Constructive heuristics

Outline

Bin packing: the insertion rules

$P||C_{\max}$: the least loaded rule

Set covering: the cheapest completion rule

Knapsack: the best ratio rule

TSP: the nearest neighbour

Lot sizing: least unit cost period covering

Local search, and what it does not give

When the constructive heuristic fails

What it gives, and what it does not

The overview of the heuristics

Learning objectives (1/2)

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

- describe a constructive heuristic by its three ingredients and say on which side its value is a bound;
- execute by hand next-fit, first-fit, best-fit, LPT, the covering, ratio and nearest-neighbour rules;

Learning objectives (2/2)

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

- compute the certified gap of a heuristic solution with a dual bound;
- recognise when a heuristic gets stuck and why it then gives no bound.

Prerequisites

- From this module, Chapter 2 «Relaxations, duality and bounds»: the dual bound, the bound from the other side to compare a heuristic with
- From this module, Chapter 3 «From the model to Python/Gurobi»: the bin packing, makespan and TSP models

What a constructive heuristic is

- It builds *one* solution, quickly, **adding one element at a time** and never backtracking
- It proves nothing about the quality of that solution
- It is not even guaranteed to reach a feasible one: it can get stuck part-way, with an element that fits nowhere — precisely because it never backtracks

The pessimistic side of the sandwich

- When the heuristic does end with a feasible solution, that solution is the other half of the sandwich of the chapter on dual bounds
- It is the **pessimistic side**: the one guaranteed by a solution that really exists
- When it fails, there is **no primal bound** at all

What a heuristic must produce (1/2)

1. a readable **pseudocode**, with the scanning order, the choice criterion, the tie-breaking rule and the failure case stated explicitly
2. the corresponding **Python function**, line by line (in `python/euristiche.py`)
3. the **trace** of the execution on an instance

What a heuristic must produce (2/2)

4. the **feasibility check** of the solution produced: constraints, bounds *and* integrality
5. the resulting **bound**, with the right name: UB in a minimisation, LB in a maximisation

Point 4 is not a formality: a solution satisfying the linear constraints but with a fractional component is feasible for the *relaxation*, not for the MILP, and its value is not a primal bound.

The three ingredients

Every constructive heuristic is described by saying three things, and only three:

1. an **order** in which the elements to be placed are looked at
2. an **insertion rule**, which says where the current element goes
3. a **feasibility check**, which says when an insertion cannot be made

Changing just one of the three gives a different heuristic, with a different value.

Observation 1: The side of the bound depends on the objective, not on the heuristic

A heuristic produces a feasible solution, and its value is attainable.

In a ***minimisation*** *that value is an upper bound:* $z(MILP) \leq UB$.

In a ***maximisation*** *it is a lower bound:* $LB \leq z(MILP)$.

Calling UB the result of a constructive heuristic on a maximisation is the commonest sign error in the course.

Bin packing: the insertion rules

Introduction: bin packing

- Bin packing is the first classic: items must be put into identical bins of limited capacity, using as few of them as possible
- Here it is not solved: it is *built*, one choice at a time
- We will see three insertion rules — next-fit, first-fit, best-fit — and how, on the same instance, they give very different results

Next-fit, first-fit, best-fit: the pseudocode (1/2)

Input: n, weights $w[1..n]$, capacity C, criterion γ.
Output: the bins used.

- start with no bin open
- for $j := 1$ to n:
  - choose b^* by the rule
  - if no open bin will do, open a new one
  - put j into b^* and update $res[b^*] \leftarrow res[b^*] - w[j]$

Next-fit, first-fit, best-fit: the pseudocode (2/2)

The rule that chooses b^*:

- **next-fit**: the current bin only, then a new one is opened
- **first-fit**: the first open bin the item fits into
- **best-fit**: among the bins it fits into, the one with the smallest $\gamma(j, b, res)$

The order and the ties

- All three scan the items **in the given order**
- Changing the order changes the result, and this must be said when a value is reported
- Ties in the criterion γ are broken on the smallest index, so the run is reproducible

The instance

- Six items of weight $w = (4, 4, 5, 3, 2, 3)$, in bins of capacity $C = 7$
- The same bins as the model of the gurobipy chapter, with two more items: on the small instance all three rules answer "three bins" and cannot be told apart

Heuristic	how it fills	UB	$z(MILP)$	gap
next-fit	[4] [4] [5] [3+2] [3]	5	3	66.7%
first-fit	[4+3] [4+2] [5] [3]	4	3	33.3%
best-fit on the fill	[4+3] [4+3] [5+2]	3	3	0.0%

The elementary bound

- The total weight is $4 + 4 + 5 + 3 + 2 + 3 = 21$ and one bin carries 7
- So at least

$$\left\lceil \frac{21}{7} \right\rceil = 3 \quad \text{bins}$$

  are needed
- Best-fit uses three, and fills all three exactly: $LB = UB = 3$
- The optimum is **proved** without ever calling the solver

Where the three rules part company (1/2)

Next-fit:

- closes a bin as soon as an item does not fit, and never looks back
- the last item, of weight 3, would fit exactly into the first bin, which still has 3 of room
- but that bin is no longer looked at: five bins instead of three

Where the three rules part company (2/2)

Between **first-fit** and **best-fit** the difference is entirely about the item of weight 2:

- first-fit puts it in the *first* bin that takes it, the second, which is left with 1 of room now useless
- best-fit puts it where it fits *exactly*, the third, which had 2
- so the final 3 still finds room in the second, and the bins stay three

Summary: bin packing

The three rules:

- next-fit looks only at the current bin, first-fit at the first that takes the item, best-fit at the one that fills best
- on the same instance they give 5, 4 and 3 bins

What to take away:

- the scanning order is part of the heuristic and must always be stated
- the elementary bound $\lceil \sum_j w_j / C \rceil$ costs nothing and here settles the problem

$P||C_{max}$: the least loaded rule

Introduction: scheduling on identical machines

- The second classic is scheduling on identical machines, written $P||C_{\max}$ in standard notation
- n jobs of duration t_j over k identical machines, minimising the instant the last one finishes
- We will see the natural rule (*list scheduling*) and the order that makes it good, which gives **LPT**

The idea

- The *makespan* is the load of the busiest machine
- **List scheduling**: the current job goes to the machine that is least loaded at that moment
- The order in which the jobs are looked at decides the result: the one that works best is by **decreasing duration**
- The rule it gives has a name of its own: **LPT** (*longest processing time first*)

Why the decreasing order

Leaving the long jobs for last makes them impossible to place:
when they arrive, every machine is already loaded
and the long job adds in full to the smallest load.

LPT: the pseudocode

Input: n, k, durations $t[1..n]$.
Output: an assignment x and its makespan.

- $L[m] \leftarrow 0$ for every m (current loads)
- for j in order of *decreasing* $t[j]$:
  - $m^* \leftarrow \arg\min_m L[m]$ (ties: the smallest index)
  - $x[j][m^*] \leftarrow 1$; $L[m^*] \leftarrow L[m^*] + t[j]$
- return x and $\max_m L[m]$

Example: seven jobs on three machines (1/2)

$t = (5, 5, 4, 4, 3, 3, 3)$, $k = 3$, total 27. The trace:

- **Steps 1–3.** The jobs 5, 5, 4 go to the three empty machines: $L = (5, 5, 4)$
- **Step 4.** Job 4: the smallest load is machine 3, which goes to 8. $L = (5, 5, 8)$
- **Steps 5–6.** The two jobs of length 3 go to machines 1 and 2: $L = (8, 8, 8)$
- **Step 7.** The last job of length 3 finds all loads equal to 8; by the tie rule it goes to machine 1, which reaches 11

Example: seven jobs on three machines (2/2)

- LPT makespan: $UB = 11$, with loads $(11, 8, 8)$

The elementary bound. The makespan is at least

$$\max\left(\max_j t_j,\ \frac{\sum_j t_j}{k}\right) = \max\left(5,\ \frac{27}{3}\right) = 9,$$

- no machine can do less than the longest job, and at least one must carry the average
- the optimum is exactly $z(MILP) = 9$, attained with $\{5, 4\}$, $\{5, 4\}$, $\{3, 3, 3\}$: LPT is off by 22.2%

Two free bounds, to be compared

- $\max_j t_j$ and $\sum_j t_j/k$ are two lower bounds computable without solving anything
- The better of the two is often already close to the optimum
- An "obvious" bound that nobody writes down is a wasted bound: the dual bound is for when the obvious ones are not enough, not instead of them

Summary: $P||C_{max}$

The rule:

- list scheduling: every job to the least loaded machine
- order by decreasing duration: this is LPT

On the seven-job instance:

- $UB = 11$ from LPT, $LB = 9$ from the two elementary bounds, $z(MILP) = 9$
- bounds computable by hand must always be written down: they are free

Set covering: the cheapest completion rule

Introduction: set covering

- In set covering one chooses elements of known cost so that every zone is covered by at least one of them
- The natural constructive rule does not look at the absolute cost, but at the **cost per newly covered zone**
- We will see the heuristic, the trace on the four-teams instance and the comparison with the dual bound

Constructive covering heuristic: the pseudocode

Input: costs $c[1..n]$, sets $S[1..m]$ (who covers zone i).
Output: a set of elements covering all zones.

- $uncov \leftarrow \{1, 2, \ldots, m\}$; $\quad y[j] \leftarrow 0$ for every j
- while $uncov \neq \emptyset$:
  - for every j not yet chosen: $new(j) \leftarrow |\{i \in uncov : j \in S_i\}|$
  - if $new(j) = 0$ for every j: return "no solution found"
  - $j^* \leftarrow \arg\min_{j:new(j)>0} c[j]/new(j)$
  - $y[j^*] \leftarrow 1$; $\quad uncov \leftarrow uncov \setminus \{i : j^* \in S_i\}$

The criterion

The criterion is the **cost per newly covered zone**,
not the absolute cost.

Picking the cheapest element regardless of how much it covers leads to far worse solutions.

Example: the four teams (1/2)

Costs $c = (4, 3, 5, 3)$, six zones (the six pairs of districts).

- **Step 1.** All six zones uncovered; ratios $4/3$, $3/3 = 1$, $5/3$, $3/3 = 1$; a tie between elements 2 and 4, element 2 is chosen and covers zones 1, 2 and 5
- **Step 2.** Zones 3, 4, 6 remain; ratios $4/2 = 2$, $5/2$, $3/2$; element 4 is chosen and covers zones 4 and 6
- **Step 3.** Zone 3 remains; ratios $4/1$ and $5/1$; element 1 is chosen

Example: the four teams (2/2)

- Solution $\{1, 2, 4\}$, of cost $UB = 10$
- Here it is the optimum: $z(MILP) = 10$
- The dual bound built in the chapter on dual bounds is 7: the **certified gap** is 30%, even though the heuristic was already optimal
- The certified gap measures what we *know*, not what we have obtained

Summary: set covering

The rule:

- at every step the element with the smallest ratio $c[j]/new(j)$ is chosen
- it stops when no zone is left uncovered

On the four-teams instance:

- $UB = 10$, which is the optimum, but the available dual bound is 7
- an optimal heuristic with a 30% certified gap: the gap says how much is known, not how much is lost

Knapsack: the best ratio rule

Introduction: the knapsack

- In the knapsack the items have a *value* as well as a weight, and there is a single resource
- It is a **maximisation**: the bound it gives is called LB, not UB
- The natural constructive rule looks at the value/weight ratio: we will see what it produces and where it goes wrong

Heuristic by value/weight ratio: the pseudocode

Input: values $p[1..n]$, weights $w[1..n]$, capacity C.
Output: a feasible subset.

- $residual \leftarrow C$; $\quad y[j] \leftarrow 0$ for every j
- for j in order of *decreasing* $p[j]/w[j]$:
  - if $w[j] \leq residual$: $\quad y[j] \leftarrow 1$; $\quad residual \leftarrow residual - w[j]$
- return y

Example: four items

Instance: $p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $C = 9$.

- The ratios are 2, $7/4$, 2, $4/3$
- Item 1 and then item 3 are taken (total weight 8), items 2 and 4 no longer fit
- Value of the solution: 16

Since the problem is a *maximisation*,

$$LB = 16 \leq z(MILP) = 17,$$

with a gap of 5.9%.

Why it goes wrong

- The optimum takes items 1 and 2, filling the knapsack exactly $(5 + 4 = 9)$
- The ratio heuristic goes wrong precisely because item 3, better by ratio, leaves an **unusable residual**
- The value/weight ratio is a local criterion: it does not see what is left to fill afterwards

Summary: knapsack

The rule:

- the items are scanned by decreasing ratio $p[j]/w[j]$ and every item that fits is taken

On the four-item instance:

- $LB = 16 \leq z(MILP) = 17$: in a maximisation the heuristic bound is a *lower* one
- the error comes from the residual that item 3 leaves unusable

TSP: the nearest neighbour

Introduction: the travelling salesman problem

- Given n cities and the distances $d_{ij} \in \mathbb{Q}_{>0}$ between every pair, find the shortest tour that visits them all once and returns to the start
- It is the problem where step-by-step construction shows best, because the solution *is* a sequence: the order is the solution
- We will see the nearest neighbour rule and the effect of the starting city

The nearest neighbour

- Start from a city and each time go to the nearest among those not yet visited
- When none are left, return to the start
- It is **feasible by construction**: every city is entered once, and the last arc closes the tour
- And it is fast, because at each step it looks only at the distances from the current city

Example: five cities (1/3)

The distances, symmetric:

	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	—

Example: five cities (2/3)

Starting from city 1:

- the nearest is 3 (distance 2); from there 2 (4); then 4 (3); 5 is left (7); and the return to 1 costs 9
- the tour $1 \to 3 \to 2 \to 4 \to 5 \to 1$ has length 25

The last arc is the one that is paid for: the rule chooses well while it has a choice, and at the last step it has none.

Example: five cities (3/3)

Changing the starting city changes the tour, and changes the length:

start	tour	length
1	$1 \to 3 \to 2 \to 4 \to 5 \to 1$	25
2	$2 \to 4 \to 1 \to 3 \to 5 \to 2$	18
3	$3 \to 1 \to 4 \to 2 \to 5 \to 3$	18
4	$4 \to 1 \to 3 \to 2 \to 5 \to 4$	19
5	$5 \to 2 \to 4 \to 1 \to 3 \to 5$	18

With five cities the distinct tours are $(5-1)!/2 = 12$: by enumeration the optimum is $1 \to 3 \to 5 \to 2 \to 4 \to 1$, of length 18.

Two things to take away

- The heuristic gives *one* feasible solution, hence an upper bound — here $z(MILP) \leq 25$ — and nothing else
  - that 18 is the optimum is known by enumeration, not because the heuristic says so
- Running the same rule from every start and keeping the best tour costs n times as much and gives a better upper bound
  - it is the simplest form of *multi-start*, and it is still a bound from one side only

Summary: TSP

The rule:

- from the current city always go to the nearest unvisited one, then close the tour
- feasible by construction, but the last arc is the one that is paid for

On the five-city instance:

- from city 1 the tour is 25, that is 38.9% above the optimum 18
- changing the start changes the value: multi-start improves the bound, but it is still one bound

Lot sizing: least unit cost period covering

Introduction: lot sizing

- Over T periods, with demand d_t to be met, a setup cost f every time production is launched and a cost h for every unit held in stock for one period
- One decides in which periods to produce, and how much
- The whole tension is there: **launching rarely costs stock, launching often costs setups**

The least unit cost rule

- From the first period with demand, try to cover $1, 2, 3, \ldots$ consecutive periods
- For each choice compute the **average cost per unit** produced — setup plus holding, divided by the quantity
- Take the covering with the lowest average cost
- Then restart from the first period still uncovered

Least unit cost: the pseudocode

Input: demands $d[1..T]$, setup cost f, unit holding cost h.

- $t \leftarrow 1$; while $t \leq T$:
  - skip the periods with $d[t] = 0$
  - for $k := 1$ to $T - t + 1$:

$$q_k \leftarrow \sum_{s=t}^{t+k-1} d[s], \qquad c_k \leftarrow \frac{f + h \sum_{s=t}^{t+k-1} (s-t)\, d[s]}{q_k}$$

  - $k^* \leftarrow \arg\min_k c_k$ (the lowest average cost per unit)
  - produce q_{k^*} in period t; $t \leftarrow t + k^*$

Observation 2: This is not the Wagner–Whitin procedure

Wagner–Whitin is an ***exact*** *dynamic-programming algorithm for the uncapacitated lot-sizing model: it solves that model to optimality in polynomial time.*

The procedure above is a heuristic, and its value is only a bound. Calling it "the Wagner–Whitin constructive heuristic" confuses two different things.

Example: five periods

Demands $d = (20, 10, 30, 40, 10)$, setup $f = 50$, holding $h = 1$.

- **Step 1.** From period 1 it pays to cover 2 periods (quantity 30, unit cost 2)
- **Step 2.** From period 3 it pays to cover 2 periods (quantity 70, unit cost ≈ 1.286)
- **Step 3.** From period 5 only that period remains (quantity 10, unit cost 5)

Cost of the heuristic solution: $UB = 200$. The optimum of the corresponding MILP is $z(MILP) = 170$, a gap of 17.6%.

Summary: lot sizing

The rule:

- cover the block of periods with the lowest average cost per unit, then restart
- the trade-off between setups and stock is all in the ratio c_k

On the five-period instance:

- $UB = 200$ against $z(MILP) = 170$: a gap of 17.6%
- Wagner–Whitin would give 170, but it is an exact algorithm, not this heuristic

Local search, and what it does not give

Introduction: local search

- After building a solution, the natural next step is trying to improve it with small changes
- We will see what a local optimum is and why it depends on the family of moves chosen
- And above all what local search does **not** give: no bound on the optimistic side

One step beyond construction

- A **local search** starts from a feasible solution and tries elementary moves (move a job, swap two items), accepting those that improve
- It stops at a **local optimum**: a solution that no move of the chosen family improves
- "Local" is relative to the family of moves: changing the moves changes the local optimum

On the LPT example

LPT solution: $L = (11, 8, 8)$, makespan 11.

- The move "move one job to another machine" improves nothing
- Moving one of the two jobs of length 3 off machine 1 brings its load to 8, but raises the receiving machine to 11
- The local search stops at 11, while the optimum is 9: to get there a *swap* between two machines is needed, not a simple move

Observation 3: A local optimum is not a better bound

Local search returns a feasible solution, hence a bound on the pessimistic side, and nothing else.

It proves nothing about the optimum, produces no bound on the optimistic side, and the fact that it stopped does not mean it has arrived.

The only way to know how good it is, is to compare it with a bound from the other side.

Summary: local search

What it is:

- it starts from a feasible solution and accepts the moves that improve
- it stops at a local optimum, relative to the family of moves chosen

What it does not give:

- no proof about the optimum, no bound on the optimistic side
- on LPT it stops at 11 while the optimum is 9: swaps are needed, not moves

When the constructive heuristic fails

Introduction: failure

- A constructive heuristic can reach a point where the current element cannot be placed anywhere
- It is not a programming mistake: it is the consequence of **never backtracking**
- We will see an instance where this happens, and what can (and cannot) be concluded

Observation 4: "No solution found" is not "no solution exists"

Three jobs of duration $(3, 3, 2)$ *on two machines with availability* $(5, 3)$. *Next-fit:*

- *job 1 goes to machine 1 (residual* 2*);*
- *job 2 does not fit and moves to machine 2 (residual* 0*);*
- *job 3 does not fit and there are no more machines:* ***failure****.*

But the problem is feasible: jobs 2 and 3 fit together on machine 1 $(3 + 2 = 5)$ *and job 1 on machine 2* $(3 \leq 3)$.

A constructive heuristic is myopic

- It decides one thing at a time and never backtracks
- Its failure is information **about the heuristic**, not about the problem
- To prove that a model is infeasible one needs the solver (`Status = INFEASIBLE`) or a proof, not a constructive heuristic that went wrong

Observation 5: A heuristic that fails gives no bound at all

If the heuristic returns no solution, nothing has been obtained: there is no primal bound. The two things to do are, in this order:

1. *change the order — the order by decreasing size fails far less often than the natural one, because it places the difficult elements first;*
2. *change the insertion rule.*

If nothing is found even so

The question becomes whether the problem is feasible,
and that is a question **for the model**, not for the heuristic.

Summary: failure

What it means:

- the heuristic is myopic: it decides one thing at a time and never backtracks
- its failure speaks about the heuristic, not about the feasibility of the problem

What to do:

- first change the order (decreasing size), then the insertion rule
- infeasibility is proved with the solver or with an argument, never with a heuristic

What it gives, and what it does not

Introduction: the exact account

- So far we have seen six heuristics on six different problems
- Now we make the exact account of what is known after running one
- And we define the **certified gap**, the only measure of quality available *before* solving the MILP

The exact account of what is known

Let $\bar{z}$ be the value of the solution found by a constructive heuristic.

- In a **minimisation** problem one knows that

$$z(MILP) \leq \bar{z}, \qquad \text{that is} \quad UB = \bar{z}$$

- In a **maximisation** problem one knows that

$$z(MILP) \geq \bar{z}, \qquad \text{that is} \quad LB = \bar{z}$$

What is not known

- In neither case does one know *how far* $\bar{z}$ is from the optimum
- To know that, a bound from the other side is needed
- And that one does not come from a feasible solution: it comes from a **relaxation** or from a **combinatorial argument**

The certified gap

With both bounds in hand one writes the **certified gap**: in absolute terms $UB - LB$, and as a percentage

$$\frac{UB - LB}{|UB|} \text{ in a minimisation,} \qquad \frac{UB - LB}{|LB|} \text{ in a maximisation.}$$

- In a minimisation the incumbent is UB, in a maximisation it is LB
- The denominator is always the solution one actually holds, never $z(MILP)$
- That is what makes it the only measure of quality available *before* solving the MILP

Why build the heuristic by hand, if the solver is there

A modern solver finds feasible solutions on its own, often better ones. Building one by hand is still worth it:

- it forces one to understand what makes a solution feasible, and that is exactly the content of the constraints
- it gives a reference value independent of the solver: if the solver returns something worse, there is a mistake in the model
- on large problems the solver does not reach the optimum, and the quality of the final solution depends on the initial one

Summary: what a heuristic gives

What it gives:

- a primal bound: $UB = \bar{z}$ in a minimisation, $LB = \bar{z}$ in a maximisation
- a solution that can really be implemented

What it does not give:

- no information on the distance from the optimum
- the certified gap needs a bound from the other side, coming from a relaxation or a combinatorial argument

The overview of the heuristics

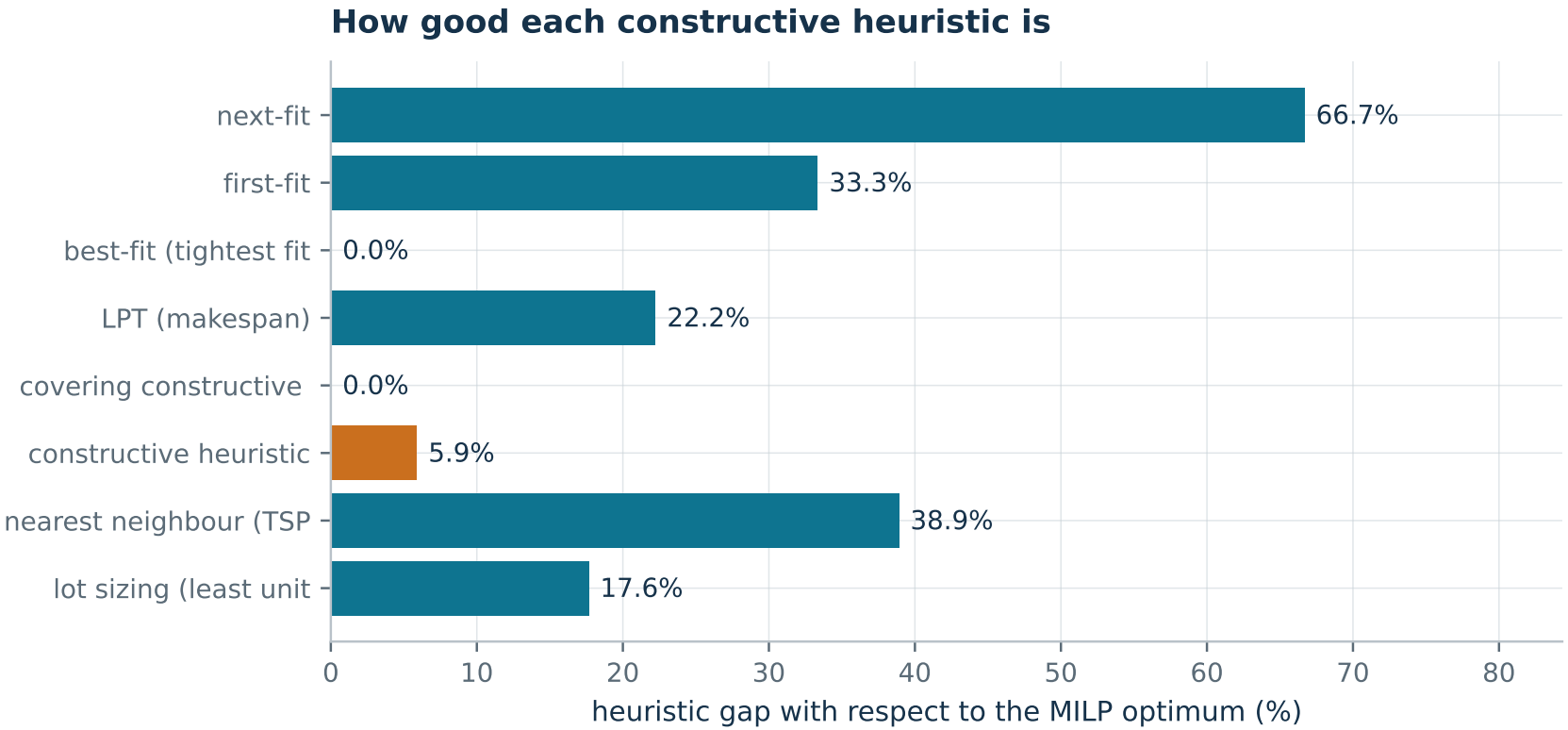
Introduction: the overall picture

- We put in a single table all the heuristics of the chapter, with their value, the MILP optimum and the gap
- It serves to see, all at once, how different from one another they are
- And to understand why the course always asks for **two bounds**

The summary table

Heuristic	Direction	value	$z(MILP)$	gap
next-fit (bin packing)	min (UB)	5	3	66.7%
first-fit (bin packing)	min (UB)	4	3	33.3%
best-fit on the fill	min (UB)	3	3	0.0%
LPT (makespan)	min (UB)	11	9	22.2%
covering (set covering)	min (UB)	10	10	0.0%
ratio (knapsack)	max (LB)	16	17	5.9%
nearest neighbour (TSP)	min (UB)	25	18	38.9%
least unit cost (lot sizing)	min (UB)	200	170	17.6%

The gap of each heuristic



Teal for minimisation problems, orange for maximisation.

What this table teaches

- Two heuristics find the optimum and six do not, and *before* solving the MILP there is no way of knowing which
- A 0% gap and a 67% gap are told apart only *afterwards*
- This is exactly why the course always asks for two bounds: a heuristic on its own says how much a solution one can actually implement costs, not how much is being lost

Summary: the overview

The table:

- eight heuristics on six problems, gaps from 0% to 66.7%
- two find the optimum, but there is no way of knowing which before solving

The moral:

- a heuristic on its own does not measure its own quality
- two bounds are always needed, one per side

Final summary (1/3)

What a constructive heuristic is:

- it builds a solution one element at a time, never backtracking
- it is described by three ingredients: order, insertion rule, feasibility check

The side of the bound:

- it depends on the objective, not on the heuristic: UB in a minimisation, LB in a maximisation

Final summary (2/3)

The rules seen:

- bin packing: next-fit, first-fit, best-fit
- $P||C_{\max}$: list scheduling by decreasing duration (LPT)
- set covering: cost per newly covered zone
- knapsack: decreasing value/weight ratio
- TSP: nearest neighbour
- lot sizing: lowest average cost per unit

Final summary (3/3)

The limits:

- a heuristic can fail: failure does not mean infeasibility
- local search stops at a local optimum, and gives no extra bound

The final account:

- a heuristic gives a bound from one side only
- the certified gap ($UB - LB$) needs a bound from the other side
- it is the only measure of quality available before solving the MILP