any assignment-like construction, such as items to bins.

- As long as all the regrets are 0, the tie-breaking rule makes REG_2 behave like CI.

Example 2.16: REG_2 against CI on the 6-city instance

On the instance of Example 2.11, from city 1. In the first two steps all the regrets are 0 (the subtours (1) and (1, 6) have one edge, or two copies of the same edge), and both REG_2 and CI insert city 6 (cost 4) and then city 3 (cost 5), obtaining $(1, 3, 6)$ of length 9. At the third step the insertion costs on the edges $(1, 3)$, $(3, 6)$, $(6, 1)$ are

city q	costs	Δ_q^1	Δ_q^2	reg_q
2	$(2, 2, 7)$	2	2	0
4	$(4, 7, 8)$	4	7	3
5	$(6, 5, 6)$	5	6	1

CI inserts city 2, the cheapest, on the edge $(1, 3)$. REG_2 inserts city 4, the one with the largest regret, on the edge $(1, 3)$: this edge is the only good place for city 4, while city 2 has two equally good places and can wait. The two constructions then continue:

	CI	REG_2
step 3	2 on $(1,3)$, cost 2: $(1,2,3,6)$	4 on $(1,3)$, cost 4: $(1,4,3,6)$
step 4	4 on $(1,2)$, cost 4: $(1,4,2,3,6)$	5 on $(3,6)$, cost 5: $(1,4,3,5,6)$
step 5	5 on $(3,6)$, cost 5: $(1,4,2,3,5,6)$	2 on $(3,5)$, cost 1: $(1,4,3,2,5,6)$
length	$4+5+2+4+5=20$	$4+5+4+5+1=19$

At step 4 of REG_2 the costs of city 2 on the edges of $(1,4,3,6)$ are $(6,2,2,7)$, regret 0, and those of city 5 are $(10,10,5,6)$, regret 1: city 5 goes first. City 2, postponed twice, is finally inserted for 1 instead of 2, between cities 3 and 5. $REG_2(I) = 19 = z(I)$, while $CI(I) = 20$.

2.10.2 Look-ahead: the pilot method

The *pilot method* turns a constructive heuristic $\mathscr{H}$, the *pilot*, into a stronger one. At each step it tries every candidate: the partial solution extended with the candidate is completed by $\mathscr{H}$, and the candidate whose completed solution is best is added for good. With NN as pilot (denoted by $PILOT\text{-}NN$), the candidates are the next cities of the tour.

Algorithm

Algorithm: Pilot method with nearest neighbour

Input: TSP instance with n cities, distances $d[1..n][1..n]$; starting city s
Output: Tour $\sigma[1..n]$

```
1  Function PilotNN(n, d, s)
2      σ := (s)
3      for len := 2 to n do
4          foreach city q ∉ σ do
5              τ_q := the path σ extended with q and completed by NN   // look ahead
6          end
7          k := the city q with minimum L(τ_q) (smallest index)
8          append k to σ                                   // the choice is final
9      end
10     return σ
```

- The pilot method is due to C. Duin and S. Voß (*The pilot method: a strategy for heuristic repetition with application to the Steiner problem in graphs*, Networks 34 (1999) 181–191). Each step runs the pilot once per candidate: with NN as pilot the running time is $O(n^4)$ instead of $O(n^2)$.

- **The pilot method is never worse than its pilot.** At the first step, the candidate chosen by NN is among those tried, and its completion is the NN tour from s, so the best completion found has length at most $NN(I)$. At every later step, the completion of the chosen candidate is again among the completions tried at the next step (the candidate

that NN would append), so the length of the best completion never increases. The final tour is the last completion, hence $PILOT\text{-}NN(I) \leq NN(I)$ from the same starting city.

Example 2.17: $PILOT\text{-}NN$ on the TSP reference instance

From city 1, the candidates and their NN completions are:

partial path	candidates: completed tour and length	chosen
(1)	2: $(1,2,4,3,5)$ 28; 3: $(1,3,2,4,5)$ 25; 4: $(1,4,2,3,5)$ 25; 5: $(1,5,2,4,3)$ 22	5
$(1,5)$	2: $(1,5,2,4,3)$ 22; 3: $(1,5,3,2,4)$ 25; 4: $(1,5,4,2,3)$ 25	2
$(1,5,2)$	3: $(1,5,2,3,4)$ 23; 4: $(1,5,2,4,3)$ 22	4
$(1,5,2,4)$	3: $(1,5,2,4,3)$ 22	3

$PILOT\text{-}NN$ goes first to city 5, the *farthest* from city 1, something NN would never do, and returns $(1,5,2,4,3)$ with $L = 9+4+3+4+2 = 22$, against $NN(I) = 25$ from the same city. The optimum 18 is not reached: the look-ahead is only as good as the pilot that performs it. On the 6-city instance of Example 2.11, $PILOT\text{-}NN$ from city 1 gives 20, against 25 for NN.

2.11 What constructive heuristics give, and what they do not

- The table collects the results on the four reference instances, with the dual bounds of the chapter «Heuristic algorithms: definitions» (UB for the KP, LB for the other problems) and the gap of the best heuristic solution.

problem	heuristic	values	dual bound	gap	$z(I)$
KP	GR_p, GR, GR_w	17, 16, 10	UB = 17	0%	17
BPP	FF, FFD, BFD	4, 3, 3	LB = 3	0%	3
$P\,\|\|\,C_{\max}$	$LIST$ (reverse list), LPT	12, 11	LB = 9	22.2%	9
TSP	NN, multi-start NN, NI, FI, CI	25, 18, 18, 18, 18	LB = 17	5.9%	18

- **A primal bound, and nothing more.** A constructive heuristic returns a feasible solution, and its value is a primal bound. For the KP and the BPP the best heuristic value meets the dual bound, and the solution is *proved* optimal; for the TSP the tour of length 18 is optimal, but the dual bound 17 cannot prove it; for $P\,||\,C_{\max}$ the gap of 22.2% is real. The heuristic itself never knows in which of these situations it is.

- **A choice is never revised.** Every heuristic of these notes stops as soon as the solution is complete, and every decision taken along the way is final. In the LPT schedule, job 7 is put on machine 1 when the three loads are equal, and the makespan 11 is fixed at that moment; in the NN tour from city 1, the edge $(5,1)$ of length 9 is the consequence of

the very first choices. Looking at the Gantt chart or at the tour, a better solution seems close; but a constructive heuristic has no mechanism to go back.

- **The order decides everything.** The examples of these notes show the same lesson on four problems: sorting the elements in the right order (efficiency, decreasing weights, longest jobs first) is the main design choice; still, no order is the best on every instance (Example 2.1, Example 2.4), and running several orders, several starting points or several randomized constructions, and keeping the best solution, is a cheap and effective protection.

- **Cheap, and useful as a start.** Constructive heuristics are fast, and they are rarely used alone: their solutions are the starting points of the methods that try to improve a complete solution.

A constructive heuristic builds one solution, one element at a time, choosing at each step the element that looks best now. The order of the choices decides the quality of the result, and a choice, once made, is never revised.

2.12 Typical mistakes

Warning

- **Expecting the greedy by efficiency to be optimal.** It can be arbitrarily bad, and so can each of the other two orders: the notes contain a family of instances for each of them.
- **Believing that sorting always helps.** FFD can be worse than FF (Example 2.4).
- **Breaking ties in the wrong way.** Every example uses one fixed rule (smallest index, first bin that fits, fullest bin for BF): changing it changes the solution.
- **Forgetting the closing edge.** The length of a tour includes the edge from the last city back to the first one; the cost of an insertion is $d_{ik} + d_{kj} - d_{ij}$, and every position, closing edge included, must be tried.
- **Reading $\mathscr{A}(I)$ as a bound on $z(I)$.** It is only a primal bound.

2.13 In summary

Summary

- A constructive heuristic builds one solution, one element at a time, and never revises a choice.
- The order of the choices is the main design decision: efficiency for the KP, decreasing weights for the BPP, longest jobs first for $P\,||\,C_{\max}$; no order wins on every instance.
- Several orders, starting points or randomized constructions, keeping the best solution, are a cheap protection.
- The solutions are starting points for local search.

Dashboard of the reference instances. Best constructive value and dual bound:

problem	best heuristic value	dual bound	gap	$z(I)$
KP (max)	17 (GR_p)	UB$(I) = 17$	0%	17
BPP (min)	3 (FFD, BFD)	LB$(I) = 3$	0%	3
$P\,||\,C_{\max}$ (min)	11 (LPT)	LB$(I) = 9$	22.2%	9
TSP (min)	18 (multi-start NN, NI, FI, CI)	LB$(I) = 17$	5.9%	18

2.14 C++ implementation

- The folder `CONSTR_HEUR_CODE` contains a `C++` implementation of the heuristics of these notes, one function per file, with the same names and the same steps as the pseudo-codes. Items, bins, jobs, machines and cities are indexed from 0; the algorithmic logic is unchanged, and ties are always broken in favour of the smallest index.
- The command `make` builds the program, and `make test` runs every example of these notes and every instance of the exercises, comparing each value and each solution with those computed by an independent implementation in Python: 202 checks, 0 errors.
- The header `Constr_heuristics.h` declares all the functions:

```
#ifndef CONSTR_HEURISTICS_HEADER
#define CONSTR_HEURISTICS_HEADER

// constructive heuristics for the KP, the BPP, P||Cmax and the TSP
// items, bins, jobs, machines and cities are indexed starting from 0
// ties are always broken in favour of the smallest index

// sorting
void KPOrder(int n, int* p, int* w, int criterion, int* order);   // 0 profit, 1 weight, 2 efficiency
void DecreasingOrder(int n, int* a, int* order);

// KP: returns the profit, x[i] = 1 if item i is selected
```

```
int GreedyOrderKP(int n, int c, int* w, int* p, int* order, int* x);

// BPP: returns the number of bins, bin[i] = bin of item i
int FirstFitDecreasing(int n, int c, int* w, int* bin);
int BestFitDecreasing(int n, int c, int* w, int* bin);

// P||Cmax: returns the makespan, mach[j] = machine of job j
int ListScheduling(int n, int m, int* t, int* order, int* mach);
int LPT(int n, int m, int* t, int* mach);

// TSP: d is the n x n distance matrix; every function returns the length of the tour
int TourLength(int n, int** d, int* tour);
int NearestNeighbour(int n, int** d, int s, int* tour);
int MultiStartNN(int n, int** d, int* tour);
int Insertion(int n, int** d, int s, int rule, int* tour);            // 0 NI, 1 FI, 2 CI
int Regret2Insertion(int n, int** d, int s, int* tour);
int PilotNN(int n, int** d, int s, int* tour);

// randomized construction
double ParkMiller(long long* seed);
int RandomizedNN(int n, int** d, int s, int alpha_num, int alpha_den, long long * seed, int* tour);

#endif
```

2.14.1 Sorting

- The orders of the KP items (Section 2.4) and the decreasing order used by FFD, BFD and LPT are computed by insertion sort, which keeps the index order among equal keys. The efficiencies are compared without divisions: $\frac{p_j}{w_j} > \frac{p_k}{w_k}$ if and only if $p_j w_k > p_k w_j$.

```
// order[0..n-1] receives the items sorted by the criterion:
// 0 = non-increasing profit, 1 = non-decreasing weight, 2 = non-increasing efficiency p/w
// insertion sort: items with equal keys keep their index order

static bool Before(int i, int j, int* p, int* w, int criterion)
{
    if (criterion == 0) return p[i] > p[j];
    if (criterion == 1) return w[i] < w[j];
    return p[i] * w[j] > p[j] * w[i];      // p_i/w_i > p_j/w_j without divisions
}

void KPOrder(int n, int* p, int* w, int criterion, int* order)
{
    for (int k = 0; k < n; k++)
    {
        int i = k;                         // item to be inserted in order[0..k-1]
        int pos = k;
        while (pos > 0 && Before(i, order[pos - 1], p, w, criterion))
        {
            order[pos] = order[pos - 1];
```

```
            pos--;
        }
        order[pos] = i;
    }
}
```

```
// order[0..n-1] receives the indices sorted by non-increasing a[i]
// insertion sort: indices with equal values keep their order

void DecreasingOrder(int n, int* a, int* order)
{
    for (int k = 0; k < n; k++)
    {
        int pos = k;
        while (pos > 0 && a[k] > a[order[pos - 1]])
        {
            order[pos] = order[pos - 1];
            pos--;
        }
        order[pos] = k;
    }
}
```

2.14.2 The 0/1 knapsack problem

```
// the items are examined in the order given by order[0..n-1]
// x[i] stores the selection decision for item i
// the function returns the profit of the solution x

int GreedyOrderKP(int n, int c, int* w, int* p, int* order, int* x)
{
    for (int i = 0; i < n; i++)
    {
        x[i] = 0;                          // item i is initially not selected
    }
    int rc = c;                            // residual capacity
    int z = 0;                             // profit of the selected items

    for (int k = 0; k < n; k++)
    {
        int i = order[k];                  // k-th item in the order
        if (w[i] <= rc)                    // feasibility test
        {
            x[i] = 1;                      // select item i
            rc = rc - w[i];                // update residual capacity
            z = z + p[i];
        }
    }
    return z;
}
```

2.14.3 The bin packing problem

```
// the items are sorted by non-increasing weight and packed by First Fit
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i

int FirstFitDecreasing(int n, int c, int* w, int* bin)
{
    int* order = new int[n];
    DecreasingOrder(n, w, order);          // sort: largest items first

    int* rc = new int[n];                  // residual capacities (at most n
    bins)
    int nb = 0;                            // number of bins opened so far

    for (int k = 0; k < n; k++)
    {
        int i = order[k];                  // k-th largest item
        int s = -1;                        // selected bin (-1 means: not found
     yet)
        for (int b = 0; b < nb; b++)       // first bin in which item i fits
        {
            if (w[i] <= rc[b])
            {
                s = b;
                break;
            }
        }
        if (s == -1)                       // no bin fits: open a new one
        {
            rc[nb] = c;
            s = nb;
            nb++;
        }
        bin[i] = s;                        // pack item i into bin s
        rc[s] = rc[s] - w[i];
    }

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

```
// the items are sorted by non-increasing weight and packed by Best Fit
// the function returns the number of bins used
// bin[i] stores the bin assigned to item i

int BestFitDecreasing(int n, int c, int* w, int* bin)
{
    int* order = new int[n];
    DecreasingOrder(n, w, order);          // sort: largest items first

    int* rc = new int[n];                  // residual capacities (at most n
    bins)
    int nb = 0;                            // number of bins opened so far

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

```
        int i = order[k];                       // k-th largest item
        int s = -1;                             // selected bin (-1 means: not found
    yet)
        for (int b = 0; b < nb; b++)            // fullest bin in which item i fits
        {
            if (w[i] <= rc[b] && (s == -1 || rc[b] < rc[s]))
            {
                s = b;
            }
        }
        if (s == -1)                            // no bin fits: open a new one
        {
            rc[nb] = c;
            s = nb;
            nb++;
        }
        bin[i] = s;                             // pack item i into bin s
        rc[s] = rc[s] - w[i];
    }

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

2.14.4 Scheduling on parallel machines

```
// the jobs are examined in the order given by order[0..n-1]
// each job is assigned to the machine with the smallest current load
// the function returns the makespan; mach[j] stores the machine of job j

int ListScheduling(int n, int m, int* t, int* order, int* mach)
{
    int* L = new int[m];                        // loads of the machines
    for (int h = 0; h < m; h++)
    {
        L[h] = 0;
    }

    for (int k = 0; k < n; k++)
    {
        int j = order[k];                       // k-th job of the list
        int s = 0;                              // least loaded machine
        for (int h = 1; h < m; h++)
        {
            if (L[h] < L[s]) s = h;
        }
        mach[j] = s;                            // assign job j to machine s
        L[s] = L[s] + t[j];
    }

    int Cmax = 0;                               // makespan = largest load
    for (int h = 0; h < m; h++)
    {
        if (L[h] > Cmax) Cmax = L[h];
    }
    delete[] L;
```

```
    return Cmax;
}
```

```
// Longest Processing Time first: list scheduling with the jobs
// sorted by non-increasing processing time
// the function returns the makespan; mach[j] stores the machine of job j

int LPT(int n, int m, int* t, int* mach)
{
    int* order = new int[n];
    DecreasingOrder(n, t, order);          // longest jobs first
    int Cmax = ListScheduling(n, m, t, order, mach);
    delete[] order;
    return Cmax;
}
```

2.14.5 The travelling salesman problem

- The distances are stored in an $n \times n$ matrix `d`, and a tour in an array of n cities.

```
// length of the tour tour[0], tour[1], ..., tour[n-1], tour[0]

int TourLength(int n, int** d, int* tour)
{
    int L = 0;
    for (int k = 0; k < n - 1; k++)
    {
        L = L + d[tour[k]][tour[k + 1]];
    }
    return L + d[tour[n - 1]][tour[0]];    // closing edge
}
```

```
// starting from city s, the tour is extended at its end with the
// nearest city not yet visited
// the function returns the length of the tour stored in tour[0..n-1]

int NearestNeighbour(int n, int** d, int s, int* tour)
{
    bool* visited = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;                           // the tour starts at city s
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int last = tour[len - 1];          // last city of the partial tour
        int k = -1;                        // nearest unvisited city
        for (int j = 0; j < n; j++)
        {
            if (!visited[j] && (k == -1 || d[last][j] < d[last][k])) k = j;
        }
        tour[len] = k;                     // append k at the end
        visited[k] = true;
    }
```

```
26
27      delete[] visited;
28      return TourLength(n, d, tour);
29  }
```

```
1   // runs the nearest neighbour heuristic from every city
2   // and keeps the shortest tour (the first one found, in case of ties)
3   // the function returns its length; the tour is stored in tour[0..n-1]
4
5   int MultiStartNN(int n, int** d, int* tour)
6   {
7       int* cur = new int[n];
8       int best = -1;                          // length of the best tour found
9       for (int s = 0; s < n; s++)
10      {
11          int L = NearestNeighbour(n, d, s, cur);
12          if (best == -1 || L < best)
13          {
14              best = L;
15              for (int k = 0; k < n; k++) tour[k] = cur[k];
16          }
17      }
18      delete[] cur;
19      return best;
20  }
```

- In `Insertion` the three selection rules share the same loop: the value of a candidate is its distance from the subtour (NI), the same distance with the sign changed, so that the minimum corresponds to the farthest city (FI), or its minimum insertion cost (CI).

```
1   // insertion heuristics: starting from the subtour (s), the city selected by
2   // the rule is inserted into the subtour where it increases the length the
        least
3   // rule: 0 = nearest insertion (NI), 1 = farthest insertion (FI), 2 = cheapest
        insertion (CI)
4   // the function returns the length of the tour stored in tour[0..n-1]
5
6   // cheapest position for city k in the subtour tour[0..len-1]:
7   // returns the increase of length, and in *pos the edge (tour[a], tour[a+1]) to
        break
8   static int BestPosition(int len, int** d, int* tour, int k, int* pos)
9   {
10      int best = -1;
11      for (int a = 0; a < len; a++)
12      {
13          int i = tour[a], j = tour[(a + 1) % len];
14          int cost = d[i][k] + d[k][j] - d[i][j];
15          if (best == -1 || cost < best)
16          {
17              best = cost;
18              *pos = a;
19          }
20      }
21      return best;
22  }
23
```

```
int Insertion(int n, int** d, int s, int rule, int* tour)
{
    bool* in = new bool[n];                 // in[i] = true if city i is in the
    subtour
    for (int i = 0; i < n; i++)
    {
        in[i] = false;
    }
    tour[0] = s;                            // the subtour (s)
    in[s] = true;

    for (int len = 1; len < n; len++)
    {
        // selection step: city k to be inserted
        int k = -1, keyk = 0, pos = 0;
        for (int q = 0; q < n; q++)
        {
            if (in[q]) continue;
            int key;                        // value of city q for the rule
            if (rule == 2)
            {
                int a;
                key = BestPosition(len, d, tour, q, &a);   // insertion cost
            }
            else
            {
                key = d[q][tour[0]];        // distance from the subtour
                for (int a = 1; a < len; a++)
                {
                    if (d[q][tour[a]] < key) key = d[q][tour[a]];
                }
                if (rule == 1) key = -key; // farthest: maximize the distance
            }
            if (k == -1 || key < keyk)
            {
                k = q;
                keyk = key;
            }
        }
        // insertion step: k goes where it increases the length the least
        BestPosition(len, d, tour, k, &pos);
        for (int a = len; a > pos + 1; a--)
        {
            tour[a] = tour[a - 1];
        }
        tour[pos + 1] = k;
        in[k] = true;
    }

    delete[] in;
    return TourLength(n, d, tour);
}
```

2.14.6 Randomized construction

- The threshold of the RCL is tested in integer arithmetic, with α given as a fraction $\frac{\alpha_{num}}{\alpha_{den}}$, so that no rounding error can change the list.

```
// Park-Miller minimal standard generator:
// seed <- 16807 * seed mod (2^31 - 1), and u = seed / (2^31 - 1) in (0,1)

double ParkMiller(long long* seed)
{
    *seed = (16807LL * (*seed)) % 2147483647LL;
    return (double)(*seed) / 2147483647.0;
}
```

```
// randomized nearest neighbour with a value-based restricted candidate list:
// RCL = { j unvisited : d[last][j] <= dmin + alpha (dmax - dmin) },
// with alpha = alpha_num / alpha_den; the next city is drawn uniformly from the RCL
// (a random number is used only when the RCL has at least two cities)
// the function returns the length of the tour stored in tour[0..n-1]

int RandomizedNN(int n, int** d, int s, int alpha_num, int alpha_den,
                 long long* seed, int* tour)
{
    bool* visited = new bool[n];
    int* RCL = new int[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int last = tour[len - 1];
        int dmin = -1, dmax = -1;          // smallest and largest distance from last
        for (int j = 0; j < n; j++)
        {
            if (visited[j]) continue;
            if (dmin == -1 || d[last][j] < dmin) dmin = d[last][j];
            if (dmax == -1 || d[last][j] > dmax) dmax = d[last][j];
        }
        int size = 0;                      // restricted candidate list
        for (int j = 0; j < n; j++)
        {
            // d <= dmin + alpha (dmax - dmin), in integer arithmetic
            if (!visited[j] && alpha_den * d[last][j] <= alpha_den * dmin + alpha_num * (dmax - dmin))
            {
                RCL[size] = j;
                size++;
            }
        }
        int k = RCL[0];
        if (size > 1)
```

```
        {
            double u = ParkMiller(seed);    // random number in (0,1)
            k = RCL[(int)(u * size)];       // uniform choice in the RCL
        }
        tour[len] = k;
        visited[k] = true;
    }

    delete[] RCL;
    delete[] visited;
    return TourLength(n, d, tour);
}
```

2.14.7 Regret and look-ahead

```
// regret-2 insertion: starting from the subtour (s), insert the city with the largest
// regret = (second cheapest insertion cost) - (cheapest insertion cost),
// ties broken by the smaller cheapest cost, then by the smaller index,
// at its cheapest position
// the function returns the length of the tour stored in tour[0..n-1]

int Regret2Insertion(int n, int** d, int s, int* tour)
{
    bool* in = new bool[n];
    for (int i = 0; i < n; i++)
    {
        in[i] = false;
    }
    tour[0] = s;
    in[s] = true;

    for (int len = 1; len < n; len++)
    {
        int k = -1, regk = 0, c1k = 0, posk = 0;
        for (int q = 0; q < n; q++)
        {
            if (in[q]) continue;
            int c1 = -1, c2 = -1, pos = 0; // cheapest and second cheapest cost
            for (int a = 0; a < len; a++)
            {
                int i = tour[a], j = tour[(a + 1) % len];
                int cost = d[i][q] + d[q][j] - d[i][j];
                if (c1 == -1 || cost < c1)
                {
                    c2 = c1;
                    c1 = cost;
                    pos = a;
                }
                else if (c2 == -1 || cost < c2)
                {
                    c2 = cost;
                }
            }
            if (c2 == -1) c2 = c1;          // a single position: no regret
            int reg = c2 - c1;
            if (k == -1 || reg > regk || (reg == regk && c1 < c1k))
```

```
            {
                k = q;
                regk = reg;
                c1k = c1;
                posk = pos;
            }
        }
        for (int a = len; a > posk + 1; a--)   // insert k after position posk
        {
            tour[a] = tour[a - 1];
        }
        tour[posk + 1] = k;
        in[k] = true;
    }

    delete[] in;
    return TourLength(n, d, tour);
}
```

```
// pilot method with the nearest neighbour heuristic as pilot:
// every candidate next city is evaluated by completing the partial tour
// with nearest neighbour; the candidate with the shortest completed tour is appended
// the function returns the length of the tour stored in tour[0..n-1]

int PilotNN(int n, int** d, int s, int* tour)
{
    bool* visited = new bool[n];
    int* trial = new int[n];                // partial tour completed by the pilot
    bool* used = new bool[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }
    tour[0] = s;
    visited[s] = true;

    for (int len = 1; len < n; len++)
    {
        int k = -1, bestL = -1;
        for (int q = 0; q < n; q++)         // candidate next city q
        {
            if (visited[q]) continue;
            for (int i = 0; i < n; i++) used[i] = visited[i];
            for (int a = 0; a < len; a++) trial[a] = tour[a];
            trial[len] = q;
            used[q] = true;
            for (int t = len + 1; t < n; t++)  // pilot: nearest neighbour completion
            {
                int last = trial[t - 1], next = -1;
                for (int j = 0; j < n; j++)
                {
                    if (!used[j] && (next == -1 || d[last][j] < d[last][next])) next = j;
                }
                trial[t] = next;
```

```
36                  used[next] = true;
37              }
38              int L = TourLength(n, d, trial);
39              if (k == -1 || L < bestL)
40              {
41                  k = q;
42                  bestL = L;
43              }
44          }
45          tour[len] = k;                      // append the best candidate
46          visited[k] = true;
47      }
48
49      delete[] used;
50      delete[] trial;
51      delete[] visited;
52      return TourLength(n, d, tour);
53  }
```

Chapter 3

Local search

Prerequisites

- From this module, the chapter «Heuristic algorithms: definitions»: neighbourhoods, local optima, intensification and diversification.
- From this module, the chapter «Constructive heuristics»: the starting solutions used here.

Objectives

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

- define the neighbourhoods of the KP, the BPP, the TSP and $P\,||\,C_{\max}$;
- execute a local search with the first improvement and the best improvement rules;
- evaluate a move incrementally, in $O(1)$ for the 2-opt move;
- escape a plateau with an auxiliary objective, and handle infeasible solutions with a penalty;
- combine neighbourhoods (variable neighbourhood descent) and restarts (multi-start).

3.1 Local search

Definition 3.1: Local search

Given a neighbourhood N, a **local search** (denoted by LS) is an improvement heuristic. It starts from a feasible solution and, at every iteration, looks in the neighbourhood of the current solution for a *better* solution. If it finds one, it moves to it and repeats; if no neighbour is better, the current solution is a local optimum with respect to N, and the search stops.

- The **neighbourhood** $N(S)$ of a feasible solution S, the **move** from S to a neighbour and the **local optimum** with respect to N are defined in the chapter «Heuristic algorithms: definitions».

A local search improves a feasible solution by moving from a solution to a better neighbour, and it stops in a local optimum with respect to N.

3.2 The problems, the assumptions and the starting solutions

- The algorithms are illustrated on the four problems of the chapter «Heuristic algorithms: definitions»: the 0/1 knapsack problem (KP), the bin packing problem (BPP), the travelling salesman problem (TSP) and $P\,||\,C_{\max}$. Their definitions and assumptions are on their pages; the table recalls the notation.

problem	data	representation of a solution	objective
KP	p_j, w_j $(j \in \mathscr{I})$, capacity c	binary vector x	max profit
BPP	w_j $(j \in \mathscr{I})$, capacity c	vector bin, packing patterns P_b	min number of bins nb
TSP	d_{ij} $(i, j \in \mathscr{V})$	tour σ	min length $L(\sigma)$
$P\,||\,C_{\max}$	t_j $(j \in \mathscr{J})$, m machines	vector $mach$, sets J_h	min makespan $C_{\max}$

- For $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ is the **load** of machine h and $C_{\max} = \max_h L_h$. For the BPP, $L_b = \sum_{j \in P_b} w_j$ is the **load** of bin b. A machine (or a bin) whose load equals the objective value is called **critical**.

- The **reference instances** are those of the chapter «Heuristic algorithms: definitions»: the KP with $p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$; the BPP with $w = (4, 4, 5, 3, 2, 3)$, $c = 7$; the TSP with 5 cities; $P\,||\,C_{\max}$ with $t = (5, 5, 4, 4, 3, 3, 3)$ and $m = 3$. Their optimal values are 17, 3, 18 and 9.

3.2.1 Starting solutions

- A local search needs a feasible **starting solution**. We use the solutions computed in the chapter «Constructive heuristics», recalled below.

problem	algorithm	starting solution	value	$z(I)$
KP	GR (greedy by efficiency)	$x = (1, 0, 1, 0)$, items $\{1, 3\}$	16	17
KP	greedy by weight GR_w	$x = (0, 0, 1, 1)$, items $\{3, 4\}$	10	17
BPP	NF	$\{1\}, \{2\}, \{3\}, \{4, 5\}, \{6\}$	5	3
BPP	FF	$\{1, 4\}, \{2, 5\}, \{3\}, \{6\}$	4	3
BPP	FFD	$\{3, 5\}, \{1, 4\}, \{2, 6\}$	3	3
TSP	NN from city 1	$\sigma = (1, 3, 2, 4, 5)$	25	18
$P\,||\,C_{\max}$	LPT	$J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$	11	9

- The solution of FFD is already optimal, so no local search can improve it: for the BPP we start from the solutions of NF and FF.

- To study the behaviour of a local search on many starting solutions we also use **random starting solutions**. They are generated with the Park–Miller minimal standard generator: given an integer **seed** $s_0 \in \{1, \ldots, 2^{31} - 2\}$, the k-th pseudo-random number is

$$s_k = 16807 \cdot s_{k-1} \bmod (2^{31} - 1), \qquad u_k = \frac{s_k}{2^{31} - 1} \in (0, 1),$$

and a pseudo-random integer in $\{0, 1, \ldots, q-1\}$ is $\lfloor q\, u_k \rfloor$.

- The sequence is completely determined by the seed: the same seed gives the same starting solutions, in the notes, in the exercises and in the code. A **random tour** keeps city 1 in the first position and shuffles the other cities (Fisher–Yates: for $k = n, n-1, \ldots, 3$, the city in position k is exchanged with the city in a random position among $2, \ldots, k$); a **random assignment** of the jobs draws the machine of the jobs $1, 2, \ldots, n$ in this order; a **random BPP solution** is computed by FF on a random order of the items (Fisher–Yates on all the positions: for $k = n, n-1, \ldots, 2$, the item in position k is exchanged with the item in a random position among $1, \ldots, k$).

3.3 Local search: the scheme

- The pseudo-code of LS reads as follows. The function *Improve* explores the neighbourhood and returns an improving neighbour of x, or *NULL* if there is none.

Algorithm

Algorithm: Local search

Input: a feasible solution x^0, a neighbourhood N, a rule $\in \{first, best\}$
Output: a local optimum x with respect to N

```
1 Function LocalSearch(x^0, N, rule)
2     x := x^0                                          // current solution
3     while true do
4         y := Improve(x, N, rule)                      // improving neighbour, or NULL
5         if y = NULL then
6             return x                                  // no improving neighbour: local optimum
7         end
8         x ← y                                         // move to the improving neighbour
9     end
```

- A local search is defined by four ingredients:
  1. the **starting solution** x^0, computed by a constructive heuristic or at random;
  2. the **neighbourhood** N, i.e., the family of moves, which depends on the representation of the solutions;
  3. the **acceptance criterion**: here, a neighbour is accepted only if it is strictly better than the current solution;
  4. the **stopping criterion**: here, the search stops in the first local optimum; in practice a time limit is often added.

- The function f that decides whether a neighbour is "better" is usually the objective of the problem, but it does not have to be: the search may be guided by a different function, as long as the solution it returns is feasible.

Proposition 3.2

For every starting solution, LS stops after a finite number of moves, in a local optimum with respect to N. If the objective function takes integer values and the problem is a minimization problem, the number of moves is at most $f(x^0) - z(I)$; for a maximization problem, at most $z(I) - f(x^0)$.

Proof. Every move strictly improves the objective, hence the sequence of the visited solutions never visits the same solution twice. The set of the feasible solutions is finite, so the sequence is finite. When it stops, *Improve* has returned *NULL*, i.e., no neighbour is better: the last solution is a local optimum. If f takes integer values, every move improves f by at least 1, and f can never go beyond $z(I)$, which gives the bounds on the number of moves. □

- On the TSP reference instance, a local search starting from the tour of NN (length 25) makes at most $25 - 18 = 7$ moves. On large instances this bound is weak, and for some neighbourhoods there are instances on which the number of moves grows exponentially with n (for the 2-opt neighbourhood of the TSP, see Englert, Röglin, Vöcking, *Algorithmica* 68, 2014). In practice, the number of moves is small, and the running time is dominated by the exploration of the neighbourhoods.

- Local search **intensifies**: it moves downhill until it reaches the bottom of the valley in which it started. Which valley, and therefore which local optimum, depends on the starting solution, on the neighbourhood and on the rule used to choose the move.

3.4 Neighbourhoods for the four problems

- The neighbourhoods are defined by small modifications of the representation of the solutions. Only **feasible** neighbours belong to the neighbourhood: a modification that produces an infeasible object is not a move. For every neighbourhood we fix the **order** in which the moves are examined, which matters when the search accepts the first improving neighbour it finds.

3.4.1 KP: flip and swap

- The **flip** neighbourhood, defined in the chapter «Heuristic algorithms: definitions», changes one component of x, i.e., adds or removes one item; it has n elements before the feasibility test. The flips are examined for $j = 1, 2, \ldots, n$.

- The **swap** neighbourhood removes one selected item j and adds one unselected item k. If S is the set of the selected items, it has $|S|\,(n - |S|) \leq n^2/4$ elements; the swaps are examined for $j \in S$ in increasing order and, for each j, for $k \notin S$ in increasing order.

- The two neighbourhoods are complementary: a flip changes the number of selected items, a swap keeps it. The neighbourhood **flip+swap** is their union, with the flips examined first.

3.4.2 BPP: move and swap

- The **move** neighbourhood moves one item j from its bin $a = bin_j$ to another non-empty bin b with $L_b + w_j \leq c$. It has at most $n\,(nb - 1)$ elements; the moves are examined for $j = 1, 2, \ldots, n$ and, for each j, for the bins in increasing order of their label.

- Moving an item into an empty bin is never useful: it cannot decrease the number of bins. The bins keep the labels of the starting solution; a bin emptied by a move simply disappears.

- The **swap** neighbourhood exchanges two items $j < k$ in different bins, provided that both bins remain feasible: $L_{bin_j} - w_j + w_k \leq c$ and $L_{bin_k} - w_k + w_j \leq c$. Exchanging two items of equal weight does not change any load, and is not considered. It has at most $\frac{n(n-1)}{2}$ elements, examined for $j = 1, \ldots, n$ and $k = j + 1, \ldots, n$.

3.4.3 TSP: 2-opt and Or-opt

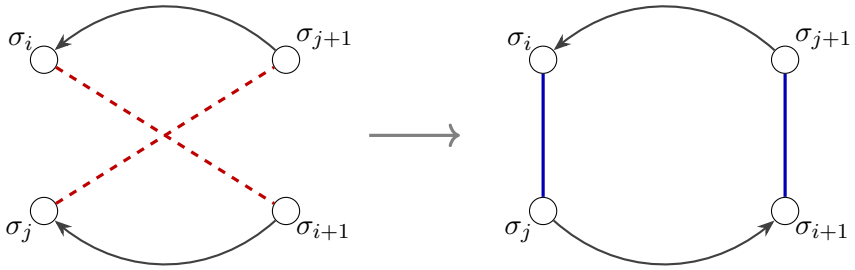
- A tour is a permutation $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$ of the cities, with $\sigma_{n+1} = \sigma_1$. We keep city 1 in the first position, $\sigma_1 = 1$.

- The **2-opt** move with positions $i < j$ removes the two edges $\{\sigma_i, \sigma_{i+1}\}$ and $\{\sigma_j, \sigma_{j+1}\}$, and reconnects the tour in the only other possible way: with the edges $\{\sigma_i, \sigma_j\}$ and $\{\sigma_{i+1}, \sigma_{j+1}\}$, travelling the path $\sigma_{i+1}, \ldots, \sigma_j$ in the opposite direction (Croes, *Operations Research* 6, 1958):

$$(\sigma_1, \ldots, \sigma_i,\ \underbrace{\sigma_{i+1}, \sigma_{i+2}, \ldots, \sigma_j}_{\text{path}},\ \sigma_{j+1}, \ldots, \sigma_n)$$

$$\downarrow$$

$$(\sigma_1, \ldots, \sigma_i,\ \underbrace{\sigma_j, \ldots, \sigma_{i+2}, \sigma_{i+1}}_{\text{reversed path}},\ \sigma_{j+1}, \ldots, \sigma_n).$$

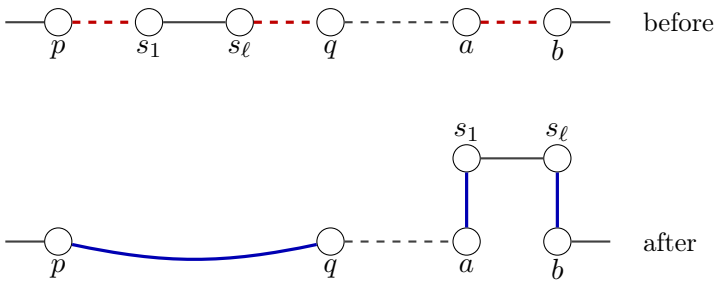


before: the two dashed edges are removed

after: two new edges, the lower path is reversed

- The moves are examined for $i = 1, \ldots, n-2$ and $j = i+2, \ldots, n$, excluding $i = 1$, $j = n$ (the two edges would share city σ_1). The 2-opt neighbourhood has $\frac{n(n-3)}{2}$ elements: 5 for the reference instance.

- The **Or-opt** move (Or, PhD thesis, Northwestern University, 1976) takes a **segment** of $\ell \in \{1, 2, 3\}$ consecutive cities $\sigma_i, \ldots, \sigma_{i+\ell-1}$ ($2 \leq i$, $i + \ell - 1 \leq n$) and inserts it, with the same orientation, between two consecutive cities σ_k and σ_{k+1} outside the segment ($k \notin \{i-1, \ldots, i+\ell-1\}$). Let $p = \sigma_{i-1}$ and $q = \sigma_{i+\ell}$ be the cities before and after the segment, $s_1 = \sigma_i$ and $s_\ell = \sigma_{i+\ell-1}$ its first and last city, $a = \sigma_k$ and $b = \sigma_{k+1}$:



- The moves are examined for $\ell = 1, 2, 3$, then for $i = 2, \ldots, n-\ell+1$, then for $k = 1, \ldots, n$. For every ℓ there are $n - \ell$ segments and $n - \ell - 1$ insertion points, so the neighbourhood has $\sum_{\ell=1}^{3}(n-\ell)(n-\ell-1)$ elements, i.e., $O(n^2)$: 170 for $n = 10$. Some of these moves give the same tour (moving a single city forward by one position is the same as moving its successor backward by one position).

- An Or-opt move removes *three* edges, $\{p, s_1\}$, $\{s_\ell, q\}$ and $\{a, b\}$. The **3-opt** neighbourhood (Lin, *Bell System Technical Journal* 44, 1965) contains all the ways of removing three edges and reconnecting the three paths: it has $O(n^3)$ elements, and Or-opt is the part of it that can be explored in $O(n^2)$ time.

3.4.4 $P\,||\,C_{\max}$: move and swap

- The **move** neighbourhood moves one job j from its machine $a = mach_j$ to another machine $h \neq a$. Every move is feasible, and the neighbourhood has $n\,(m-1)$ elements, examined for $j = 1, \ldots, n$ and, for each j, for $h = 1, \ldots, m$.

- The **swap** neighbourhood exchanges the machines of two jobs $j < k$ on different machines with $t_j \neq t_k$ (otherwise no load changes). It has at most $\frac{n(n-1)}{2}$ elements, examined for $j = 1, \ldots, n$ and $k = j+1, \ldots, n$. A swap never changes the number of jobs on a machine.

3.4.5 Summary

problem	neighbourhood	move	size
KP	flip	add or remove item j	n
KP	swap	item j out, item k in	$\lvert S\rvert(n - \lvert S\rvert) \leq \frac{n^2}{4}$
BPP	move	item j to another bin	$\leq n\,(nb-1)$
BPP	swap	exchange items j and k	$\leq \frac{n(n-1)}{2}$
TSP	2-opt	remove 2 edges, reverse a path	$\frac{n(n-3)}{2}$
TSP	Or-opt	move a segment of 1, 2 or 3 cities	$\sum_{\ell=1}^{3}(n-\ell)(n-\ell-1)$
TSP	3-opt	remove 3 edges, reconnect	$O(n^3)$
$P\,\|\|\,C_{\max}$	move	job j to another machine	$n\,(m-1)$
$P\,\|\|\,C_{\max}$	swap	exchange jobs j and k	$\leq \frac{n(n-1)}{2}$

- The sizes are those *before* the feasibility test (KP, BPP) and include moves that do not change the objective. The size of a neighbourhood is the number of neighbours to evaluate at every iteration, so it determines the cost of an iteration.

3.5 First improvement and best improvement

- The function *Improve* of *LS* can stop at the *first* improving neighbour it finds, or examine the whole neighbourhood and return the *best* neighbour. The pseudo-codes are written for a minimization problem; for a maximization problem, "<" becomes ">".

3.5.1 First improvement

The *first improvement* rule examines the neighbours of the current solution in a fixed order and returns the first neighbour that is better than the current solution. If no neighbour is better, it returns *NULL*.

Algorithm

Algorithm: First improvement

Input: a feasible solution x, a neighbourhood N, an objective function f (to be minimized)

Output: the first neighbour $y \in N(x)$ with $f(y) < f(x)$, or *NULL*

```
1 Function FirstImprovement(x, N, f)
2     for y ∈ N(x), in the order of the neighbourhood do
3         if f(y) < f(x) then
4             return y                  // improving neighbour: accept it at once
5         end
6     end
7     return NULL                                   // x is a local optimum
```

3.5.2 Best improvement

The *best improvement* rule examines all the neighbours of the current solution and returns the best one, if it is better than the current solution; ties are broken in favour of the neighbour examined first. If no neighbour is better, it returns *NULL*.

Algorithm

Algorithm: Best improvement

Input: a feasible solution x, a neighbourhood N, an objective function f (to be minimized)

Output: a best neighbour $y \in N(x)$ if $f(y) < f(x)$, otherwise *NULL*

```
1  Function BestImprovement(x, N, f)
2      ȳ := NULL                      // best improving neighbour found so far
3      f̄ := f(x)                                                  // its value
4      for y ∈ N(x), in the order of the neighbourhood do
5          if f(y) < f̄ then
6              ȳ ← y                  // better than all the neighbours seen so far
7              f̄ ← f(y)
8          end
9      end
10     return ȳ                                  // NULL if x is a local optimum
```

3.5.3 Comparison

- In a local optimum both rules examine the whole neighbourhood: the last iteration costs the same.

- Before reaching it, **first improvement** makes **cheaper** iterations, since it stops at the first improving neighbour, but each move usually improves less, so it tends to make **more** moves. **Best improvement** makes the steepest descent: each iteration costs a full exploration of the neighbourhood, and each move improves as much as possible.

- The two rules follow different paths, so they may stop in **different local optima**. Neither rule is better than the other on all instances: the steepest move now can lead into a worse valley later.

- With first improvement the result depends on the order in which the neighbours are examined; randomizing that order is a simple way to obtain different local optima from the same starting solution.

Example 3.1: First and best improvement on the KP

We start from the solution of the greedy by weight GR_w on the KP reference instance ($c = 9$), i.e., $x = (0, 0, 1, 1)$, of weight 6 and profit 10, with the neighbourhood flip+swap. The neighbours are examined in the order of Section 3.4:

move	neighbour	weight	profit	
flip 1	$(1, 0, 1, 1)$	11	—	infeasible
flip 2	$(0, 1, 1, 1)$	10	—	infeasible
flip 3	$(0, 0, 0, 1)$	3	4	
flip 4	$(0, 0, 1, 0)$	3	6	
swap 3 out, 1 in	$(1, 0, 0, 1)$	8	14	← first improvement
swap 3 out, 2 in	$(0, 1, 0, 1)$	7	11	
swap 4 out, 1 in	$(1, 0, 1, 0)$	8	16	← best improvement
swap 4 out, 2 in	$(0, 1, 1, 0)$	7	13	

First improvement accepts $(1, 0, 0, 1)$, of profit 14, after examining 3 feasible neighbours. From $(1, 0, 0, 1)$ (weight 8) the neighbours are examined again from the beginning:

move	neighbour	weight	profit	
flip 1	$(0, 0, 0, 1)$	3	4	
flip 2	$(1, 1, 0, 1)$	12	—	infeasible
flip 3	$(1, 0, 1, 1)$	11	—	infeasible
flip 4	$(1, 0, 0, 0)$	5	10	
swap 1 out, 2 in	$(0, 1, 0, 1)$	7	11	
swap 1 out, 3 in	$(0, 0, 1, 1)$	6	10	
swap 4 out, 2 in	$(1, 1, 0, 0)$	9	17	← first improvement

and $(1, 1, 0, 0)$, of profit 17, is accepted after 5 feasible neighbours. It is the optimal solution,

so no neighbour is better: the 6 feasible neighbours are examined and the search stops. In total: 2 moves, $3 + 5 + 6 = 14$ feasible neighbours examined.

Best improvement examines all the 6 feasible neighbours of $(0, 0, 1, 1)$ and moves to $(1, 0, 1, 0)$, of profit 16. This is the solution of GR, whose neighbourhood is in the chapter «Heuristic algorithms: definitions»: the best neighbour is $(1, 1, 0, 0)$, of profit 17, reached with the swap "3 out, 2 in". The search stops there after examining its 6 feasible neighbours. In total: 2 moves, $6 + 6 + 6 = 18$ feasible neighbours examined.

The two rules reach the optimum along different paths, $10 \to 14 \to 17$ and $10 \to 16 \to 17$, and first improvement examines fewer neighbours. The flip neighbourhood alone would not have moved at all: from $(0, 0, 1, 1)$ every feasible flip removes an item, so $(0, 0, 1, 1)$, of profit 10, is a local optimum for flip.

3.6 Evaluating a move incrementally

- Every iteration of LS evaluates many neighbours, and most of the running time is spent there. Computing $f(y)$ **from scratch** costs $O(n)$ for each of the four problems: the profit of a KP solution, the loads of the bins or of the machines, the length of a tour are sums of n terms.

- A move changes only a few components of the solution. The **incremental evaluation** computes the **change** $\Delta = f(y) - f(x)$ from the components that change, keeping some information about the current solution (its profit and weight, its loads), without recomputing $f(y)$.

Proposition 3.3

Let σ' be the tour obtained from σ by the 2-opt move with positions $i < j$, and let $u_1 = \sigma_i$, $v_1 = \sigma_{i+1}$, $u_2 = \sigma_j$, $v_2 = \sigma_{j+1}$. Then $L(\sigma') = L(\sigma) + \Delta$, with

$$\Delta \;=\; d_{u_1u_2} + d_{v_1v_2} - d_{u_1v_1} - d_{u_2v_2}.$$

Proof. The tours σ and σ' have the same edges, except that σ contains $\{u_1, v_1\}$ and $\{u_2, v_2\}$ while σ' contains $\{u_1, u_2\}$ and $\{v_1, v_2\}$. The reversed path $v_1, \ldots, u_2$ uses the same edges in the opposite direction, and its length does not change because the distances are symmetric. □

The change of length of a 2-opt move is computed in $O(1)$ time:

$$\Delta \;=\; \underbrace{d_{u_1u_2} + d_{v_1v_2}}_{\text{added edges}} - \underbrace{d_{u_1v_1} + d_{u_2v_2}}_{\text{removed edges}}$$

and the move is improving if and only if $\Delta < 0$.

- For example, on the tour $(1, 3, 2, 4, 5)$ of the reference instance, of length 25, the move with positions $i = 2$, $j = 4$ removes $\{3, 2\}$ and $\{4, 5\}$ and adds $\{3, 4\}$ and $\{2, 5\}$: $\Delta = d_{34} + d_{25} - d_{32} - d_{45} = 4 + 4 - 4 - 7 = -3$. The new tour $(1, 3, 4, 2, 5)$ has length $25 - 3 = 22$, as a direct computation confirms: $2 + 4 + 3 + 4 + 9 = 22$.

- **Why it matters.** The 2-opt neighbourhood has $\frac{n(n-3)}{2}$ moves. With the incremental evaluation an iteration of best improvement costs $O(n^2)$; recomputing every tour from scratch would cost $O(n^3)$. For $n = 1000$ cities, an iteration evaluates 498 500 moves: about $5 \cdot 10^5$ operations instead of about $5 \cdot 10^8$. The reversal of the path, which costs $O(n)$, is performed only once per iteration, for the accepted move.

- The same idea applies to the other neighbourhoods:

move	change of the solution	time
KP, flip of j	profit $\pm p_j$, weight $\pm w_j$	$O(1)$
KP, swap j out, k in	profit $+p_k - p_j$, weight $+w_k - w_j$	$O(1)$
BPP, move j from a to b	$L_a - w_j$, $L_b + w_j$; bin a is emptied iff $L_a = w_j$	$O(1)$
TSP, 2-opt	$d_{u_1u_2} + d_{v_1v_2} - d_{u_1v_1} - d_{u_2v_2}$	$O(1)$
TSP, Or-opt	$d_{pq} + d_{as_1} + d_{s_\ell b} - d_{ps_1} - d_{s_\ell q} - d_{ab}$	$O(1)$
$P \mid\mid C_{\max}$, move j from a to h	$L_a - t_j$, $L_h + t_j$; $C_{\max} = \max\{L_a - t_j,\ L_h + t_j,\ \max_{q \neq a,h} L_q\}$	$O(m)$

- The Or-opt formula follows as for 2-opt: the move removes the edges $\{p, s_1\}$, $\{s_\ell, q\}$, $\{a, b\}$ and adds $\{p, q\}$, $\{a, s_1\}$, $\{s_\ell, b\}$, and the segment keeps its orientation.

- The cost of an iteration of LS is the size of the neighbourhood times the cost of evaluating a move (plus the cost of applying the accepted move, which is paid once per iteration):

problem	neighbourhood	evaluation of a move	cost of an iteration
KP	flip	$O(1)$	$O(n)$
KP	swap	$O(1)$	$O(n^2)$
BPP	move	$O(1)$	$O(n \cdot nb)$
BPP	swap	$O(1)$	$O(n^2)$
TSP	2-opt, Or-opt	$O(1)$	$O(n^2)$
TSP	3-opt	$O(1)$	$O(n^3)$
$P \mid\mid C_{\max}$	move	$O(m)$	$O(nm^2)$
$P \mid\mid C_{\max}$	swap	$O(m)$	$O(n^2m)$

- The incremental evaluation does not change the local optima, nor the path followed by the search: it changes only the time needed to follow it. This is why the design of a local search always starts with two questions: *which moves?* and *how fast can a move be evaluated?*

3.7 TSP: 2-opt and Or-opt

3.7.1 The 2-opt local search

The *2-opt* local search (denoted by $2OPT$) is a local search for the TSP with the 2-opt neighbourhood. At every iteration it evaluates the 2-opt moves of the current tour with the incremental formula, applies an improving move chosen by the first- or the best-improvement rule, and stops when no move shortens the tour.

Algorithm

Algorithm: 2-opt step

Input: n cities, distances $d[1..n][1..n]$, tour $\sigma[1..n]$ with $\sigma[1] = 1$, rule $\in \{first, best\}$
Output: *true* if an improving 2-opt move was applied to σ, *false* otherwise

```
1  Function TwoOptStep(n, d, σ, rule)
2      Δ̄ := 0                                  // best length change found so far
3      (ī, j̄) := (0, 0)                        // best move found so far
4      for i := 1 to n − 2 do
5          for j := i + 2 to n do
6              if i = 1 and j = n then continue   // the two edges are adjacent
7              u1 := σ[i]; v1 := σ[i + 1]          // first removed edge
8              u2 := σ[j]; v2 := σ[j mod n + 1]    // second removed edge
9              Δ := d[u1][u2] + d[v1][v2] − d[u1][v1] − d[u2][v2]
10             if Δ < Δ̄ then
11                 Δ̄ ← Δ; (ī, j̄) ← (i, j)          // improving move
12                 if rule = first then
13                     leave both loops
14                 end
15             end
16         end
17     end
18     if ī = 0 then return false          // no improving move: local optimum
19     reverse the path σ[ī + 1], ..., σ[j̄]          // apply the move
20     return true
```

- In the implementation, *Improve* and the move to the improving neighbour are merged in a single function *Step*, which explores the neighbourhood with the incremental evaluation and, if it finds an improving move, applies it to the current solution and returns *true*; otherwise it returns *false*. *LS* calls *Step* until it returns *false*.

- $2OPT$ is *LocalSearch* with *TwoOptStep*; it never changes $\sigma[1] = 1$, since $i \geq 1$ and the reversed path starts at position $i + 1 \geq 2$.

- An iteration costs $O(n^2)$ time; the reversal of the path costs $O(n)$.

- When the distances are Euclidean, a 2-opt local optimum has no crossing edges: if the edges $\{u_1, v_1\}$ and $\{u_2, v_2\}$ cross at a point X, the triangle inequality gives $d_{u_1u_2} \leq d_{u_1X} + d_{Xu_2}$ and $d_{v_1v_2} \leq d_{v_1X} + d_{Xv_2}$, so $\Delta \leq 0$, with $\Delta < 0$ unless the four cities are on a line. The 2-opt move "uncrosses" the two edges.

Example 3.2: *2OPT* on the TSP reference instance

We start from the tour of *NN* from city 1, $\sigma = (1, 3, 2, 4, 5)$, of length 25. Its 5 moves are

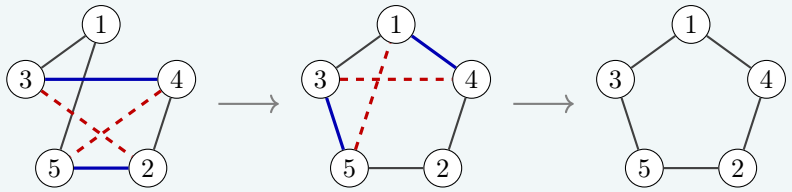
(i, j)	removed edges	added edges	Δ	new tour, length
$(1, 3)$	$\{1, 3\}, \{2, 4\}$	$\{1, 2\}, \{3, 4\}$	$5 + 4 - 2 - 3 = +4$	$(1, 2, 3, 4, 5)$, 29
$(1, 4)$	$\{1, 3\}, \{4, 5\}$	$\{1, 4\}, \{3, 5\}$	$2 + 7 - 2 - 7 = 0$	$(1, 4, 2, 3, 5)$, 25
$(2, 4)$	$\{3, 2\}, \{4, 5\}$	$\{3, 4\}, \{2, 5\}$	$4 + 4 - 4 - 7 = -3$	$(1, 3, 4, 2, 5)$, 22
$(2, 5)$	$\{3, 2\}, \{5, 1\}$	$\{3, 5\}, \{2, 1\}$	$7 + 5 - 4 - 9 = -1$	$(1, 3, 5, 4, 2)$, 24
$(3, 5)$	$\{2, 4\}, \{5, 1\}$	$\{2, 5\}, \{4, 1\}$	$4 + 2 - 3 - 9 = -6$	$(1, 3, 2, 5, 4)$, 19

First improvement accepts the move $(2, 4)$, the first with $\Delta < 0$, after 3 evaluations: $\sigma = (1, 3, 4, 2, 5)$, of length 22. Its moves have $\Delta = +1, +6, +3, -4, 0$; the move $(2, 5)$, which removes $\{3, 4\}$ and $\{5, 1\}$ and adds $\{3, 5\}$ and $\{4, 1\}$, is accepted after 4 evaluations: $\sigma = (1, 3, 5, 2, 4)$, of length 18. All its 5 moves have $\Delta > 0$ $(+7, +4, +1, +4, +6)$: the search stops after $3 + 4 + 5 = 12$ evaluations.

Best improvement accepts the move $(3, 5)$, with $\Delta = -6$: $\sigma = (1, 3, 2, 5, 4)$, of length 19. Its moves have $\Delta = +6, +4, -1, +3, +6$, and the move $(2, 4)$, which removes $\{3, 2\}$ and $\{5, 4\}$ and adds $\{3, 5\}$ and $\{2, 4\}$, gives $\sigma = (1, 3, 5, 2, 4)$, of length 18. The search stops after $5 + 5 + 5 = 15$ evaluations.

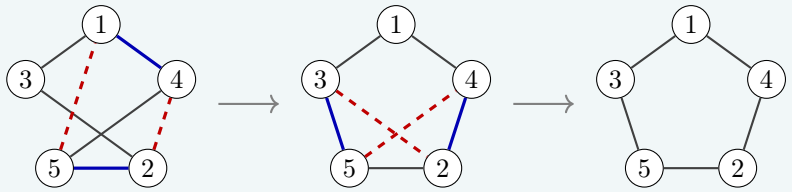
Both rules reach the optimal tour, with 2 moves each. The figure shows the two paths; the cities are placed in the order of the optimal tour, so the optimal tour is the boundary of the pentagon (the instance is not Euclidean: the drawing is only a schematic). Dashed red edges are removed, blue edges are added.

first improvement



$(1, 3, 2, 4, 5)$, 25 $(1, 3, 4, 2, 5)$, 22 $(1, 3, 5, 2, 4)$, 18

best improvement



$(1, 3, 2, 4, 5)$, 25 $(1, 3, 2, 5, 4)$, 19 $(1, 3, 5, 2, 4)$, 18

3.7.2 The Or-opt local search

The *Or-opt* local search (denoted by $OROPT$) is a local search for the TSP with the Or-opt neighbourhood. At every iteration it evaluates, with the incremental formula, the insertion of every segment of 1, 2 or 3 consecutive cities between two other consecutive cities, applies an improving move chosen by the first- or the best-improvement rule, and stops when no move shortens the tour.

Algorithm

Algorithm: Or-opt step

Input: n cities, distances $d[1..n][1..n]$, tour $\sigma[1..n]$ with $\sigma[1] = 1$, rule $\in \{first, best\}$
Output: *true* if an improving Or-opt move was applied to σ, *false* otherwise

Function *OrOptStep*$(n, d, \sigma, rule)$
  $\bar{\Delta} := 0$; $(\bar{\ell}, \bar{\imath}, \bar{k}) := (0, 0, 0)$ `// best move found so far`
  for $\ell := 1$ **to** 3 **do**
    for $i := 2$ **to** $n - \ell + 1$ **do**
      $p := \sigma[i-1]$; $s_1 := \sigma[i]$ `// predecessor, first city of the segment`
      $s_\ell := \sigma[i+\ell-1]$; $q := \sigma[(i+\ell-1) \bmod n + 1]$ `// last city, successor`
      for $k := 1$ **to** n **do**
        if $i - 1 \le k \le i + \ell - 1$ **then continue** `// same place, or inside the segment`
        $a := \sigma[k]$; $b := \sigma[k \bmod n + 1]$ `// edge where the segment goes`
        $\Delta := d[p][q] + d[a][s_1] + d[s_\ell][b] - d[p][s_1] - d[s_\ell][q] - d[a][b]$
        if $\Delta < \bar{\Delta}$ **then**
          $\bar{\Delta} \leftarrow \Delta$; $(\bar{\ell}, \bar{\imath}, \bar{k}) \leftarrow (\ell, i, k)$ `// improving move`
          if *rule* = *first* **then**
            leave the three loops
          end
        end
      end
    end
  end
  if $\bar{\ell} = 0$ **then return** *false* `// no improving move: local optimum`
  move $\sigma[\bar{\imath}], \ldots, \sigma[\bar{\imath} + \bar{\ell} - 1]$ between $\sigma[\bar{k}]$ and $\sigma[\bar{k} \bmod n + 1]$ `// apply`
  return *true*

- Or-opt moves a short piece of the tour to a better place, without reversing anything: it is the natural neighbourhood when a few cities have been "left behind" by the construction. An iteration costs $O(n^2)$ time.

- A 2-opt local optimum need not be an Or-opt local optimum, and vice versa: the two neighbourhoods contain different moves.

3.7.3 First and best improvement on a medium instance

- On 5 cities the 2-opt neighbourhood is small and both rules reach the optimum. To see the difference between the two rules, and the role of Or-opt, we use a medium instance with 10 cities. This instance is not metric: $d_{68} = 8 > d_{67} + d_{78} = 4 + 3$ and $d_{6,10} = 11 > d_{67} + d_{7,10} = 4 + 6$.

Example 3.3: A TSP instance with 10 cities

The cities have integer coordinates, and d_{ij} is the Euclidean distance rounded to the nearest integer:

city	1	2	3	4	5	6	7	8	9	10
coordinates	$(1,1)$	$(3,8)$	$(6,10)$	$(10,9)$	$(13,6)$	$(12,1)$	$(8,3)$	$(5,4)$	$(8,7)$	$(2,5)$

d_{ij}	1	2	3	4	5	6	7	8	9	10
1	—	7	10	12	13	11	7	5	9	4
2	7	—	4	7	10	11	7	4	5	3
3	10	4	—	4	8	11	7	6	4	6
4	12	7	4	—	4	8	6	7	3	9
5	13	10	8	4	—	5	6	8	5	11
6	11	11	11	8	5	—	4	8	7	11
7	7	7	7	6	6	4	—	3	4	6
8	5	4	6	7	8	8	3	—	4	3
9	9	5	4	3	5	7	4	4	—	6
10	4	3	6	9	11	11	6	3	6	—

Among its 181 440 distinct tours, the unique optimal one is $(1, 10, 2, 3, 9, 4, 5, 6, 7, 8)$, of length $z(I) = 39$. The tour of NN from city 1 is $(1, 10, 2, 3, 4, 9, 7, 8, 5, 6)$, of length 49, and the 2-opt neighbourhood has $\frac{10 \cdot 7}{2} = 35$ moves.

First improvement makes 4 moves:

removed edges	new tour	Δ	length	evaluations
$\{3,4\}, \{8,5\}$	$(1, 10, 2, 3, 8, 7, 9, 4, 5, 6)$	-2	47	23
$\{3,8\}, \{6,1\}$	$(1, 10, 2, 3, 6, 5, 4, 9, 7, 8)$	-1	46	25
$\{3,6\}, \{4,9\}$	$(1, 10, 2, 3, 4, 5, 6, 9, 7, 8)$	-3	43	22
$\{6,9\}, \{7,8\}$	$(1, 10, 2, 3, 4, 5, 6, 7, 9, 8)$	-2	41	33

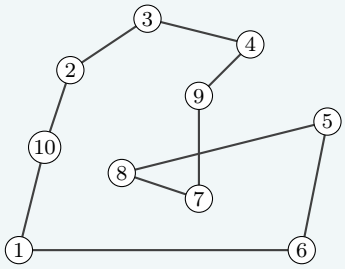
and stops in the local optimum $(1, 10, 2, 3, 4, 5, 6, 7, 9, 8)$, of length 41, after 138 evaluations in total (the last 35 to certify the local optimum).

Best improvement makes 3 moves:

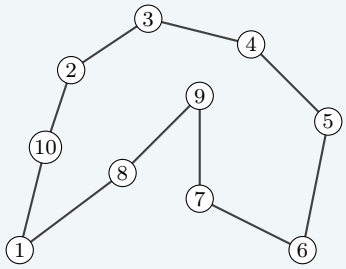
removed edges	new tour	Δ	length
$\{7,8\}, \{6,1\}$	$(1, 10, 2, 3, 4, 9, 7, 6, 5, 8)$	-5	44
$\{9,7\}, \{5,8\}$	$(1, 10, 2, 3, 4, 9, 5, 6, 7, 8)$	-4	40
$\{3,4\}, \{9,5\}$	$(1, 10, 2, 3, 9, 4, 5, 6, 7, 8)$	-1	39

and stops in the optimal tour after $4 \cdot 35 = 140$ evaluations.

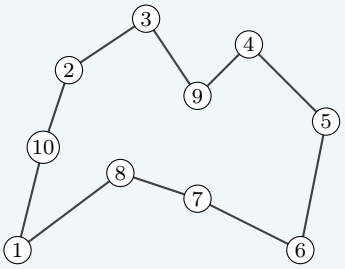
From the same starting tour, the two rules stop in **different local optima**, after almost the same number of evaluations. Among all the 181 440 tours, only 3 are 2-opt local optima, of length 39, 41 and 43.



NN: 49



first improvement: 41



best improvement: 39

Example 3.4: *OROPT* from a 2-opt local optimum

On the instance of Example 3.3, first improvement stopped in the 2-opt local optimum $\sigma = (1, 10, 2, 3, 4, 5, 6, 7, 9, 8)$, of length 41. The Or-opt move that takes the segment (9), with $\ell = 1$, between $p = 7$ and $q = 8$, and inserts it between $a = 3$ and $b = 4$ has

$$\Delta = d_{78} + d_{39} + d_{94} - d_{79} - d_{98} - d_{34} = 3 + 4 + 3 - 4 - 4 - 4 = -2.$$

It is the first improving move found, after 60 evaluations, and also the best of the 170 moves. The new tour $(1, 10, 2, 3, 9, 4, 5, 6, 7, 8)$, of length 39, is optimal. Among all the 181 440 tours, only 3 are Or-opt local optima, of length 39, 43 and 44, and only the optimal tour is a local optimum for both 2-opt and Or-opt.

3.7.4 3-opt and Lin–Kernighan

- The **3-opt** neighbourhood removes three edges and reconnects the three paths in all the possible ways (Lin, 1965). Its local optima are better than those of 2-opt, but an iteration costs $O(n^3)$ time, and the neighbourhood is explored only partially in practice.

- The **Lin–Kernighan** heuristic (Lin, Kernighan, *Operations Research* 21, 1973) builds a *variable* number k of edge exchanges, one at a time, choosing every exchange greedily and stopping when the total gain cannot become positive. It explores part of the k-opt neighbourhood for every k, in a time close to that of 2-opt, and its implementations are among the best heuristics for the TSP.

- The sequence 2-opt, 3-opt, k-opt shows the compromise of the chapter «Heuristic algorithms: definitions»: a larger neighbourhood has better local optima, and costs more to explore.

3.8 $P\,||\,C_{\max}$: move and swap

- The moves of $P\,||\,C_{\max}$ transfer processing time between two machines: a move of job j from machine a to machine b transfers $\delta = t_j$ from a to b; a swap of jobs j (on a) and k (on b) transfers $\delta = t_j - t_k$, which can be negative. In both cases the new loads are $L_a - \delta$ and $L_b + \delta$, and $F(L, a, b, \delta)$ denotes the resulting change of the objective.

3.8.1 The move local search

The *move* local search for $P\,||\,C_{\max}$ examines the reassignments of one job to another machine, applies an improving one chosen by the first- or the best-improvement rule, updates the two loads involved, and stops when no reassignment decreases the objective.

Algorithm

Algorithm: Move step for $P\,||\,C_{\max}$

Input: n jobs, m machines, times $t[1..n]$, assignment $mach[1..n]$, loads $L[1..m]$, rule
Output: *true* if an improving move was applied, *false* otherwise

```
1  Function PCmaxMoveStep(n, m, t, mach, L, rule)
2      Δ̄ := 0; (j̄, h̄) := (0, 0)                          // best move found so far
3      for j := 1 to n do
4          a := mach[j]                                    // machine of job j
5          for h := 1 to m do
6              if h = a then
7                  continue
8              end
9              Δ := F(L, a, h, t[j])                       // change of the objective
10             if Δ < Δ̄ then
11                 Δ̄ ← Δ; (j̄, h̄) ← (j, h)                  // improving move
12                 if rule = first then
13                     leave both loops
14                 end
15             end
16         end
17     end
18     if j̄ = 0 then return false          // no improving move: local optimum
19     L[mach[j̄]] ← L[mach[j̄]] − t[j̄]; L[h̄] ← L[h̄] + t[j̄]   // update the loads
20     mach[j̄] ← h̄                                        // apply the move
21     return true
```

3.8.2 The swap local search

The *swap* local search for $P\,||\,C_{\max}$ examines the exchanges of two jobs on different machines, applies an improving one chosen by the first- or the best-improvement rule, updates the two loads involved, and stops when no exchange decreases the objective.

Algorithm

Algorithm: Swap step for $P\,||\,C_{\max}$

Input: n jobs, m machines, times $t[1..n]$, assignment $mach[1..n]$, loads $L[1..m]$, rule
Output: *true* if an improving swap was applied, *false* otherwise

```
1  Function PCmaxSwapStep(n, m, t, mach, L, rule)
2      Δ̄ := 0; (j̄, k̄) := (0, 0)                          // best swap found so far
3      for j := 1 to n do
4          for k := j + 1 to n do
5              a := mach[j]; b := mach[k]
6              if a = b or t[j] = t[k] then continue      // the loads would not change
7              Δ := F(L, a, b, t[j] − t[k])               // change of the objective
8              if Δ < Δ̄ then
9                  Δ̄ ← Δ; (j̄, k̄) ← (j, k)                  // improving swap
10                 if rule = first then
11                     leave both loops
12                 end
13             end
14         end
15     end
16     if j̄ = 0 then return false          // no improving swap: local optimum
17     a := mach[j̄]; b := mach[k̄]; δ := t[j̄] − t[k̄]
18     L[a] ← L[a] − δ; L[b] ← L[b] + δ                   // update the loads
19     mach[j̄] ← b; mach[k̄] ← a                           // apply the swap
20     return true
```

- For the makespan, the change of the objective is

$$F(L,a,b,\delta) \;=\; \max\Big\{\, L_a-\delta,\; L_b+\delta,\; \max_{q\neq a,b} L_q \Big\} \;-\; \max_{q} L_q,$$

computed in $O(m)$ time from the loads $L_1,\dots,L_m$ of the current solution.

- A move or a swap is improving if and only if $F(L,a,b,\delta)<0$. Since only L_a and L_b change, the makespan can decrease only if a is the **unique** critical machine: if another machine has load $C_{\max}$, no move between two machines can lower the makespan.

- An iteration costs $O(nm^2)$ time for move and $O(n^2m)$ for swap, because of the $O(m)$ evaluation of F; keeping the loads sorted reduces it, but the simple version is enough here.

- A move changes the number of jobs on two machines, a swap keeps the number of jobs on every machine: if a machine has too many jobs, only a move can fix it, and the two neighbourhoods complement each other.

Example 3.5: The solution of *LPT* is a local optimum for move, but not for swap

The solution of *LPT* on the $P\,||\,C_{\max}$ reference instance has $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, loads $(11, 8, 8)$ and $C_{\max} = 11$. Machine 1 is the unique critical machine.

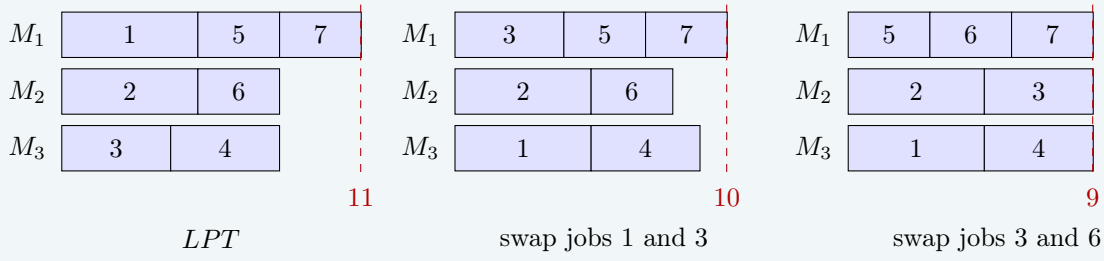
Move. The 14 moves give the loads

move	loads	$C_{\max}$	move	loads	$C_{\max}$
job 1: $M_1 \to M_2$	$(6, 13, 8)$	13	job 4: $M_3 \to M_2$	$(11, 12, 4)$	12
job 1: $M_1 \to M_3$	$(6, 8, 13)$	13	job 5: $M_1 \to M_2$	$(8, 11, 8)$	11
job 2: $M_2 \to M_1$	$(16, 3, 8)$	16	job 5: $M_1 \to M_3$	$(8, 8, 11)$	11
job 2: $M_2 \to M_3$	$(11, 3, 13)$	13	job 6: $M_2 \to M_1$	$(14, 5, 8)$	14
job 3: $M_3 \to M_1$	$(15, 8, 4)$	15	job 6: $M_2 \to M_3$	$(11, 5, 11)$	11
job 3: $M_3 \to M_2$	$(11, 12, 4)$	12	job 7: $M_1 \to M_2$	$(8, 11, 8)$	11
job 4: $M_3 \to M_1$	$(15, 8, 4)$	15	job 7: $M_1 \to M_3$	$(8, 8, 11)$	11

No move decreases the makespan: moving job 1 away from machine 1 makes the receiving machine reach $8 + 5 = 13$, and moving job 5 or 7 lowers machine 1 to 8 but raises the receiving machine to $8 + 3 = 11$. The solution of *LPT* is a **local optimum for move**, with $C_{\max} = 11$.

Swap. The first swap examined exchanges job 1 (on M_1, $t_1 = 5$) and job 3 (on M_3, $t_3 = 4$): $\delta = 1$, the loads become $(10, 8, 9)$ and $C_{\max} = 10$. It is improving, and first improvement accepts it at once; best improvement examines all the 13 swaps and finds two more with $C_{\max} = 10$ (jobs 1 and 4, jobs 1 and 6), so it accepts the same swap, the first one found. Now $J_1 = \{3, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{1, 4\}$.

In the second iteration, the swap of job 3 (on M_1, $t_3 = 4$) and job 6 (on M_2, $t_6 = 3$) has $\delta = 1$ and gives the loads $(9, 9, 9)$: it is found after 9 evaluations, and it is the only swap with $C_{\max} < 10$, so both rules accept it. The new solution $J_1 = \{5, 6, 7\}$, $J_2 = \{2, 3\}$, $J_3 = \{1, 4\}$ has $C_{\max} = 9 = \lceil 27/3 \rceil$: it is **optimal**.



- On this example the move neighbourhood stops at 11, and the swap neighbourhood reaches the optimum 9. As observed for the KP in the chapter «Heuristic algorithms: definitions», **the local optimum depends on the neighbourhood**.

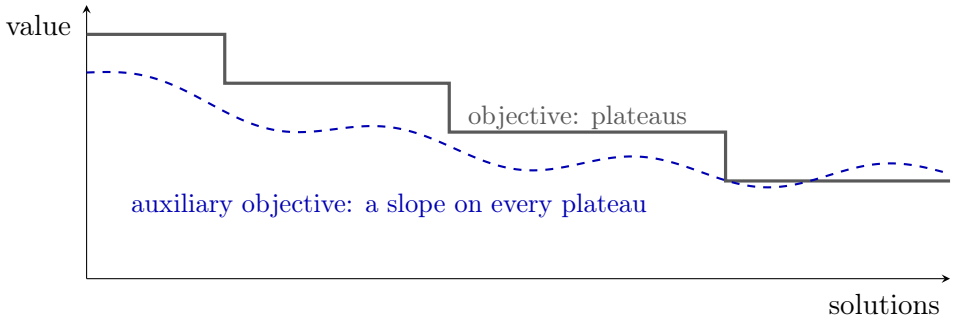
- A larger neighbourhood does not always help. On the instance with $m = 4$ machines and 12 jobs with times $t = (11, 10, 9, 8, 7, 7, 6, 5, 4, 4, 3, 2)$, whose optimal makespan is $z(I) = 19 = \lceil 76/4 \rceil$, LPT gives $J_1 = \{1, 8, 10\}$, $J_2 = \{2, 7, 11\}$, $J_3 = \{3, 6, 12\}$, $J_4 = \{4, 5, 9\}$, with loads $(20, 19, 18, 19)$: this solution is a local optimum for both move and swap.

3.9 Flat landscapes and auxiliary objectives

3.9.1 Plateaus

- The objectives of the BPP and of $P\,||\,C_{\max}$ are very "coarse": the number of bins takes few values, and the makespan depends only on the critical machines. Many solutions have the same value, and most moves do not change it: the landscape is made of large **plateaus**.

- On a plateau a local search that accepts only strictly improving moves stops, even if a sequence of moves of value zero would lead to a better solution. The search is **blind**: the objective does not say which neighbours are closer to an improvement.



- How flat? On the $P\,||\,C_{\max}$ instance with $m = 4$ and 12 jobs of the previous section, we generate 100 random assignments (seed 1) and examine their $100 \cdot 12 \cdot 3 = 3600$ move neighbours: 1290 of them (35.8%) have the *same* makespan as the current solution. For the BPP the situation is extreme: on the instance with $c = 10$ and weights $w = (7, 6, 6, 5, 5, 4, 4, 4, 3, 3, 2, 1)$, for which $z(I) = 5 = \lceil 50/10 \rceil$, the 100 solutions computed by FF on random orders of the items (seed 1) have 2374 move and swap neighbours, and **all** of them have the same number of bins as the current solution.

3.9.2 The auxiliary objective

- The remedy is to guide the search with an **auxiliary objective**, which distinguishes the solutions with the same value and rewards those that are "closer" to an improvement. For both problems it is a function of the loads:

$$\text{BPP:}\quad \max \sum_b L_b^2 = \sum_b \Big(\sum_{j \in P_b} w_j \Big)^2, \qquad P \,||\, C_{\max}: \quad \min \sum_{h=1}^{m} L_h^2.$$

- Both problems redistribute a *fixed* total, $\sum_j w_j$ or $\sum_j t_j$, among the bins or the machines. For a fixed total, the sum of the squares is small when the loads are *balanced* and large when they are *unbalanced*. For $P \,||\, C_{\max}$ balanced loads are exactly what we want; for the BPP we want the opposite: a few full bins and the others empty.

Proposition 3.4

If the amount δ is transferred from the load L_a to the load L_b ($a \neq b$), the sum of the squared loads changes by

$$\big(L_a - \delta\big)^2 + \big(L_b + \delta\big)^2 - L_a^2 - L_b^2 \;=\; 2\,\delta\,\big(L_b - L_a + \delta\big).$$

Proof. Expanding the squares, $(L_a-\delta)^2-L_a^2 = -2\delta L_a+\delta^2$ and $(L_b+\delta)^2-L_b^2 = 2\delta L_b+\delta^2$; the sum is $2\delta(L_b - L_a + \delta)$. The other loads do not change. □

- The change is computed in $O(1)$ time: the auxiliary objective is also cheaper to evaluate than the makespan. For a move, $\delta > 0$ is the weight (or time) of the moved element, and the sum of the squares **increases** if and only if $L_b + \delta > L_a$: the element goes to a load that, after the move, is larger than the load it left.

3.9.3 The move local search for the BPP

The *move* local search for the BPP examines the moves of one item into another non-empty bin where it fits, applies an improving one chosen by the first- or the best-improvement rule, and stops when no move improves. The objective is either the number of bins, $f = nb$, or the auxiliary objective $f = SQ = \sum_b L_b^2$; the **gain** G of a move is the improvement of f.

Algorithm

Algorithm: Move step for the BPP

Input: n items, weights $w[1..n]$, capacity c, bins $bin[1..n]$, loads L, $f \in \{nb, SQ\}$, rule

Output: *true* if an improving move was applied, *false* otherwise

```
1  Function BPPMoveStep(n, w, c, bin, L, f, rule)
2      Ḡ := 0; (j̄, b̄) := (0, 0)                          // best move found so far
3      for j := 1 to n do
4          a := bin[j]                                    // bin of item j
5          for every bin b, in increasing order do
6              if b = a or L[b] = 0 or L[b] + w[j] > c then continue   // same, empty
                or too full
7              if f = SQ then
8                  G := 2 w[j] (L[b] − L[a] + w[j])       // Proposition 3.4
9              else
10                 G := 1 if L[a] = w[j] else 0           // bin a is emptied
11             end
12             if G > Ḡ then
13                 Ḡ ← G; (j̄, b̄) ← (j, b)                // improving move
14                 if rule = first then
15                     leave both loops
16                 end
17             end
18         end
19     end
20     if j̄ = 0 then return false          // no improving move: local optimum
21     L[bin[j̄]] ← L[bin[j̄]] − w[j̄]; L[b̄] ← L[b̄] + w[j̄]   // update the loads
22     bin[j̄] ← b̄                                         // apply the move
23     return true
```

3.9.4 The swap local search for the BPP

The *swap* local search for the BPP examines the exchanges of two items of different weight in different bins that keep both bins feasible, applies an improving one chosen by the first- or the best-improvement rule, and stops when no exchange improves. A swap never changes the number of bins, so it is used only with the auxiliary objective SQ.

Algorithm

Algorithm: Swap step for the BPP

Input: n items, weights $w[1..n]$, capacity c, bins $bin[1..n]$, loads L, rule
Output: *true* if an improving swap was applied, *false* otherwise

```
1  Function BPPSwapStep(n, w, c, bin, L, rule)
2      Ḡ := 0; (j̄, k̄) := (0, 0)                          // best swap found so far
3      for j := 1 to n do
4          for k := j + 1 to n do
5              a := bin[j]; b := bin[k]
6              if a = b or w[j] = w[k] then continue// the loads would not change
7              δ := w[j] − w[k]                          // weight transferred from a to b
8              if L[b] + δ > c or L[a] − δ > c then continue      // a bin would be
                overfull
9              G := 2 δ (L[b] − L[a] + δ)                          // Proposition 3.4
10             if G > Ḡ then
11                 Ḡ ← G; (j̄, k̄) ← (j, k)                        // improving swap
12                 if rule = first then
13                     leave both loops
14                 end
15             end
16         end
17     end
18     if j̄ = 0 then return false          // no improving swap:  local optimum
19     a := bin[j̄]; b := bin[k̄]; δ := w[j̄] − w[k̄]
20     L[a] ← L[a] − δ; L[b] ← L[b] + δ                    // update the loads
21     bin[j̄] ← b; bin[k̄] ← a                               // apply the swap
22     return true
```

3.9.5 Properties of the auxiliary objective

Proposition 3.5

For the BPP, with the neighbourhoods move and swap:

1. a move that empties a bin increases $\sum_b L_b^2$;
2. a swap does not change the number of bins.

Hence every move or swap that decreases the number of bins also increases $\sum_b L_b^2$.

Proof. If the move of item j from bin a to bin b empties bin a, then $L_a = w_j$, and bin b is not empty, so $L_b > 0$. By Proposition 3.4 with $\delta = w_j$ the change is $2w_j(L_b - w_j + w_j) = 2w_j L_b > 0$. A swap exchanges one item of bin a with one item of bin b, so both bins remain non-empty and no bin is created. □

Proposition 3.6

For $P\,||\,C_{\max}$, with the neighbourhoods move and swap:

1. a move or a swap that decreases $\sum_h L_h^2$ does not increase the makespan;
2. a move or a swap that decreases the makespan decreases $\sum_h L_h^2$.

Proof. Let δ be transferred from machine a to machine b; exchanging the names of a and b if necessary, $\delta > 0$. Only L_a and L_b change. (1) By Proposition 3.4, $\sum_h L_h^2$ decreases if and only if $L_b + \delta < L_a$; then both new loads $L_a - \delta$ and $L_b + \delta$ are smaller than $L_a \le C_{\max}$, and the makespan does not increase. (2) If the makespan decreases, machine b, whose load increases, is not critical after the move, so $L_b + \delta < C_{\max}$; machine a must have been the unique critical machine, $L_a = C_{\max}$. Hence $L_b + \delta < L_a$, and $\sum_h L_h^2$ decreases. □

- The two propositions say that, for the auxiliary objective, every move that improves the original objective is improving, and so are some moves that the original objective sees as neutral; for $P\,||\,C_{\max}$, no move that worsens the makespan is improving. For the BPP a move that increases $\sum_b L_b^2$ never increases the number of bins, since no bin is created.

- The auxiliary objective is a guide, not the objective: a solution with more bins can have a larger $\sum_b L_b^2$. With $c = 7$, the loads $(5, 5)$ use 2 bins and give $25 + 25 = 50$, while the loads $(7, 2, 1)$ use 3 bins and give $49 + 4 + 1 = 54$. The quality of the solution returned is always measured with the original objective.

3.9.6 Examples

Example 3.6: The number of bins is blind, the auxiliary objective is not

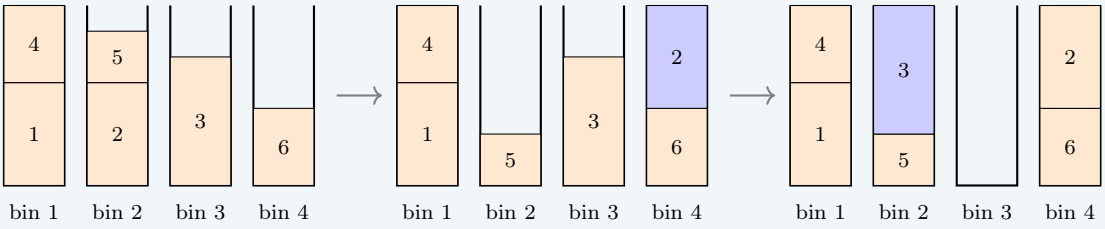
The solution of FF on the BPP reference instance ($c = 7$) has the packing patterns $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, with loads $(7, 6, 5, 3)$, $nb = 4$ and $SQ = 49 + 36 + 25 + 9 = 119$. Its feasible moves are

move	loads	nb	gain $2w_j(L_b - L_a + w_j)$	SQ
item 1 ($w_1 = 4$): bin $1 \to 4$	$(3, 6, 5, 7)$	4	$2 \cdot 4 \cdot (3 - 7 + 4) = 0$	119
item 2 ($w_2 = 4$): bin $2 \to 4$	$(7, 2, 5, 7)$	4	$2 \cdot 4 \cdot (3 - 6 + 4) = 8$	127
item 4 ($w_4 = 3$): bin $1 \to 4$	$(4, 6, 5, 6)$	4	$2 \cdot 3 \cdot (3 - 7 + 3) = -6$	113
item 5 ($w_5 = 2$): bin $2 \to 3$	$(7, 4, 7, 3)$	4	$2 \cdot 2 \cdot (5 - 6 + 2) = 4$	123
item 5 ($w_5 = 2$): bin $2 \to 4$	$(7, 4, 5, 5)$	4	$2 \cdot 2 \cdot (3 - 6 + 2) = -4$	115

With $f = nb$, no move empties a bin: items 3 and 6, alone in their bins, fit nowhere else. The solution of FF is a local optimum with 4 bins, and so it is for the swap neighbourhood, by Proposition 3.5.

With $f = SQ$, both rules accept the move of item 2 into bin 4 (gain 8, $SQ = 127$), which leaves item 5 alone in bin 2. In the next iteration, two moves have the largest gain, 20: item 3 into bin 2, with gain $2 \cdot 5 \cdot (2 - 5 + 5) = 20$, and item 5 into bin 3, with gain $2 \cdot 2 \cdot (5 - 2 + 2) = 20$. Item 3 is examined first, so both rules choose it. It empties bin 3:

the loads become $(7, 7, 7)$, $SQ = 147$, and the packing patterns $\{1, 4\}$, $\{3, 5\}$, $\{2, 6\}$ use 3 bins, which is **optimal**. The first move did not change the number of bins: it prepared the second one.



FF: $nb = 4$, $SQ = 119$ item 2 to bin 4: $SQ = 127$ item 3 to bin 2: $SQ = 147$

Example 3.7: Starting from NF

The solution of NF on the BPP reference instance has the packing patterns $\{1\}$, $\{2\}$, $\{3\}$, $\{4, 5\}$, $\{6\}$ (bins $1, \ldots, 5$), loads $(4, 4, 5, 5, 3)$, $nb = 5$ and $SQ = 91$.

With $f = nb$, the first move examined, item 1 into bin 5 ($3 + 4 = 7$), empties bin 1: the solution has 4 bins, $\{2\}, \{3\}, \{4, 5\}, \{1, 6\}$. It is a local optimum: items 2 and 3 fit in no other bin, and bins 4 and 5 contain two items each. Both rules stop with 4 bins.

With $f = SQ$, best improvement makes three moves:

move	packing patterns	gain	SQ
item 1 into bin 5	$\{2\}, \{3\}, \{4, 5\}, \{1, 6\}$	$2 \cdot 4 \cdot (3 - 4 + 4) = 24$	115
item 4 into bin 2	$\{2, 4\}, \{3\}, \{5\}, \{1, 6\}$	$2 \cdot 3 \cdot (4 - 5 + 3) = 12$	127
item 3 into bin 4	$\{2, 4\}, \{3, 5\}, \{1, 6\}$	$2 \cdot 5 \cdot (2 - 5 + 5) = 20$	147

and reaches an optimal solution with 3 bins, through a move (the second) that does not change the number of bins. First improvement makes the same three moves.

3.9.7 Experiments on two medium instances

- On a single small instance anything can happen; the following experiments repeat the local searches from 100 random starting solutions (seed 1), with best improvement.

- $P \,||\, C_{\max}$ with $m = 4$ and $t = (11, 10, 9, 8, 7, 7, 6, 5, 4, 4, 3, 2)$, $z(I) = 19$. The 100 random assignments have an average makespan of 33.32. The table gives the average makespan of the local optima, and the number of runs that end in an optimal solution:

	$f = C_{\max}$		$f = \sum_h L_h^2$	
neighbourhood	average	optimal	average	optimal
move	20.94	3	20.39	4
swap	24.31	7	23.09	20

- The auxiliary objective improves both neighbourhoods. Swap alone is poor because it cannot change the number of jobs on the machines, which in a random assignment is

often far from balanced.

- BPP with $c = 10$ and $w = (7, 6, 6, 5, 5, 4, 4, 4, 3, 3, 2, 1)$, $z(I) = 5$. The 100 solutions of FF on random orders of the items use 5 bins in 14 cases, 6 bins in 85 cases and 7 bins in 1 case. The table gives the number of runs that end with 5 bins:

neighbourhood	$f = nb$	$f = \sum_b L_b^2$
none (the solutions of FF)	14	14
move	14	76
swap	—	14

- With $f = nb$ the move local search does not improve any of the 100 solutions: they are all on a plateau. With the auxiliary objective, 62 more runs reach the optimum. Swap alone cannot change the number of bins, but it can prepare moves that do.

3.10 KP: feasibility and penalties

- The representation of the KP by a binary vector x describes also infeasible objects, whose weight exceeds the capacity. There are two ways of dealing with them:
  1. **stay feasible**: the neighbourhood contains only the feasible neighbours, as done so far;
  2. **penalize**: the infeasible vectors are allowed, and the search maximizes a function that subtracts from the profit a penalty proportional to the excess weight.

- Staying feasible can make the search short-sighted. In the chapter «Heuristic algorithms: definitions», $x = (1, 0, 1, 0)$ is a local optimum for flip: to reach the optimal solution $(1, 1, 0, 0)$ with flips one has to add item 2 and remove item 3, and the intermediate vector is either $(1, 1, 1, 0)$, infeasible, or $(1, 0, 0, 0)$, of profit $10 < 16$. The swap neighbourhood solves this case by performing the two flips at once; penalties solve it by letting the search walk through the infeasible vector.

- With penalties, the search space is the whole set $\{0, 1\}^n$, and the function to maximize is

$$g_\alpha(x) = \sum_{j \in \mathscr{I}} p_j x_j - \alpha \cdot \max\Big\{0, \sum_{j \in \mathscr{I}} w_j x_j - c\Big\},$$

where $\alpha > 0$ is the **penalty** for each unit of excess weight. For a feasible x, $g_\alpha(x)$ is the profit of x.

3.10.1 Staying feasible

The *flip+swap* local search for the KP examines the flips and then the swaps of the current solution, discards those that exceed the capacity, applies an improving one chosen by the first- or the best-improvement rule, and stops when no feasible neighbour has a larger profit.

Algorithm

Algorithm: Flip+swap step for the KP

Input: n items, profits $p[1..n]$, weights $w[1..n]$, capacity c, feasible solution $x[1..n]$, rule

Output: *true* if an improving move was applied to x, *false* otherwise

```
1  Function KPFlipSwapStep(n, p, w, c, x, rule)
2      W := Σ_{j=1}^{n} w[j] x[j]                          // weight of the current solution
3      Δ̄ := 0; (j̄, k̄) := (0, 0)                  // best move found so far (k̄ = 0: flip)
4      for j := 1 to n do                                          // flip of item j
5          Δp := (1 − 2x[j]) p[j]; Δw := (1 − 2x[j]) w[j]   // remove, or add, item j
6          if W + Δw ≤ c and Δp > Δ̄ then
7              Δ̄ ← Δp; (j̄, k̄) ← (j, 0)
8              if rule = first then
9                  leave all the loops
10             end
11         end
12     end
13     for j := 1 to n with x[j] = 1 do                    // swap: j out, k in
14         for k := 1 to n with x[k] = 0 do
15             Δp := p[k] − p[j]; Δw := w[k] − w[j]
16             if W + Δw ≤ c and Δp > Δ̄ then
17                 Δ̄ ← Δp; (j̄, k̄) ← (j, k)
18                 if rule = first then
19                     leave all the loops
20                 end
21             end
22         end
23     end
24     if j̄ = 0 then return false                  // no improving move: local optimum
25     x[j̄] ← 1 − x[j̄]                        // flip, or remove the item that goes out
26     if k̄ ≠ 0 then
27         x[k̄] ← 1                                        // add the item that comes in
28     end
29     return true
```

- With the first-improvement rule, if an improving flip is found the swaps are not examined. Only flips that *add* an item can improve the profit; flips that remove an item are kept in the neighbourhood for uniformity. An iteration costs $O(n^2)$ time, because of the swaps.

3.10.2 Penalizing the excess weight

The *penalty flip* local search for the KP examines the flips of the current vector, feasible or not, evaluates them with g_α, applies an improving one chosen by the first- or the best-improvement rule, and stops when no flip increases g_α.

Algorithm

Algorithm: Penalty flip step for the KP

Input: n items, profits $p[1..n]$, weights $w[1..n]$, capacity c, penalty α, vector $x[1..n]$, rule

Output: *true* if an improving flip was applied to x, *false* otherwise

Function *KPPenaltyFlipStep*$(n, p, w, c, \alpha, x, rule)$
  $P := \sum_{j=1}^{n} p[j]\,x[j]$; $W := \sum_{j=1}^{n} w[j]\,x[j]$ // profit and weight of x
  $g := P - \alpha \max\{0, W - c\}$ // value of x
  $\bar{\Delta} := 0$; $\bar{\jmath} := 0$ // best flip found so far
  for $j := 1$ **to** n **do**
    $P' := P + (1 - 2x[j])\,p[j]$; $W' := W + (1 - 2x[j])\,w[j]$ // flip of item j
    $\Delta := P' - \alpha \max\{0, W' - c\} - g$ // change of g_α
    if $\Delta > \bar{\Delta}$ **then**
      $\bar{\Delta} \leftarrow \Delta$; $\bar{\jmath} \leftarrow j$ // improving flip
      if *rule* = *first* **then**
        leave the loop
      end
    end
  end
  if $\bar{\jmath} = 0$ **then return** *false* // no improving flip: local optimum
  $x[\bar{\jmath}] \leftarrow 1 - x[\bar{\jmath}]$ // apply the flip
  return *true*

- The local optimum may be infeasible. The search must therefore remember the **best feasible solution** visited, which is the solution it returns.

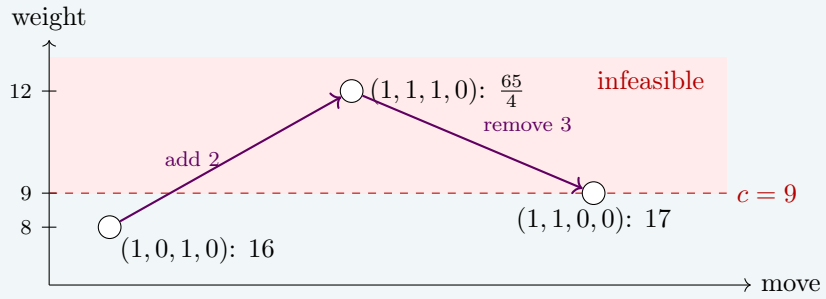
Example 3.8: Penalties on the KP reference instance

We start from the solution of GR, $x = (1, 0, 1, 0)$, of weight 8 and profit 16, which is a local optimum for the flip neighbourhood restricted to feasible solutions. The values g_α of its four flips are

flip	vector	weight	excess	$\alpha = 1$	$\alpha = 2$	$\alpha = \frac{9}{4}$	$\alpha = 3$
1	$(0, 0, 1, 0)$	3	0	6	6	6	6
2	$(1, 1, 1, 0)$	12	3	20	17	$\frac{65}{4}$	14
3	$(1, 0, 0, 0)$	5	0	10	10	10	10
4	$(1, 0, 1, 1)$	11	2	18	16	$\frac{31}{2}$	14

The two rules make the same moves:

- $\alpha = 1$: the search adds item 2 ($g_1 = 20$), then item 4 ($g_1 = 27 - 6 = 21$), and stops in $(1, 1, 1, 1)$, of weight 15: every flip of it removes one item and gives 16, 18, 18 or 20. The penalty is too small: the local optimum is infeasible, and the best feasible solution visited is the starting one, of profit 16.
- $\alpha = 2$: the search adds item 2 ($g_2 = 17 > 16$) and stops in $(1, 1, 1, 0)$, infeasible: removing item 3 gives $(1, 1, 0, 0)$ with $g_2 = 17$, not strictly better.
- $\alpha = \frac{9}{4}$: the search adds item 2 ($g = \frac{65}{4} = 16.25 > 16$) and then removes item 3, reaching $(1, 1, 0, 0)$, of profit 17: the **optimal** solution, through an infeasible vector (see the figure).
- $\alpha = 3$: no flip improves, and the search stops at once in $(1, 0, 1, 0)$, as the feasible search.



Proposition 3.7

Let the data be integer and let $\alpha > \max_{j \in \mathscr{I}} p_j$. Then:

1. from a feasible vector, no flip that produces an infeasible vector increases g_α;
2. every infeasible vector has a flip that increases g_α.

Proof. (1) If adding item k to a feasible vector produces an excess $e \geq 1$, then g_α changes by $p_k - \alpha e \leq p_k - \alpha < 0$. (2) Let x be infeasible, with excess $e \geq 1$, and let j be a selected item. Removing j decreases the excess by $\min\{w_j, e\} \geq 1$, so g_α changes by $-p_j + \alpha \min\{w_j, e\} \geq -p_j + \alpha > 0$. □

- With a **large** penalty the local optima are feasible, but the search never leaves the feasible region; with a **small** penalty it crosses the infeasible region, but may stop there. The useful values depend on the instance (here $\frac{9}{4}$ works, 1, 2 and 3 do not), so in practice α is adjusted during the search: increased while the search stays among infeasible vectors, decreased while it stays among feasible ones.

3.11 Variable neighbourhood descent

- The examples of the previous sections show that a local optimum for one neighbourhood is often not a local optimum for another: the solution of LPT for move and swap, the 2-opt local optimum of length 41 for 2-opt and Or-opt. The natural idea is to use **several neighbourhoods** $N_1, \dots, N_K$, and to stop only when none of them contains an improving neighbour.

The *variable neighbourhood descent* (denoted by VND) explores the neighbourhoods $N_1, \dots, N_K$ in this order. It looks for an improving move in the current neighbourhood N_k: if it finds one, it applies it and goes back to N_1; otherwise it passes to N_{k+1}. It stops when the current solution is a local optimum for all the K neighbourhoods.

Algorithm

Algorithm: Variable neighbourhood descent

Input: a feasible solution x, the steps $Step_1, \dots, Step_K$ of K neighbourhoods, a rule
Output: a local optimum x for all the K neighbourhoods

```
1  Function VND(x, Step_1, ..., Step_K, rule)
2      k := 1                                          // current neighbourhood
3      while k ≤ K do
4          if Step_k(x, rule) then
5              k ← 1                  // improving move applied: back to the first
                neighbourhood
6          else
7              k ← k + 1              // local optimum for N_k: next neighbourhood
8          end
9      end
10     return x
```

- VND stops after a finite number of moves, by the argument of Proposition 3.2. When it stops, $Step_K$ has failed and, since the last improving move, all of $Step_1, \dots, Step_{K-1}$ have failed on the same solution: the solution returned is a local optimum for every N_k, i.e., for the union $N_1 \cup \dots \cup N_K$.

- The neighbourhoods are ordered from the **smallest and cheapest** to the largest: the expensive neighbourhoods are explored only when the cheap ones are exhausted. VND is the deterministic part of the variable neighbourhood search of Mladenović and Hansen (*Computers & Operations Research* 24, 1997).

Example 3.9: VND on $P||C_{\max}$ with move and swap

On the reference instance, from the solution of LPT, with N_1 = move and N_2 = swap, and the makespan as objective:

k	result of $Step_k$	loads	next
1	no improving move	$(11, 8, 8)$	$k = 2$
2	swap jobs 1 and 3	$(10, 8, 9)$	$k = 1$
1	no improving move	$(10, 8, 9)$	$k = 2$
2	swap jobs 3 and 6	$(9, 9, 9)$	$k = 1$
1	no improving move	$(9, 9, 9)$	$k = 2$
2	no improving swap	$(9, 9, 9)$	stop

In the solution with loads $(10, 8, 9)$ the best move gives makespan 11 (job 5 or job 7 from machine 1 to machine 2), so move is again exhausted, and swap finds the optimum.

Example 3.10: VND on the TSP with 2-opt and Or-opt

On the instance with 10 cities of Example 3.3, from the tour of NN (length 49), with N_1 = 2-opt, N_2 = Or-opt and the first-improvement rule, VND makes the four 2-opt moves of the 2-opt local search ($49 \to 47 \to 46 \to 43 \to 41$), then the Or-opt move of Example 3.4, which inserts city 9 between cities 3 and 4 ($41 \to 39$), and stops: the optimal tour is a local optimum for both neighbourhoods. With the best-improvement rule, 2-opt alone already reaches the optimum.

- On the two medium instances of Section 3.9, with 100 random starting solutions (seed 1) and best improvement, VND with N_1 = move and N_2 = swap gives:

instance	objective	runs ending in an optimum	average value
$P \,\|\|\, C_{\max}$, $m = 4$, $n = 12$	$C_{\max}$	15	20.62
$P \,\|\|\, C_{\max}$, $m = 4$, $n = 12$	$\sum_h L_h^2$	68	19.32
BPP, $c = 10$, $n = 12$	$\sum_b L_b^2$	89	5.11

- Compared with the single neighbourhoods of Section 3.9, VND with the auxiliary objective reaches the optimum in 68 runs out of 100 for $P \,||\, C_{\max}$ (move alone: 4) and in 89 for the BPP (move alone: 76). Combining neighbourhoods and using the auxiliary objective are independent improvements, and they add up.

3.12 Multi-start local search

- Local search stops in the local optimum of the valley where it starts. The simplest way to explore other valleys is to **start again** from a different solution, and keep the best local optimum.

The *multi-start local search* (denoted by $MSLS$) generates M starting solutions, at random or with a randomized constructive heuristic, runs a local search from each of them, independently, and returns the best local optimum found.

Algorithm

Algorithm: Multi-start 2-opt

Input: n cities, distances d, number of starts M, seed s of the generator, rule
Output: the best 2-opt local optimum $\bar{\sigma}$ found, and its length $\bar{L}$

```
1  Function MultiStartTwoOpt(n, d, M, s, rule)
2      L̄ := +∞                          // length of the best tour found so far
3      for r := 1 to M do
4          σ := RandomTour(n, s)        // random starting tour; updates the seed
5          while TwoOptStep(n, d, σ, rule) do
               // repeat until σ is a 2-opt local optimum
6          end
7          if L(σ) < L̄ then
8              L̄ ← L(σ); σ̄ ← σ                    // new best local optimum
9          end
10     end
11     return (σ̄, L̄)
```

- The M local searches are **independent**: they can run in parallel, and the result does not depend on their order. Multi-start is the simplest form of **diversification**.

Example 3.11: $MSLS$ on the TSP with 10 cities

On the instance of Example 3.3, 100 random tours (seed 1), each improved by a local search, end in the following local optima:

neighbourhood	rule	runs ending in the optimum 39	other local optima (runs)
2-opt	first	46	41 (54)
2-opt	best	87	41 (8), 43 (5)
Or-opt	first	72	43 (26), 44 (2)
Or-opt	best	77	43 (22), 44 (1)
VND (2-opt, Or-opt)	first	100	—
VND (2-opt, Or-opt)	best	100	—

With 2-opt and first improvement, the first random tour ends in the local optimum of length

41 and the third one in the optimum. On this instance, best improvement is much better than first improvement, and VND always finds the optimum, the only tour that is a local optimum for both neighbourhoods.

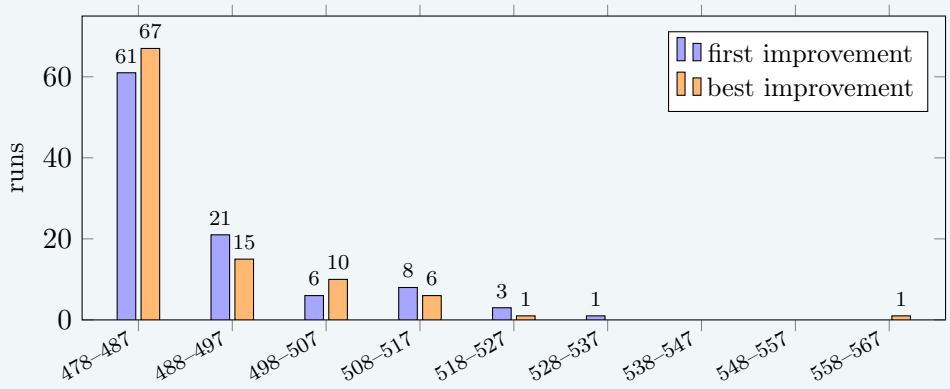
Example 3.12: $MSLS$ on the TSP with 30 cities

The 30 cities have coordinates $(\lfloor 101u_{2k-1} \rfloor, \lfloor 101u_{2k} \rfloor)$, $k = 1, \ldots, 30$, where $u_1, u_2, \ldots$ are generated with seed 12345, and d_{ij} is the Euclidean distance rounded to the nearest integer. The optimal tour, computed with a MIP solver, has length $z(I) = 478$. The best of the 30 tours of NN (one for each starting city) has length 537.

The 100 random tours (seed 1) have an average length of 1768.73 (between 1364 and 2080). After the 2-opt local search:

	first improvement	best improvement
distinct local optima	59	54
runs ending in the optimum	14	12
average length (relative error)	488.45 (2.2%)	487.24 (1.9%)
worst length (relative error)	537 (12.3%)	567 (18.6%)
first run that finds the optimum	2	9

The local optima are distributed as follows (number of runs per class of length):



Local search turns tours of length about 1769 into tours within 2% of the optimum on average; but every run ends in its own valley, and only about one run in eight reaches the optimum.

3.12.1 The limits of multi-start

- **No memory.** Each run starts from scratch and ignores what the previous runs found. If a run reaches the optimum with probability q, independently of the others, M runs all miss it with probability $(1-q)^M$. With $q = 0.12$, as in the last example with best improvement, 10 runs miss the optimum with probability $0.88^{10} \approx 0.28$, and 37 runs are needed to bring this probability below 1%.

- **The probability q shrinks.** The number of local optima grows very quickly with n: on 10 cities only 3 tours are 2-opt local optima, on 30 cities 100 runs find more than 50 distinct ones. On large instances a random start almost never reaches the optimum, and $(1-q)^M$ is close to 1 for every reasonable M.

- **Most of the time is spent descending.** A random start is far from the good solutions, and the local search spends most of its moves on the descent from it; a good constructive heuristic gives better starting solutions, but few different ones.

- A better strategy would **use the local optima already found**: restart *near* a good local optimum rather than anywhere, or continue the search from it by accepting some worse moves. These are the ideas of the metaheuristics.

3.12.2 What local search gives, and what it does not

- Local search improves a feasible solution until it reaches a local optimum: a better **primal bound**, often much better than the starting one.

- It does not give a dual bound, and the fact that it stopped does not mean that it arrived. On the 30-city instance we know that 478 is optimal only because a MIP solver proved it; the local search alone could not tell the local optimum of length 479, found in many runs, from the optimum.

A local optimum is not a better bound: it is a feasible solution, i.e., one side of the gap. Its quality can be assessed only against a dual bound computed in another way.

3.13 Typical mistakes

Warning

- **Forgetting that a local optimum depends on the neighbourhood.** The LPT solution is a local optimum for move, but not for swap.
- **Using the number of bins as the only objective.** It is blind: most neighbours have the same value, and the search stops on a plateau; the auxiliary objective SQ tells them apart.
- **Ignoring the order of examination.** The moves are examined in a fixed order and ties go to the first one examined (in Example 3.6 item 3 comes before item 5).
- **Computing the wrong Δ for a 2-opt move.** Only the two removed and the two added edges count; the two edges cannot share a city.
- **Concluding that a local optimum is optimal.** Only a dual bound can say so.

3.14 In summary

Summary

- A local search moves to a better neighbour until none exists: it returns a local optimum, a better primal bound.
- First improvement is faster per step, best improvement takes the largest gain; both stop in a local optimum of the neighbourhood.
- Incremental evaluation, auxiliary objectives and penalties make the search fast and able to cross plateaus and infeasible regions.
- Variable neighbourhood descent and multi-start enlarge what the search can reach, but the restarts have no memory.

Dashboard of the reference instances. Starting solution and local optimum:

problem	start	local search	$z(I)$
TSP	25 (NN)	18 (2-opt)	18
$P \mid\mid C_{\max}$	11 (LPT)	9 (swap, $=$ LB(I))	9
BPP	4 (FF)	3 (move, SQ)	3

The dual bounds come from the chapter «Heuristic algorithms: definitions»: the local search alone does not certify its result.

3.15 C++ implementation

- The code is in the folder `LOCAL_SEARCH_CODE`: one function per file in `src/`, the declarations in `src/LocalSearch.h`. The command `make` builds the program, and `make test` runs all the examples of these notes and all the instances of the exercises, and compares the results (the values along every trajectory, and the statistics of the experiments) with those computed by an independent engine.

- In this section we use 0-based indexing for items, jobs, machines, bins and cities, and city 0 stays in the first position of the tour. The algorithmic logic is unchanged. The constants `FIRST`, `BEST` select the rule, `CMAX`, `SQ`, `NB` the objective; `Vec` is `std::vector<int>` and `Mat` a vector of `Vec`.

- The generic **local search** and the **variable neighbourhood descent** receive the steps as functions without arguments, which act on the current solution:

```cpp
// generic local search: Step looks for an improving neighbour of the current solution
// and, if it finds one, moves to it and returns true
// the function returns the number of moves performed

int LocalSearch(const std::function<bool()>& Step)
{
    int moves = 0;                          // number of moves performed
    while (Step())                          // an improving neighbour was found and accepted
    {
        moves = moves + 1;
    }
    return moves;                           // the current solution is a local optimum
}
```

```cpp
// variable neighbourhood descent: Steps[0], ..., Steps[K-1] explore the neighbourhoods N_1, ..., N_K
// after every improving move the search restarts from the first neighbourhood
// the function returns the number of moves performed

int VND(const std::vector<std::function<bool()>>& Steps)
{
    int K = (int)Steps.size();              // number of neighbourhoods
    int k = 0;                              // current neighbourhood
    int moves = 0;                          // number of moves performed
    while (k < K)
    {
        if (Steps[k]())                     // improving neighbour found in N_k, and accepted
        {
            moves = moves + 1;
            k = 0;                          // back to the first neighbourhood
```

```
        }
        else
        {
            k = k + 1;                    // local optimum for N_k: try the next neighbourhood
        }
    }
    return moves;                         // local optimum for all the K neighbourhoods
}
```

- For example, the 2-opt local search on a tour `pi`, and VND with 2-opt and Or-opt, are

```
LocalSearch([&]() { return TwoOptStep(n, d, pi, BEST); });
VND({ [&]() { return TwoOptStep(n, d, pi, FIRST); },
      [&]() { return OrOptStep(n, d, pi, FIRST); } });
```

- The **2-opt step**:

```
// one step of the 2-opt local search on the tour pi[0..n-1] (pi[0] never moves)
// the move (i, j) removes the edges {pi[i], pi[i+1]} and {pi[j], pi[j+1]},
// reverses the path pi[i+1..j] and adds the edges {pi[i], pi[j]} and {pi[i+1], pi[j+1]}

bool TwoOptStep(int n, const Mat& d, Vec& pi, int rule)
{
    int bestDelta = 0;                    // best length change found so far
    int bi = -1, bj = -1;                 // best move found so far
    bool stop = false;
    for (int i = 0; i <= n - 3 && !stop; ++i)
    {
        for (int j = i + 2; j <= n - 1; ++j)
        {
            if (i == 0 && j == n - 1) continue;          // the two edges are adjacent
            int u1 = pi[i], v1 = pi[i + 1];               // first removed edge
            int u2 = pi[j], v2 = pi[(j + 1) % n];         // second removed edge
            int delta = d[u1][u2] + d[v1][v2] - d[u1][v1] - d[u2][v2];
            if (delta < bestDelta)
            {
                bestDelta = delta; bi = i; bj = j;        // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;           // no improving move: local optimum

    for (int a = bi + 1, b = bj; a < b; ++a, --b)        // reverse pi[bi+1..bj]
    {
        int tmp = pi[a]; pi[a] = pi[b]; pi[b] = tmp;
```

```
    }
    return true;
}
```

- The **Or-opt step**:

```
// one step of the Or-opt local search on the tour pi[0..n-1] (pi[0] never moves)
// the move (l, i, k) takes the segment pi[i..i+l-1] of l = 1, 2, 3 consecutive cities
// and inserts it, with the same orientation, between pi[k] and pi[k+1]

bool OrOptStep(int n, const Mat& d, Vec& pi, int rule)
{
    int bestDelta = 0;                  // best length change found so far
    int bl = -1, bi = -1, bk = -1;      // best move found so far
    bool stop = false;
    for (int l = 1; l <= 3 && !stop; ++l)
    {
        for (int i = 1; i + l - 1 <= n - 1 && !stop; ++i)
        {
            int p = pi[i - 1], s1 = pi[i];                // predecessor and first city of the segment
            int sl = pi[i + l - 1], q = pi[(i + l) % n];  // last city of the segment and successor
            for (int k = 0; k <= n - 1; ++k)
            {
                if (k >= i - 1 && k <= i + l - 1) continue;   // same position, or inside the segment
                int a = pi[k], b = pi[(k + 1) % n];           // edge where the segment is inserted
                int delta = d[p][q] + d[a][s1] + d[sl][b] - d[p][s1] - d[sl][q] - d[a][b];
                if (delta < bestDelta)
                {
                    bestDelta = delta; bl = l; bi = i; bk = k;   // improving move
                    if (rule == FIRST) { stop = true; break; }
                }
            }
        }
    }
    if (bl == -1) return false;         // no improving move: local optimum

    Vec newpi;                          // the new tour
    for (int k = 0; k <= n - 1; ++k)
    {
        if (k >= bi && k <= bi + bl - 1) continue;        // the segment is copied after pi[bk]
        newpi.push_back(pi[k]);
        if (k == bk)
            for (int r = bi; r <= bi + bl - 1; ++r) newpi.push_back(pi[r]);
    }
    pi = newpi;
    return true;
}
```

- The change F of the objective of $P\,||\,C_{\max}$, for the makespan and for the sum of the squared loads, and the **move** and **swap steps** for $P\,||\,C_{\max}$:

```
// change of the objective of P||Cmax when the time delta is transferred from machine a to machine b
// (a move transfers delta = t_j, a swap transfers delta = t_j - t_k)
// obj = CMAX: change of the makespan; obj = SQ: change of the sum of the squared loads

int ObjectiveChange(int m, const Vec& L, int a, int b, int delta, int obj)
{
    if (obj == SQ)
        return 2 * delta * (L[b] - L[a] + delta);

    int Cmax = 0, newC = 0;                                  // makespan before and after
    for (int q = 0; q < m; ++q)
    {
        if (L[q] > Cmax) Cmax = L[q];
        int newL = L[q];
        if (q == a) newL = L[q] - delta;
        if (q == b) newL = L[q] + delta;
        if (newL > newC) newC = newL;
    }
    return newC - Cmax;
}
```

```
// one step of the move local search for P||Cmax
// mach[j] is the machine of job j, L[h] the load of machine h
// obj = CMAX (makespan) or SQ (sum of the squared loads); both are minimized

bool PCmaxMoveStep(int n, int m, const Vec& t, Vec& mach, Vec& L, int obj, int rule)
{
    int bestDelta = 0;                                       // best change of the objective found so far
    int bj = -1, bh = -1;                                    // best move found so far
    bool stop = false;
    for (int j = 0; j < n && !stop; ++j)
    {
        int a = mach[j];                                     // machine of job j
        for (int h = 0; h < m; ++h)
        {
            if (h == a) continue;
            int delta = ObjectiveChange(m, L, a, h, t[j], obj);   // job j from a to h
            if (delta < bestDelta)
            {
                bestDelta = delta; bj = j; bh = h;           // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bj == -1) return false;                              // no improving move: local optimum
```

```
25
26      L[mach[bj]] = L[mach[bj]] - t[bj];                  // apply the move
27      L[bh] = L[bh] + t[bj];
28      mach[bj] = bh;
29      return true;
30  }
```

```
1  // one step of the swap local search for P||Cmax
2  // the move (j, k) exchanges the machines of the jobs j and k
3  // obj = CMAX (makespan) or SQ (sum of the squared loads); both are minimized
4
5  bool PCmaxSwapStep(int n, int m, const Vec& t, Vec& mach, Vec& L, int obj, int
       rule)
6  {
7       int bestDelta = 0;                                  // best change of the
       objective found so far
8       int bj = -1, bk = -1;                               // best move found so
       far
9       bool stop = false;
10      for (int j = 0; j < n && !stop; ++j)
11      {
12          for (int k = j + 1; k < n; ++k)
13          {
14              int a = mach[j], b = mach[k];
15              if (a == b || t[j] == t[k]) continue;       // the loads would not
       change
16              int delta = ObjectiveChange(m, L, a, b, t[j] - t[k], obj);   // t_j
        - t_k from a to b
17              if (delta < bestDelta)
18              {
19                  bestDelta = delta; bj = j; bk = k;      // improving move
20                  if (rule == FIRST) { stop = true; break; }
21              }
22          }
23      }
24      if (bj == -1) return false;                         // no improving move:
       local optimum
25
26      int a = mach[bj], b = mach[bk];                     // apply the move
27      L[a] = L[a] - (t[bj] - t[bk]);
28      L[b] = L[b] + (t[bj] - t[bk]);
29      mach[bj] = b; mach[bk] = a;
30      return true;
31  }
```

- The **move** and **swap steps** for the BPP:

```
1  // one step of the move local search for the BPP
2  // bin[i] is the bin of item i, L[b] the load of bin b (L[b] = 0 if bin b is
       empty)
3  // obj = NB (number of bins, minimized) or SQ (sum of the squared loads,
       maximized)
4
```

```
bool BPPMoveStep(int n, const Vec& w, int c, Vec& bin, Vec& L, int obj, int
    rule)
{
    int bestGain = 0;                                   // best gain found so
    far
    int bi = -1, bb = -1;                               // best move found so
    far
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)
    {
        int a = bin[i];                                 // bin of item i
        for (int b = 0; b < (int)L.size(); ++b)
        {
            if (b == a || L[b] == 0) continue;          // same bin, or empty
    bin
            if (L[b] + w[i] > c) continue;              // item i does not fit
    into bin b
            int gain;                                   // improvement of the
    objective
            if (obj == SQ)
                gain = 2 * w[i] * (L[b] - L[a] + w[i]);
            else
                gain = (L[a] == w[i]) ? 1 : 0;          // bin a becomes empty
            if (gain > bestGain)
            {
                bestGain = gain; bi = i; bb = b;        // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                         // no improving move:
    local optimum

    L[bin[bi]] = L[bin[bi]] - w[bi];                    // apply the move
    L[bb] = L[bb] + w[bi];
    bin[bi] = bb;
    return true;
}
```

```
// one step of the swap local search for the BPP, with the auxiliary objective
// sum of the squared loads (maximized); a swap never changes the number of
    bins
// the move (i, k) exchanges the bins of the items i and k

bool BPPSwapStep(int n, const Vec& w, int c, Vec& bin, Vec& L, int rule)
{
    int bestGain = 0;                                   // best gain found so
    far
    int bi = -1, bk = -1;                               // best move found so
    far
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)
    {
        for (int k = i + 1; k < n; ++k)
        {
            int a = bin[i], b = bin[k];
```

```
            if (a == b || w[i] == w[k]) continue;        // the loads would not change
            int dw = w[i] - w[k];                        // weight transferred from a to b
            if (L[b] + dw > c || L[a] - dw > c) continue;   // a bin would be overfull
            int gain = 2 * dw * (L[b] - L[a] + dw);      // improvement of the objective
            if (gain > bestGain)
            {
                bestGain = gain; bi = i; bk = k;         // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                          // no improving move: local optimum

    int a = bin[bi], b = bin[bk], dw = w[bi] - w[bk];    // apply the move
    L[a] = L[a] - dw;
    L[b] = L[b] + dw;
    bin[bi] = b; bin[bk] = a;
    return true;
}
```

- The **flip+swap step** and the **penalty flip step** for the KP (the penalty $\alpha =$ `alpha_num`/`alpha_den` is a fraction, and all the values are multiplied by `alpha_den` to work with integers):

```
// one step of the local search for the KP with the neighbourhood flip + swap,
// restricted to feasible solutions; x[i] = 1 if item i is selected
// the flips are examined first, then the swaps (item i out, item k in)

bool KPFlipSwapStep(int n, const Vec& p, const Vec& w, int c, Vec& x, int rule)
{
    int W = 0;                                           // weight of the current solution
    for (int i = 0; i < n; ++i) W = W + w[i] * x[i];

    int bestDp = 0;                                      // best profit increase found so far
    int bi = -1, bk = -1;                                // best move found so far (bk = -1: flip)
    bool stop = false;
    for (int i = 0; i < n && !stop; ++i)                 // flip of item i
    {
        int dp = (1 - 2 * x[i]) * p[i];                  // remove, or add, item i
        int dw = (1 - 2 * x[i]) * w[i];
        if (W + dw <= c && dp > bestDp)
        {
            bestDp = dp; bi = i; bk = -1;                // improving move
            if (rule == FIRST) stop = true;
        }
```

```
    }
    for (int i = 0; i < n && !stop; ++i)                // swap: item i out, item k in
    {
        if (x[i] == 0) continue;
        for (int k = 0; k < n; ++k)
        {
            if (x[k] == 1) continue;
            int dp = p[k] - p[i];
            int dw = w[k] - w[i];
            if (W + dw <= c && dp > bestDp)
            {
                bestDp = dp; bi = i; bk = k;            // improving move
                if (rule == FIRST) { stop = true; break; }
            }
        }
    }
    if (bi == -1) return false;                         // no improving move: local optimum

    x[bi] = 1 - x[bi];                                  // apply the flip, or the swap
    if (bk != -1) x[bk] = 1;
    return true;
}
```

```
// one step of the flip local search for the KP with a penalty for the excess weight:
// g(x) = profit(x) - alpha * max(0, weight(x) - c), with alpha = alpha_num / alpha_den
// infeasible solutions are allowed; all the values are multiplied by alpha_den

bool KPPenaltyFlipStep(int n, const Vec& p, const Vec& w, int c, int alpha_num, int alpha_den, Vec& x, int rule)
{
    int P = 0, W = 0;                                   // profit and weight of the current solution
    for (int i = 0; i < n; ++i) { P = P + p[i] * x[i]; W = W + w[i] * x[i]; }
    int excess = (W > c) ? W - c : 0;
    int g = alpha_den * P - alpha_num * excess;         // alpha_den * g(x)

    int bestDelta = 0;                                  // best increase of alpha_den * g found so far
    int bi = -1;                                        // best move found so far
    for (int i = 0; i < n; ++i)                         // flip of item i
    {
        int newP = P + (1 - 2 * x[i]) * p[i];
        int newW = W + (1 - 2 * x[i]) * w[i];
        int newExcess = (newW > c) ? newW - c : 0;
        int delta = alpha_den * newP - alpha_num * newExcess - g;
        if (delta > bestDelta)
        {
            bestDelta = delta; bi = i;                  // improving move
            if (rule == FIRST) break;
        }
    }
```

```
    if (bi == -1) return false;                          // no improving move: local optimum

    x[bi] = 1 - x[bi];                                   // apply the flip
    return true;
}
```

- The **multi-start 2-opt**, the Park–Miller generator and the random tours:

```
// multi-start local search for the TSP: M random tours, each improved by 2-opt
// the random tours are generated by the Park-Miller generator, starting from seed
// best receives the best local optimum found; the function returns its length

int MultiStartTwoOpt(int n, const Mat& d, int M, long long seed, int rule, Vec& best)
{
    int bestLength = -1;                                 // length of the best tour found so far
    for (int s = 1; s <= M; ++s)
    {
        Vec pi = RandomTour(n, seed);                    // random starting tour
        LocalSearch([&]() { return TwoOptStep(n, d, pi, rule); });
        int length = TourLength(n, d, pi);               // length of the local optimum
        if (bestLength == -1 || length < bestLength)
        {
            bestLength = length;                         // new best tour
            best = pi;
        }
    }
    return bestLength;
}
```

```
// Park-Miller minimal standard generator: seed = 16807 * seed mod (2^31 - 1)
// returns u = seed / (2^31 - 1), a pseudo-random number in (0, 1)

double ParkMiller(long long& seed)
{
    seed = (16807LL * seed) % 2147483647LL;
    return (double)seed / 2147483647.0;
}
```

```
// pseudo-random integer in {0, 1, ..., k-1}: floor(u * k), computed in exact integer arithmetic

int RandomInt(long long& seed, int k)
{
    ParkMiller(seed);
    return (int)((seed * (long long)k) / 2147483647LL);
}
```

```cpp
// random tour with city 0 in the first position (Fisher-Yates shuffle of the
    positions 1..n-1)

Vec RandomTour(int n, long long& seed)
{
    Vec pi(n);
    for (int i = 0; i < n; ++i) pi[i] = i;
    for (int i = n - 1; i >= 2; --i)
    {
        int j = 1 + RandomInt(seed, i);                  // j in {1, ..., i}
        int tmp = pi[i]; pi[i] = pi[j]; pi[j] = tmp;
    }
    return pi;
}
```

Chapter 4

Metaheuristics

Prerequisites

- From this module, the chapter «Local search»: neighbourhoods, first and best improvement, variable neighbourhood descent and multi-start.
- From this module, the chapters «Constructive heuristics» (randomized construction) and «Heuristic algorithms: definitions» (intensification and diversification, experimental evaluation).

Objectives

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

- execute by hand simulated annealing, tabu search, iterated local search and variable neighbourhood search, with given random numbers;
- execute GRASP, large neighbourhood search and a genetic or memetic algorithm on small instances;
- explain how each method escapes from a local optimum, and which parameters tune it;
- compare the methods on the same instances, over several seeds;
- say what metaheuristics do not give: a dual bound.

4.1 Metaheuristics

Definition 4.1: Metaheuristic

A **metaheuristic** is a general strategy that guides a local search beyond the first local optimum, to explore other regions of the set of feasible solutions.

- A metaheuristic is a *scheme*, not an algorithm: an algorithm is obtained by choosing the representation, the neighbourhoods, the local search and the parameters.
- Every metaheuristic balances **intensification** and **diversification**, defined in the chapter «Heuristic algorithms: definitions».

A local search stops in the first local optimum; a metaheuristic escapes from it by accepting worse solutions, by jumping to a new starting solution, or by combining several good solutions.

4.2 The problems and the assumptions

- The metaheuristics are illustrated on the four problems of the chapter «Heuristic algorithms: definitions»: the 0/1 knapsack problem (KP), the bin packing problem (BPP), the travelling salesman problem (TSP) and $P\,||\,C_{\max}$. Their definitions and assumptions are on their pages; we recall the notation.

problem	data	solution	objective
KP	items $\mathscr{I}$, profits p_j, weights w_j, capacity c	subset of items	max total profit
BPP	items $\mathscr{I}$, weights w_j, capacity c	patterns P_b	min number of bins
TSP	cities $\mathscr{V}$, distances d_{ij}	tour σ	min length $L(\sigma)$
$P\,\|\|\,C_{\max}$	jobs $\mathscr{J}$, times t_j, machines m	sets $J_1, \ldots, J_m$	min makespan

- Every instance satisfies the assumptions of its problem: $w_j \leq c$ for all items and $\sum_j w_j > c$ for the KP and the BPP; $n \geq 3$ and symmetric distances for the TSP; $n > m$ for $P\,||\,C_{\max}$. For $P\,||\,C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ is the **load** of machine h and $C_{\max} = \max_h L_h$.
- As in the chapter «Heuristic algorithms: definitions», $\mathscr{A}(I)$ is the value of the solution computed by the algorithm $\mathscr{A}$ on the instance I, and $z(I)$ is the optimal value. We write $f(S)$ for the value of a solution S, and we describe the algorithms for a **minimization** problem; for the KP, a maximization problem, the inequalities are reversed.

4.2.1 The neighbourhoods

- All the metaheuristics of this chapter are built on the neighbourhoods of the chapter «Local search». We recall them by name, with the moves that define them; n is the number of items, cities or jobs.

problem	name	move	size
KP	flip	add one item to the knapsack, or remove one	n
	swap	remove one selected item and add one unselected item	$\leq \frac{n^2}{4}$
BPP	move	move one item to another bin	$\leq n \cdot nb$
	swap	exchange two items packed in different bins	$\leq \frac{n(n-1)}{2}$
TSP	2-opt	remove two non-adjacent edges of the tour and reconnect it with the two other edges, reversing the segment between them	$\frac{n(n-3)}{2}$
	Or-opt	move a segment of 1, 2 or 3 consecutive cities to another position	$O(n^2)$
$P \,\|\|\, C_{\max}$	move	move one job to another machine	$n(m-1)$
	swap	exchange two jobs assigned to different machines	$\leq \frac{n(n-1)}{2}$

- A TSP tour is stored as a permutation $\sigma = (\sigma_1, \ldots, \sigma_n)$ with $\sigma_1 = 1$. The **2-opt move** (i, j), with $1 \leq i$, $i + 2 \leq j \leq n$ and $(i, j) \neq (1, n)$, removes the edges (σ_i, σ_{i+1}) and (σ_j, σ_{j+1}) (with $\sigma_{n+1} = \sigma_1$), adds the edges (σ_i, σ_j) and $(\sigma_{i+1}, \sigma_{j+1})$, and reverses the segment $\sigma_{i+1}, \ldots, \sigma_j$. Its effect on the length is computed in $O(1)$:

$$\delta(i, j) \;=\; d_{\sigma_i \sigma_j} + d_{\sigma_{i+1} \sigma_{j+1}} - d_{\sigma_i \sigma_{i+1}} - d_{\sigma_j \sigma_{j+1}}.$$

- The objectives of the BPP and of $P \,||\, C_{\max}$ are **flat**: most moves do not change the number of bins, or the makespan. Two solutions with the same value are then compared with an **auxiliary objective**:
  - BPP: the sum of the squared loads $\sum_b \left(\sum_{j \in P_b} w_j \right)^2$, to be *maximized*: it grows when the items concentrate in fewer, fuller bins;
  - $P \,||\, C_{\max}$: the sum of the squared loads $\sum_h L_h^2$, to be *minimized*: it decreases when the loads become balanced.

  For $P \,||\, C_{\max}$ we compare two solutions **lexicographically**: first by $C_{\max}$, and if the makespans are equal, by $\sum_h L_h^2$. For move and swap this selects the same improving moves as the auxiliary objective $\sum_h L_h^2$ of the chapter «Local search»: a transfer that decreases $\sum_h L_h^2$ never increases $C_{\max}$, and a transfer that decreases $C_{\max}$ also decreases $\sum_h L_h^2$.
- For the KP the flip neighbourhood contains infeasible solutions (adding an item may exceed the capacity). A local search either discards them or accepts them with a **penalty** on the excess weight; both options appear in this chapter.

- The **local searches** used inside the metaheuristics are the descents of the chapter «Local search»: TwoOptLS (2-opt, best improvement) for the TSP, MoveDescent (move, best improvement, lexicographic comparison) for $P\,||\,C_{\max}$, and KPLocalSearch (feasible flips and swaps, best improvement) for the KP. In every best-improvement descent, ties between equally good moves are broken in favour of the first move found, scanning the indices in increasing order.

4.2.2 Random numbers

- Most metaheuristics make **random choices**. To make every example and every exercise reproducible, all the random numbers of this chapter, and of its `C++` code, are produced by the **minimal standard generator** of Park and Miller (1988):

$$seed \leftarrow 16807 \cdot seed \bmod (2^{31}-1), \qquad u = \frac{seed}{2^{31}-1} \in (0,1).$$

A call Rand() updates *seed* and returns u. A random integer in $\{0,1,\dots,K-1\}$ is $\lfloor u \cdot K \rfloor$, and a random element of a list of K elements is the element in position $1+\lfloor u \cdot K \rfloor$.

- The initial value of *seed*, the **seed**, is an integer in $\{1,\dots,2^{31}-2\}$. The same seed always gives the same sequence of numbers; different seeds give different runs of the same algorithm. With $seed = 1$ the first numbers are, rounded to four decimals,

$$0.0000,\ 0.1315,\ 0.7556,\ 0.4587,\ 0.5328,\ 0.2190,\ 0.0470,\ 0.6789,\ \dots$$

  The first number produced from a small seed is always very close to 0; it is a property of the generator, not an error.

- In the worked examples the random numbers are written in the text, rounded to four decimals, so that every step can be checked by hand; the rounding never changes a decision.

4.2.3 The instances

- The worked examples use the four **reference instances** of the chapter «Heuristic algorithms: definitions», whose optimal values are known:

problem	reference instance	optimal value and solution
KP	$p=(10,7,6,4)$, $w=(5,4,3,3)$, $c=9$	$z(I)=17$, items $\{1,2\}$
BPP	$w=(4,4,5,3,2,3)$, $c=7$	$z(I)=3$, patterns $\{1,4\},\{2,6\},\{3,5\}$
TSP	5 cities, distances d_{ij} below	$z(I)=18$, tour $(1,3,5,2,4)$
$P\,\|\|\,C_{\max}$	$t=(5,5,4,4,3,3,3)$, $m=3$	$z(I)=9$, sets $\{1,3\},\{2,4\},\{5,6,7\}$

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	—

- The reference instances are small enough to follow every iteration by hand, but too small to show the behaviour of a metaheuristic over thousands of iterations. For the experiments we use one **medium instance** per problem, generated once with the random number generator of Section 4.2.2 and fixed below. Their optimal values have been computed with an exact algorithm, and are used only to measure the quality of the heuristic solutions.

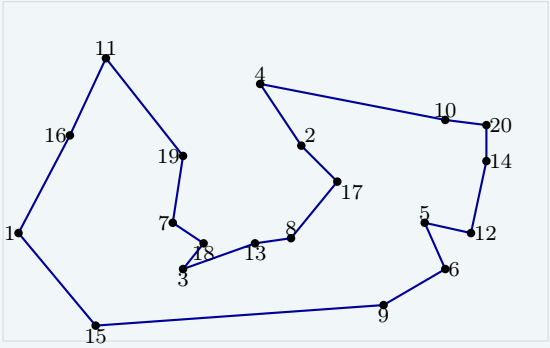
Example 4.1: The medium TSP instance (TSP20)

20 cities in the plane, with integer coordinates (x_i, y_i), and distances $d_{ij} = \lfloor \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} + \frac{1}{2} \rfloor$ (the Euclidean distance rounded to the nearest integer):

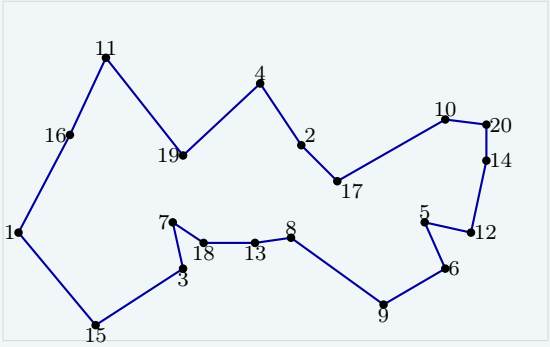
i	1	2	3	4	5	6	7	8	9	10
x_i	0	55	32	47	79	83	30	53	71	83
y_i	18	35	11	47	20	11	20	17	4	40

i	11	12	13	14	15	16	17	18	19	20
x_i	17	88	46	91	15	10	62	36	32	91
y_i	52	18	16	32	0	37	28	16	33	39

nearest neighbour from city 1: $L(\sigma) = 326$



an optimal tour: $L(\sigma) = 291$



The nearest neighbour tour from city 1 has length 326; the 2-opt local search started from it stops at a local optimum of length 319, after two improving moves. The optimal value is $z(I) = 291$: the local optimum is 9.6% above it.

Example 4.2: The medium instances of the KP, the BPP and $P\,||\,C_{\max}$

KP20: $n = 20$ items, capacity $c = 389$ (half of the total weight, rounded down),

j	1	2	3	4	5	6	7	8	9	10
p_j	54	55	55	52	58	42	53	53	22	64
w_j	39	43	49	47	44	35	43	44	16	50
j	11	12	13	14	15	16	17	18	19	20
p_j	54	51	39	49	53	46	28	64	58	34
w_j	40	42	28	38	46	39	14	50	45	27

The greedy algorithm by efficiency finds profit 490, and the local search with flips and swaps started from it reaches 514; the optimal value is $z(I) = 521$.

PCM15: $m = 4$ machines and $n = 15$ jobs with processing times

$$t = (10, 23, 26, 16, 23, 11, 14, 18, 18, 24, 29, 22, 13, 13, 15).$$

The total time is 275, so $z(I) \geq \lceil 275/4 \rceil = 69$, and an assignment of makespan 69 exists: $z(I) = 69$. LPT finds makespan 73.

BPP24: bins of capacity $c = 100$ and $n = 24$ items with weights

$$w = (30, 40, 57, 22, 37, 23, 34, 34, 55, 51, 21, 47, 22, 24, 22, 34, 20, 51, 25, 54, 35, 27, 55, 48).$$

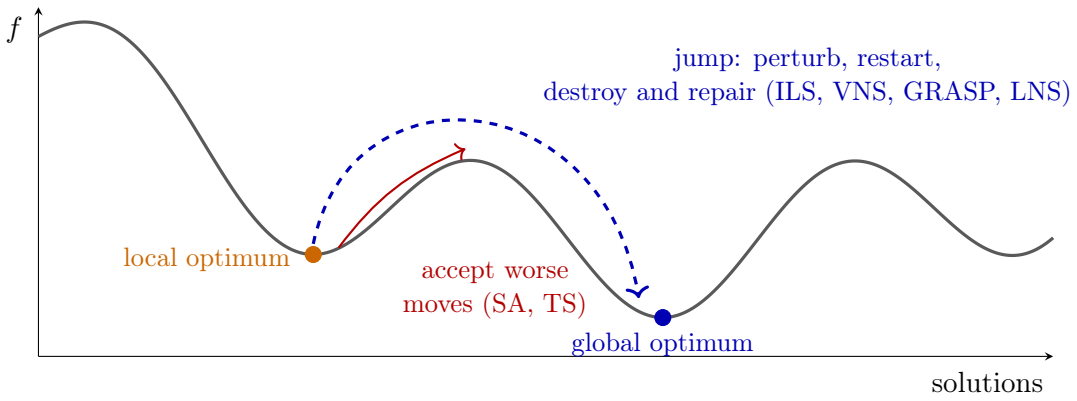
The total weight is 868, so at least $\lceil 868/100 \rceil = 9$ bins are needed, and a packing into 9 bins exists: $z(I) = 9$. FF and FFD both use 10 bins.

4.3 How to escape from a local optimum

- A local search moves from a solution to a better neighbour, and stops at the first **local optimum**: a solution with no better neighbour. In the landscape of the objective function, it descends to the bottom of the valley where it starts, and stays there. This is pure **intensification**.

- Repeating the local search from many starting solutions (multi-start) explores many valleys, but each start ignores everything learnt before, and good valleys are usually found close to other good valleys.

- The table below lists the ways to escape from a local optimum; it is the map of the chapter.



how to escape	idea	metaheuristic
accept worse moves *at random*	a worse neighbour is accepted with a probability that decreases with the worsening and with time	simulated annealing (Section 4.4)
accept worse moves *deterministically, with memory*	move to the best neighbour even if it is worse, and forbid going back to the solutions just visited	tabu search (Section 4.5)
perturb the local optimum and restart	apply a random modification that the local search cannot undo, and descend again	iterated local search (Section 4.6)
change the neighbourhood	when the search is stuck, use a larger neighbourhood, of which the local optimum is no longer a local optimum	variable neighbourhood search (Section 4.7)
restart from a randomized construction	build a different good solution at every restart, and improve it	GRASP (Section 4.8)
destroy and repair	remove a large part of the solution and rebuild it with a heuristic or an exact algorithm	large neighbourhood search (Section 4.9)
work with a *population*	combine the parts of several good solutions into new ones	genetic and memetic algorithms (Section 4.10)

- The first two ways keep **one current solution** and change the **acceptance rule** of the local search: they walk out of the valley through worse solutions. The next four **jump**: they generate a new starting point, far enough to leave the valley but close enough to keep what is good. The last one keeps **many solutions** at the same time.

- Every metaheuristic must balance **intensification** (descend to the bottom of the current valley) and **diversification** (leave it for another one). For each method we point out which component does what.

- A metaheuristic is a *scheme*, not an algorithm: to obtain an algorithm one must choose the representation, the neighbourhoods, the local search and the parameters, and these choices matter at least as much as the scheme. We present every metaheuristic on the problems where its idea is most natural, not on all four.

4.4 Simulated annealing

Simulated annealing (denoted by SA) is a local search that chooses a *random* neighbour of the current solution and moves to it if it is better; if it is worse, it moves to it anyway with a probability that decreases with the worsening and with a parameter T, the *temperature*. The temperature starts high, so that at the beginning almost every move is accepted, and is lowered gradually, so that at the end only improving moves are accepted. The best solution visited is returned.

- The pseudo-code of SA, for a minimization problem with objective f and neighbourhood N, reads as follows:

Algorithm

Algorithm: Simulated annealing

Input: starting solution S, initial temperature T_0, cooling factor $\alpha \in (0,1)$, number L of moves per temperature, final temperature $T_{\min}$

Output: best solution found

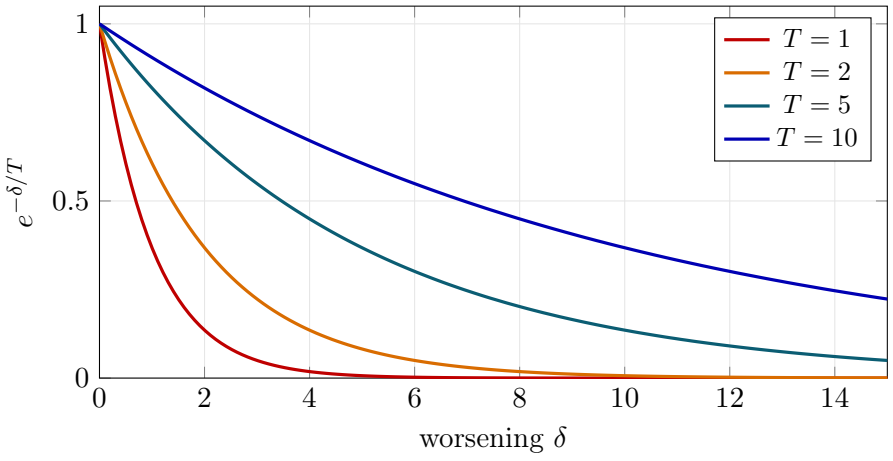
Function *SimulatedAnnealing*$(S, T_0, \alpha, L, T_{\min})$
  $cur := S$; $fcur := f(cur)$ // current solution
  $best := cur$; $fbest := fcur$ // best solution found
  $T := T_0$ // temperature
  while $T > T_{\min}$ **do**
    for $l := 1$ **to** L **do**
      $S' :=$ random neighbour in $N(cur)$
      $\delta := f(S') - fcur$
      $v :=$ RAND()
      if $\delta \leq 0$ ***or*** $v < e^{-\delta/T}$ **then**
        $cur := S'$; $fcur := fcur + \delta$ // move accepted
        if $fcur < fbest$ **then**
          $best := cur$; $fbest := fcur$
        end
      end
    end
    $T := \alpha \cdot T$ // cooling
  end
  return *best*

- The name comes from metallurgy: *annealing* is the process of heating a metal and cooling it slowly, so that its atoms settle into a state of minimum energy; cooled too fast, the metal freezes into a disordered state of higher energy. Kirkpatrick, Gelatt and Vecchi (1983) and, independently, Černý (1985) proposed to simulate this process on an optimization problem, with the objective f in the role of the energy. The acceptance rule is the one used by Metropolis et al. (1953) to simulate the atoms.

Metropolis acceptance rule. At temperature T, a neighbour S' of the current solution, with $\delta = f(S') - f(cur)$, is accepted with probability

$$P(\text{accept } S') = \begin{cases} 1 & \text{if } \delta \leq 0, \\ e^{-\delta/T} & \text{if } \delta > 0. \end{cases}$$

- The probability is implemented by drawing $v = \text{RAND}()$, uniform in $(0, 1)$, and accepting if $v < e^{-\delta/T}$, which happens with probability $e^{-\delta/T}$.
- A worsening move is accepted more easily if the worsening δ is **small**, and if the temperature T is **high**. As $T \to \infty$ every move is accepted, and SA becomes a random walk (pure diversification); as $T \to 0$ only the moves with $\delta \leq 0$ are accepted, and SA becomes a local search with random choice of the neighbour (pure intensification). Lowering the temperature moves gradually from the first behaviour to the second.
- Moves with $\delta = 0$ are always accepted: SA walks freely on the plateaus of flat objectives.
- SA never stops at a local optimum: a local optimum is left as soon as a worsening move is drawn and accepted. For this reason the **best** solution visited must be stored separately: the current solution at the end is not necessarily the best one.
- Only **one random neighbour** is evaluated per iteration, not the whole neighbourhood: an iteration of SA is very cheap, and SA performs many more iterations than a local search. For the TSP, a random 2-opt move is drawn by numbering the $M = \frac{n(n-3)}{2}$ moves (i, j) in lexicographic order and choosing the one in position $1 + \lfloor \text{RAND}() \cdot M \rfloor$; its δ is computed in $O(1)$.



- The figure shows the acceptance probability as a function of the worsening, for four temperatures. At $T = 10$ a worsening of 5 is accepted with probability $e^{-0.5} \approx 0.61$; at $T = 1$ with probability $e^{-5} \approx 0.007$. The temperature has the same unit as the objective: T is "large" or "small" only compared with the typical values of δ.

4.4.1 The cooling schedule and the parameters

- The **cooling schedule** is the sequence of temperatures. The pseudo-code uses the **geometric** schedule $T \leftarrow \alpha T$, the most common in practice, with α between 0.8 and 0.99, and L moves at each temperature. Other schedules are the *linear* one, $T_k = T_0 - k\beta$, and the *logarithmic* one, $T_k = \frac{\Gamma}{\ln(k+1)}$, which matters in the theory of SA but is far too slow in practice.

- **Initial temperature T_0.** It must be high enough to accept most worsening moves at the beginning. A common rule fixes a target initial acceptance ratio χ_0 (for instance $\chi_0 = 0.8$), estimates the average worsening $\bar{\delta}$ of a sample of random moves, and sets $T_0 = -\bar{\delta}/\ln \chi_0$, so that $e^{-\bar{\delta}/T_0} = \chi_0$.

- **Moves per temperature L.** The search must be given time to adapt to each temperature; L is often taken proportional to the size of the neighbourhood.

- **Stopping rule.** The pseudo-code stops when $T \leq T_{\min}$. With the geometric schedule the number of temperatures is the smallest integer K with $T_0\alpha^K \leq T_{\min}$, i.e., $K = \lceil \ln(T_{\min}/T_0)/\ln \alpha \rceil$ when the ratio is not an integer, and the number of iterations is $K \cdot L$. Other common rules stop after a number of temperatures without any accepted move, or at a time limit.

- Slow cooling (α close to 1) gives better solutions but more iterations: the parameters trade quality for time, as every heuristic does.

4.4.2 Examples

Example 4.3: SA with the 2-opt neighbourhood on the TSP reference instance

We start from the tour computed by the nearest neighbour heuristic from city 1 in the chapter «Constructive heuristics», $\sigma = (1, 3, 2, 4, 5)$ of length 25, with $T_0 = 5$, $\alpha = 0.8$, $L = 2$ and $T_{\min} = 3$: the temperatures are 5, 4 and 3.2 (then $2.56 \leq 3$ stops the search), for a total of 6 iterations. The $M = 5$ moves of 2-opt on 5 cities are numbered

$$1 : (1, 3), \quad 2 : (1, 4), \quad 3 : (2, 4), \quad 4 : (2, 5), \quad 5 : (3, 5),$$

and the random numbers, with $seed = 1$, are those of Section 4.2.2: the first number of each iteration (u) chooses the move $1 + \lfloor 5u \rfloor$, the second (v) is used in the acceptance test.

it	T	u	move	δ	v	$e^{-\delta/T}$	decision	current tour	L
1	5	0.0000	$1 : (1, 3)$	4	0.1315	0.4493	accept	$(1, 2, 3, 4, 5)$	29
2	5	0.7556	$4 : (2, 5)$	-7	—	—	accept	$(1, 2, 5, 4, 3)$	22
3	4	0.5328	$3 : (2, 4)$	2	0.2190	0.6065	accept	$(1, 2, 4, 5, 3)$	24
4	4	0.0470	$1 : (1, 3)$	-6	—	—	accept	$(1, 4, 2, 5, 3)$	18
5	3.2	0.6793	$4 : (2, 5)$	4	0.9347	0.2865	reject	$(1, 4, 2, 5, 3)$	18
6	3.2	0.3835	$2 : (1, 4)$	4	0.5194	0.2865	reject	$(1, 4, 2, 5, 3)$	18

At iterations 1 and 3 a **worse** tour is accepted ($25 \to 29$ and $22 \to 24$), because v is smaller than the acceptance probability; at iterations 5 and 6 the temperature is lower, the worsening moves are less likely to be accepted, and both are rejected. The best tour, $(1, 4, 2, 5, 3)$, is the optimal tour $(1, 3, 5, 2, 4)$ traversed in the opposite direction: $SA(I) = 18 = z(I)$.

- On this small instance the optimal tour is the only 2-opt local optimum among the 12 tours, so a 2-opt local search would reach it as well: the example shows the mechanics of SA; its ability to escape from local optima shows on instances with bad local optima.

4.4.3 Infeasible solutions: simulated annealing with a penalty

- For the KP, the flip neighbourhood of a solution close to the capacity contains many infeasible solutions. Instead of discarding them, SA can **visit** them, with a penalty on the excess weight that makes them less attractive:

$$f_\beta(x) = \sum_{j \in \mathscr{I}} p_j x_j - \beta \cdot \max\left\{0, \sum_{j \in \mathscr{I}} w_j x_j - c\right\}, \qquad \beta > 0.$$

SA maximizes f_β over all $x \in \{0, 1\}^n$; the worsening of a move from x to y is $\delta = f_\beta(x) - f_\beta(y)$, and the best solution is updated with **feasible** solutions only.

- On a feasible solution f_β is the profit. Passing through an infeasible solution, SA can reach feasible solutions that are not reachable through feasible flips only: the infeasible region acts as a **bridge** between valleys.
- The **penalty coefficient** β is a parameter: if it is too small the search wanders in the infeasible region, where the profit is larger; if it is too large the infeasible solutions are never accepted, and the bridge is closed.

Example 4.4: *SA* with a penalty on the KP reference instance

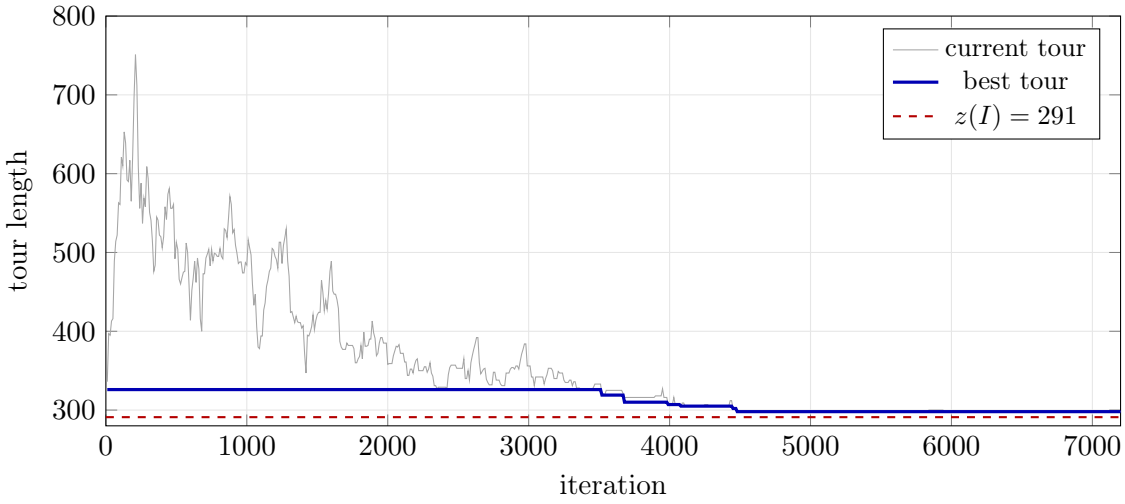
We start from $x = (1, 0, 1, 0)$, the solution $\{1, 3\}$ of the greedy algorithm by efficiency, of profit 16. It is a local optimum for the flip neighbourhood (chapter «Heuristic algorithms: definitions»): its feasible neighbours have profit 6 and 10. We use $\beta = 3$, $T_0 = 4$, $\alpha = 0.8$, $L = 2$, $T_{\min} = 2.5$ (temperatures 4, 3.2, 2.56: 6 iterations) and $seed = 3$. At each iteration u chooses the item $k = 1 + \lfloor 4u \rfloor$ to flip, and v is used in the acceptance test.

it	T	u	k	y	$\sum w_j y_j$	$f_3(y)$	δ	v vs $e^{-\delta/T}$	decision
1	4	0.0000	1	$(0, 0, 1, 0)$	3	6	10	$0.3946 \geq 0.0821$	reject
2	4	0.2668	2	$(1, 1, 1, 0)$	12	14	2	$0.3760 < 0.6065$	accept
3	3.2	0.5983	3	$(1, 1, 0, 0)$	9	17	-3	—	accept
4	3.2	0.1411	1	$(0, 1, 0, 0)$	4	7	10	$0.0366 < 0.0439$	accept
5	2.56	0.0379	1	$(1, 1, 0, 0)$	9	17	-10	—	accept
6	2.56	0.1505	1	$(0, 1, 0, 0)$	4	7	10	$0.5582 \geq 0.0201$	reject

At iteration 2 the search moves to the **infeasible** solution $(1, 1, 1, 0)$, of weight $12 > 9$: its profit is 23 and its penalized value $23 - 3 \cdot 3 = 14$, a worsening of 2 accepted with probability 0.6065. From there, removing item 3 gives the feasible solution $(1, 1, 0, 0)$ of profit 17: the optimum, $SA(I) = 17 = z(I)$. Iteration 4 shows that at this temperature even a worsening of 10 can be accepted, with a small probability; iteration 5 goes back to the optimum.

4.4.4 An experiment on the medium TSP instance

- We run *SA* with the 2-opt neighbourhood on TSP20 (Example 4.1), from the nearest neighbour tour of length 326, with $L = 100$ and $T_{\min} = 0.5$. The figure shows one run, with $T_0 = 20$, $\alpha = 0.95$ and $seed = 1$: the length of the current tour (gray) and of the best tour (blue) along the 7200 iterations, every 10 iterations. The data are produced by the `C++` code of this chapter (`make data`).



- At the beginning the temperature is high compared with the typical worsening of a 2-opt move, and the current tour gets much **worse** than the starting one (up to length 765): the search is almost a random walk. As T decreases, fewer worsening moves are accepted and the current tour descends into a valley; at the end the search is a local search. The best

tour, of length 298, is found at iteration 4479: it is better than the 2-opt local optimum (319), but it is not optimal.

- The table reports, for 10 runs with $seed = 1, \ldots, 10$ and several parameters, the number of iterations, the minimum, the average and the maximum length of the best tour, and the number of runs that reach the optimum 291.

α	T_0	iterations	min	average	max	optimal runs
0.9	1	700	291	310.3	322	1
0.9	5	2200	291	297.3	305	4
0.9	20	3600	291	300.3	323	4
0.9	100	5100	291	297.8	311	3
0.95	1	1400	291	302.3	319	3
0.95	5	4500	291	294.3	298	5
0.95	20	7200	291	294.3	298	5
0.95	100	10400	291	293.2	299	7
0.99	1	6900	291	298.8	305	4
0.99	5	23000	291	291.7	298	9
0.99	20	36800	291	291.0	291	10
0.99	100	52800	291	291.0	291	10

- With $T_0 = 1$ the temperature is too low from the beginning: few worsening moves are accepted, and SA behaves like a randomized local search. Slower cooling ($\alpha = 0.99$) spends more iterations at every temperature range and reaches the optimum in almost every run, at the price of many more iterations. A single run says little: the comparison must look at the distribution over several seeds, as recalled in the chapter «Heuristic algorithms: definitions».

4.4.5 Properties

- At a fixed temperature T, the sequence of current solutions of SA is a Markov chain: the next solution depends only on the current one. If the neighbourhood is symmetric ($S' \in N(S)$ if and only if $S \in N(S')$, all neighbourhoods having the same size) and connected (every solution can reach every other one through moves), the chain has a unique stationary distribution, the **Boltzmann distribution**:

$$\pi_T(S) = \frac{e^{-f(S)/T}}{\sum_{S'} e^{-f(S')/T}} \qquad \text{for every solution } S.$$

- For two solutions S_1 and S_2, $\frac{\pi_T(S_1)}{\pi_T(S_2)} = e^{(f(S_2)-f(S_1))/T}$: better solutions are more probable, and the more so the lower the temperature. As $T \to 0$, π_T concentrates on the optimal solutions.

- Hajek (1988) proved that with the logarithmic schedule $T_k = \frac{\Gamma}{\ln(k+1)}$, with Γ at least the depth of the deepest valley that does not contain an optimal solution, the probability that the current solution is optimal tends to 1. The result is a guarantee *in the limit*: the number of iterations needed can be larger than the number of solutions, and in practice the faster geometric schedule is used, without guarantees.

4.5 Tabu search

Tabu search (denoted by TS) explores the neighbourhood of the current solution completely and moves to the *best* neighbour, even if it is worse than the current solution. To avoid going back to the solutions just visited, it keeps a short-term memory, the *tabu list*: the moves that would undo the recent moves are forbidden (*tabu*) for a number of iterations, the *tenure*, unless they lead to a solution better than the best one found (*aspiration criterion*).

- The pseudo-code of TS for $P\,||\,C_{\max}$, with the move neighbourhood and the lexicographic comparison of Section 4.2.1, reads as follows; the attribute (j, h) forbids moving job j back to machine h.

Algorithm

Algorithm: Tabu search for $P\,||\,C_{\max}$

Input: times $t[1..n]$, m machines, starting assignment $mach[1..n]$, $tenure$, $maxIter$
Output: best assignment found

Function *TabuSearch*$(t, m, mach, tenure, maxIter)$
  $L :=$ loads of $mach$; $tabu[j][h] := 0$ for all j, h
  $best := mach$; $bestC := \max_h L[h]$
  LB $:= \max\{\max_j t[j],\ \lceil \sum_j t[j]/m \rceil\}$ `// stop if reached`
  $it := 0$
  while $it < maxIter$ ***and*** $bestC >$ LB **do**
    $it := it + 1$; $bj := 0$ `// best non-tabu move` (bj, bh)
    for $j := 1$ **to** n **do**
      for $h := 1$ **to** m, $h \neq mach[j]$ **do**
        $L' :=$ the loads after moving job j from $mach[j]$ to h
        if $tabu[j][h] \geq it$ ***and not*** $\max_k L'[k] < bestC$ **then continue**`// tabu`
        if $bj = 0$ ***or*** L' *is lexicographically better than* bL **then**
          $(bj, bh, bL) := (j, h, L')$
      end
    end
    if $bj = 0$ **then break**
    $tabu[bj][mach[bj]] := it + tenure$ `// forbid going back`
    $mach[bj] := bh$; $L := bL$ `// even if worse`
    if $\max_k L[k] < bestC$ **then**
      $bestC := \max_k L[k]$; $best := mach$
    end
  end
  return $best$

- Tabu search was proposed by Glover (1986); the reference book is Glover and Laguna, *Tabu Search*, Kluwer, 1997. Unlike SA, it is **deterministic**: it escapes from a local optimum because it always moves, to the least bad neighbour if no neighbour is better.

- Without memory, this rule would **cycle**: from a local optimum S the search moves to the best neighbour S', which is worse, and from S' the best neighbour is typically S again. The tabu list breaks the cycle by forbidding the reverse move.

- **Attributes.** Storing the visited solutions and comparing every neighbour with them would be too expensive. The tabu list stores **attributes** of the recent moves instead: here the pair (j, h) "job j has left machine h"; for the TSP, the edges just removed from the tour.

- An attribute forbids more than the single solution just left: every solution with job j on machine h is tabu, even the ones never visited. The **aspiration criterion** repairs this excess of caution: a tabu move is allowed if it leads to a solution better than the best one found, which certainly has not been visited.

- **Tenure.** If it is too short, the search can still cycle through longer cycles; if it is too long, too many moves are forbidden and the search is driven far from the good regions. Typical values are between 5 and 20, or proportional to $\sqrt{n}$; drawing the tenure at random in an interval at every iteration (robust tabu search, Taillard 1991) makes cycles unlikely.

- **Stopping rule.** A maximum number of iterations, or of iterations without improvement of the best solution. For $P\,||\,C_{\max}$ we can also stop when the makespan reaches the lower bound LB: then the solution is optimal, and the search can only waste time.

- **Long-term memory.** Besides the tabu list (short-term memory), TS can record how often each attribute appears in the visited solutions. *Diversification* penalizes the frequent attributes, to push the search toward unexplored regions; *intensification* restarts from the best solutions found, and fixes the attributes they share.

4.5.1 Example

Example 4.5: TS on the $P\,||\,C_{\max}$ reference instance

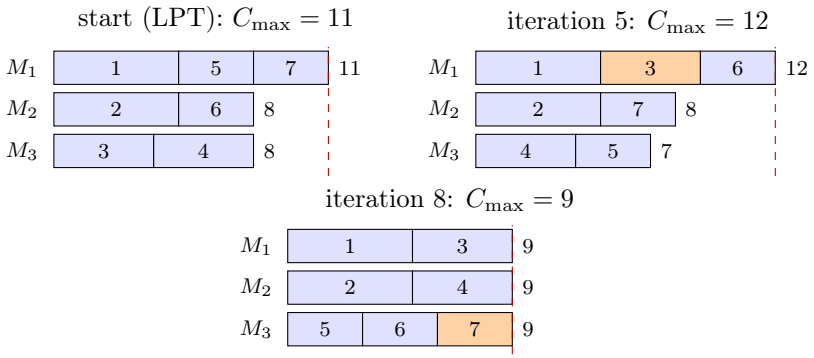
We start from the solution computed by LPT in the chapter «Constructive heuristics», $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, with loads $(11, 8, 8)$, $C_{\max} = 11$ and $\sum_h L_h^2 = 249$. It is a local optimum for the move neighbourhood: at the first iteration the 14 neighbours are

move	$1 \to M_2$	$1 \to M_3$	$2 \to M_1$	$2 \to M_3$	$3 \to M_1$	$3 \to M_2$	$4 \to M_1$
$(C_{\max}, \sum_h L_h^2)$	$(13, 269)$	$(13, 269)$	$(16, 329)$	$(13, 299)$	$(15, 305)$	$(12, 281)$	$(15, 305)$
move	$4 \to M_2$	$5 \to M_2$	$5 \to M_3$	$6 \to M_1$	$6 \to M_3$	$7 \to M_2$	$7 \to M_3$
$(C_{\max}, \sum_h L_h^2)$	$(12, 281)$	$(11, 249)$	$(11, 249)$	$(14, 285)$	$(11, 267)$	$(11, 249)$	$(11, 249)$

No neighbour has $C_{\max} < 11$, and the four best ones have $C_{\max} = 11$ and $\sum_h L_h^2 = 249$, as the current solution: a local search stops here. TS, with $tenure = 6$, moves instead to the best non-tabu neighbour (the first of the four, job 5 to M_2), and continues:

it	move	loads	$C_{\max}$	$\sum_h L_h^2$	$bestC$	new tabu attribute
1	job 5: $M_1 \to M_2$	$(8, 11, 8)$	11	249	11	$(5, M_1)$ until it. 7
2	job 5: $M_2 \to M_3$	$(8, 8, 11)$	11	249	11	$(5, M_2)$ until it. 8
3	job 6: $M_2 \to M_1$	$(11, 5, 11)$	11	267	11	$(6, M_2)$ until it. 9
4	job 7: $M_1 \to M_2$	$(8, 8, 11)$	11	249	11	$(7, M_1)$ until it. 10
5	job 3: $M_3 \to M_1$	$(12, 8, 7)$	12	257	11	$(3, M_3)$ until it. 11
6	job 6: $M_1 \to M_3$	$(9, 8, 10)$	10	245	10	$(6, M_1)$ until it. 12
7	job 4: $M_3 \to M_2$	$(9, 12, 6)$	12	261	10	$(4, M_3)$ until it. 13
8	job 7: $M_2 \to M_3$	$(9, 9, 9)$	9	243	9	$(7, M_2)$ until it. 14

- At iteration 2 the best neighbour would move job 5 back to M_1, returning to the starting solution: the move is tabu (attribute $(5, M_1)$ until iteration 7), and the search moves job 5 to M_3.
- At iteration 5 the four neighbours with $C_{\max} = 11$ are all tabu, and none satisfies the aspiration criterion ($C_{\max} < bestC = 11$): the best non-tabu move **worsens** the makespan to 12. This is how TS leaves the plateau.
- At iteration 6 the makespan drops to 10, and at iteration 8 to 9: $C_{\max} = \text{LB} = 9$, so the solution $J_1 = \{1, 3\}$, $J_2 = \{2, 4\}$, $J_3 = \{5, 6, 7\}$ is optimal and the search stops: $TS(I) = 9 = z(I)$.



- The figure shows the Gantt charts of the starting solution, of the solution after the worsening move of iteration 5 (the moved job in orange), and of the optimal solution reached at iteration 8.
- With $tenure \leq 5$, TS does not reach the optimum on this instance within 50 iterations: the tabu list is too short to prevent the search from going back to the plateau of makespan 11.

4.5.2 Tabu search for the TSP, and the effect of the tenure

- For the TSP with the 2-opt neighbourhood, the natural attributes are the **edges**: when a move removes the edges (σ_i, σ_{i+1}) and (σ_j, σ_{j+1}), these two edges may not be added back to the tour for *tenure* iterations, unless the move gives a tour shorter than the best one. Everything else is as in the pseudo-code of Section 4.5: at every iteration all the $\frac{n(n-3)}{2}$ moves are evaluated in $O(1)$ each, and the best non-tabu one is performed.

- The tables report the result of TS on the medium instances, from the nearest neighbour tour (326) for TSP20 with 200 iterations, and from the LPT solution (73) for PCM15 with 300 iterations, for several values of the tenure; for PCM15 the table also gives the number of iterations performed (300 unless the lower bound is reached).

TSP20: TS with 2-opt, 200 iterations ($z(I) = 291$)						
tenure	0	1	2	5	10	20
best length	319	291	291	291	291	291

PCM15: TS with the move neighbourhood, at most 300 iterations ($z(I) = 69$)										
tenure	0	1	2	3	5	7	10	15	20	30
best $C_{\max}$	73	73	73	73	73	72	71	69	70	70
iterations	300	300	300	300	300	300	300	231	300	300

- With $tenure = 0$ there is no memory, and TS cycles: on TSP20 it never leaves the first 2-opt local optimum. On PCM15 the move neighbourhood has a large plateau, and TS needs a long tabu list to cross it; but a tabu list that is too long forbids too much, and with $tenure \geq 20$ the optimum is lost again. The tenure must be tuned, and its best value depends on the problem and on the instance.

4.6 Iterated local search

Iterated local search (denoted by ILS) applies a local search, and then repeats: it *perturbs* the current local optimum with a random modification (a *kick*) that the local search cannot easily undo, applies the local search to the perturbed solution, and decides with an *acceptance criterion* whether the new local optimum replaces the current one. It is a walk among the local optima.

- The pseudo-code of ILS for the TSP, with the 2-opt local search and the double-bridge kick, reads as follows:

Algorithm

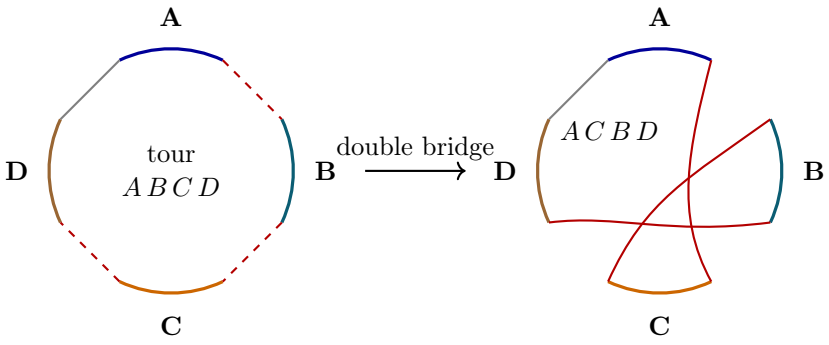
Algorithm: Iterated local search for the TSP

Input: distances d, starting tour $\sigma[1..n]$, number of iterations $maxIter$
Output: the last accepted local optimum

Function *IteratedLocalSearch*$(d, \sigma, maxIter)$
  $cur := \text{TwoOptLS}(\sigma)$; $lcur := L(cur)$ `// first local optimum`
  for $it := 1$ **to** $maxIter$ **do**
    $nw := \text{DoubleBridge}(cur)$ `// perturbation`
    $nw := \text{TwoOptLS}(nw)$; $lnew := L(nw)$ `// local search`
    if $lnew < lcur$ **then**
      $cur := nw$; $lcur := lnew$ `// acceptance`
    end
  end
  return cur

Function *DoubleBridge*(σ)
  $p_1 := 1 + \lfloor \text{Rand}() \cdot (n-3) \rfloor$
  $p_2 := p_1 + 1 + \lfloor \text{Rand}() \cdot (n-2-p_1) \rfloor$
  $p_3 := p_2 + 1 + \lfloor \text{Rand}() \cdot (n-1-p_2) \rfloor$
  $A := \sigma[1..p_1]$; $B := \sigma[p_1{+}1..p_2]$; $C := \sigma[p_2{+}1..p_3]$; $D := \sigma[p_3{+}1..n]$
  return (A, C, B, D)

- The scheme was described in its general form by Lourenço, Martin and Stützle (2003), in the *Handbook of Metaheuristics*; for the TSP it goes back to the "large-step Markov chains" of Martin, Otto and Felten (1991), who introduced the double-bridge kick.
- ILS has three components, and each one has a role:
  - the **local search** intensifies: it brings every perturbed solution to the bottom of its valley;
  - the **perturbation** diversifies: it moves the search to another valley;
  - the **acceptance criterion** balances the two: accepting only better local optima (as in the pseudo-code) intensifies, accepting every new local optimum diversifies; intermediate rules accept a worse local optimum with a probability, as in SA.
- The **strength** of the perturbation is the key parameter. If it is too weak, the local search undoes it and returns to the same local optimum; if it is too strong, the perturbed solution has nothing in common with the local optimum, and ILS becomes a random multi-start. A good kick changes the solution in a way that the local search *cannot* undo, but keeps most of it.



- The **double bridge** cuts the tour into four segments A, B, C, D and reconnects them as $A\,C\,B\,D$: it removes three edges (dashed) and adds three new ones (red), while the edge between the end of D and the beginning of A is kept. No segment is reversed, whereas every 2-opt move reverses a segment: when B and C have at least two cities each, a single 2-opt move cannot undo the kick, and the 2-opt local search, which accepts only improving moves, rarely goes back to the same local optimum. (If B and C are single cities, the kick changes only two edges and is itself a 2-opt move: the exercise «Iterated local search and VNS 2» uses this case.)

4.6.1 Example

Example 4.6: *ILS* on the medium TSP instance

On TSP20, the 2-opt local search from the nearest neighbour tour (length 326) stops at the local optimum

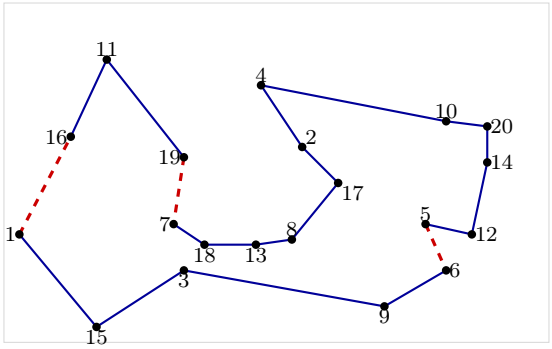
$$cur = (1, 16, 11, 19, 7, 18, 13, 8, 17, 2, 4, 10, 20, 14, 12, 5, 6, 9, 3, 15), \qquad L(cur) = 319.$$

With $seed = 1$, the three random numbers of the first kick are 0.0000, 0.1315, 0.7556, so

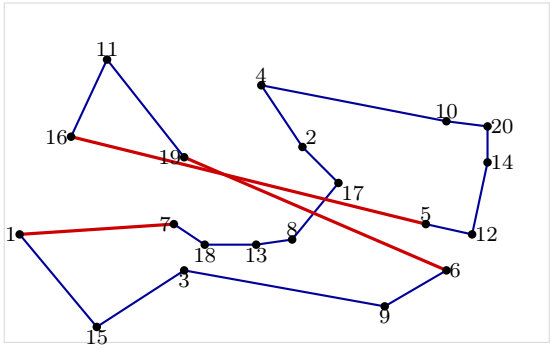
$$p_1 = 1 + \lfloor 0.0000 \cdot 17 \rfloor = 1, \quad p_2 = 1 + 1 + \lfloor 0.1315 \cdot 17 \rfloor = 4, \quad p_3 = 4 + 1 + \lfloor 0.7556 \cdot 15 \rfloor = 16,$$

and the segments are $A = (1)$, $B = (16, 11, 19)$, $C = (7, 18, 13, 8, 17, 2, 4, 10, 20, 14, 12, 5)$, $D = (6, 9, 3, 15)$. The kick removes the edges $(1, 16)$, $(19, 7)$, $(5, 6)$ and adds $(1, 7)$, $(5, 16)$, $(19, 6)$: the perturbed tour $A\,C\,B\,D$ has length 432, much worse. The 2-opt local search brings it to a new local optimum of length $303 < 319$, which is accepted. The second kick ($p = (8, 14, 16)$) gives, after 2-opt, a tour of length $291 = z(I)$; the third gives again a tour of length 291, not better, and it is rejected.

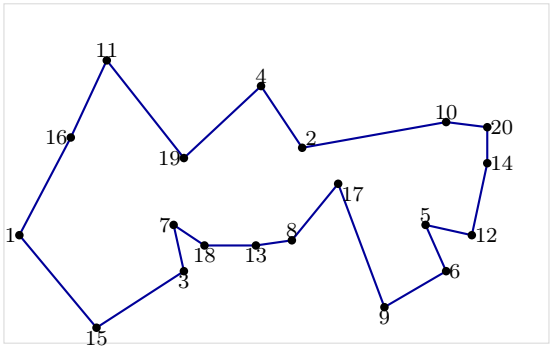
(a) local optimum: $L = 319$



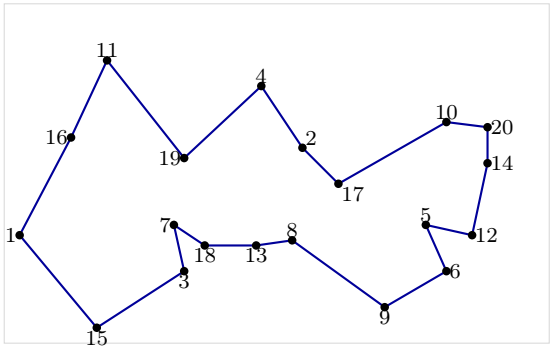
(b) after the double bridge: $L = 432$



(c) after 2-opt: $L = 303$



(d) after the second iteration: $L = 291$



- Panel (a) shows the local optimum with the three edges removed by the kick (dashed), panel (b) the perturbed tour with the three new edges, panels (c) and (d) the local optima after the first and the second iteration. The kick moves the segment C before B; the 2-opt local search then repairs the long edges it created, keeping the good parts of the old tour.
- With 100 iterations and $seed = 1, \ldots, 10$, ILS reaches the optimum 291 in every run; the iteration at which the optimum is reached is

$$2,\ 3,\ 1,\ 7,\ 8,\ 11,\ 15,\ 6,\ 2,\ 3$$

for the ten seeds. Each iteration costs one kick and one 2-opt local search, which starts from a solution close to a local optimum and needs few moves.

4.7 Variable neighbourhood search

Variable neighbourhood search (denoted by VNS) uses a sequence of neighbourhoods $N_1, N_2, \ldots, N_{k_{\max}}$ of increasing size. From the current local optimum it draws a random solution in N_k (*shaking*), applies a local search to it, and moves to the new local optimum if it is better, restarting from $k = 1$; otherwise it tries the next, larger neighbourhood N_{k+1}.

- The pseudo-code of VNS for $P\,||\,C_{\max}$, where N_k contains the assignments obtained with k moves and the local search is MoveDescent, reads as follows:

Algorithm

Algorithm: Variable neighbourhood search for $P\,||\,C_{\max}$

Input: times $t[1..n]$, m machines, starting assignment $mach[1..n]$, $k_{\max}$, $maxIter$
Output: an assignment $mach$ and its makespan C

Function *VNS*$(t, m, mach, k_{\max}, maxIter)$
  LB $:= \max\{\max_j t[j],\ \lceil \sum_j t[j]/m \rceil\}$
  $mach :=$ MoveDescent$(mach)$; $L :=$ loads of $mach$; $C := \max_h L[h]$
  $it := 0$; $k := 1$
  while $it < maxIter$ ***and*** $C >$ LB **do**
    $it := it + 1$
    $nw :=$ Shake$(mach, k)$ `// random solution in` N_k
    $nw :=$ MoveDescent(nw); $Lnew :=$ loads of nw `// local search`
    if $Lnew$ *is lexicographically better than* L **then**
      $mach := nw$; $L := Lnew$; $k := 1$; $C := \max_h L[h]$ `// move`
    else
      if $k < k_{\max}$ **then** $k := k + 1$ **else** $k := 1$
    end
  end
  return $mach$, C

Function *Shake*$(mach, k)$
  for $s := 1$ **to** k **do**
    $j := 1 + \lfloor$Rand$() \cdot n \rfloor$; $g := 1 + \lfloor$Rand$() \cdot (m-1) \rfloor$
    $mach[j] :=$ the g-th machine different from $mach[j]$
  end
  return $mach$

- VNS was proposed by Mladenović and Hansen (1997). It rests on three observations: a local optimum for one neighbourhood need not be a local optimum for another; a global optimum is a local optimum for every neighbourhood; and the local optima of different neighbourhoods are often close to each other.

- **Shaking** diversifies, and the size k controls how much: small k explores the solutions close to the current local optimum, larger k goes farther only when the closer ones have failed. The **local search** intensifies. Returning to $k = 1$ after every improvement concentrates the search near the new, better solution.

- The variable neighbourhood descent (VND) of the chapter «Local search» also uses several neighbourhoods, but *deterministically*: it explores them completely, one after the other, and stops at a solution that is a local optimum for all of them. VNS explores the large neighbourhoods only by *random samples*, which is what makes very large neighbourhoods usable; the local search inside VNS can itself be a VND.

- Here N_k contains the assignments obtained by moving k jobs (or fewer, if the same job is drawn twice). Since a swap of two jobs is the combination of two moves, N_2 contains the swap neighbourhood: shaking with $k = 2$ can reproduce the swaps that the move neighbourhood cannot see.

4.7.1 Example

Example 4.7: VNS on the $P\,||\,C_{\max}$ reference instance

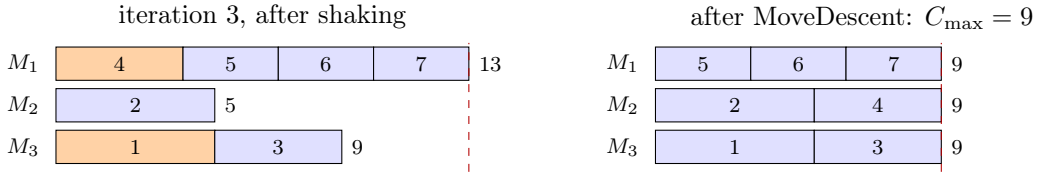
We start from the LPT solution, with loads $(11, 8, 8)$, and use $k_{\max} = 3$ and $seed = 1$ (random numbers 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789). MoveDescent does not change the LPT solution, which is a local optimum for the move neighbourhood (Example 4.5). Each random move uses two numbers: $j = 1 + \lfloor 7u \rfloor$ and $g = 1 + \lfloor 2u \rfloor$.

- **Iteration 1**, $k = 1$: job $j = 1 + \lfloor 7 \cdot 0.0000 \rfloor = 1$ moves from M_1 to the first other machine ($g = 1 + \lfloor 2 \cdot 0.1315 \rfloor = 1$), M_2: loads $(6, 13, 8)$. MoveDescent moves job 6 from M_2 to M_1: loads $(9, 10, 8)$, $C_{\max} = 10$, better than 11. The new solution $J_1 = \{5, 6, 7\}$, $J_2 = \{1, 2\}$, $J_3 = \{3, 4\}$ is accepted, and $k = 1$.

- **Iteration 2**, $k = 1$: job $j = 1 + \lfloor 7 \cdot 0.7556 \rfloor = 6$ moves from M_1 to M_2 ($g = 1 + \lfloor 2 \cdot 0.4587 \rfloor = 1$): loads $(6, 13, 8)$. MoveDescent moves job 6 back to M_1 and returns the current solution, which is not better: $k = 2$.

- **Iteration 3**, $k = 2$: job $j = 1 + \lfloor 7 \cdot 0.5328 \rfloor = 4$ moves from M_3 to the first other machine M_1 ($g = 1 + \lfloor 2 \cdot 0.2190 \rfloor = 1$); then job $j = 1 + \lfloor 7 \cdot 0.0470 \rfloor = 1$ moves from M_2 to the second other machine M_3 ($g = 1 + \lfloor 2 \cdot 0.6789 \rfloor = 2$): loads $(13, 5, 9)$. MoveDescent moves job 4 from M_1 to M_2: loads $(9, 9, 9)$. The solution is accepted, and since $C_{\max} = 9 = \text{LB}$ the search stops: $VNS(I) = 9 = z(I)$.

In terms of the current solution of iteration 3, the two random moves and the descent have exchanged jobs 1 and 4: a swap, which the move neighbourhood of MoveDescent cannot perform.



- On PCM15, from the LPT solution (73) and with at most 300 iterations, VNS gives the following makespans for $seed = 1, \ldots, 10$, and the number of iterations performed:

seed	1	2	3	4	5	6	7	8	9	10
$k_{\max} = 1$: $C_{\max}$	69	70	70	70	69	69	69	69	70	69
iterations	27	300	300	300	48	59	20	43	300	13
$k_{\max} = 2$: $C_{\max}$	70	69	69	69	69	70	69	70	70	69
iterations	300	41	126	285	33	300	31	300	300	26
$k_{\max} = 3$: $C_{\max}$	70	69	69	69	69	69	69	70	70	69
iterations	300	32	38	29	26	11	25	300	300	25

- With $k_{\max} = 3$, VNS reaches the optimum 69 in 7 runs out of 10, usually in a few tens of iterations; the best tabu search of Section 4.5.2 needed a carefully tuned tenure. When VNS fails, the search is stuck at 70: the shaking explores at most three moves away, and the optimum is farther. Larger $k_{\max}$, or restarts, can be added at the price of more iterations.

4.8 GRASP

The *greedy randomized adaptive search procedure* (denoted by $GRASP$) is a multi-start method: at every iteration it builds a solution with a *randomized* greedy algorithm, which chooses the next element at random among the best candidates (the *restricted candidate list*, RCL) instead of always taking the best one, and improves the solution with a local search. The best local optimum found is returned.

- The pseudo-code of $GRASP$ for the KP, with the randomized greedy algorithm by efficiency and KPLocalSearch, reads as follows:

Algorithm

Algorithm: GRASP for the KP

Input: KP instance $p[1..n]$, $w[1..n]$, c; parameter $\alpha \in [0,1]$; number of iterations $maxIter$

Output: best solution found

Function *GRASP*$(p, w, c, \alpha, maxIter)$
  $zbest := -1$
  for $it := 1$ **to** $maxIter$ **do**
    $x :=$ RandomizedGreedy(p, w, c, α) `// construction`
    $x :=$ KPLocalSearch(x) `// local search`
    if $\sum_j p[j]x[j] > zbest$ **then** $best := x$; $zbest := \sum_j p[j]x[j]$
  end
  return *best*

Function *RandomizedGreedy*(p, w, c, α)
  $x := (0, \ldots, 0)$; $rc := c$ `// residual capacity`
  while *some item with* $x[j] = 0$ *has* $w[j] \leq rc$ **do**
    $e_{\max}$, $e_{\min} :=$ maximum and minimum of $p[j]/w[j]$ over these candidate items
    $RCL :=$ candidates with $p[j]/w[j] \geq e_{\max} - \alpha\,(e_{\max} - e_{\min})$, by non-increasing $p[j]/w[j]$
    $j :=$ the element of RCL in position $1 + \lfloor$Rand$() \cdot |RCL| \rfloor$
    $x[j] := 1$; $rc := rc - w[j]$
  end
  return x

- *GRASP* was introduced by Feo and Resende (1989) for the set covering problem and presented as a general metaheuristic in Feo and Resende (1995). The randomized construction with a restricted candidate list is the one of the chapter «Constructive heuristics».
- The RCL is defined by a threshold on the greedy score, controlled by α:

KP (maximize the efficiency p_j/w_j of the items that fit):

$$RCL = \left\{ j \text{ candidate} : \frac{p_j}{w_j} \;\geq\; e_{\max} - \alpha\,(e_{\max} - e_{\min}) \right\}.$$

TSP (minimize the distance from the last city ℓ of the partial tour):

$$RCL = \left\{ j \text{ unvisited} : d_{\ell j} \;\leq\; d_{\min} + \alpha\,(d_{\max} - d_{\min}) \right\}.$$

- With $\alpha = 0$ the RCL contains only the best candidates, and the construction is the greedy algorithm: every iteration builds the same solution (up to ties), and *GRASP* is a single local search. With $\alpha = 1$ the RCL contains all the candidates, and the construction is random. Intermediate values give solutions that are **different** (diversification) and **good** (a good start for the local search, which then intensifies).
- The iterations of *GRASP* are **independent**: nothing is learnt from one to the next, and

the iterations can be run in parallel. Extensions add memory: *reactive* GRASP adapts α to the results, and *path relinking* explores the solutions between two good local optima.

Example 4.8: *GRASP* on the KP reference instance

With $\alpha = \frac{1}{2}$ and $seed = 2$ (random numbers 0.0000, 0.2631, 0.5112, 0.9173) we perform two iterations. The efficiencies are $\frac{10}{5} = 2$, $\frac{7}{4} = 1.75$, $\frac{6}{3} = 2$, $\frac{4}{3} \approx 1.33$.

Iteration 1. With $rc = 9$ all items fit: $e_{\max} = 2$, $e_{\min} = \frac{4}{3}$, the threshold is $2 - \frac{1}{2} \cdot \frac{2}{3} = \frac{5}{3} \approx 1.67$, and $RCL = (1, 3, 2)$. With $u = 0.0000$ the first element is chosen: item 1, $rc = 4$. The candidates are items 2, 3, 4, the threshold is again $\frac{5}{3}$, $RCL = (3, 2)$, and $u = 0.2631$ chooses position $1 + \lfloor 2 \cdot 0.2631 \rfloor = 1$: item 3, $rc = 1$, and no item fits. The construction gives $\{1, 3\}$, of profit 16; KPLocalSearch swaps item 3 out and item 2 in: $\{1, 2\}$, profit 17.

Iteration 2. $RCL = (1, 3, 2)$ and $u = 0.5112$ chooses position $1 + \lfloor 3 \cdot 0.5112 \rfloor = 2$: item 3, $rc = 6$. The candidates are items 1, 2, 4, the threshold is $\frac{5}{3}$, $RCL = (1, 2)$, and $u = 0.9173$ chooses position 2: item 2, $rc = 2$. The construction gives $\{2, 3\}$, of profit 13, a *different* and worse solution than the greedy one; the local search swaps item 3 out and item 1 in: $\{1, 2\}$, profit 17.

$GRASP(I) = 17 = z(I)$. On this small instance every start leads to the optimum; on larger instances different starts lead to different local optima.

4.8.1 GRASP for the TSP, and the effect of α

- For the TSP the randomized construction is the nearest neighbour heuristic with an RCL: from the last city ℓ of the partial tour, the next city is chosen at random among the unvisited cities whose distance from ℓ is at most $d_{\min} + \alpha(d_{\max} - d_{\min})$, and the local search is TwoOptLS.

- The table reports the results of *GRASP* with 20 iterations on TSP20 and on KP20, for 10 runs ($seed = 1, \ldots, 10$): minimum, average and maximum value of the best solution, and number of runs that reach the optimum.

	TSP20 ($z(I) = 291$)				KP20 ($z(I) = 521$)			
α	min	average	max	optimal	min	average	max	optimal
0	319	319.0	319	0	514	514.0	514	0
0.1	291	291.0	291	10	521	521.0	521	10
0.25	291	291.0	291	10	521	521.0	521	10
0.5	291	291.0	291	10	521	521.0	521	10
1	291	291.0	291	10	519	520.8	521	9

- With $\alpha = 0$ all the iterations are the same: the result is the local optimum reached from the greedy solution (319 for TSP20, 514 for KP20), in every run. A little randomization is enough to reach the optimum in every run on these instances; with $\alpha = 1$ the constructions are random, and on KP20 one run misses the optimum.

4.9 Large neighbourhood search

Large neighbourhood search (denoted by LNS) improves a solution by *destroying* a part of it and *repairing* it: a destroy method removes some elements (items, cities, jobs) from the current solution, a repair method reinserts them, and the new solution replaces the current one if an acceptance criterion is satisfied. The neighbourhood of a solution is the set of all the solutions that destroy and repair can produce, and it is *large*.

- The pseudo-code of LNS for the TSP, which removes q random cities and reinserts them by cheapest insertion, reads as follows:

Algorithm

Algorithm: Large neighbourhood search for the TSP

Input: distances d, starting tour σ, number q of cities to remove, number of iterations $maxIter$

Output: the current tour at the end

Function *LNS*$(d, \sigma, q, maxIter)$
  $cur := \sigma$; $lcur := L(cur)$
  for $it := 1$ **to** $maxIter$ **do**
    $nw := cur$
    $removed :=$ RandomRemoval(nw, q) `// destroy`
    CheapestInsertionRepair$(nw, removed)$ `// repair`
    if $L(nw) < lcur$ **then** $cur := nw$; $lcur := L(nw)$ `// acceptance`
  end
  return cur

Function *RandomRemoval*(σ, q)
  for $s := 1$ **to** q **do**
    $k := 2 + \lfloor$Rand$() \cdot (|\sigma| - 1) \rfloor$ `// city 1 is never removed`
    append $\sigma[k]$ to $removed$, and delete it from σ
  end
  return $removed$

Function *CheapestInsertionRepair*$(\sigma, removed)$
  foreach *city* v *of* $removed$*, in the order of removal* **do**
    insert v between the consecutive cities a, b of σ of minimum $d_{av} + d_{vb} - d_{ab}$
  end

- LNS was introduced by Shaw (1998) for vehicle routing problems, where the repair was done by constraint programming. The **size** of the neighbourhood grows very quickly with the number q of removed elements: for the TSP there are $\binom{n-1}{q}$ ways to choose the removed cities, and many ways to reinsert them. Such a neighbourhood cannot be listed and evaluated move by move, as in a local search: it is explored **implicitly**, by the repair method.
- The repair method is a **sub-algorithm**: a constructive heuristic (cheapest insertion, FFD), or an *exact* algorithm for the subproblem left by the destroy method. The stronger

the repair, the better the best neighbour found; the larger q, the larger the neighbourhood, and the more expensive its exploration.

- The destroy method diversifies (it is usually random, and q controls how far the search goes); the repair method intensifies (it rebuilds the destroyed part in the best way it can). Removing *related* elements (cities close to each other, items of the same bins) makes the repair more effective than removing unrelated ones, which can only be put back where they were.

Example 4.9: *LNS* on the TSP reference instance

From the nearest neighbour tour $(1,3,2,4,5)$ of length 25, with $q=2$ and $seed=1$. The first random number 0.0000 removes the city in position $2+\lfloor 4\cdot 0.0000\rfloor = 2$, city 3; the partial tour is $(1,2,4,5)$, and 0.1315 removes the city in position $2+\lfloor 3\cdot 0.1315\rfloor = 2$, city 2. The partial tour $(1,4,5)$ has length $2+7+9=18$. Cheapest insertion of city 3:

$$(1,4):\ d_{13}+d_{34}-d_{14}=2+4-2=4, \qquad (4,5):\ 4+7-7=4, \qquad (5,1):\ 7+2-9=0,$$

so city 3 goes between 5 and 1: $(1,4,5,3)$. Cheapest insertion of city 2:

$$(1,4):\ 5+3-2=6, \qquad (4,5):\ 3+4-7=0, \qquad (5,3):\ 4+4-7=1,$$
$$(3,1):\ 4+5-2=7,$$

so city 2 goes between 4 and 5: $(1,4,2,5,3)$, of length 18, accepted. It is the optimal tour, traversed in the opposite direction: one destroy-and-repair step has changed four edges of the tour.

4.9.1 Large neighbourhood search for the BPP: heuristic and exact repair

- For the BPP the natural destroy method **empties bins**: the least loaded bin, which is the one we would like to get rid of, and $q-1$ other bins chosen at random. The repair method repacks the removed items into new bins, while the other bins are kept as they are. The new packing is accepted if it uses fewer bins, or the same number of bins and a value of the auxiliary objective $\sum_b (\sum_{j\in P_b} w_j)^2$ not smaller: on the plateaus of the flat objective, the search moves toward fuller bins.

- The repair solves a **BPP subproblem**, on the removed items only, and can do it in two ways: with FFD, or **exactly**, by a branch-and-bound that tries every bin for every item and keeps the packing with the fewest bins. The subproblem has only the items of q bins, so an exact repair is affordable.

Algorithm

Algorithm: Large neighbourhood search for the BPP

Input: weights $w[1..n]$, capacity c, packing $bin[1..n]$ with nb bins, q, *exact*, *maxIter*
Output: a packing bin and its number of bins nb

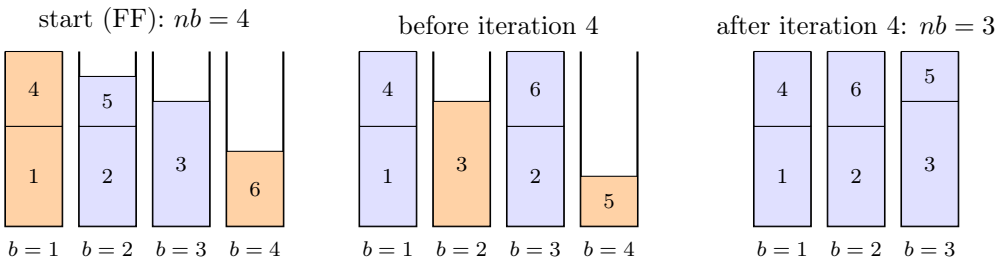
Function *LNS-BPP*$(w, c, bin, nb, q, exact, maxIter)$
  LB $:= \lceil \sum_j w[j]/c \rceil$; $it := 0$
  while $it < maxIter$ ***and*** $nb >$ LB **do**
    $it := it + 1$
    empty := the least loaded bin and $q-1$ other bins at random `// destroy`
    items := the items packed in the bins of *empty*
    newbin := the other bins, renumbered $1, \ldots, nb - q$
    if *exact* **then** pack *items* optimally in k new bins **else** pack *items* by FFD in k new bins `// repair`
    $nb' := nb - q + k$
    if $nb' < nb$ ***or*** $(nb' = nb$ ***and*** $\sum_b (load\ of\ b\ in\ newbin)^2 \geq \sum_b (load\ of\ b\ in\ bin)^2)$ **then**
      $bin := newbin$; $nb := nb'$ `// acceptance`
    end
  end
  return *bin*, *nb*

Example 4.10: *LNS* on the BPP reference instance

We start from the packing computed by First Fit, $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, with loads $(7, 6, 5, 3)$, and use $q = 2$, repair by FFD and $seed = 3$ (random numbers 0.0000, 0.3946, 0.2668, 0.3760). At every iteration the least loaded bin is emptied, together with the bin in position $1 + \lfloor 3u \rfloor$ among the other three.

it	loads	emptied	items	FFD	nb'	$\sum_b (\cdot)^2$
1	$(7, 6, 5, 3)$	P_4, P_1	$\{1, 4, 6\}$	$\{1, 4\}, \{6\}$	4	$119 \to 119$
2	$(6, 5, 7, 3)$	P_4, P_2	$\{3, 6\}$	$\{3\}, \{6\}$	4	$119 \to 119$
3	$(6, 7, 5, 3)$	P_4, P_1	$\{2, 5, 6\}$	$\{2, 6\}, \{5\}$	4	$119 \to 127$
4	$(7, 5, 7, 2)$	P_4, P_2	$\{3, 5\}$	$\{3, 5\}$	3	$127 \to 147$

Iterations 1 and 2 only reshuffle the items and keep the same loads (the auxiliary objective stays 119, and the new packing is accepted because it is not worse); iteration 3 builds the full bin $\{2, 6\}$, raising the auxiliary objective to 127; iteration 4 empties the bins $\{5\}$ and $\{3\}$, and FFD packs items 3 and 5 together: 3 bins, the optimum, and the search stops since $nb =$ LB.



- On BPP24, starting from the FFD packing (10 bins), with 200 iterations and $seed = 1, \ldots, 10$, the result depends on the repair:

q	repair	bins at the end (10 seeds)	iterations to reach 9
2	FFD	$10, 10, 10, 10, 10, 10, 10, 10, 10, 10$	never
2	exact	$10, 10, 10, 10, 10, 10, 10, 10, 10, 10$	never
3	FFD	$10, 10, 10, 10, 10, 10, 10, 10, 10, 10$	never
3	exact	$9, 9, 9, 9, 9, 9, 9, 9, 9, 9$	$4, 4, 12, 38, 3, 20, 5, 2, 7, 5$
4	FFD	$10, 10, 10, 10, 10, 10, 10, 10, 10, 10$	never
4	exact	$9, 9, 9, 9, 9, 9, 9, 9, 9, 9$	$2, 3, 9, 12, 4, 5, 1, 3, 10, 2$

- With the FFD repair, *LNS* **never** improves the FFD packing: the items removed from an FFD packing are repacked by FFD in the same way. With the exact repair and $q \geq 3$ it reaches the optimum in every run, in a few iterations: it finds three bins whose items fit into two, which is what FFD cannot see. With $q = 2$ even the exact repair fails, because the items of two bins of an FFD packing never fit into one.

A large neighbourhood is explored by solving a **subproblem**: the part of the solution left free by the destroy method is re-optimized, while the rest is fixed. The more powerful the method that solves the subproblem, the more powerful the neighbourhood.

4.9.2 Adaptive large neighbourhood search

- Which destroy method works best depends on the instance, and on the phase of the search. **Adaptive** LNS (ALNS), introduced by Ropke and Pisinger (2006), uses several destroy (and repair) methods, chooses one at random at every iteration, and **learns** which ones work, by giving to each method o a weight w_o:

$$P(\text{choose } o) = \frac{w_o}{\sum_{o'} w_{o'}}, \qquad w_o \leftarrow (1-\rho)\, w_o + \rho\, \psi,$$

where $\rho \in [0, 1]$ is the *reaction factor* and the score ψ rewards the outcome of the iteration in which o was used: $\psi = 3$ if the new solution is the best found so far, 2 if it is better than the current one, 1 if it is as good, 0 if it is worse.

Algorithm

Algorithm: Adaptive large neighbourhood search for the TSP

Input: distances d, starting tour σ, q, $maxIter$, reaction factor ρ
Output: best tour found

Function $ALNS(d, \sigma, q, maxIter, \rho)$
  $cur := \sigma$; $best := \sigma$; $w_1 := 1$; $w_2 := 1$
  for $it := 1$ **to** $maxIter$ **do**
    if $\textsc{Rand}() \cdot (w_1 + w_2) < w_1$ **then** $o := 1$ **else** $o := 2$ `// roulette wheel`
    $nw := cur$
    if $o = 1$ **then** $removed := \textsc{RandomRemoval}(nw, q)$ **else** $removed := \textsc{WorstRemoval}(nw, q)$
    $\textsc{CheapestInsertionRepair}(nw, removed)$
    $\psi := 3$, 2, 1 or 0 if $L(nw) < L(best)$, $L(nw) < L(cur)$, $L(nw) = L(cur)$, $L(nw) > L(cur)$
    if $L(nw) \leq L(cur)$ **then** $cur := nw$
    if $L(cur) < L(best)$ **then** $best := cur$
    $w_o := (1 - \rho)\, w_o + \rho\, \psi$ `// adaptive weights`
  end
  return $best$

- WorstRemoval removes, q times, the city whose removal shortens the tour the most: the city v, between a and b, of maximum $d_{av} + d_{vb} - d_{ab}$. It is a deterministic, greedy destroy method: it removes the cities that seem badly placed.
- On TSP20, from the nearest neighbour tour, with $q = 5$, 200 iterations and $\rho = 0.1$, ALNS reaches the optimum in 7 runs out of 10 (average 299.4).
- At the end the weight of the random removal is between 0.77 and 0.98, the weight of the worst removal between 0.03 and 0.08: the method has learnt that, from a good tour, removing the worst-placed cities and reinserting them by cheapest insertion almost always rebuilds a tour that is not better, and it has almost stopped using it.
- In these runs plain LNS with random removal ($q = 5$) does slightly better: adaptivity pays off when there are many methods, of unknown quality, and long runs.

4.10 Population-based methods: genetic and memetic algorithms

A *genetic algorithm* (denoted by GA) keeps a *population* of solutions, encoded as strings (*chromosomes*). At every iteration it *selects* two parents, preferring the fitter ones, *recombines* them into a child (*crossover*), changes a few random genes of the child (*mutation*), *repairs* it if it is infeasible, and inserts it into the population in place of a worse individual. A *memetic algorithm* also applies a local search to every child.

- The pseudo-code of a steady-state GA for the KP with binary encoding reads as follows; with *memetic* = true it is a memetic algorithm.

Algorithm

Algorithm: Genetic and memetic algorithm for the KP

Input: KP instance $p[1..n]$, $w[1..n]$, c; population size P; mutation probability p_m; $maxIter$; *memetic*

Output: best individual of the final population

Function *GeneticKP*$(p, w, c, P, p_m, maxIter, memetic)$
  for $s := 1$ **to** P **do** `// random initial population`
    for $j := 1$ **to** n **do** $pop[s][j] := 1$ if RAND() < 0.5, else 0
    REPAIRKP($pop[s]$); **if** *memetic* **then** KPLOCALSEARCH($pop[s]$)
    $fit[s] := \sum_j p[j]\, pop[s][j]$
  end
  for $it := 1$ **to** $maxIter$ **do**
    $a :=$ TOURNAMENT(); $b :=$ TOURNAMENT() `// selection`
    $cut := 1 + \lfloor$RAND$() \cdot (n-1) \rfloor$ `// one-point crossover`
    $child := (pop[a][1..cut],\ pop[b][cut{+}1..n])$
    for $j := 1$ **to** n **do if** RAND$() < p_m$ **then** $child[j] := 1 - child[j]$
    `// mutation`
    REPAIRKP($child$); **if** *memetic* **then** KPLOCALSEARCH($child$)
    $worst :=$ the last index s of minimum $fit[s]$
    if $child \notin pop$ ***and*** $\sum_j p[j]\, child[j] > fit[worst]$ **then** `// replacement`
      $pop[worst] := child$; $fit[worst] := \sum_j p[j]\, child[j]$
    end
  end
  return *the individual of maximum fit*
Function *Tournament()*
  $a := 1 + \lfloor$RAND$() \cdot P \rfloor$; $b := 1 + \lfloor$RAND$() \cdot P \rfloor$
  if $fit[a] \geq fit[b]$ **then return** a **else return** b

- Genetic algorithms were introduced by Holland (1975) as a model of natural evolution, and made popular as optimization methods by Goldberg (1989). Memetic algorithms, in which every individual is improved by a local search, were proposed by Moscato (1989).

- **Encoding.** For the KP the binary vector x is a natural chromosome: the gene j says whether item j is selected. The **fitness** of an individual is its profit.
- **Selection.** A binary tournament draws two individuals at random and keeps the fitter one: good individuals are more likely to become parents, but every individual has a chance.
- **Crossover.** One-point crossover takes the genes $1, \ldots, cut$ from the first parent and the genes $cut + 1, \ldots, n$ from the second one. The idea is that good solutions share good *building blocks*, and recombining two good parents can put the good blocks of both into one child.
- **Mutation.** Each gene is flipped with a small probability p_m, typically $\frac{1}{n}$: it brings back into the population genes that selection may have lost.
- **Repair.** The child can be infeasible. RepairKP removes the selected item of minimum efficiency until the capacity is respected (ties: the item of largest index). Alternatives are penalties, as in Section 4.4.3, or encodings in which every chromosome is feasible.
- **Replacement.** In the steady-state version of the pseudo-code, the child replaces the worst individual if it is better and not already in the population; duplicates would make the population lose its diversity.
- Selection and replacement **intensify** (the population concentrates on good solutions); crossover and mutation **diversify**. Without local search, a *GA* is good at finding promising regions but slow at reaching their local optima: the memetic algorithm adds the missing intensification.

Example 4.11: *GA* on the KP reference instance

With $P = 4$, $p_m = \frac{1}{4}$ and $seed = 36$. The first 16 random numbers build the initial population (gene 1 if the number is smaller than 0.5):

$$\begin{aligned} pop[1]: &\ 0.0003,\ 0.7354,\ 0.2018,\ 0.5114 \ \rightarrow (1,0,1,0) \\ pop[2]: &\ 0.1796,\ 0.8825,\ 0.6936,\ 0.4391 \ \rightarrow (1,0,0,1) \\ pop[3]: &\ 0.4547,\ 0.6489,\ 0.8061,\ 0.6990 \ \rightarrow (1,0,0,0) \\ pop[4]: &\ 0.9148,\ 0.2446,\ 0.9246,\ 0.0692 \ \rightarrow (0,1,0,1) \end{aligned}$$

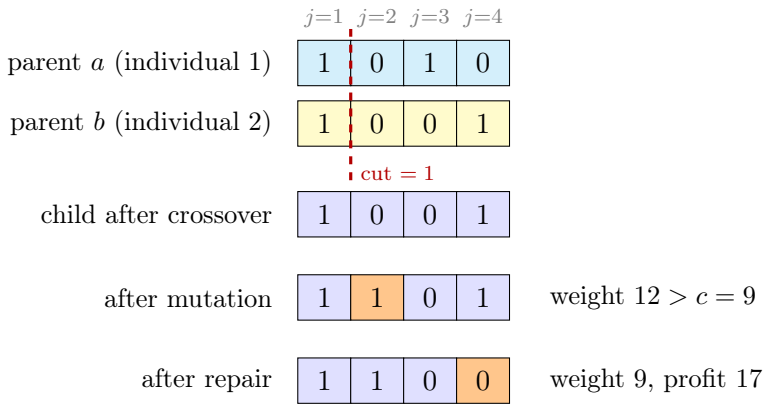
All four individuals are feasible: $pop[1] = (1,0,1,0)$, $pop[2] = (1,0,0,1)$, $pop[3] = (1,0,0,0)$, $pop[4] = (0,1,0,1)$, with fitness $(16, 14, 10, 11)$.

Iteration 1 (random numbers 0.1614, 0.2771, 0.8030, 0.4063, 0.0295, 0.7238, 0.2032, 0.4957, 0.4620). The first tournament draws individuals $1 + \lfloor 4 \cdot 0.1614 \rfloor = 1$ and $1 + \lfloor 4 \cdot 0.2771 \rfloor = 2$, and selects $a = 1$ (fitness $16 > 14$); the second draws 4 and 2 and selects $b = 2$ (fitness $14 > 11$). The cut is $1 + \lfloor 3 \cdot 0.0295 \rfloor = 1$: the child takes gene 1 from $pop[1]$ and genes 2, 3, 4 from $pop[2]$, $child = (1,0,0,1)$. Only the second mutation number, 0.2032, is smaller than $p_m = 0.25$: gene 2 flips, $child = (1,1,0,1)$, of weight $12 > 9$. The repair removes item 4, the selected item of minimum efficiency: $child = (1,1,0,0)$, of weight 9 and profit 17. It replaces the worst individual, $pop[3]$ of fitness 10.

Iteration 2 selects $a = 2$ and $b = 3$, cuts after gene 1 and builds $(1,1,0,0)$, with no mutation: it is already in the population, and it is discarded. **Iteration 3** selects $a = 2$

and $b = 1$, cuts after gene 3, builds $(1,0,0,0)$, and the mutation flips genes 1 and 4: $(0,0,0,1)$, of profit 4, worse than every individual, discarded.

The best individual is $(1,1,0,0)$: $GA(I) = 17 = z(I)$, found by crossover, mutation and repair together.



- On KP20, with $P = 10$, $p_m = 0.05$ and 300 iterations, for $seed = 1, \ldots, 10$:

	min	average	max	optimal	best profit per seed
GA	508	516.9	521	4	521, 519, 521, 508, 514, 521, 510, 519, 521, 515
memetic	521	521.0	521	10	521, 521, 521, 521, 521, 521, 521, 521, 521, 521

- The plain GA reaches the optimum 521 in 4 runs out of 10; the memetic algorithm in every run. The local search makes every individual a local optimum, and the crossover then recombines local optima instead of arbitrary solutions.

4.11 Comparison and conclusions

4.11.1 The methods on the medium instances

- The table collects the results of the experiments of the previous sections: for each method, the best value found in 10 runs ($seed = 1, \ldots, 10$), as minimum, average and maximum, and the number of runs that reach the optimum. Deterministic methods give the same value in every run.

instance	method	parameters	min	average	max	optimal
TSP20	NN	—	326	326	326	0
	NN + 2-opt	—	319	319	319	0
	SA	$T_0 = 20$, $\alpha = 0.95$	291	294.3	298	5
	SA	$T_0 = 20$, $\alpha = 0.99$	291	291.0	291	10
	TS	*tenure* = 5, 200 it.	291	291	291	10
	ILS	100 it.	291	291.0	291	10
	GRASP	$\alpha = 0.25$, 20 it.	291	291.0	291	10
	LNS	$q = 5$, 200 it.	291	293.8	319	9
	ALNS	$q = 5$, 200 it.	291	299.4	319	7
KP20	greedy	—	490	490	490	0
	greedy + local search	—	514	514	514	0
	GRASP	$\alpha = 0.25$, 20 it.	521	521.0	521	10
	GA	$P = 10$, 300 it.	508	516.9	521	4
	memetic	$P = 10$, 300 it.	521	521.0	521	10
PCM15	LPT	—	73	73	73	0
	TS	*tenure* = 15, 300 it.	69	69	69	10
	VNS	$k_{\max} = 3$, 300 it.	69	69.3	70	7
BPP24	FFD	—	10	10	10	0
	LNS, FFD repair	$q = 3$, 200 it.	10	10.0	10	0
	LNS, exact repair	$q = 3$, 200 it.	9	9.0	9	10

- Every metaheuristic improves on the local search it is built on, and with suitable parameters several of them reach the optimum in every run. The table, however, also shows how much the result depends on the parameters (the tenure of *TS*, the strength q of the destruction, the cooling of *SA*) and on the components (the repair of *LNS*, the local search of the memetic algorithm). On four small instances it does not rank the methods: such a ranking needs many instances and equal running times, as discussed in the chapter «Heuristic algorithms: definitions».

4.11.2 Intensification and diversification

- Every metaheuristic of this chapter combines a component that **intensifies** (searches around the good solutions already found) and a component that **diversifies** (moves the search to unexplored regions), and has parameters that set the balance between the two.

method	intensification	diversification	balance set by
SA	accepting only improving moves at low temperature	accepting worse moves at high temperature	T_0, cooling factor α, L
TS	moving to the best neighbour; restarting from elite solutions	tabu list; penalties on frequent attributes	tenure, aspiration criterion
ILS	local search after every kick; accepting only better local optima	the kick	strength of the kick, acceptance criterion
VNS	local search; return to $k = 1$ after an improvement	shaking in N_k with growing k	$k_{\max}$, the neighbourhoods N_k
GRASP	local search on every constructed solution	randomized construction	α
LNS / ALNS	repair (heuristic or exact), acceptance	random destroy	q, choice of the methods, weights
GA / memetic	selection, replacement, local search (memetic)	crossover, mutation	population size, p_m

4.11.3 Parameters and tuning

- A metaheuristic is an algorithm only after its parameters are fixed, and the experiments of this chapter show that the parameters can make the difference between finding the optimum in every run and never finding it. Choosing them is called **tuning**.
- The parameters must be tuned on **training** instances and evaluated on *different* ones, otherwise the picture is optimistic. Tuning one parameter at a time ignores their interactions (for SA, the best T_0 depends on α); tools such as `irace` (López-Ibáñez et al., 2016) tune all of them together. A method that works well for a wide range of parameters is preferable to one that works very well only for a narrow range.

4.11.4 What these methods do not use

- None of the metaheuristics of this chapter uses a **relaxation** of the problem or a **MIP solver**. They manipulate feasible (or penalized) solutions only, and they produce only *primal* bounds.
- As a consequence, they cannot tell how good their solutions are. On the medium instances we could measure the quality only because the optimal values had been computed separately, by exact algorithms.
- The two exceptions confirm the rule: for $P\,||\,C_{\max}$ and the BPP, TS, VNS and LNS stopped as soon as they reached a simple lower bound, which is a *dual* bound that certifies optimality; when the optimum is above the bound, no such certificate exists, and the search goes on until the iteration limit.
- The exact repair of LNS (Section 4.9.1), finally, showed that re-optimizing a part of the solution exactly, with the rest fixed, gives very powerful neighbourhoods.

Metaheuristics escape from local optima, but they give only one side of the gap: the primal bound.

4.12 Typical mistakes

Warning

- **Reporting one run.** The results depend on the seed: report minimum, average and maximum over several runs.
- **Tuning and evaluating on the same instances.** The picture is optimistic; use training instances for the parameters and different ones for the evaluation.
- **Losing the best solution.** In simulated annealing and tabu search the current solution can be worse than the best one found: the best is stored apart.
- **Choosing the tenure without testing it.** With tenure 0 tabu search is a local search; a tenure that is too small cycles, one that is too large forbids almost every move.
- **Miscounting the random numbers.** Every iteration consumes a fixed number

of them (ten in the genetic algorithm of the exercises, with $n = 5$); a wrong count changes every later step.

4.13 In summary

Summary

- A metaheuristic escapes from local optima by accepting worse solutions (SA, TS), by perturbing (ILS, VNS), by randomizing the construction (GRASP), by destroying and repairing (LNS) or by recombining a population (GA).
- Each method balances intensification and diversification through a few parameters, and the results depend strongly on them.
- They use no relaxation: they produce primal bounds only, and cannot tell how good their solutions are.

Dashboard of the medium instances. Best value of the methods against the optimum (the optimum computed by an exact algorithm):

instance	constructive	local search	best metaheuristic	optimum
TSP20 (min)	326 (NN)	319 (2-opt)	291	291
KP20 (max)	490 (greedy)	514	521	521
PCM15 (min)	73 (LPT)	—	69	69
BPP24 (min)	10 (FFD)	—	9	9

4.14 C++ implementation

- All the functions of this section are in the folder `METAHEUR_CODE/src`, one function per file, and are declared in the header `Metaheuristics.h`. The items, cities and jobs are indexed $0, 1, \ldots, n-1$, the machines $0, \ldots, m-1$ and the bins $0, \ldots, nb-1$: the algorithmic logic is the one of the pseudo-codes, with 0-based indices. Vectors are `Vec` (`vector<int>`) and matrices `Matrix` (`vector<Vec>`).
- The command `make test` compiles the code and runs every example of this chapter and every instance of the exercises, checking the value found against the expected one; `make data` writes the data of the figure of Section 4.4.4.
- The random numbers are produced by `Rand`, the generator of Section 4.2.2. The structure `Random` contains the seed and, optionally, a list of given numbers, which are returned instead: the exercises give their random numbers explicitly.

```
// Park-Miller minimal standard generator: seed = 16807 * seed mod (2^31 - 1),
    u = seed /
// (2^31 - 1)
// if a list of numbers is given, they are returned in order instead
```

```
4
5  double Rand(Random& r)
6  {
7      double u;
8      if (r.given.size() > 0)
9      {
10         u = r.given[r.used];
11     }
12     else
13     {
14         r.seed = (16807 * r.seed) % 2147483647;
15         u = (double) r.seed / 2147483647.0;
16     }
17     r.used = r.used + 1;
18     return u;
19 }
```

4.14.1 Local searches and auxiliary functions

- The 2-opt local search for the TSP, with the $O(1)$ evaluation of a move:

```
1  // length L(pi) of the tour pi
2
3  int TourLength(const Vec& pi, const Matrix& d)
4  {
5      int n = pi.size();
6      int len = 0;
7      for (int k = 0; k < n; ++k)
8      {
9          len = len + d[pi[k]][pi[(k + 1) % n]];
10     }
11     return len;
12 }
```

```
1  // change of length of the 2-opt move (i,j): the edges (pi[i],pi[i+1]) and (pi[
       j],pi[j+1])
2  // are replaced by (pi[i],pi[j]) and (pi[i+1],pi[j+1]); computed in O(1)
3
4  int TwoOptDelta(const Vec& pi, const Matrix& d, int i, int j)
5  {
6      int n = pi.size();
7      int a = pi[i], b = pi[i + 1], c = pi[j], e = pi[(j + 1) % n];
8      return d[a][c] + d[b][e] - d[a][b] - d[c][e];
9  }
```

```
1  // performs the 2-opt move (i,j): reverses the segment pi[i+1..j]
2
3  void ApplyTwoOpt(Vec& pi, int i, int j)
4  {
5      int lo = i + 1, hi = j;
6      while (lo < hi)
7      {
8          int tmp = pi[lo]; pi[lo] = pi[hi]; pi[hi] = tmp;
9          lo = lo + 1; hi = hi - 1;
10     }
11 }
```

```cpp
// 2-opt local search with best improvement: stops at a local optimum
// ties between moves are broken in favour of the first move (i,j) in lexicographic order

void TwoOptLS(Vec& pi, const Matrix& d)
{
    int n = pi.size();
    while (true)
    {
        int bdelta = 0, bi = -1, bj = -1;
        for (int i = 0; i < n - 1; ++i)
        {
            for (int j = i + 2; j < n; ++j)
            {
                if (i == 0 && j == n - 1) continue;      // same two edges: no change
                int delta = TwoOptDelta(pi, d, i, j);
                if (delta < bdelta) { bdelta = delta; bi = i; bj = j; }
            }
        }
        // no improving move: local optimum
        if (bi < 0) return;
        ApplyTwoOpt(pi, bi, bj);
    }
}
```

```cpp
// nearest neighbour tour from city 0 (ties: smallest index)

Vec NearestNeighbour(const Matrix& d)
{
    int n = d.size();
    Vec pi(1, 0);
    vector<bool> visited(n, false);
    visited[0] = true;
    for (int k = 1; k < n; ++k)
    {
        int last = pi.back(), next = -1;
        for (int j = 0; j < n; ++j)
        {
            if (!visited[j] && (next < 0 || d[last][j] < d[last][next])) next = j;
        }
        pi.push_back(next);
        visited[next] = true;
    }
    return pi;
}
```

- The move descent for $P\,||\,C_{\max}$ with the lexicographic comparison, and the LPT starting solution:

```cpp
// loads L[h] of the machines for the assignment mach

Vec Loads(const Vec& mach, const Vec& t, int m)
{
    Vec L(m, 0);
```

```
    for (int j = 0; j < (int) t.size(); ++j) L[mach[j]] = L[mach[j]] + t[j];
    return L;
}
```

```
// true if the loads L1 are better than L2 in the lexicographic order
// (makespan first, then sum of the squared loads)

bool LexLess(const Vec& L1, const Vec& L2)
{
    int C1 = 0, C2 = 0, S1 = 0, S2 = 0;
    for (int h = 0; h < (int) L1.size(); ++h)
    {
        if (L1[h] > C1) C1 = L1[h];
        if (L2[h] > C2) C2 = L2[h];
        S1 = S1 + L1[h] * L1[h];
        S2 = S2 + L2[h] * L2[h];
    }
    return C1 < C2 || (C1 == C2 && S1 < S2);
}
```

```
// local search for P||Cmax with the move neighbourhood and best improvement,
// with respect to the lexicographic order (makespan, sum of squared loads)

void MoveDescent(const Vec& t, int m, Vec& mach)
{
    int n = t.size();
    Vec L = Loads(mach, t, m);
    while (true)
    {
        int bj = -1, bh = -1;
        Vec bL = L;
        for (int j = 0; j < n; ++j)
        {
            for (int h = 0; h < m; ++h)
            {
                if (h == mach[j]) continue;
                Vec L2 = L;
                L2[mach[j]] = L2[mach[j]] - t[j];
                L2[h] = L2[h] + t[j];
                if (LexLess(L2, bL)) { bj = j; bh = h; bL = L2; }
            }
        }
        if (bj < 0) return;                                  // local optimum
        mach[bj] = bh;
        L = bL;
    }
}
```

```
// longest processing time first: jobs by non-increasing t (ties: smallest
    index),
// each assigned to the least loaded machine (ties: smallest index)

Vec LPT(const Vec& t, int m)
{
    int n = t.size();
    Vec order(n), mach(n), L(m, 0);
    for (int j = 0; j < n; ++j) order[j] = j;
```

```
    for (int a = 1; a < n; ++a)                              // insertion sort, stable
    {
        int j = order[a], b = a;
        while (b > 0 && t[order[b - 1]] < t[j]) { order[b] = order[b - 1]; b = b - 1; }
        order[b] = j;
    }
    for (int a = 0; a < n; ++a)
    {
        int j = order[a], h = 0;
        for (int k = 1; k < m; ++k) if (L[k] < L[h]) h = k;
        mach[j] = h;
        L[h] = L[h] + t[j];
    }
    return mach;
}
```

- The local search for the KP with feasible flips and swaps:

```
// total profit of the items selected by x

int Profit(const Vec& p, const Vec& x)
{
    int P = 0;
    for (int i = 0; i < (int) p.size(); ++i) if (x[i] == 1) P = P + p[i];
    return P;
}
```

```
// total weight of the items selected by x

int Weight(const Vec& w, const Vec& x)
{
    int W = 0;
    for (int i = 0; i < (int) w.size(); ++i) if (x[i] == 1) W = W + w[i];
    return W;
}
```

```
// local search for the KP with best improvement on the feasible neighbours of the flip
// neighbourhood (only additions can improve) and of the swap neighbourhood

void KPLocalSearch(const Vec& p, const Vec& w, int c, Vec& x)
{
    int n = p.size();
    while (true)
    {
        int W = Weight(w, x), P = Profit(p, x);
        // bi = -1: add item bk; else swap bi out, bk in
        int bP = P, bi = -1, bk = -1;
        for (int k = 0; k < n; ++k)
            if (x[k] == 0 && W + w[k] <= c && P + p[k] > bP)
            { bP = P + p[k]; bi = -1; bk = k; }
        for (int i = 0; i < n; ++i)
        {
            if (x[i] != 1) continue;
```

```
            for (int k = 0; k < n; ++k)
                if (x[k] == 0 && W - w[i] + w[k] <= c && P - p[i] + p[k] > bP)
                { bP = P - p[i] + p[k]; bi = i; bk = k; }
        }
        if (bk < 0) return;                                  // local optimum
        if (bi >= 0) x[bi] = 0;
        x[bk] = 1;
    }
}
```

4.14.2 Simulated annealing

- Simulated annealing for the TSP (Section 4.4, with the 2-opt neighbourhood), and for the KP with the penalized objective of Section 4.4.3:

```
// simulated annealing for the TSP with the 2-opt neighbourhood and geometric cooling
// pi: starting tour on input, best tour found on output; returns its length
// if curve is not NULL, the length of the current tour after each move is appended to it

int SimulatedAnnealingTSP(const Matrix& d, Vec& pi, double T0, double alpha, int L,
                          double Tmin, Random& r, Vec* curve)
{
    int n = pi.size();
    // the M = n(n-3)/2 moves (i,j) of 2-opt
    Vec mi, mj;
    for (int i = 0; i < n - 1; ++i)
        for (int j = i + 2; j < n; ++j)
            if (!(i == 0 && j == n - 1)) { mi.push_back(i); mj.push_back(j); }
    int M = mi.size();

    Vec cur = pi;                                            // current solution
    int fcur = TourLength(cur, d);
    Vec best = cur;                                          // best solution found
    int fbest = fcur;
    double T = T0;                                           // temperature
    while (T > Tmin)
    {
        for (int l = 0; l < L; ++l)                          // L moves at temperature T
        {
            int k = (int) (Rand(r) * M);                     // random neighbour
            int delta = TwoOptDelta(cur, d, mi[k], mj[k]);
            double v = Rand(r);
            if (delta <= 0 || v < exp(-delta / T))           // Metropolis acceptance rule
            {
                ApplyTwoOpt(cur, mi[k], mj[k]);
                fcur = fcur + delta;
                if (fcur < fbest) { best = cur; fbest = fcur; }
            }
            if (curve != NULL) curve->push_back(fcur);
        }
        T = alpha * T;                                       // cooling
```

```
    }
    pi = best;
    return fbest;
}
```

```
// simulated annealing for the KP with the flip neighbourhood and the penalized objective
// f(x) = profit(x) - beta * max{0, weight(x) - c}; infeasible solutions may be visited,
// the best solution is updated with feasible solutions only
// x: feasible starting solution on input, best solution found on output; returns its
// profit

int SimulatedAnnealingKP(const Vec& p, const Vec& w, int c, Vec& x, int beta,
                         double T0, double alpha, int L, double Tmin, Random& r)
{
    int n = p.size();
    int excess = Weight(w, x) - c;
    int f = Profit(p, x) - beta * (excess > 0 ? excess : 0);
    Vec best = x;
    int zbest = Profit(p, x);
    double T = T0;
    while (T > Tmin)
    {
        for (int l = 0; l < L; ++l)
        {
            int k = (int) (Rand(r) * n);                    // random flip
            Vec y = x;
            y[k] = 1 - y[k];
            int ey = Weight(w, y) - c;
            int fy = Profit(p, y) - beta * (ey > 0 ? ey : 0);
            // maximization: delta > 0 is a worsening
            int delta = f - fy;
            double v = Rand(r);
            if (delta <= 0 || v < exp(-delta / T))
            {
                x = y; f = fy;
                if (ey <= 0 && Profit(p, x) > zbest) { best = x; zbest = Profit(p, x); }
            }
        }
        T = alpha * T;
    }
    x = best;
    return zbest;
}
```

4.14.3 Tabu search

- Tabu search for $P||C_{\max}$ (Section 4.5) and for the TSP (Section 4.5.2):

```
// tabu search for P||Cmax with the move neighbourhood (one job to another machine);
// the best non-tabu move is performed even if it is worse (makespan first, then sum of
// squares);
// after moving job j away from machine h, the attribute (j,h) is tabu for tenure
// iterations;
// a tabu move is allowed if it gives a makespan smaller than the best one (aspiration)
// mach: starting assignment on input, best assignment on output; returns its makespan

int TabuSearchPCmax(const Vec& t, int m, Vec& mach, int tenure, int maxIter)
{
    int n = t.size();
    Vec L = Loads(mach, t, m);
    // tabu[j][h] >= it: j may not go to h
    Matrix tabu(n, Vec(m, 0));
    Vec best = mach;
    int bestC = 0, sum = 0, LB = 0;
    for (int h = 0; h < m; ++h) if (L[h] > bestC) bestC = L[h];
    for (int j = 0; j < n; ++j) { sum = sum + t[j]; if (t[j] > LB) LB = t[j]; }
    // LB = max{ max t_j, ceil(sum/m) }
    if ((sum + m - 1) / m > LB) LB = (sum + m - 1) / m;
    int it = 0;
    while (it < maxIter && bestC > LB)
    {
        it = it + 1;
        int bj = -1, bh = -1;
        Vec bL;
        for (int j = 0; j < n; ++j)
        {
            for (int h = 0; h < m; ++h)
            {
                if (h == mach[j]) continue;
                Vec L2 = L;
                L2[mach[j]] = L2[mach[j]] - t[j];
                L2[h] = L2[h] + t[j];
                int C2 = 0;
                for (int k = 0; k < m; ++k) if (L2[k] > C2) C2 = L2[k];
                // tabu and no aspiration
                if (tabu[j][h] >= it && !(C2 < bestC)) continue;
                if (bj < 0 || LexLess(L2, bL)) { bj = j; bh = h; bL = L2; }
            }
        }
        if (bj < 0) break;
        // j may not go back for tenure iterations
        tabu[bj][mach[bj]] = it + tenure;
        mach[bj] = bh;
        L = bL;
        int C = 0;
        for (int k = 0; k < m; ++k) if (L[k] > C) C = L[k];
        if (C < bestC) { bestC = C; best = mach; }
```

```
    }
    mach = best;
    return bestC;
}
```

```
// tabu search for the TSP with the 2-opt neighbourhood
// the two edges removed by a move may not be added back for tenure iterations,
// unless the move gives a tour shorter than the best one (aspiration criterion)

int TabuSearchTSP(const Matrix& d, Vec& pi, int tenure, int maxIter)
{
    int n = pi.size();
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    Vec best = cur;
    int lbest = lcur;
    // tabu[a][b] >= it: edge (a,b) is tabu
    Matrix tabu(n, Vec(n, 0));
    for (int it = 1; it <= maxIter; ++it)
    {
        int bdelta = 0, bi = -1, bj = -1;
        for (int i = 0; i < n - 1; ++i)
        {
            for (int j = i + 2; j < n; ++j)
            {
                if (i == 0 && j == n - 1) continue;
                int a = cur[i], b = cur[i + 1], c = cur[j], e = cur[(j + 1) % n];
                int delta = d[a][c] + d[b][e] - d[a][b] - d[c][e];
                bool isTabu = tabu[a][c] >= it || tabu[b][e] >= it;
                if (isTabu && !(lcur + delta < lbest)) continue;
                if (bi < 0 || delta < bdelta) { bdelta = delta; bi = i; bj = j; }
            }
        }
        if (bi < 0) break;                                  // every move is tabu
        int a = cur[bi], b = cur[bi + 1], c = cur[bj], e = cur[(bj + 1) % n];
        tabu[a][b] = tabu[b][a] = it + tenure;              // removed edges become tabu
        tabu[c][e] = tabu[e][c] = it + tenure;
        // best non-tabu move, even if worse
        ApplyTwoOpt(cur, bi, bj);
        lcur = lcur + bdelta;
        if (lcur < lbest) { best = cur; lbest = lcur; }
    }
    pi = best;
    return lbest;
}
```

4.14.4 Iterated local search and variable neighbourhood search

- Iterated local search for the TSP with the double-bridge kick (Section 4.6):

```
// double-bridge kick: the tour A B C D, cut at the positions p1 < p2 < p3, becomes A C B
// D

Vec DoubleBridge(const Vec& pi, Random& r)
{
    int n = pi.size();
    int p1 = 1 + (int) (Rand(r) * (n - 3));
    int p2 = p1 + 1 + (int) (Rand(r) * (n - 2 - p1));
    int p3 = p2 + 1 + (int) (Rand(r) * (n - 1 - p2));
    Vec kick;
    for (int k = 0; k < p1; ++k) kick.push_back(pi[k]);     // A
    for (int k = p2; k < p3; ++k) kick.push_back(pi[k]);    // C
    for (int k = p1; k < p2; ++k) kick.push_back(pi[k]);    // B
    for (int k = p3; k < n; ++k) kick.push_back(pi[k]);     // D
    return kick;
}
```

```
// iterated local search for the TSP: double-bridge kick + 2-opt local search,
// the new local optimum is accepted if it is better than the current one

int IteratedLocalSearchTSP(const Matrix& d, Vec& pi, int maxIter, Random& r)
{
    Vec cur = pi;
    TwoOptLS(cur, d);                                       // first local optimum
    int lcur = TourLength(cur, d);
    for (int it = 1; it <= maxIter; ++it)
    {
        Vec nw = DoubleBridge(cur, r);                      // perturbation
        TwoOptLS(nw, d);                                    // local search
        int lnew = TourLength(nw, d);
        if (lnew < lcur) { cur = nw; lcur = lnew; }         // acceptance criterion
    }
    pi = cur;
    return lcur;
}
```

- Variable neighbourhood search for $P\,||\,C_{\max}$ (Section 4.7):

```
// shaking in the k-th neighbourhood: k random moves, each one moving a random job j
// to a random machine different from mach[j]

Vec Shake(const Vec& t, int m, const Vec& mach, int k, Random& r)
{
    int n = t.size();
    Vec sh = mach;
    for (int s = 0; s < k; ++s)
    {
        int j = (int) (Rand(r) * n);
        int g = (int) (Rand(r) * (m - 1));
```

```
        // the (g+1)-th machine other than sh[j]
        sh[j] = (g < sh[j]) ? g : g + 1;
    }
    return sh;
}
```

```
// basic variable neighbourhood search for P||Cmax: shaking with k random moves,
// local search by MoveDescent; k returns to 1 after an improvement, otherwise it grows
// up to kmax

int VNSPCmax(const Vec& t, int m, Vec& mach, int kmax, int maxIter, Random& r)
{
    int n = t.size(), sum = 0, LB = 0;
    for (int j = 0; j < n; ++j) { sum = sum + t[j]; if (t[j] > LB) LB = t[j]; }
    if ((sum + m - 1) / m > LB) LB = (sum + m - 1) / m;
    MoveDescent(t, m, mach);
    Vec L = Loads(mach, t, m);
    int C = 0;
    for (int h = 0; h < m; ++h) if (L[h] > C) C = L[h];
    int it = 0, k = 1;
    while (it < maxIter && C > LB)
    {
        it = it + 1;
        Vec nw = Shake(t, m, mach, k, r);              // shaking
        MoveDescent(t, m, nw);                         // local search
        Vec Lnew = Loads(nw, t, m);
        if (LexLess(Lnew, L))                          // move or not
        {
            mach = nw; L = Lnew; k = 1;
            C = 0;
            for (int h = 0; h < m; ++h) if (L[h] > C) C = L[h];
        }
        else
        {
            k = (k < kmax) ? k + 1 : 1;
        }
    }
    return C;
}
```

4.14.5 GRASP

- GRASP for the KP (Section 4.8) and for the TSP (Section 4.8.1):

```
// greedy randomized construction for the KP: at each step the candidates are the items
// that fit,
// RCL = { i candidate : p_i/w_i >= emax - alpha (emax - emin) }, sorted by non-increasing
// efficiency (ties: smallest index), and the item is chosen at random in the RCL

Vec RandomizedGreedyKP(const Vec& p, const Vec& w, int c, double alpha, Random& r)
{
    int n = p.size(), rc = c;                              // rc: residual capacity
    Vec x(n, 0);
    while (true)
    {
        double emax = -1, emin = -1;
        for (int i = 0; i < n; ++i)
        {
            if (x[i] == 1 || w[i] > rc) continue;
            double e = (double) p[i] / w[i];
            if (emax < 0 || e > emax) emax = e;
            if (emin < 0 || e < emin) emin = e;
        }
        if (emax < 0) return x;                            // no item fits
        double thr = emax - alpha * (emax - emin);
        Vec rcl;
        for (int i = 0; i < n; ++i)
        {
            if (x[i] == 1 || w[i] > rc || (double) p[i] / w[i] < thr - 1e-9) continue;
            int pos = rcl.size();                          // insertion in sorted position
            rcl.push_back(i);
            while (pos > 0 && p[rcl[pos - 1]] * w[i] < p[i] * w[rcl[pos - 1]])
            { rcl[pos] = rcl[pos - 1]; pos = pos - 1; }
            rcl[pos] = i;
        }
        int i = rcl[(int) (Rand(r) * rcl.size())];
        x[i] = 1;
        rc = rc - w[i];
    }
}
```

```
// GRASP for the KP: greedy randomized construction + local search, maxIter times

int GraspKP(const Vec& p, const Vec& w, int c, Vec& best, double alpha, int maxIter,
            Random& r)
{
    int zbest = -1;
    for (int it = 1; it <= maxIter; ++it)
```

```
    {
        Vec x = RandomizedGreedyKP(p, w, c, alpha, r);   // construction
        KPLocalSearch(p, w, c, x);                       // local search
        int z = Profit(p, x);
        if (z > zbest) { best = x; zbest = z; }
    }
    return zbest;
}
```

```
// randomized nearest neighbour from city 0: the next city is chosen at random in the
// restricted candidate list RCL = { j unvisited : d[last][j] <= dmin + alpha (dmax -
// dmin) },
// sorted by non-decreasing distance (ties: smallest index)

Vec RandomizedNN(const Matrix& d, double alpha, Random& r)
{
    int n = d.size();
    Vec pi(1, 0);
    vector<bool> visited(n, false);
    visited[0] = true;
    for (int k = 1; k < n; ++k)
    {
        int last = pi.back();
        int dmin = -1, dmax = -1;
        for (int j = 0; j < n; ++j)
        {
            if (visited[j]) continue;
            if (dmin < 0 || d[last][j] < dmin) dmin = d[last][j];
            if (dmax < 0 || d[last][j] > dmax) dmax = d[last][j];
        }
        double thr = dmin + alpha * (dmax - dmin);
        Vec rcl;
        for (int j = 0; j < n; ++j)
        {
            if (visited[j] || d[last][j] > thr + 1e-9) continue;
            int pos = rcl.size();                        // insertion in sorted position
            rcl.push_back(j);
            while (pos > 0 && d[last][rcl[pos - 1]] > d[last][j])
            { rcl[pos] = rcl[pos - 1]; pos = pos - 1; }
            rcl[pos] = j;
        }
        int next = rcl[(int) (Rand(r) * rcl.size())];
        pi.push_back(next);
        visited[next] = true;
    }
    return pi;
}
```

```
// GRASP for the TSP: randomized nearest neighbour + 2-opt local search, maxIter times

int GraspTSP(const Matrix& d, Vec& best, double alpha, int maxIter, Random& r)
{
    int lbest = -1;
    for (int it = 1; it <= maxIter; ++it)
```

```
    {
        Vec pi = RandomizedNN(d, alpha, r);              // greedy randomized
    construction
        TwoOptLS(pi, d);                                 // local search
        int len = TourLength(pi, d);
        if (lbest < 0 || len < lbest) { best = pi; lbest = len; }
    }
    return lbest;
}
```

4.14.6 Large neighbourhood search

- LNS and ALNS for the TSP (Sections 4.9 and 4.9.2):

```
// destroy: removes q random cities from the tour pi (never the first city) and
    returns
// them

Vec RandomRemoval(Vec& pi, int q, Random& r)
{
    Vec removed;
    for (int s = 0; s < q; ++s)
    {
        int k = 1 + (int) (Rand(r) * (pi.size() - 1));
        removed.push_back(pi[k]);
        pi.erase(pi.begin() + k);
    }
    return removed;
}
```

```
// destroy: removes, q times, the city whose removal saves the most length (
    never the
// first city)

Vec WorstRemoval(Vec& pi, int q, const Matrix& d)
{
    Vec removed;
    for (int s = 0; s < q; ++s)
    {
        int n = pi.size(), bk = -1, bs = 0;
        for (int k = 1; k < n; ++k)
        {
            int a = pi[k - 1], v = pi[k], b = pi[(k + 1) % n];
            int saving = d[a][v] + d[v][b] - d[a][b];
            if (bk < 0 || saving > bs) { bk = k; bs = saving; }
        }
        removed.push_back(pi[bk]);
        pi.erase(pi.begin() + bk);
    }
    return removed;
}
```

```
// repair: inserts the removed cities, in the order of removal, each in the
    position
// of minimum insertion cost d[a][v] + d[v][b] - d[a][b] (ties: first position)

void CheapestInsertionRepair(Vec& pi, const Vec& removed, const Matrix& d)
```

```
{
    for (int s = 0; s < (int) removed.size(); ++s)
    {
        int v = removed[s], n = pi.size(), bpos = -1, bcost = 0;
        for (int k = 0; k < n; ++k)
        {
            int a = pi[k], b = pi[(k + 1) % n];
            int cost = d[a][v] + d[v][b] - d[a][b];
            if (bpos < 0 || cost < bcost) { bpos = k; bcost = cost; }
        }
        pi.insert(pi.begin() + bpos + 1, v);
    }
}
```

```
// large neighbourhood search for the TSP: random removal of q cities + cheapest
// insertion,
// the new tour is accepted if it is shorter than the current one

int LnsTSP(const Matrix& d, Vec& pi, int q, int maxIter, Random& r)
{
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    for (int it = 1; it <= maxIter; ++it)
    {
        Vec nw = cur;
        Vec removed = RandomRemoval(nw, q, r);          // destroy
        CheapestInsertionRepair(nw, removed, d);        // repair
        int lnew = TourLength(nw, d);
        if (lnew < lcur) { cur = nw; lcur = lnew; }     // acceptance
    }
    pi = cur;
    return lcur;
}
```

```
// adaptive large neighbourhood search for the TSP with two destroy operators
// (o = 0: random removal, o = 1: worst removal) and cheapest insertion as repair;
// the operator is chosen with probability proportional to its weight wgt[o],
// updated as wgt[o] = (1 - rho) wgt[o] + rho psi, with psi = 3, 2, 1, 0 if the new tour
// is a new best, better than the current one, as good as the current one, worse

int AlnsTSP(const Matrix& d, Vec& pi, int q, int maxIter, double rho, Random& r,
            double* wgt)
{
    Vec cur = pi;
    int lcur = TourLength(cur, d);
    Vec best = cur;
    int lbest = lcur;
    wgt[0] = 1.0; wgt[1] = 1.0;
    for (int it = 1; it <= maxIter; ++it)
    {
        int o = (Rand(r) * (wgt[0] + wgt[1]) < wgt[0]) ? 0 : 1;     // roulette wheel
        Vec nw = cur;
```

```
        Vec removed = (o == 0) ? RandomRemoval(nw, q, r) : WorstRemoval(nw, q, d);
        CheapestInsertionRepair(nw, removed, d);
        int lnew = TourLength(nw, d);
        double psi;
        if (lnew < lbest) psi = 3.0;
        else if (lnew < lcur) psi = 2.0;
        else if (lnew == lcur) psi = 1.0;
        else psi = 0.0;
        if (lnew <= lcur) { cur = nw; lcur = lnew; }
        if (lcur < lbest) { best = cur; lbest = lcur; }
        wgt[o] = (1 - rho) * wgt[o] + rho * psi;                    // adaptive weights
    }
    pi = best;
    return lbest;
}
```

- LNS for the BPP (Section 4.9.1), with the First Fit starting packing, the destroy method and the two repair methods:

```
// first fit on the items in the given order; returns the number of bins nb, bin[i] = bin
// of item i

int FirstFit(const Vec& w, int c, const Vec& order, Vec& bin)
{
    Vec load;
    bin.assign(w.size(), -1);
    for (int a = 0; a < (int) order.size(); ++a)
    {
        int i = order[a], b = 0;
        while (b < (int) load.size() && load[b] + w[i] > c) b = b + 1;
        if (b == (int) load.size()) load.push_back(0);   // open a new bin
        bin[i] = b;
        load[b] = load[b] + w[i];
    }
    return load.size();
}
```

```
// destroy for the BPP: the least loaded bin (ties: smallest index) and q-1 other bins at
// random

Vec DestroyBins(const Vec& bin, int nb, const Vec& w, int q, Random& r)
{
    Vec load(nb, 0);
    for (int i = 0; i < (int) w.size(); ++i) load[bin[i]] = load[bin[i]] + w[i];
    int e = 0;
    for (int b = 1; b < nb; ++b) if (load[b] < load[e]) e = b;
    Vec empty(1, e);
    while ((int) empty.size() < q && (int) empty.size() < nb)
    {
        Vec others;
        for (int b = 0; b < nb; ++b)
```

```
        {
            bool chosen = false;
            for (int s = 0; s < (int) empty.size(); ++s) if (empty[s] == b)
    chosen = true;
            if (!chosen) others.push_back(b);
        }
        empty.push_back(others[(int) (Rand(r) * others.size())]);
    }
    return empty;
}
```

```
// heuristic repair for the BPP: first fit decreasing on the removed items, in
    new bins;
// asg[s] = bin (0, 1, ...) of items[s]; returns the number of new bins

int RepairFFD(const Vec& items, const Vec& w, int c, Vec& asg)
{
    int k = items.size();
    Vec order(k);
    for (int s = 0; s < k; ++s) order[s] = s;
    for (int a = 1; a < k; ++a)                          // non-increasing
    weight, stable
    {
        int s = order[a], b = a;
        while (b > 0 && w[items[order[b - 1]]] < w[items[s]])
        { order[b] = order[b - 1]; b = b - 1; }
        order[b] = s;
    }
    Vec load;
    asg.assign(k, -1);
    for (int a = 0; a < k; ++a)
    {
        int s = order[a], b = 0;
        while (b < (int) load.size() && load[b] + w[items[s]] > c) b = b + 1;
        if (b == (int) load.size()) load.push_back(0);
        asg[s] = b;
        load[b] = load[b] + w[items[s]];
    }
    return load.size();
}
```

```
// exact repair for the BPP: optimal packing of the removed items in new bins,
    by a
// depth-first branch-and-bound (items by non-increasing weight; each item goes
     in an
// open bin or in a new one)

static void Branch(int a, const Vec& order, const Vec& items, const Vec& w, int
     c,
                   Vec& load, Vec& cur, Vec& asg, int& best)
{
    if ((int) load.size() >= best) return;               // bound: cannot
    improve
    if (a == (int) order.size()) { best = load.size(); asg = cur; return; }
    int s = order[a];
    for (int b = 0; b < (int) load.size(); ++b)
    {
        if (load[b] + w[items[s]] > c) continue;
```

```
        load[b] = load[b] + w[items[s]]; cur[s] = b;
        Branch(a + 1, order, items, w, c, load, cur, asg, best);
        load[b] = load[b] - w[items[s]];
    }
    load.push_back(w[items[s]]); cur[s] = load.size() - 1;
    Branch(a + 1, order, items, w, c, load, cur, asg, best);
    load.pop_back();
}

int RepairExact(const Vec& items, const Vec& w, int c, Vec& asg)
{
    int k = items.size();
    Vec order(k);
    for (int s = 0; s < k; ++s) order[s] = s;
    for (int a = 1; a < k; ++a)
    {
        int s = order[a], b = a;
        while (b > 0 && w[items[order[b - 1]]] < w[items[s]])
        { order[b] = order[b - 1]; b = b - 1; }
        order[b] = s;
    }
    Vec load, cur(k, -1);
    int best = k + 1;
    Branch(0, order, items, w, c, load, cur, asg, best);
    return best;
}
```

```
// large neighbourhood search for the BPP: empty q bins (DestroyBins) and repack their
// items with FFD (exact = false) or optimally (exact = true); the new packing is
// accepted if it uses fewer bins, or the same number of bins and a sum of squared loads
// not smaller
// bin, nb: starting packing on input, final packing on output; returns the number of bins

int LnsBPP(const Vec& w, int c, Vec& bin, int nb, int q, bool exact, int maxIter,
           Random& r)
{
    int n = w.size(), sum = 0;
    for (int i = 0; i < n; ++i) sum = sum + w[i];
    int LB = (sum + c - 1) / c;
    int it = 0;
    while (it < maxIter && nb > LB)
    {
        it = it + 1;
        Vec empty = DestroyBins(bin, nb, w, q, r);      // destroy
        vector<bool> isEmptied(nb, false);
        for (int s = 0; s < (int) empty.size(); ++s) isEmptied[empty[s]] = true;
        Vec items, newbin(n, -1), newIndex(nb, -1);
        int kept = 0;
        for (int b = 0; b < nb; ++b)
            if (!isEmptied[b]) { newIndex[b] = kept; kept = kept + 1; }
        for (int i = 0; i < n; ++i)
        {
```

```
            if (isEmptied[bin[i]]) items.push_back(i);
            else newbin[i] = newIndex[bin[i]];
        }
        Vec asg;                                              // repair
        int k = exact ? RepairExact(items, w, c, asg) : RepairFFD(items, w, c,
    asg);
        for (int s = 0; s < (int) items.size(); ++s) newbin[items[s]] = kept +
    asg[s];
        int nb2 = kept + k;
        Vec load(nb, 0), load2(nb2, 0);
        for (int i = 0; i < n; ++i) { load[bin[i]] += w[i]; load2[newbin[i]] +=
     w[i]; }
        // auxiliary objective: sum of squared loads
        long long aux = 0, aux2 = 0;
        for (int b = 0; b < nb; ++b) aux = aux + load[b] * load[b];
        for (int b = 0; b < nb2; ++b) aux2 = aux2 + load2[b] * load2[b];
        // acceptance
        if (nb2 < nb || (nb2 == nb && aux2 >= aux)) { bin = newbin; nb = nb2; }
    }
    return nb;
}
```

4.14.7 Genetic and memetic algorithm

- The genetic and memetic algorithm for the KP (Section 4.10):

```
// repair of an infeasible KP solution: removes the selected item of minimum
    efficiency
// (ties: largest index) until the capacity is respected

void RepairKP(const Vec& p, const Vec& w, int c, Vec& x)
{
    int n = p.size(), W = Weight(w, x);
    while (W > c)
    {
        int s = -1;
        for (int i = 0; i < n; ++i)
            if (x[i] == 1 && (s < 0 || p[i] * w[s] <= p[s] * w[i])) s = i;
        x[s] = 0;
        W = W - w[s];
    }
}
```

```
// binary tournament: two random individuals, the fitter one wins (ties: the
    first drawn)

int Tournament(const Vec& fit, Random& r)
{
    int P = fit.size();
    int a = (int) (Rand(r) * P);
    int b = (int) (Rand(r) * P);
    return (fit[a] >= fit[b]) ? a : b;
}
```

```
// steady-state genetic algorithm for the KP with binary encoding: binary
    tournament
// selection,
```

```cpp
// one-point crossover, bit-flip mutation with probability pm, repair; if memetic is true,
// every new individual is also improved by KPLocalSearch; the child replaces the worst
// individual (ties: last index) if it is better and not already in the population

int GeneticKP(const Vec& p, const Vec& w, int c, Vec& best, int P, double pm, int maxIter,
              bool memetic, Random& r)
{
    int n = p.size();
    Matrix pop(P, Vec(n, 0));
    Vec fit(P, 0);
    for (int s = 0; s < P; ++s)                          // random initial population
    {
        for (int i = 0; i < n; ++i) pop[s][i] = (Rand(r) < 0.5) ? 1 : 0;
        RepairKP(p, w, c, pop[s]);
        if (memetic) KPLocalSearch(p, w, c, pop[s]);
        fit[s] = Profit(p, pop[s]);
    }
    for (int it = 1; it <= maxIter; ++it)
    {
        int a = Tournament(fit, r);                      // selection
        int b = Tournament(fit, r);
        int cut = 1 + (int) (Rand(r) * (n - 1));         // one-point crossover
        Vec child(n);
        for (int i = 0; i < n; ++i) child[i] = (i < cut) ? pop[a][i] : pop[b][i];
        for (int i = 0; i < n; ++i)                      // mutation
            if (Rand(r) < pm) child[i] = 1 - child[i];
        RepairKP(p, w, c, child);                        // repair
        if (memetic) KPLocalSearch(p, w, c, child);      // memetic: local search
        int fc = Profit(p, child);
        int worst = 0;
        for (int s = 1; s < P; ++s) if (fit[s] <= fit[worst]) worst = s;
        bool present = false;
        for (int s = 0; s < P; ++s) if (pop[s] == child) present = true;
        // replacement
        if (!present && fc > fit[worst]) { pop[worst] = child; fit[worst] = fc; }
    }
    int sb = 0;
    for (int s = 1; s < P; ++s) if (fit[s] > fit[sb]) sb = s;
    best = pop[sb];
    return fit[sb];
}
```

References

- V. Černý, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications* 45 (1985) 41–51.
- T.A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, *Operations Research Letters* 8 (1989) 67–71.
- T.A. Feo, M.G.C. Resende, Greedy randomized adaptive search procedures, *Journal of Global Optimization* 6 (1995) 109–133.
- F. Glover, Future paths for integer programming and links to artificial intelligence, *Computers & Operations Research* 13 (1986) 533–549.
- F. Glover, M. Laguna, *Tabu Search*, Kluwer, 1997.
- D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
- B. Hajek, Cooling schedules for optimal annealing, *Mathematics of Operations Research* 13 (1988) 311–329.
- J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, *Science* 220 (1983) 671–680.
- M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration, *Operations Research Perspectives* 3 (2016) 43–58.
- H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search, in F. Glover, G.A. Kochenberger (eds.), *Handbook of Metaheuristics*, Kluwer, 2003, 321–353.
- O. Martin, S.W. Otto, E.W. Felten, Large-step Markov chains for the traveling salesman problem, *Complex Systems* 5 (1991) 299–326.
- N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, *The Journal of Chemical Physics* 21 (1953) 1087–1092.
- N. Mladenović, P. Hansen, Variable neighborhood search, *Computers & Operations Research* 24 (1997) 1097–1100.
- P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms, Technical Report C3P 826, Caltech, 1989.
- S.K. Park, K.W. Miller, Random number generators: good ones are hard to find, *Communications of the ACM* 31 (1988) 1192–1201.
- S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, *Transportation Science* 40 (2006) 455–472.

- P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in *Principles and Practice of Constraint Programming – CP98*, Lecture Notes in Computer Science 1520, Springer, 1998, 417–431.
- É. Taillard, Robust taboo search for the quadratic assignment problem, *Parallel Computing* 17 (1991) 443–455.

Chapter 5

MIP heuristics for the classical problems

Prerequisites

- From the module *Integer Linear Programming*, Chapter 10 «MIP heuristics»: the methods of this chapter (rounding, diving, the feasibility pump, RINS, local branching, proximity search, relax-and-fix and fix-and-optimize), their pseudo-codes and the conditions under which they work.
- From the module *Combinatorial Optimization Problems*, the formulations of the classical problems: the models of the 0/1 knapsack problem, of the bin packing problem, of the travelling salesman problem, of $P\,||\,C_{\max}$ and of lot sizing.
- From this module, the chapters on heuristic algorithms, from «Heuristic algorithms: definitions» to «Metaheuristics» («Constructive heuristics», «Local search»): the constructive heuristics (GR, FF, NN, LPT), local search and metaheuristics, whose solutions are the starting points of this chapter.

Objectives

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

- write the MIP models of the classical problems in the form required by the MIP-based heuristics, and compute their LP relaxations;
- execute by hand rounding, diving, the feasibility pump, RINS, local branching and proximity search on the 0/1 knapsack problem, on bin packing and on $P\,||\,C_{\max}$;
- execute relax-and-fix and fix-and-optimize on a lot sizing instance;
- explain why each method succeeds or fails on each problem;
- compute the gap of a heuristic solution when a dual bound is available.

5.1 MIP-based heuristics

Definition 5.1: MIP-based heuristic

A heuristic algorithm for a problem with a MIP model is **MIP-based** (a **matheuristic**) if it uses the LP relaxation of the model and a MIP solver to search for a feasible solution.

- The MIP-based heuristics are defined in the module *Integer Linear Programming*, Chapter 10 «MIP heuristics». In this chapter they are applied to the classical problems, with the reference instances of the previous chapters.

- The notation is that of that chapter: a MIP with integer variables $\mathscr{D}$, $\tilde{\boldsymbol{x}}$ is an optimal solution of the LP relaxation, $\bar{\boldsymbol{x}}$ a feasible solution (the incumbent), $[\,v\,] = \lfloor v + \frac{1}{2} \rfloor$ the rounding to the nearest integer, $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = \sum_{j \in \mathscr{D}} |x_j - \bar{x}_j|$ the distance from an integer point. A maximization problem, such as the KP, is handled by exchanging min and max, $\leq$ and $\geq$ in the comparisons of objective values.

- The examples are solved with the MIP solver Gurobi, used from Python through `gurobipy`. When an LP has several optimal solutions, the text reports the one returned by the solver; the codes are in the last section.

Constructive heuristics, local search and metaheuristics work on a representation of the solutions; a MIP-based heuristic works on the model of the problem, through its LP relaxation and a MIP solver.

5.2 The problems and their MIP models

- The heuristics are illustrated on the four problems of the previous chapters: the 0/1 knapsack problem (KP), the bin packing problem (BPP), the travelling salesman problem (TSP) and $P\,||\,C_{\max}$. Their definitions and assumptions are on their pages.

- The reference instances are recalled in the following table, with the values computed in the previous chapters on heuristic algorithms.

problem	reference instance	values
KP	$p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$	GR: $\{1, 3\}$, profit 16; $z(I) = 17$, $\{1, 2\}$
BPP	$w = (4, 4, 5, 3, 2, 3)$, $c = 7$	FF: 4 bins; $z(I) = 3$
TSP	5 cities, the reference distances	NN from city 1: 25; $z(I) = 18$
$P\,||\,C_{\max}$	$t = (5, 5, 4, 4, 3, 3, 3)$, $m = 3$	LPT: $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, 11; $z(I) = 9$

- The MIP models are those of the module *Combinatorial Optimization Problems*: the ILP_{KP} model of the chapter «Formulations of knapsack problems», the ILP_{BPP} model of the chapter «Formulations of packing problems», the $\text{ILP}^{\text{DFJ}}_{\text{TSP}}$ model of the chapter «Formulations of routing problems» and the $\text{ILP}_{P||C_{\max}}$ model of the chapter «Formulations of scheduling problems». We recall the three models used in the examples, with the variables and the notation of those chapters.

5.2.1 The 0/1 knapsack problem

- The binary variable x_j is equal to 1 if item j is selected in the knapsack, and to 0 otherwise, for every $j \in \mathscr{I}$.

Using these variables, an ILP model for the 0/1 knapsack problem (KP) reads as follows:

$$\max \quad \sum_{j \in \mathscr{I}} p_j \, x_j \tag{5.1a}$$

$$\text{subject to} \quad \sum_{j \in \mathscr{I}} w_j \, x_j \leq c, \tag{5.1b}$$

$$x_j \in \{0,1\}, \qquad \forall j \in \mathscr{I}. \tag{5.1c}$$

This ILP model is referred to as the ILP_{KP} model.

5.2.2 The bin packing problem

- Given an upper bound m on the number of bins used in an optimal solution, and the set $\mathscr{B} = \{1, 2, \ldots, m\}$ of the available bins, the binary variable x_{jb} is equal to 1 if item j is packed in bin b, and the binary variable y_b is equal to 1 if bin b is used, for every $j \in \mathscr{I}$ and $b \in \mathscr{B}$.

Using these variables, an ILP model for the bin packing problem (BPP) reads as follows:

$$\min \quad \sum_{b \in \mathscr{B}} y_b \tag{5.2a}$$

$$\text{subject to} \quad \sum_{b \in \mathscr{B}} x_{jb} = 1, \qquad \forall j \in \mathscr{I}, \tag{5.2b}$$

$$-\sum_{j \in \mathscr{I}} w_j \, x_{jb} + c \, y_b \geq 0, \qquad \forall b \in \mathscr{B}, \tag{5.2c}$$

$$x_{jb} \in \{0,1\}, \qquad \forall j \in \mathscr{I}, \ \forall b \in \mathscr{B}, \tag{5.2d}$$

$$y_b \in \{0,1\}, \qquad \forall b \in \mathscr{B}. \tag{5.2e}$$

This ILP model is referred to as the ILP_{BPP} model.

- The content of bin b is the packing pattern $P_b = \{j \in \mathscr{I} : x_{jb} = 1\}$. The number n of items is always a valid upper bound m; on the reference instance we use $m = n = 6$.

5.2.3 The travelling salesman problem

- The $\text{ILP}_{\text{TSP}}^{\text{DFJ}}$ model has a binary variable x_{ij} for every arc $(i,j) \in A$, the constraints on the arcs leaving and entering every city, and the subtour elimination constraints on the subsets $S \subseteq \mathscr{V} \setminus \{1\}$ with $|S| \geq 2$. On the reference instance, with $n = 5$ cities, it has 20 variables and $5 + 5 + 11 = 21$ constraints.

5.2.4 The scheduling problem $P \,||\, C_{\max}$

- Given the set $\mathscr{H} = \{1, 2, \ldots, m\}$ of the machines, the binary variable x_{jh} is equal to 1 if job j is assigned to machine h, for every $j \in \mathscr{J}$ and $h \in \mathscr{H}$, and the non-negative continuous variable $C_{\max}$ is the makespan.

Using these variables, an ILP model for $P||C_{\max}$ reads as follows:

$$\min \quad C_{\max} \tag{5.3a}$$

$$\text{subject to} \quad \sum_{h \in \mathscr{H}} x_{jh} = 1, \qquad \forall j \in \mathscr{J}, \tag{5.3b}$$

$$-\sum_{j \in \mathscr{J}} t_j \, x_{jh} + C_{\max} \geq 0, \qquad \forall h \in \mathscr{H}, \tag{5.3c}$$

$$x_{jh} \in \{0,1\}, \qquad \forall j \in \mathscr{J}, \ \forall h \in \mathscr{H}, \tag{5.3d}$$

$$C_{\max} \geq 0. \tag{5.3e}$$

This ILP model is referred to as the $\text{ILP}_{P||C_{\max}}$ model.

- $C_{\max}$ is the only continuous variable of the four models: given the assignment, the completion sets it to the largest load $L_h = \sum_{j \in \mathscr{J}} t_j \, x_{jh}$.

5.2.5 The LP relaxations on the reference instances

- The following table reports the size of the four models on the reference instances, the value $z(\text{LP})$ of their LP relaxations, computed with Gurobi, and the optimal value $z(I)$.

problem	variables	constraints	$z(\text{LP})$	$z(I)$	LP solution integer?
KP	4	1	17.75	17	no
BPP	42	12	3	3	no
TSP	20	21	18	18	yes
$P \,||\, C_{\max}$	22	10	9	9	no

- On all four instances the LP value, rounded in the direction of the bound (down for the KP, up for the three minimization problems), is already the optimal value. The relaxations are *strong* on these instances: the difficulty is to find a solution that attains the bound, not to prove it.

- This is the typical situation in which MIP-based heuristics are useful.

- On the TSP reference instance the LP solution of the $\text{ILP}_{\text{TSP}}^{\text{DFJ}}$ model is itself the optimal tour $(1,3,5,2,4)$, with $\tilde{x}_{13} = \tilde{x}_{35} = \tilde{x}_{52} = \tilde{x}_{24} = \tilde{x}_{41} = 1$: every method of this chapter finds it at once. The LP relaxation of the $\text{ILP}_{\text{TSP}}^{\text{MTZ}}$ model is weaker: its value is 17. The TSP is therefore not used in the other examples of this chapter.

Example 5.1: The LP solutions of the reference instances

KP. The LP solution is $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$: the items are taken by non-increasing efficiency, and the split item 2 is taken for the fraction $\frac{9-8}{4} = \frac{1}{4}$ of its weight, so $z(\text{LP}) = 10 + \frac{7}{4} + 6 = 17.75$.

$P\,||\,C_{\max}$**.** The LP solution returned by the solver is (zeros omitted)

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
$\tilde{x}_{j1}$		$\frac{3}{5}$				1	1
$\tilde{x}_{j2}$			$\frac{1}{2}$	1	1		
$\tilde{x}_{j3}$	1	$\frac{2}{5}$	$\frac{1}{2}$				

with loads $L_1 = 3 + 5 \cdot \frac{3}{5} + 3 = 9$, $L_2 = 4 \cdot \frac{1}{2} + 4 + 3 = 9$, $L_3 = 5 + 5 \cdot \frac{2}{5} + 4 \cdot \frac{1}{2} = 9$ and $\tilde{C}_{\max} = 9 = \frac{27}{3}$. The LP has infinitely many optimal solutions: every fractional assignment with three loads equal to 9 is optimal.

BPP. The LP solution returned by the solver puts item 1 into bin 6, item 2 into bin 4, item 3 for $\frac{4}{5}$ into bin 3 and for $\frac{1}{5}$ into bin 4, item 4 into bin 6, item 5 into bin 1, item 6 into bin 3, and opens the bins by the fractions

$$\tilde{\boldsymbol{y}} = \left(\frac{2}{7}, 0, 1, \frac{5}{7}, 0, 1\right),$$

just enough to contain their loads: 2, 0, 7, 5, 0, 7. Its value is $\frac{21}{7} = 3$.

5.3 Rounding

- *Rounding* rounds the LP solution to the nearest integer point and completes the continuous variables; it fails if the rounded point violates a constraint. Rounding down is always feasible when all the coefficients of A are non-negative and $b \geq 0$ (the observation on rounding down of Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*); an equality constraint, such as the assignment constraints of the BPP and of $P\,||\,C_{\max}$, is almost always violated by independent roundings.

Example 5.2: Rounding on the reference instances

KP. The LP solution $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$ is rounded to $\bar{\boldsymbol{x}} = (1,0,1,0)$: weight $8 \leq 9$, profit 16. It is the solution of GR of the chapter «Constructive heuristics». Rounding $\tilde{x}_2 = \frac{1}{4}$ up instead would give $(1,1,1,0)$, of weight $12 > 9$: infeasible.

$P\,||\,C_{\max}$. In the LP solution of Example 5.1, $\left[\frac{3}{5}\right] = 1$ and $\left[\frac{2}{5}\right] = 0$ send job 2 to machine 1, but $\left[\frac{1}{2}\right] = 1$ assigns job 3 to *both* machines 2 and 3:

$$\bar{x}_{31} + \bar{x}_{32} + \bar{x}_{33} = 0 + 1 + 1 = 2 \neq 1.$$

The assignment constraint (5.3b) of job 3 is violated: **rounding fails**.

BPP. Rounding the LP solution of Example 5.1 gives $\bar{\boldsymbol{y}} = (0, 0, 1, 1, 0, 1)$ and puts item 3 into bin 3. Two capacity constraints (5.2c) are violated: bin 1 contains item 5 but $\bar{y}_1 = 0$ ($-2 + 7 \cdot 0 \geq 0$ is false), and bin 3 contains items 3 and 6 ($-(5 + 3) + 7 \cdot 1 \geq 0$ is false). **Rounding fails**.

- The three outcomes summarize the method: on the KP, with a single packing constraint, rounding is safe (the observation on rounding down of Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*); on the problems with assignment constraints it fails, because the LP spreads a job or an item over several machines or bins, and rounding cannot decide where it goes.

5.4 Diving

- *Diving* fixes one fractional variable at a time and re-solves the LP: fractional diving fixes the least fractional variable to its nearest integer, guided diving fixes the variable closest to the incumbent to the incumbent value; it fails when the LP becomes infeasible. Ties are broken in favour of the variable that comes first in the order of the variables ($x_{11}, x_{12}, \ldots$ for the assignment models, then $C_{\max}$ or $y_1, y_2, \ldots$).

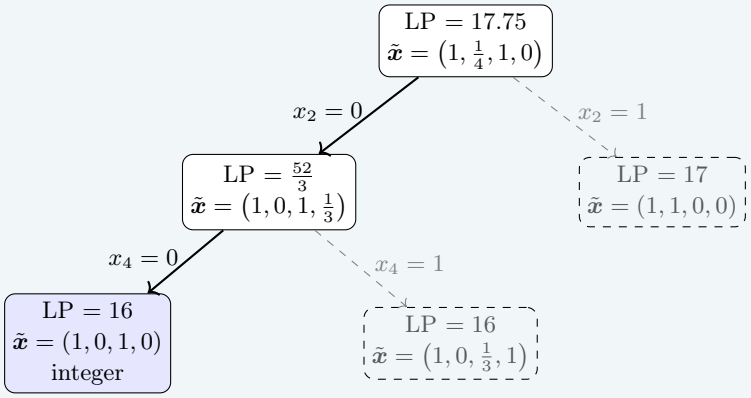
Example 5.3: Fractional diving on the KP

Step 1. The LP solution is $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$, of value 17.75. The only fractional variable is x_2, rounded to $\left[\frac{1}{4}\right] = 0$: fix $x_2 := 0$.

Step 2. Without item 2, the LP takes items 1 and 3 (weight 8) and the fraction $\frac{1}{3}$ of item 4: $\tilde{\boldsymbol{x}} = \left(1, 0, 1, \frac{1}{3}\right)$, of value $10 + 6 + \frac{4}{3} = \frac{52}{3} \approx 17.33$. Fix $x_4 := \left[\frac{1}{3}\right] = 0$.

Step 3. The LP solution $\tilde{\boldsymbol{x}} = (1, 0, 1, 0)$, of value 16, is integer: the dive returns $\{1, 3\}$, of profit 16.

The dive is the path drawn with solid lines in the branch-and-bound tree below; the dashed nodes are the branches it never visits, with the value of their LP relaxations.



The branch $x_2 = 1$, abandoned at the first step, contains the optimal solution $\{1, 2\}$ of profit 17, which is an optimal solution of its LP relaxation. The first decision of the dive, $\left[\frac{1}{4}\right] = 0$, was the wrong one.

Example 5.4: Fractional diving on $P\,||\,C_{\max}$

The dive starts from the LP solution of Example 5.1. The table reports, for each step, the value of the LP, its fractional variables and the fixing; a fractionality $|\tilde{x}_j - [\tilde{x}_j]|$ equal for two variables is broken by the order $x_{11}, x_{12}, x_{13}, x_{21}, \ldots$

step	LP value	fractional variables	fixing
1	9	$x_{21} = \frac{3}{5}$, $x_{23} = \frac{2}{5}$, $x_{32} = \frac{1}{2}$, $x_{33} = \frac{1}{2}$	$x_{21} := 1$
2	9	$x_{11} = \frac{1}{5}$, $x_{13} = \frac{4}{5}$, $x_{32} = \frac{3}{4}$, $x_{33} = \frac{1}{4}$	$x_{11} := 0$
3	9	$x_{32} = \frac{3}{4}$, $x_{33} = \frac{1}{4}$, $x_{41} = \frac{1}{4}$, $x_{43} = \frac{3}{4}$	$x_{32} := 1$
4	9	$x_{41} = \frac{1}{4}$, $x_{43} = \frac{3}{4}$, $x_{52} = \frac{2}{3}$, $x_{53} = \frac{1}{3}$	$x_{41} := 0$
5	9	$x_{61} = \frac{1}{3}$, $x_{62} = \frac{2}{3}$	$x_{61} := 0$
6	9	$x_{12} = \frac{3}{5}$, $x_{13} = \frac{2}{5}$, $x_{71} = \frac{1}{3}$, $x_{72} = \frac{2}{3}$	$x_{71} := 0$
7	9.5	$x_{42} = \frac{1}{8}$, $x_{43} = \frac{7}{8}$	$x_{42} := 0$
8	9.5	$x_{62} = \frac{1}{6}$, $x_{63} = \frac{5}{6}$	$x_{62} := 0$
9	9.5	$x_{12} = \frac{1}{2}$, $x_{13} = \frac{1}{2}$	$x_{12} := 1$
10	9.5	$x_{72} = \frac{1}{6}$, $x_{73} = \frac{5}{6}$	$x_{72} := 0$
11	10	none	—

The last LP solution is integer: $J_1 = \{2, 5\}$, $J_2 = \{1, 3\}$, $J_3 = \{4, 6, 7\}$, with loads 8, 9, 10 and makespan 10. The dive improves on rounding, which failed, and on *LPT* (11), but it does not reach the optimum 9.

- For six steps the LP value stays at 9: the LP of $P\,||\,C_{\max}$ has a huge set of optimal solutions, and fixing one variable just moves the LP to another of them. The LP value tells nothing about the quality of the decisions until the load of a machine is forced above 9, at step 7, and then it is too late to undo the decisions that caused it. Min-max objectives are as flat for the LP as they are for local search.

Example 5.5: Guided diving on $P\,||\,C_{\max}$

The incumbent is the solution of LPT, $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, of makespan 11, i.e., $\bar{x}_{11} = \bar{x}_{51} = \bar{x}_{71} = \bar{x}_{22} = \bar{x}_{62} = \bar{x}_{33} = \bar{x}_{43} = 1$ and all the other $\bar{x}_{jh} = 0$.

Step 1. In the LP solution of Example 5.1 the fractional variables and their distances from the incumbent are

$$|\tilde{x}_{21} - \bar{x}_{21}| = \tfrac{3}{5}, \quad |\tilde{x}_{23} - \bar{x}_{23}| = \tfrac{2}{5}, \quad |\tilde{x}_{32} - \bar{x}_{32}| = \tfrac{1}{2}, \quad |\tilde{x}_{33} - \bar{x}_{33}| = \tfrac{1}{2}.$$

The closest is x_{23}: fix $x_{23} := \bar{x}_{23} = 0$.

Step 2. The new LP solution, of value 9, has the fractional variables $x_{12} = \frac{2}{5}$, $x_{13} = \frac{3}{5}$, $x_{31} = \frac{1}{2}$, $x_{32} = \frac{1}{2}$, at distances $\frac{2}{5}$, $\frac{3}{5}$, $\frac{1}{2}$, $\frac{1}{2}$ from the incumbent. Fix $x_{12} := \bar{x}_{12} = 0$.

Step 3. The LP solution is integer, of value 9: $J_1 = \{5, 6, 7\}$, $J_2 = \{2, 4\}$, $J_3 = \{1, 3\}$, all loads equal to 9. Guided diving finds an **optimal** solution after two LPs.

- Guided by a solution of makespan 11, the dive ends at a solution of makespan 9: the incumbent only decides the *values* of the two fixed variables, both equal to 0, while the LP rebuilds the rest of the assignment.

- On the BPP reference instance, fractional diving takes 8 fixings, all with LP value 3; the ninth, $x_{12} := 1$, puts item 1 into bin 2, which already contains item 2 by an earlier fixing, and $4 + 4 > 7$ makes the LP infeasible: **the dive fails**. The fixings $x_{jb} = 1$ commit items to bins one at a time, and the LP of the BPP is so weak (3 at every step) that it gives no warning before the dead end.

5.5 The feasibility pump

- The *feasibility pump* alternates the projection of an integer point onto the LP feasible region (one LP minimizing Δ) and the rounding of the projection, with a flip when the rounding returns the same integer point and a random restart when the pump cycles, until the distance is 0. The random numbers are those of the *minimal standard* generator of Park and Miller with the seeds $1, 2, \ldots$, as in the chapter «Metaheuristics».

Example 5.6: The feasibility pump on $P\,||\,C_{\max}$

Iteration 0. The LP solution of Example 5.1 is rounded as in Example 5.2: job 3 is assigned to machines 2 and 3, and $\bar{\boldsymbol{x}}^0$ is infeasible.

Iteration 1. The projection minimizes $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^0)$, which is linear since all the integer variables are binary. The solver returns the point of $\mathcal{L}$ that keeps all the assignments of $\bar{\boldsymbol{x}}^0$ and removes job 3 from machine 2, i.e., it moves x_{32} from 1 to 0: $\Delta = 1$ (removing job 3 from machine 3 would be equally close). The result is integer:

$$J_1 = \{2, 6, 7\}, \qquad J_2 = \{4, 5\}, \qquad J_3 = \{1, 3\},$$

with loads 11, 7, 9, and its rounding is the new integer point $\bar{\boldsymbol{x}}^1$.

Iteration 2. $\bar{\boldsymbol{x}}^1 \in \mathcal{L}$, the distance is 0, and the pump returns a solution of makespan 11, after two projections.

- The solution has the same makespan as LPT: the projection takes care of the assignment constraints and ignores the loads, since $C_{\max}$ is continuous and does not enter Δ. The pump answers the question "is there a feasible solution?", which for $P\,||\,C_{\max}$ was not a difficult question.

Example 5.7: The feasibility pump on the BPP

On the BPP reference instance, where rounding and diving fail, the pump is run with the seeds $1, 2, \ldots, 10$ of the random numbers. The table reports the number of projections, of flips and of restarts, the number of bins opened by the solution found ($\sum_b \bar{y}_b$, the value of the solution) and the number of non-empty bins.

seed	1	2	3	4	5	6	7	8	9	10
projections	15	12	9	9	6	6	19	5	10	20
flips	8	5	3	4	3	3	8	2	5	8
restarts	4	3	3	2	1	1	5	1	2	4
bins opened	4	4	4	5	5	4	3	4	4	5
non-empty bins	4	4	4	3	5	4	3	4	4	4

The pump finds a feasible solution with every seed, but always after some restarts: without them it would cycle. Only with seed 7 the solution is optimal: $\{3, 5\}$, $\{2, 4\}$, $\{1, 6\}$, of value 3. With seed 4 the pump opens 5 bins, two of which stay empty: the packing $\{3, 5\}$, $\{1, 6\}$, $\{2, 4\}$ is optimal, but the solution has value 5, because nothing in the distance function asks to close the empty bins.

- The pump is a method to find a *first* feasible solution, and it does that well. The quality of the solution is the job of the improvement methods, which start from an incumbent.

5.6 RINS, local branching and proximity search

- The three methods start from an incumbent $\bar{\boldsymbol{x}}$ and solve a sub-MIP with the cutoff constraint $f(\boldsymbol{x}) \leq f(\bar{\boldsymbol{x}}) - \theta$, which here has $\theta = 1$ since the objective values are integer: RINS fixes the integer variables on which the LP solution and the incumbent agree; local branching adds $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq k$ and enlarges k when it finds nothing; proximity search minimizes $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}})$.

5.6.1 RINS

Example 5.8: RINS on the KP

The incumbent is the solution $\bar{\boldsymbol{x}} = (1, 0, 1, 0)$ of GR, of profit 16, and the LP solution is $\tilde{\boldsymbol{x}} = \left(1, \frac{1}{4}, 1, 0\right)$. They agree on $x_1 = 1$, $x_3 = 1$, $x_4 = 0$, which are fixed. The sub-MIP, with the capacity constraint (5.1b) and the cutoff constraint, is

$$\begin{array}{rrrcr} \max & 16 & +\,7\,x_2 & & \\ \text{subject to} & 8 & +\,4\,x_2 & \leq & 9 \\ & 16 & +\,7\,x_2 & \geq & 17 \\ & & x_2 & \in & \{0,1\} \end{array}$$

and it is infeasible: the capacity requires $x_2 = 0$, the cutoff $x_2 = 1$. RINS returns the incumbent. The optimal solution $\{1, 2\}$ has $x_3 = 0$, but x_3 is fixed to 1 because the LP and the incumbent agree on it.

Example 5.9: RINS on $P\,||\,C_{\max}$

The incumbent is the solution of LPT ($J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, makespan 11), and $\tilde{\boldsymbol{x}}$ is the LP solution of Example 5.1. The table compares, for each job, the machine of the incumbent with the LP solution.

	$j=1$	$j=2$	$j=3$	$j=4$	$j=5$	$j=6$	$j=7$
machine in $\bar{\boldsymbol{x}}$	1	2	3	3	1	2	1
machines in $\tilde{\boldsymbol{x}}$	3	1 ($\frac{3}{5}$), 3 ($\frac{2}{5}$)	2 ($\frac{1}{2}$), 3 ($\frac{1}{2}$)	2	2	1	1
fixed variables	$x_{12} = 0$	—	$x_{31} = 0$	$x_{41} = 0$	$x_{53} = 0$	$x_{63} = 0$	$x_{71} = 1$, $x_{72} = x_{73} = 0$

Only job 7 is on the same machine in both solutions; for the other jobs, the two solutions agree only on some machines that the job does *not* use. Eight of the 21 binary variables are fixed. The sub-MIP with the cutoff $C_{\max} \leq 10$ is solved by Gurobi, which returns

$$J_1 = \{5, 6, 7\}, \qquad J_2 = \{2, 3\}, \qquad J_3 = \{1, 4\},$$

with all loads equal to 9: **RINS finds an optimal solution**.

- On the KP the incumbent and the LP solution are too similar, and the neighbourhood of RINS has a single free variable; on $P\,||\,C_{\max}$ they are very different, and the neighbourhood is large enough to contain the optimum.

5.6.2 Local branching

- **Local branching and local search.** With $k = 1$ the neighbourhood is the flip neighbourhood of the KP in the chapter «Heuristic algorithms: definitions»; with $k = 2$ it contains the swap neighbourhood as well. For the assignment variables of the BPP and of $P\,||\,C_{\max}$ the distance has a simple meaning.

Observation 5.2

Let $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$ be two feasible assignments of $P\,||\,C_{\max}$. Then $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) = 2\,r$, where r is the number of jobs assigned to different machines in $\boldsymbol{x}$ and in $\bar{\boldsymbol{x}}$.

Proof. A job on the same machine in $\boldsymbol{x}$ and $\bar{\boldsymbol{x}}$ contributes 0 to Δ. A job on machine h in $\bar{\boldsymbol{x}}$ and on machine $h' \neq h$ in $\boldsymbol{x}$ contributes $1 - x_{jh} = 1$ and $x_{jh'} = 1$, and 0 for the other machines. □

- Hence, for $P\,||\,C_{\max}$, the neighbourhood $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq 2$ is the **move** neighbourhood of the chapter «Local search» (one job changes machine), and $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}) \leq 4$ contains the **swap** neighbourhood (two jobs change machine). The same holds for the variables x_{jb} of the BPP.

Example 5.10: Local branching on the KP

Start from $\bar{\boldsymbol{x}}^1 = (1, 0, 1, 0)$, of profit 16, with $k = 1$, $k_{\max} = 3$, $k_{step} = 1$ and $\theta = 1$.

$k = 1$. The neighbours at distance 1 are the flips of $\bar{\boldsymbol{x}}^1$; with the cutoff $\sum_j p_j x_j \geq 17$ no one is feasible. Add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 2$ and set $k = 2$.

$k = 2$. The sub-MIP with $2 \leq \Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \leq 2$ finds the swap $(1, 1, 0, 0)$, of profit 17: the new incumbent $\bar{\boldsymbol{x}}^2$. Add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 3$.

$k = 2$, around $\bar{\boldsymbol{x}}^2$. No solution of profit at least 18: add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^2) \geq 3$ and set $k = 3$.
$k = 3$. No solution of profit at least 18 either; k becomes $4 > k_{\max}$ and the method stops with $\{1, 2\}$, of profit 17: the optimum.

Example 5.11: Local branching on $P\,||\,C_{\max}$

Start from the solution $\bar{\boldsymbol{x}}^1$ of LPT, of makespan 11, with $k = 2$, $k_{\max} = 6$, $k_{step} = 2$ and $\theta = 1$, so that the cutoff is $C_{\max} \leq 10$, then 9, and so on.

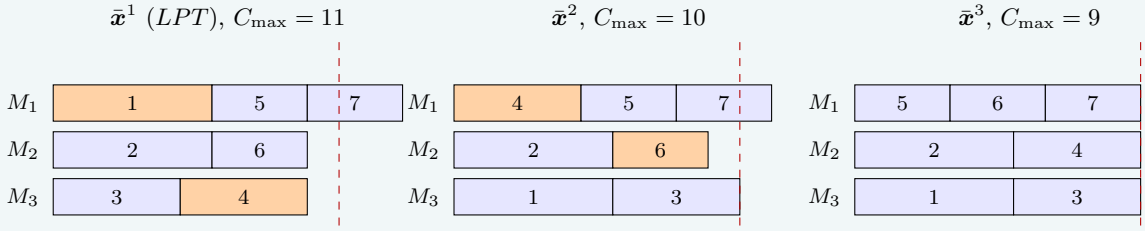
$k = 2$. By Observation 5.2 the neighbourhood is the move neighbourhood, and LPT is a local optimum for it (chapter «Local search»): the sub-MIP is infeasible. Add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 3$ and set $k = 4$.

$k = 4$. The best solution with two jobs reassigned has makespan 10; the solver returns the swap of jobs 1 and 4, $\bar{\boldsymbol{x}}^2$ below. Add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^1) \geq 5$.

$k = 4$, around $\bar{\boldsymbol{x}}^2$. The solver returns the swap of jobs 4 and 6, of makespan 9: $\bar{\boldsymbol{x}}^3$. Add $\Delta(\boldsymbol{x}, \bar{\boldsymbol{x}}^2) \geq 5$.

$k = 4$ and $k = 6$, around $\bar{\boldsymbol{x}}^3$. No solution of makespan 8: k becomes $8 > k_{\max}$, and the

method stops with makespan 9, the optimum.



The dashed line is the lower bound 9; the jobs swapped to obtain the next solution are coloured.

- In the language of the chapter «Local search», local branching performs a local search with a neighbourhood whose size is controlled by k, explored exactly by the solver, and it enlarges the neighbourhood when it gets stuck — exactly the idea of variable neighbourhood search, with the solver doing the exploration.

5.6.3 Proximity search

- A proximity sub-MIP that is infeasible with $\theta = 1$ proves that the incumbent is optimal, when the objective takes integer values (the proposition of Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*).

Example 5.12: Proximity search on the KP

From $\bar{\boldsymbol{x}} = (1, 0, 1, 0)$, of profit 16: the closest solution of profit at least 17 is $(1, 1, 0, 0)$, at distance 2 (item 3 out, item 2 in), of profit 17. Around it, no solution has profit at least 18: by the optimality condition of proximity search (Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*), $\{1, 2\}$ is **optimal**.

Example 5.13: Proximity search on $P\,||\,C_{\max}$

From the solution of LPT, of makespan 11:

iteration	cutoff	$\min \Delta$	solution returned	$C_{\max}$
1	$C_{\max} \leq 10$	4	$J_1 = \{5, 6, 7\}$, $J_2 = \{1, 2\}$, $J_3 = \{3, 4\}$	10
2	$C_{\max} \leq 9$	4	$J_1 = \{5, 6, 7\}$, $J_2 = \{1, 4\}$, $J_3 = \{2, 3\}$	9
3	$C_{\max} \leq 8$	—	infeasible	—

By Observation 5.2, distance 4 means two jobs reassigned: jobs 1 and 6 at the first iteration, jobs 2 and 4 at the second. No solution at distance 2 improves the makespan, as for local branching with $k = 2$. At the third iteration the sub-MIP is infeasible, and the optimality condition of proximity search (Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*) proves that makespan 9 is optimal.

- The proof of optimality comes from the solver, which proves the infeasibility of the last sub-MIP with its own dual bounds: proximity search, used with $\theta = 1$, is an exact method that improves the incumbent along the way.

5.7 Relax-and-fix and fix-and-optimize on lot sizing

- Relax-and-fix and fix-and-optimize decompose the MIP into blocks of integer variables. Lot sizing is a natural application, because its variables are indexed by the periods $t \in T = \{1, 2, \dots, m\}$ of a planning horizon, and the decisions of a period mainly interact with those of the nearby periods. The blocks are the setup variables of consecutive periods.

- The model is the ILP_{CLSP} model of the chapter «Formulations of lot sizing problems», with the strong big-M value $M_t = \min\{cap_t, D_{t,m}\}$, where $D_{t,m} = \sum_{k=t}^{m} d_k$; for an uncapacitated instance, $cap_t = D_{t,m}$. Its variables are the production x_t, the inventory s_t at the end of period t and the setup y_t, for every period $t \in T$.

Using these variables, an ILP model for the capacitated lot sizing problem (CLS) reads as follows:

$$\min \quad \sum_{t \in T} (sc_t\, y_t + vc_t\, x_t + hc_t\, s_t) \tag{5.4a}$$

$$\text{subject to} \quad s_{t-1} + x_t = d_t + s_t, \qquad \forall t \in T, \tag{5.4b}$$

$$x_t \le M_t\, y_t, \qquad \forall t \in T, \tag{5.4c}$$

$$x_t \le cap_t, \qquad \forall t \in T, \tag{5.4d}$$

$$s_0 = s_m = 0, \tag{5.4e}$$

$$x_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{5.4f}$$

$$s_t \in \mathbb{Z}_{\ge 0}, \qquad \forall t \in T, \tag{5.4g}$$

$$y_t \in \{0, 1\}, \qquad \forall t \in T. \tag{5.4h}$$

This ILP model is referred to as the ILP_{CLSP} model.

- Once the setups y are fixed, the rest is an LP with integer data and a network matrix, which has an integer optimal solution. The hard decisions are the m binary variables: as in the codes, the heuristics treat x_t and s_t as continuous variables, with $x_t \ge 0$ and $s_t \ge 0$ in place of (5.4f) and (5.4g).

- The LP relaxation is weak, because a fraction $y_t = x_t / M_t$ of a setup is enough to produce x_t.

Example 5.14: Relax-and-fix on an uncapacitated lot sizing instance

The instance of the module *MIP Modelling*, Chapter 4 «Constructive heuristics», has $m = 5$ periods, $d = (20, 10, 30, 40, 10)$, $sc_t = 50$, $vc_t = 0$ and $hc_t = 1$ for every $t \in T$, and no capacity. Its LP relaxation has value $\frac{36985}{396} \approx 93.40$, the constructive heuristic least unit cost of that chapter gives 200, and the optimal value is 170. Relax-and-fix with one period per block, $B_t = \{y_t\}$, finds a solution of value 170: **optimal**.

Example 5.15: Relax-and-fix on a capacitated instance

Consider $m = 6$ periods, $d = (50, 30, 10, 10, 40, 10)$, $sc_t = 40$, $vc_t = 0$, $hc_t = 1$ and $cap_t = 80$ for every $t \in T$. The LP relaxation has value $\frac{2150}{21} \approx 102.38$ and the optimal value is 160, with setups in periods 1, 2, 5. Relax-and-fix with $B_t = \{y_t\}$:

step r	value of the MIP of step r	y of the solution	fixing
1	$\frac{2465}{21} \approx 117.38$	$(1,\ 0.375,\ 0.143,\ 0.167,\ 1,\ 0)$	$y_1 := 1$
2	$\frac{2780}{21} \approx 132.38$	$(1,\ 0,\ 0.143,\ 0.167,\ 1,\ 0)$	$y_2 := 0$
3	$\frac{500}{3} \approx 166.67$	$(1,\ 0,\ 1,\ 0.167,\ 1,\ 0)$	$y_3 := 1$
4	170	$(1, 0, 1, 0, 1, 0)$	$y_4 := 0$
5	170	$(1, 0, 1, 0, 1, 0)$	$y_5 := 1$
6	170	$(1, 0, 1, 0, 1, 0)$	$y_6 := 0$

(the fractional values are rounded to three decimals). The result has setups in periods 1, 3, 5: production 80, 20, 50, stocks 30, 10, 10 at the end of periods 1, 3, 5, cost $3 \cdot 40 + 50 = 170$. The wrong decision is taken at step 2: with the future relaxed, $y_2 = 0$ gives 132.38 and $y_2 = 1$ gives 142.38, so relax-and-fix saves the setup of period 2; but then the capacity forces a setup in period 3, which the relaxation had paid only for a fraction 0.143.

Example 5.16: Fix-and-optimize on the capacitated instance

Start from the solution of relax-and-fix of Example 5.15, $\bar{\boldsymbol{y}} = (1, 0, 1, 0, 1, 0)$ of cost 170, with the windows of two consecutive periods $W_t = \{y_t, y_{t+1}\}$, $t = 1, 2, \ldots, 5$.

window	incumbent after the window	
$\{y_1, y_2\}$	170	no improvement
$\{y_2, y_3\}$	160	$\bar{\boldsymbol{y}} = (1, 1, 0, 0, 1, 0)$
$\{y_3, y_4\}$, $\{y_4, y_5\}$, $\{y_5, y_6\}$	160	no improvement
second pass, all windows	160	no improvement: stop

The window $\{y_2, y_3\}$ revises the wrong decision of relax-and-fix: the setup moves from period 3 to period 2. The plan produces 50, 50, 50 in periods 1, 2, 5, with stocks 20 and 10 at the end of periods 2, 3 and 10 at the end of period 5: cost $3 \cdot 40 + 40 = 160$, the **optimum**.

5.8 Heuristics inside the solver

- The heuristics of Gurobi, their parameters and the MIP start are described in Chapter 10 «MIP heuristics» and in Chapter 9 «MIP solvers: what they do» of the module *Integer Linear Programming*. A heuristic solution computed outside the solver — by FFD, LPT, a local search — can be passed to it as a MIP start (the attribute `Start` of the variables).

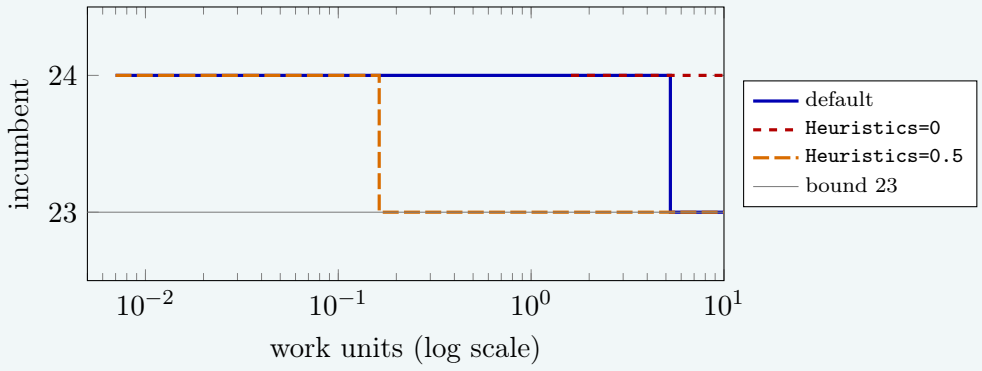
Example 5.17: The effect of the heuristics of Gurobi on a medium BPP instance

The instance has $n = 60$ items, with weights $w_j = 20 + (s_j \bmod 41)$, where $s_1, s_2, \ldots$ is the Park–Miller sequence from the seed 1, and capacity $c = 100$:

58	29	22	22	55	45	38	40	34	32
43	57	20	57	53	32	40	57	50	33
29	49	37	20	27	22	55	40	48	40
52	24	21	48	31	36	38	46	30	20
31	58	40	28	54	50	26	31	27	54
26	26	22	30	33	55	27	50	47	26

The total weight is 2271, so $LB = \lceil 2271/100 \rceil = 23$; FFD uses 24 bins, and the ILP_{BPP} model is built with $m = 24$ bins. Gurobi 13.0.3 is run on one thread with a limit of 10 *work units* (a deterministic measure of the computing effort, roughly a second each), with different parameters.

parameters	incumbent	bound	nodes	work	incumbents found (work: value)
default	23	23	2533	5.27	0.007: 24; 5.273: 23
`Heuristics=0`	24	23	4809	10.00	1.610: 24
`Heuristics=0.5`	23	23	1	0.16	0.007: 24; 0.163: 23
`MIPFocus=1`	23	23	1	0.16	0.007: 24; 0.160: 23
`NoRelHeurWork=1`	23	23	2533	6.48	0.007: 24; 6.477: 23
`RINS=0`	23	23	2959	5.86	0.007: 24; 5.861: 23



With the default parameters, a solution of 24 bins is found at once, and the optimal one only after 2533 nodes. Spending more effort in the heuristics (`Heuristics=0.5` or `MIPFocus=1`) finds 23 at the root, about 30 times faster; without heuristics the first solution arrives after 1.61 work units, and the optimum is not found within the limit. Since the bound is 23 from the root, the whole difficulty of this instance is primal: finding the solution, not proving it optimal.

- The numbers depend on the instance, on the version of the solver and on the machine; a single instance proves nothing (chapter «Heuristic algorithms: definitions»). The lesson is qualitative: on instances whose bound is strong, as the BPP with $LB = z(I)$, the time to solve is the time to find a good solution, and the heuristics of the solver — or a good MIP start — decide it.

5.9 What the MIP-based heuristics give

- The chapter «Heuristic algorithms: definitions» ended with a warning: a heuristic gives only one side of the gap, and without a dual bound nothing can be said about the quality of its solution. The heuristics of the previous chapters give *only* primal bounds.

- A MIP-based heuristic works with a MIP model, and the model comes with its LP relaxation; run inside a MIP solver, it comes with the dual bound of the branch-and-bound as well. For the first time, the heuristic **knows how far it is from the optimum** while it runs: the gap of the chapter «Heuristic algorithms: definitions» is computed at every moment, and when it is 0 the solution is *proved* optimal.

	heuristics of the previous chapters			MIP-based heuristics		
instance	solution	dual bound	gap	solution	dual bound	gap
KP	GR: 16	LP: 17	6.25%	LB, PS: 17	PS: 17	0%
BPP	FF: 4	$\lceil 21/7 \rceil = 3$	33.3%	pump (seed 7): 3	LP: 3	0%
TSP	NN: 25	two-edge: 17	47.1%	rounding: 18	LP: 18	0%
$P \,\|\|\, C_{\max}$	LPT: 11	9	22.2%	RINS, LB, PS: 9	LP: 9	0%
lot sizing, capacitated	RF: 170	LP: 102.38	66.0%	FO: 160	solver: 160	0%
medium BPP	FFD: 24	23	4.3%	Gurobi: 23	23	0%

- On the KP the integer rounding of the LP value is $\lfloor 17.75 \rfloor = 17$, and proximity search proves the optimality of 17 by itself, by the optimality condition of proximity search (Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*). On the capacitated lot sizing instance the LP bound is weak, and the gap is closed by the branch-and-bound of the solver, which proves that 160 is optimal.

A MIP-based heuristic does not only find solutions: inside a MIP solver it works with a dual bound, so the gap of its solution is always known, and a gap equal to 0 certifies optimality.

5.10 Typical mistakes

Warning

- **Rounding a problem with equality constraints.** Independent roundings of an assignment constraint almost never sum to 1: on the BPP and on $P \,||\, C_{\max}$ rounding fails.
- **Believing that a dive can undo a decision.** A dive is one path of the tree, without backtracking: the first wrong fixing (on the KP, $x_2 := 0$) is never revised.
- **Using the feasibility pump to get a good solution.** After the first LP the pump ignores the objective: it returns a feasible solution, possibly with empty bins left open.
- **Trusting hard fixing.** RINS fixes the variables on which the LP and the incumbent agree, so it cannot find a better solution that changes one of them.
- **Forgetting the cutoff constraint.** Without $f(\boldsymbol{x}) \leq f(\bar{\boldsymbol{x}}) - \theta$ the sub-MIP returns the incumbent solution again; with it, an infeasible sub-MIP proves a local optimum.
- **Claiming optimality without the conditions.** The optimality condition of proximity search (Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*) needs $\theta = 1$ and integer objective values on all the feasible solutions.
- **Ignoring the myopia of relax-and-fix.** The relaxed future is optimistic: a decision that looks cheap can force a setup later.
- **Generalizing from one instance or one solver run.** Paths and times depend on the version of the solver and on the tie-breaking rules, so the lessons are qualitative.

5.11 In summary

Summary

- The MIP-based heuristics of Chapter 10 «MIP heuristics» of the module *Integer Linear Programming* apply to the classical problems through their MIP models.
- Rounding and diving build a solution from the LP; the feasibility pump looks only for feasibility; RINS, local branching and proximity search improve an incumbent solution with a sub-MIP.
- Relax-and-fix builds a solution along the time, fix-and-optimize improves it window by window.
- Inside a MIP solver the dual bound is available: the gap of the heuristic solution is known, and a gap equal to 0 certifies optimality.

Dashboard of the reference instances. Best solution found by the methods of this chapter against the dual bound (max for the KP, min for the others):

instance	solution	dual bound	gap
KP	17	17	0%
BPP	3	3	0%
$P \mid\mid C_{\max}$	9	9	0%
lot sizing, capacitated	160	160	0%
medium BPP	23	23	0%

5.12 Python implementation

- The codes of the methods (rounding, diving, the feasibility pump, RINS, local branching, proximity search, relax-and-fix and fix-and-optimize) are those of Chapter 10 «MIP heuristics» of the module *Integer Linear Programming*. Here we give the models of the classical problems (`models.py`) and the experiment with the parameters of Gurobi (`gurobi_heuristics.py`); the folder `MIP_HEUR_CODE` contains all the functions and `instances.py`, with the instances of the chapter and of the exercises. The command `python3 run_tests.py` runs every example of the chapter and every instance of the exercises, checking each value against the expected value, written in the file.

```
import math
import gurobipy as gp
from gurobipy import GRB

EPS = 1e-6                                  # integrality and
    comparison tolerance

ENV = gp.Env(empty=True)                    # one silent, single-
    thread (deterministic) environment
ENV.setParam("OutputFlag", 0)
ENV.setParam("Threads", 1)
ENV.start()


def int_index(m):
    """positions j (in m.getVars()) of the integer and binary variables
       : the set I"""
    return [j for j, v in enumerate(m.getVars()) if v.VType != GRB.
       CONTINUOUS]


def nearest(v):
    """[v] = floor(v + 1/2): rounding to the nearest integer, 1/2
       rounded up"""
    return math.floor(v + 0.5 + EPS)


def is_integral(v):
    return abs(v - round(v)) <= EPS

```

```
26
27 def solve_lp(m):
28     """optimal solution of the LP relaxation of m (with its current
           bounds), None if infeasible"""
29     m.update()
30     r = m.relax()
31     r.optimize()
32     if r.Status != GRB.OPTIMAL:
33         return None
34     return [v.X for v in r.getVars()]
35
36
37 def obj_of(m, x):
38     """value of the objective function of m at the point x"""
39     e = m.getObjective()
40     return e.getConstant() + sum(e.getCoeff(k) * x[e.getVar(k).index]
           for k in range(e.size()))
41
42
43 def round_point(x_tilde, I):
44     """x_bar = [x_tilde]: round the integer components, copy the
           continuous ones"""
45     x_bar = list(x_tilde)
46     for j in I:
47         x_bar[j] = nearest(x_tilde[j])
48     return x_bar
49
50
51 def evaluate(m, x_int):
52     """fix the integer variables to x_int and optimize the continuous
           ones:
53         returns (value, x), or None if the point cannot be completed to
               a feasible solution"""
54     f = m.copy()
55     V = f.getVars()
56     for j in int_index(f):
57         V[j].LB = V[j].UB = x_int[j]
58     f.optimize()
59     if f.Status != GRB.OPTIMAL:
60         return None
61     return f.ObjVal, [v.X for v in V]
62
63
64 def solve_mip(m):
65     """optimal value and solution of the MIP m, None if infeasible"""
66     m.optimize()
67     if m.SolCount == 0:
68         return None
69     return m.ObjVal, [v.X for v in m.getVars()]
70
71
72 def distance(m, x_bar, I):
73     """the linear expression Delta(x, x_bar) = sum_{j in I} |x_j -
```

```
        x_bar_j| in the variables of m:
        x_j - l_j if x_bar_j is at its lower bound, u_j - x_j if at its
            upper bound,
        otherwise an auxiliary variable d_j >= |x_j - x_bar_j|"""
    V = m.getVars()
    expr = gp.LinExpr()
    for j in I:
        if x_bar[j] == V[j].LB:
            expr += V[j] - x_bar[j]
        elif x_bar[j] == V[j].UB:
            expr += x_bar[j] - V[j]
        else:
            d = m.addVar(name=f"d[{j}]")
            m.addConstr(d >= V[j] - x_bar[j])
            m.addConstr(d >= x_bar[j] - V[j])
            expr += d
    return expr


def delta(x, x_bar, I):
    """value of Delta(x, x_bar)"""
    return sum(abs(x[j] - x_bar[j]) for j in I)


def add_cutoff(m, z, theta):
    """require a solution better than z by at least theta"""
    obj = m.getObjective()
    if m.ModelSense == GRB.MINIMIZE:
        m.addConstr(obj <= z - theta, name="cutoff")
    else:
        m.addConstr(obj >= z + theta, name="cutoff")


def note(trace, *items):
    """record a step of the algorithm (used to print the examples of
        the chapter)"""
    if trace is not None:
        trace.append(items)
```

```
from itertools import combinations


def build_kp(p, w, c):
    m = gp.Model("KP", env=ENV)
    n = len(p)
    x = m.addVars(n, vtype=GRB.BINARY, name="x")
    m.setObjective(gp.quicksum(p[j] * x[j] for j in range(n)), GRB.
        MAXIMIZE)
    m.addConstr(gp.quicksum(w[j] * x[j] for j in range(n)) <= c, name="
        capacity")
    m.update()
    return m

```

```
13
14 def build_bpp(w, c, nb):
15     m = gp.Model("BPP", env=ENV)
16     n = len(w)
17     x = m.addVars(n, nb, vtype=GRB.BINARY, name="x")
18     y = m.addVars(nb, vtype=GRB.BINARY, name="y")
19     m.setObjective(y.sum(), GRB.MINIMIZE)
20     m.addConstrs((x.sum(j, "*") == 1 for j in range(n)), name="assign")
21     m.addConstrs((-gp.quicksum(w[j] * x[j, b] for j in range(n)) + c *
           y[b] >= 0
22                   for b in range(nb)), name="capacity")
23     m.update()
24     return m
25
26
27 def build_tsp(d):
28     """DFJ model: one binary variable per arc (i,j), out- and in-degree
           , subtour elimination constraints"""
29     m = gp.Model("TSP", env=ENV)
30     n = len(d)
31     A = [(i, j) for i in range(n) for j in range(n) if i != j]
32     x = m.addVars(A, vtype=GRB.BINARY, name="x")
33     m.setObjective(gp.quicksum(d[i][j] * x[i, j] for i, j in A), GRB.
           MINIMIZE)
34     m.addConstrs((x.sum(i, "*") == 1 for i in range(n)), name="out")
35     m.addConstrs((x.sum("*", j) == 1 for j in range(n)), name="in")
36     for k in range(2, n):                          # S subset of the cities
           except the first one, |S| >= 2
37         for S in combinations(range(1, n), k):
38             m.addConstr(gp.quicksum(x[i, j] for i in S for j in S if i
                   != j) <= len(S) - 1)
39     m.update()
40     return m
41
42
43 def build_pcm(t, m):
44     model = gp.Model("PCmax", env=ENV)
45     n = len(t)
46     x = model.addVars(n, m, vtype=GRB.BINARY, name="x")
47     Cmax = model.addVar(name="Cmax")
48     model.setObjective(Cmax, GRB.MINIMIZE)
49     model.addConstrs((x.sum(j, "*") == 1 for j in range(n)), name="
           assign")
50     model.addConstrs((-gp.quicksum(t[j] * x[j, h] for j in range(n)) +
           Cmax >= 0
51                       for h in range(m)), name="load")
52     model.update()
53     return model
54
55
56 def build_lot_sizing(d, sc, vc, hc, cap=None):
57     """big-M model of lot sizing: production x_t, inventory s_t, setup
           y_t, M_t = min{cap_t, D_tm};
```

```
        cap = None: uncapacitated; x_t and s_t continuous (with y fixed,
            the LP has an integer optimum)"""
    m = gp.Model("LS", env=ENV)
    T = len(d)
    y = m.addVars(T, vtype=GRB.BINARY, name="y")
    x = m.addVars(T, name="x")
    s = m.addVars(T, name="s")
    m.setObjective(gp.quicksum(sc[t] * y[t] + vc[t] * x[t] + hc[t] * s[
        t] for t in range(T)), GRB.MINIMIZE)
    for t in range(T):
        m.addConstr((s[t - 1] if t > 0 else 0) + x[t] == d[t] + s[t],
            name=f"balance[{t}]")
        M = sum(d[t:]) if cap is None else min(cap[t], sum(d[t:]))
        m.addConstr(x[t] <= M * y[t], name=f"setup[{t}]")
        if cap is not None:
            m.addConstr(x[t] <= cap[t], name=f"capacity[{t}]")
    m.addConstr(s[T - 1] == 0, name="end")      # s_0 = s_m = 0
    m.update()
    return m


def build_ip2(A, b, c, U):
    """max c1 x1 + c2 x2 subject to A x <= b, 0 <= x <= U, x integer"""
    m = gp.Model("IP2", env=ENV)
    x = m.addVars(2, lb=0, ub=U, vtype=GRB.INTEGER, name="x")
    m.setObjective(c[0] * x[0] + c[1] * x[1], GRB.MAXIMIZE)
    for k in range(len(A)):
        m.addConstr(A[k][0] * x[0] + A[k][1] * x[1] <= b[k])
    m.update()
    return m


def pcm_solution(m, sets):
    """the solution of the P||Cmax model with job sets J_1, ..., J_m
        (1-based jobs)"""
    x = [0] * m.NumVars
    index = {v.VarName: k for k, v in enumerate(m.getVars())}
    for h, J in enumerate(sets):
        for j in J:
            x[index[f"x[{j - 1},{h}]"]] = 1
    return evaluate(m, x)[1]
```

```
def run_with_parameters(m, params, work_limit):
    """solve m with the given Gurobi parameters for at most work_limit
        work units;
        returns incumbent, dual bound, nodes, work and the list of the
            incumbents found (work, value)"""
    s = m.copy()
    s.Params.WorkLimit = work_limit
    for name, value in params.items():
        s.setParam(name, value)
    found = []
    def callback(model, where):                 # called by Gurobi at
```

```
        every new incumbent
        if where == GRB.Callback.MIPSOL:
            found.append((model.cbGet(GRB.Callback.WORK), model.cbGet(
                GRB.Callback.MIPSOL_OBJ)))
    s.optimize(callback)
    z = s.ObjVal if s.SolCount > 0 else None
    return z, s.ObjBound, int(s.NodeCount), s.Work, found
```