Constructive heuristics (1/2)

Outline

- Why constructive heuristics?
- Constructive heuristics
- The problems and the assumptions
- The greedy paradigm
- The 0/1 knapsack problem: which order?
- The bin packing problem: first sort, then pack
  - First Fit Decreasing
  - Best Fit Decreasing
  - What sorting gives, and what it does not
- Scheduling on parallel machines: longest jobs first
  - List scheduling
  - Longest processing time first

Learning 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.

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.

Why constructive heuristics?

Introduction: Why constructive heuristics?

- The problems we work on are NP-hard: for large instances an optimal solution may be out of reach, but a good feasible solution is still needed
- A constructive heuristic is the most natural way to obtain one: it builds the solution itself, one piece at a time
- Its solutions are also the starting points of the methods that improve a complete solution, which we will see later in the course

Building a solution step by step

- A **constructive heuristic** builds one feasible solution step by step: it starts from an empty solution and extends it with one element at a time, until the solution is complete.
- In this chapter we study how the choice of the element to add is made, and why this choice decides the quality of the result.

Summary

Constructive heuristics:

- build one feasible solution, one element at a time
- start from an empty solution and stop when the solution is complete

The question of the chapter:

- how is the element to add chosen at every step?
- why does this choice decide the quality of the result?

Constructive heuristics

Introduction: Constructive heuristics

- In this section we give the formal definition of a constructive heuristic
- It fixes the words used in the whole chapter: **partial solution** and **element**
- The rest of the chapter studies how the element to add is chosen

Constructive heuristic

Definition 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 (the greedy paradigm, in the next sections).

Summary

Constructive heuristic:

- starts from the empty partial solution
- extends it with one element at a time, until the solution is complete

Elements:

- items, assignments of items to bins and of jobs to machines, edges of a tour
- how the element is chosen decides the quality of the result

The problems and the assumptions

Introduction: The problems and the assumptions

- In this section we recall the problems on which the heuristics are illustrated
- We fix the notation used by all the heuristics of the chapter
- We recall the four reference instances, whose optimal values are known: they let us measure how good each heuristic is

The problems of this chapter

- 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),
  - $P \,||\, C_{\max}$;
- 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.

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 reference instances

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

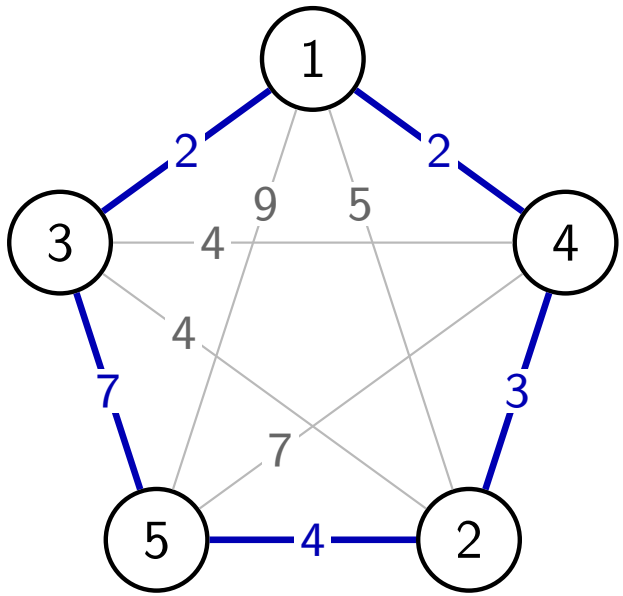
problem	reference instance	optimum
KP	$p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$	$\{1, 2\}$, $z(I) = 17$
BPP	$w = (4, 4, 5, 3, 2, 3)$, $c = 7$	$\{1, 4\}, \{2, 6\}, \{3, 5\}$, $z(I) = 3$
$P \mid\mid C_{\max}$	$t = (5, 5, 4, 4, 3, 3, 3)$, $m = 3$	$\{1, 3\}, \{2, 4\}, \{5, 6, 7\}$, $z(I) = 9$
TSP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

The TSP reference instance (1/2)

- 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 next figure shows the distances of the instance and, in bold, its optimal tour.

The TSP reference instance (2/2)

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	—



Summary

The problems:

- the KP, the BPP, the TSP and $P \,||\, C_{\max}$; the VCP for the greedy coloring and *DSATUR*
- $\mathscr{A}(I)$ is the value of the heuristic $\mathscr{A}$, $z(I)$ the optimal value

The reference instances:

- four instances with known optimal values: 17, 3, 9 and 18
- the five cities of the TSP instance are drawn on a pentagon, whose boundary is the optimal tour

The greedy paradigm

Introduction: The greedy paradigm

- In this section we describe the simplest way to choose the element to add: take the one that looks best now
- We write a template of pseudo-code that every heuristic of the chapter fills in
- We discuss the order of the choices, static and dynamic criteria, myopia, running time, and when greedy is exact

Three questions at every step

- 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),
  - **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.

Greedy algorithm

Definition 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 that follows 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 the chapter fills in.

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
11 end
```

The order decides everything

- The template examines the elements in the order induced by the selection criterion.
- 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;
  - examples: 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:
  - it must be recomputed at every step;
  - examples: distance from the last city of a partial tour, residual capacity of the bins, load of the machines.

Myopia and running time

- **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 this chapter no greedy algorithm is exact, and the question becomes *which* criterion gives good solutions.

The template for the four problems (1/2)

The 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

The template for the four problems (2/2)

problem	element	selection criterion	feasibility test
$P \mid\mid C_{\max}$	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

Summary

The greedy paradigm:

- three questions at every step: the candidates, the selection criterion, the feasibility test
- a greedy algorithm adds the candidate that looks best now; a choice is never revised

Design:

- the order induced by the criterion decides everything
- static criteria (sort once) and dynamic criteria (recompute at every step)
- myopia: fast, but a good choice now can force bad choices later
- for the four NP-hard problems no greedy algorithm is exact

The 0/1 knapsack problem: which order?

Introduction: The 0/1 knapsack problem: which order?

- In this section we apply the greedy paradigm to the KP
- The feasibility test is fixed, so the heuristic is determined by the order of the items: we compare three natural orders
- We see that each order can be arbitrarily bad, and how combining solutions protects against it

Greedy heuristics for the KP

- 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, by weight and by efficiency.

Three orders

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.

Greedy algorithm with order ρ

- 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.
- It selects each item if and only if it fits into the residual capacity.

Algorithm: Greedy with order (1/2)

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
```

Algorithm: Greedy with order (2/2)

function *GreedyOrderKP*(n, c, w, p, ρ) (continued)

```
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
14 end
```

Each criterion captures one aspect

- 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;
  - the efficiency balances the two, but ignores how well the items fill the knapsack.

Example: three orders on the KP reference instance (1/3)

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).

Example: three orders on the KP reference instance (2/3)

The executions, with the residual capacity after each step, are:

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

	step 3	step 4
GR_p	item 3: out	item 4: out
GR_w	item 2: out	item 1: out
GR	item 2: out	item 4: out

Example: three orders on the KP reference instance (3/3)

Hence

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

- 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.

No order is the best (1/2)

- 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 $z(I) = c$;

No order is the best (2/2)

Still with $c \geq 3$:

- **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.

Combining two solutions

- 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.
- 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.

Summary

Greedy for the KP:

- the feasibility test is fixed: the heuristic is determined by the order of the items
- three orders: profit (GR_p), weight (GR_w), efficiency (GR)
- on the reference instance: $GR_p(I) = 17$, $GR_w(I) = 10$, $GR(I) = 16$

No order is the best:

- each of the three orders can be arbitrarily bad
- Ext-Greedy, the better of GR and of the best single item, guarantees $\frac{z(I)}{GR^e(I)} \leq 2$

The bin packing problem: first sort, then pack

Introduction: The bin packing problem: first sort, then pack

- In this section we choose the order of the items for the BPP, not only the bin
- We study two heuristics, First Fit Decreasing and Best Fit Decreasing, and run them on the reference instance
- We see what sorting gives (a better worst-case guarantee) and what it does not (a better solution on every instance)

Greedy heuristics in the input order

- 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;
  - 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

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

First sort, then pack

- 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.

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: First Fit Decreasing (1/3)

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
```

Algorithm: First Fit Decreasing (2/3)

function *FirstFitDecreasing*(n, w, c) (continued)

```
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
```

Algorithm: First Fit Decreasing (3/3)

function *FirstFitDecreasing*(n, w, c) (continued)

```
    for k := 1 to n do (continued)
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
24 end
```

FFD and FF

- 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: FF and FFD on the BPP reference instance (1/4)

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 $FF(I) = 4$.

Example: FF and FFD on the BPP reference instance (2/4)

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; $rc[3] = 7 - 4 = 3$.

Example: FF and FFD on the BPP reference instance (3/4)

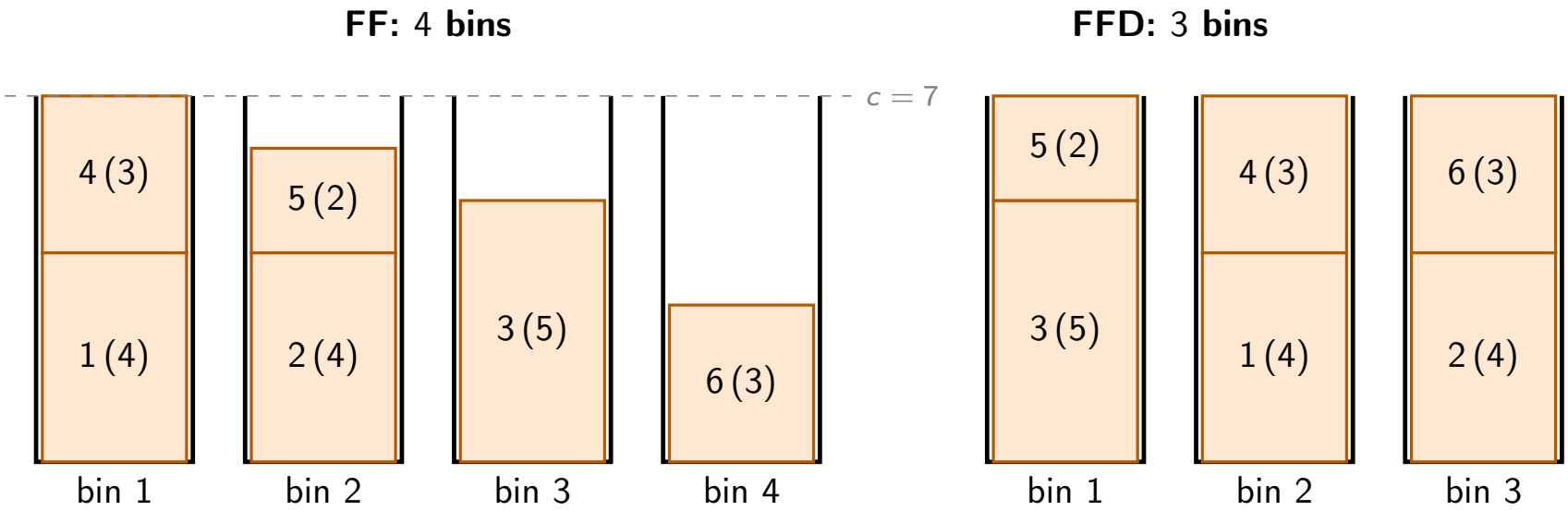
4. Item 4 (weight 3): $3 > rc[1] = 2$, $3 \leq rc[2] = 3$: bin 2; $rc[2] = 0$.
5. Item 6 (weight 3): $3 > rc[1] = 2$, $3 > rc[2] = 0$, $3 \leq rc[3] = 3$: bin 3; $rc[3] = 0$.
6. Item 5 (weight 2): $2 \leq rc[1] = 2$: bin 1; $rc[1] = 0$.

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

Example: FF and FFD on the BPP reference instance (4/4)



- 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.

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: Best Fit Decreasing (1/3)

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
```

Algorithm: Best Fit Decreasing (2/3)

function *BestFitDecreasing*(n, w, c) (continued)

```
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
```

Algorithm: Best Fit Decreasing (3/3)

```
function BestFitDecreasing(n, w, c) (continued)
    for k := 1 to n do (continued)
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
23 end
```

Again, the packing loop is that of BF,

preceded by the sorting.

Example: 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 $BFD(I) = 3$.

A better guarantee (1/2)

- 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 better guarantee (2/2)

- 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.

The next example shows an instance on which
FF finds the optimum and FFD does not.

Example: sorting can hurt (1/2)

Let $c = 7$ and $w = (2, 2, 3, 2, 2, 3)$, with total weight 14, so that

$$z(I) \geq \left\lceil \frac{14}{7} \right\rceil = 2.$$

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: $FFD(I) = 3$.

Example: sorting can hurt (2/2)

- 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.

Summary

First sort, then pack:

- offline, the order of the items can be chosen: the large items first, the small ones fill the gaps
- FFD and BFD: the loops of FF and BF, preceded by the sorting
- on the reference instance $FF(I) = 4$, $FFD(I) = BFD(I) = 3 = z(I)$

What sorting gives, and what it does not:

- a better worst case: $FFD(I) \leq \frac{11}{9} z(I) + \frac{6}{9}$, against the ratio $\frac{17}{10}$ of FF and BF
- not a better solution on every instance: FFD can be worse than FF

Scheduling on parallel machines: longest jobs first

Introduction: Scheduling on parallel machines: longest jobs first

- In this section we apply the same idea to $P \,||\, C_{\max}$: a rule chooses the machine, and the order of the jobs is a second choice
- We study list scheduling ($LIST$), which takes the jobs in any given order, and LPT, which takes the longest jobs first
- We compare their worst-case guarantees, due to Graham, and see that the bound of LPT is attained by the reference instance

The least loaded machine

- 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.

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.
- It assigns each job to a machine with the smallest current load (in case of ties, the machine with the smallest index).

Algorithm: List Scheduling (1/2)

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
```

Algorithm: List Scheduling (2/2)

function *ListScheduling*(n, m, t, ρ) (continued)

```
 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]
13 end
```

An online rule

- *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.

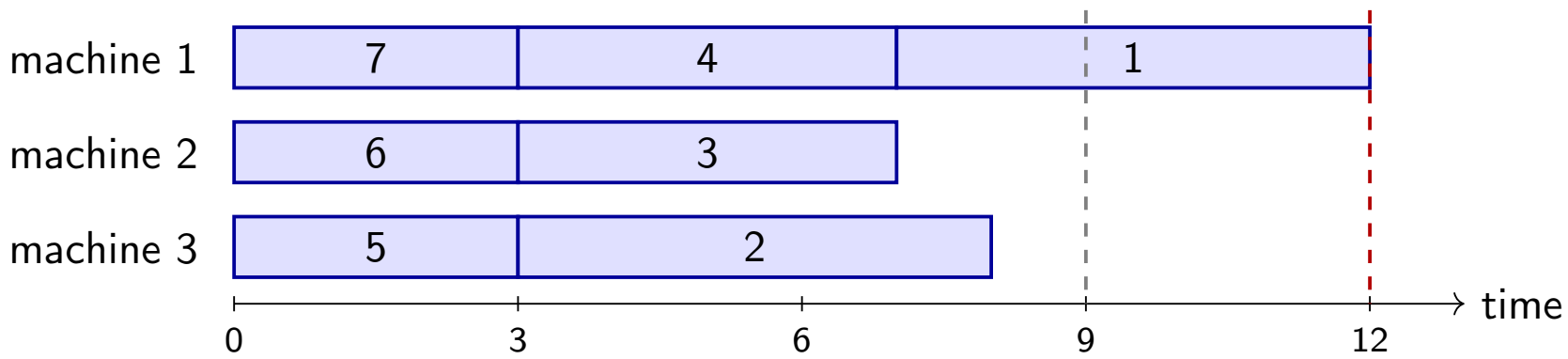
Example: *LIST* on the reference instance, shortest jobs first (1/2)

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

Example: *LIST* on the reference instance, shortest jobs first (2/2)

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).

The guarantee of *LIST*

How far from the optimum can *LIST* be, whatever the order of the list?

Proposition 1

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 an approximation algorithm

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

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

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: Longest Processing Time first

Input : $P \,||\, C_{\max}$ instance with n jobs, times $t[1..n]$, m machines

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

```
1 function LPT(n, m, t)
2 |   ρ := the jobs sorted by non-increasing t[j] (ties by
  |    index)                          // longest jobs first
3 |   return ListScheduling(n, m, t, ρ)
4 end
```

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.

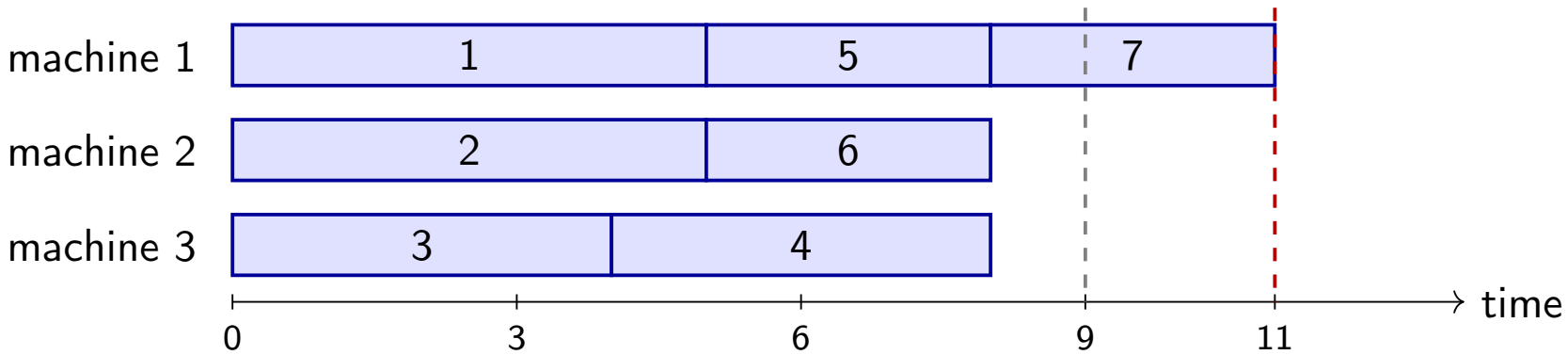
Example: *LPT* on the reference instance (1/3)

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: $(11, 8, 8)$.

Example: *LPT* on the reference instance (2/3)

Hence $J_1 = \{1, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{3, 4\}$, $mach = (1, 2, 3, 3, 1, 2, 1)$, and $LPT(I) = 11$:



Example: *LPT* on the reference instance (3/3)

- 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.

The guarantee of *LPT*

With the longest jobs first, the guarantee becomes:

Proposition 2

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 an approximation algorithm

LPT is a $\left(\frac{4}{3} - \frac{1}{3m}\right)$-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 bound of *LPT* is tight

- 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 the proposition above therefore cannot be improved.

Summary

List scheduling:

- each job of the list goes to a least loaded machine; an online rule, $O(nm)$ time
- $LIST(I) \leq \left(2 - \frac{1}{m}\right) z(I)$ for every list; on the reverse list $LIST(I) = 12$

Longest processing time first:

- $LIST$ on the jobs sorted by non-increasing processing time
- $LPT(I) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I)$; on the reference instance $LPT(I) = 11$
- the bound is tight: $\frac{11}{9}$ for $m = 3$, attained by the reference instance

Summary of the first part (1/2)

Constructive heuristics and the greedy paradigm:

- one feasible solution, built one element at a time from the empty partial solution
- a greedy algorithm adds the element that looks best now, and never revises a choice
- the order induced by the selection criterion decides everything

The 0/1 knapsack problem:

- three orders: profit, weight, efficiency; each can be arbitrarily bad
- combining solutions protects: Ext-Greedy guarantees $\frac{z(I)}{GR^e(I)} \leq 2$

Summary of the first part (2/2)

The bin packing problem: first sort, then pack:

- FFD and BFD: the large items first; $3 = z(I)$ bins on the reference instance, against 4 for FF
- $FFD(I) \leq \frac{11}{9} z(I) + \frac{6}{9}$, but sorting can hurt on some instances

Scheduling on parallel machines: longest jobs first:

- $LIST$: each job to a least loaded machine, $LIST(I) \leq \left(2 - \frac{1}{m}\right) z(I)$
- LPT: the longest jobs first, $LPT(I) \leq \left(\frac{4}{3} - \frac{1}{3m}\right) z(I)$, a tight bound

In the second part: the TSP, the VCP, randomized construction, regret and look-ahead.