Local search (2/2)

Outline (1/2)

$P \,||\, C_{\max}$: move and swap
- The move local search
- The swap local search

Flat landscapes and auxiliary objectives
- Plateaus
- The auxiliary objective
- The move local search for the BPP
- The swap local search for the BPP
- Properties of the auxiliary objective
- Examples
- Experiments on two medium instances

Outline (2/2)

KP: feasibility and penalties
- Staying feasible
- Penalizing the excess weight

Variable neighbourhood descent

Multi-start local search
- The limits of multi-start
- What local search gives, and what it does not

C++ implementation

Typical mistakes

In summary

Where we were

- A local search starts from a feasible solution and moves to a better neighbour until none exists: it stops in a local optimum with respect to its neighbourhood N.
- The neighbourhoods: flip and swap for the KP, move and swap for the BPP and $P \,||\, C_{\max}$, 2-opt and Or-opt for the TSP.
- First improvement and best improvement can stop in different local optima.
- The incremental evaluation computes the change Δ of a move in $O(1)$ for 2-opt and Or-opt.
- **Now**: $P \,||\, C_{\max}$, plateaus and auxiliary objectives, penalties for the KP, variable neighbourhood descent and multi-start.

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

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

- In this section we write the move and the swap local searches for $P \,||\, C_{\max}$
- Both moves transfer processing time between two machines: the change of the makespan is computed from the loads
- On the reference instance the solution of LPT is a local optimum for move, but not for swap

Transferring processing time

- 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

$$L_a - \delta \qquad \text{and} \qquad L_b + \delta,$$

and $F(L, a, b, \delta)$ denotes the resulting change of the objective.

The move local search

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

Algorithm: Move step for $P \,||\, C_{\max}$ (1/3)

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
```

Algorithm: Move step for $P \,||\, C_{\max}$ (2/3)

function *PCmaxMoveStep(n, m, t, mach, L, rule)* (continued)

```
    for j := 1 to n (continued) do
        for h := 1 to m (continued) do
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
```

Algorithm: Move step for $P \,||\, C_{\max}$ (3/3)

function *PCmaxMoveStep*$(n, m, t, mach, L, rule)$ (continued)

```
18    if j̄ = 0 then return false        // no improving move: local optimum
19    L[mach[j⃗]] ← L[mach[j⃗]] − t[j⃗]; L[h̄] ← L[h̄] + t[j⃗]
      // update the loads
20    mach[j⃗] ← h̄                       // apply the move
21    return true
22  end
```

The swap local search

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

Algorithm: Swap step for $P \,||\, C_{\max}$ (1/3)

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
```

Algorithm: Swap step for $P \,||\, C_{\max}$ (2/3)

function *PCmaxSwapStep*$(n, m, t, mach, L, rule)$ (continued)

```
    for j := 1 to n (continued) do
        for k := j + 1 to n (continued) do
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
```

Algorithm: Swap step for $P \,||\, C_{\max}$ (3/3)

function *PCmaxSwapStep*$(n, m, t, mach, L, rule)$ (continued)

```
16    if j̄ = 0 then return false        // no improving swap: local optimum
17    a := mach[j]; b := mach[k̄]; δ := t[j] − t[k̄]
18    L[a] ← L[a] − δ; L[b] ← L[b] + δ   // update the loads
19    mach[j] ← b; mach[k̄] ← a           // apply the swap
20    return true
21  end
```

The change of the makespan

For the makespan, the change of the objective is

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

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

Improving moves and critical machines

- 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.

Cost of an iteration, and move versus swap

- 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: the two neighbourhoods complement each other.

Example: the solution of *LPT* (1/6)

- The solution of *LPT* on the $P \,||\, C_{\max}$ reference instance has

$$J_1 = \{1, 5, 7\}, \qquad J_2 = \{2, 6\}, \qquad J_3 = \{3, 4\},$$

  loads $(11, 8, 8)$ and $C_{\max} = 11$.

- Machine 1 is the unique critical machine.

- We show that it is a local optimum for move, but not for swap.

Example: the solution of *LPT* (2/6)

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

Example: the solution of *LPT* (3/6)

- No move decreases the makespan:
  - moving job 1 away from machine 1 makes the receiving machine reach $8 + 5 = 13$;
  - 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 $C_{\max} = 11$.

Example: the solution of *LPT* (4/6)

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 $J_1 = \{3, 5, 7\}$, $J_2 = \{2, 6\}$, $J_3 = \{1, 4\}$.

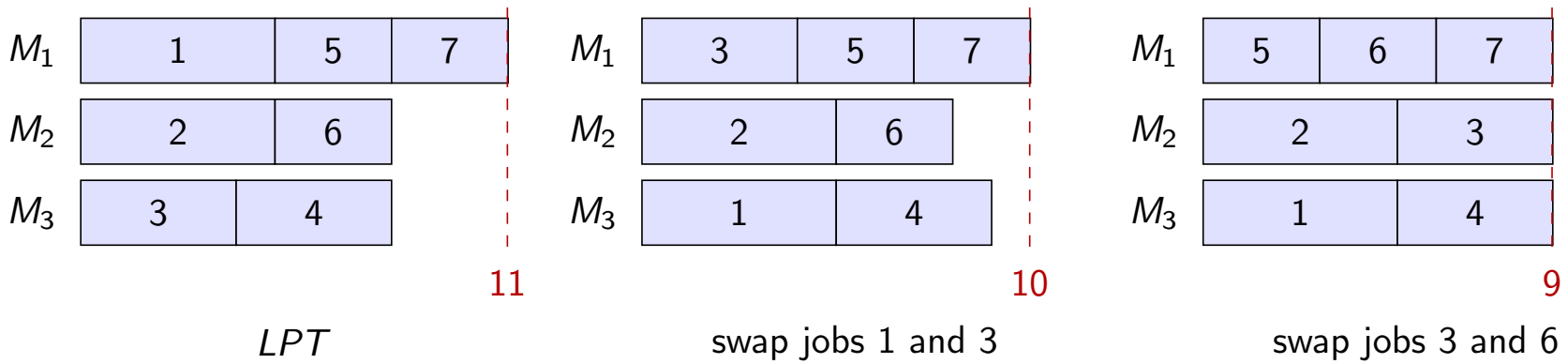
Example: the solution of *LPT* (5/6)

- 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

$$J_1 = \{5, 6, 7\}, \qquad J_2 = \{2, 3\}, \qquad J_3 = \{1, 4\}$$

has $C_{\max} = 9 = \lceil 27/3 \rceil$: it is **optimal**.

Example: the solution of *LPT* (6/6)



LPT swap jobs 1 and 3 swap jobs 3 and 6

- From *LPT* ($C_{\max} = 11$) two swaps reach the optimum $C_{\max} = 9$.

The local optimum depends on the neighbourhood

- 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

- Consider 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
$$J_1 = \{1, 8, 10\}, \quad J_2 = \{2, 7, 11\},$$
$$J_3 = \{3, 6, 12\}, \quad J_4 = \{4, 5, 9\},$$
with loads $(20, 19, 18, 19)$.
- This solution is a local optimum for both move and swap.

Summary

Move and swap for $P \,||\, C_{\max}$:

- Both transfer an amount δ between two machines; $F(L, a, b, \delta)$ is computed in $O(m)$
- The makespan can decrease only if a is the unique critical machine
- A move changes the number of jobs on the machines, a swap does not

On the reference instance:

- *LPT* (11) is a local optimum for move; two swaps reach the optimum 9
- The local optimum depends on the neighbourhood

Flat landscapes and auxiliary objectives

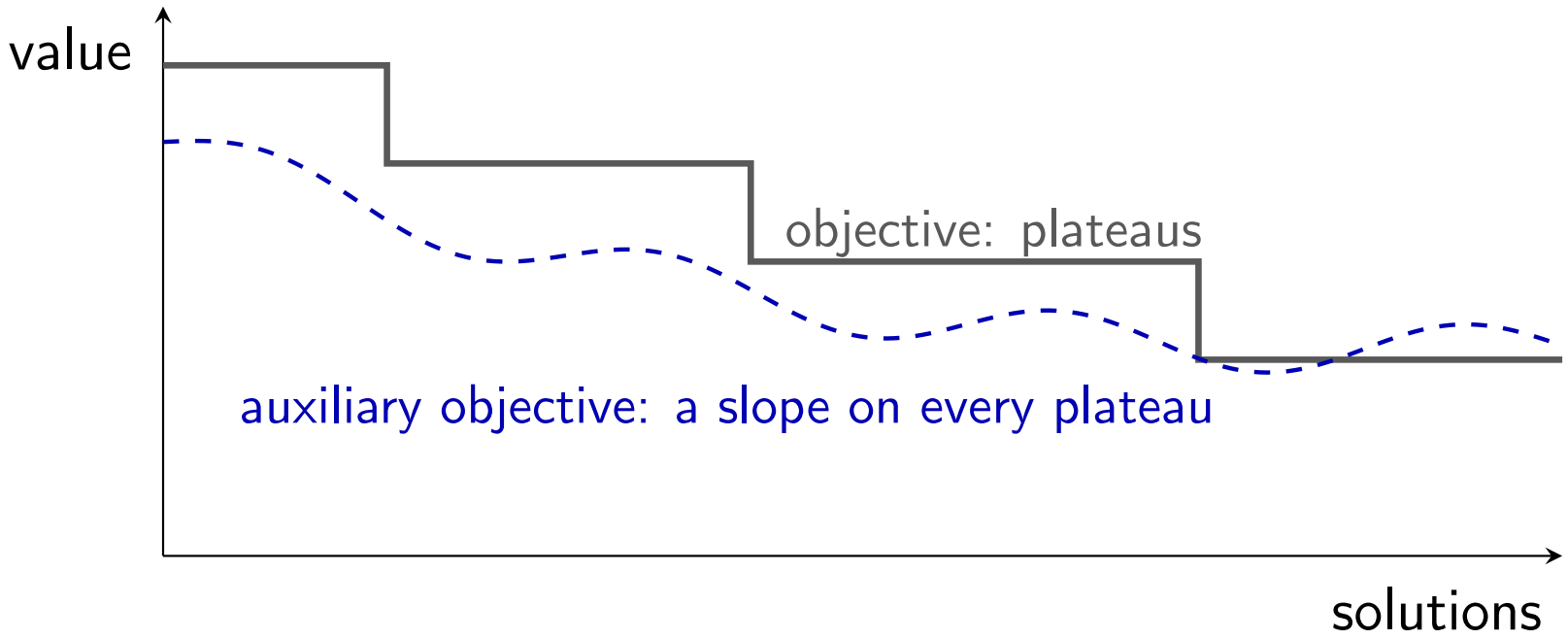
Introduction: Flat landscapes and auxiliary objectives

- The objectives of the BPP and of $P \,||\, C_{\max}$ take few values: most neighbours have the same value
- In this section we see why a local search stops on these plateaus, and how an auxiliary objective gives it a direction
- We write the move and the swap local searches for the BPP, and we test them on small and medium instances

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.

Plateaus in a picture



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.

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:

$$\text{BPP:} \quad \max \sum_b L_b^2 = \sum_b \left(\sum_{j \in P_b} w_j \right)^2,$$

$$P \,||\, C_{\max}: \quad \min \sum_{h=1}^{m} L_h^2.$$

Balanced and unbalanced loads

- 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 1

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

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

Proof

Expanding the squares,

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

the sum is

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

The other loads do not change. □

The auxiliary objective is cheap to evaluate

- 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.

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.
- It 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: Move step for the BPP (1/4)

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
```

Algorithm: Move step for the BPP (2/4)

function *BPPMoveStep*$(n, w, c, bin, L, f, rule)$ (continued)

```
    for j := 1 to n (continued) do
        for every bin b, in increasing order (continued) do
7           if f = SQ then
8               G := 2 w[j] (L[b] − L[a] + w[j])
                // Proposition 1
9           else
10              G := 1 if L[a] = w[j] else 0     // bin a is
                emptied
11          end
```

Algorithm: Move step for the BPP (3/4)

function *BPPMoveStep*$(n, w, c, bin, L, f, rule)$ (continued)

```
    for j := 1 to n (continued) do
        for every bin b, in increasing order (continued) do
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
```

Algorithm: Move step for the BPP (4/4)

function *BPPMoveStep*$(n, w, c, bin, L, f, rule)$ (continued)

```
20    if j̄ = 0 then return false        // no improving move: local optimum
21    L[bin[j]] ← L[bin[j]] − w[j]; L[b̄] ← L[b̄] + w[j]   // update the loads
22    bin[j] ← b̄                                  // apply the move
23    return true
24 end
```

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.
- It 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: Swap step for the BPP (1/3)

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
```

Algorithm: Swap step for the BPP (2/3)

function *BPPSwapStep*$(n, w, c, bin, L, rule)$ (continued)

```
    for j := 1 to n (continued) do
        for k := j + 1 to n (continued) do
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 1
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
```

Algorithm: Swap step for the BPP (3/3)

function *BPPSwapStep*$(n, w, c, bin, L, rule)$ (continued)

18 **if** $\bar{j} = 0$ **then return** *false* `// no improving swap: local optimum`

19 $a := bin[\vec{j}]$; $b := bin[\bar{k}]$; $\delta := w[\vec{j}] - w[\bar{k}]$

20 $L[a] \leftarrow L[a] - \delta$; $L[b] \leftarrow L[b] + \delta$ `// update the loads`

21 $bin[\vec{j}] \leftarrow b$; $bin[\bar{k}] \leftarrow a$ `// apply the swap`

22 **return** *true*

23 **end**

How do the auxiliary objectives relate
to the original objectives?

Two propositions, one for each problem.

Proposition 2

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 1 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

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 (1/2)

- 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 1, $\sum_h L_h^2$ decreases if and only if

$$L_b + \delta < L_a;$$

then both new loads $L_a - \delta$ and $L_b + \delta$ are smaller than $L_a \leq C_{\max}$, and the makespan does not increase.

Proof (2/2)

- (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.

□

What the two propositions say

- 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.

A guide, not the objective

- 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 $25 + 25 = 50$;
  - 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.

Example: the number of bins is blind (1/5)

- The solution of *FF* on the BPP reference instance ($c = 7$) has the packing patterns

$$P_1 = \{1, 4\}, \qquad P_2 = \{2, 5\}, \qquad P_3 = \{3\}, \qquad P_4 = \{6\},$$

with loads $(7, 6, 5, 3)$ and $nb = 4$.

- Its auxiliary objective is

$$SQ = 49 + 36 + 25 + 9 = 119.$$

- We show that the number of bins is blind, and the auxiliary objective is not.

Example: the number of bins is blind (2/5)

The feasible moves of the solution of *FF*:

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

Example: the number of bins is blind (3/5)

With $f = nb$:

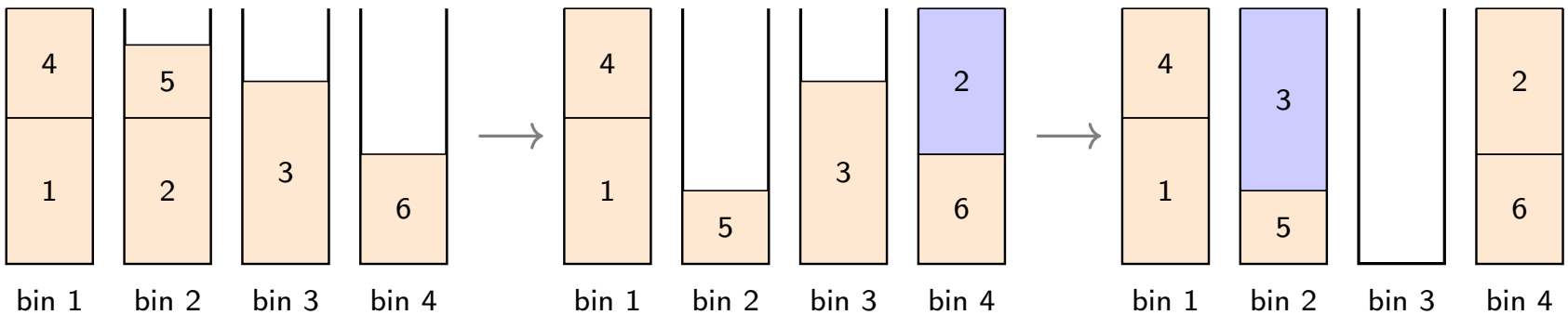
- No move empties a bin: items 3 and 6, alone in their bins, fit nowhere else.
- The solution of *FF* is a local optimum with 4 bins, and so it is for the swap neighbourhood, by Proposition 2.

Example: the number of bins is blind (4/5)

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.

Example: the number of bins is blind (5/5)



- Two moves with $f = SQ$: the first one does not change nb, the second one empties bin 3.

Example: starting from *NF* (1/2)

- 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.

Example: starting from *NF* (2/2)

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.

Experiments on two medium instances (1/4)

- 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

Experiments on two medium instances (2/4)

- 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.

Experiments on two medium instances (3/4)

- 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

Experiments on two medium instances (4/4)

- 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.

Summary (1/2)

Plateaus:

- The number of bins and the makespan take few values: most moves do not change them
- On a plateau the search is blind and stops

The auxiliary objective:

- BPP: $\max \sum_b L_b^2$; $P \,||\, C_{\max}$: $\min \sum_h L_h^2$
- Transferring δ from L_a to L_b changes it by $2\delta(L_b - L_a + \delta)$, in $O(1)$
- Every move that improves the original objective is improving for it, and so are some neutral moves

Summary (2/2)

The BPP:

- From FF: 4 bins with $f = nb$, the optimum 3 with $f = SQ$, through a move that prepares the next one
- From NF: 4 bins with $f = nb$, the optimum 3 with $f = SQ$

The experiments:

- $P \,||\, C_{\max}$: the auxiliary objective improves both move and swap
- BPP: with move, 14 optimal runs with $f = nb$, 76 with $f = \sum_b L_b^2$

KP: feasibility and penalties

Introduction: KP: feasibility and penalties

- A binary vector can exceed the capacity of the knapsack: it is not a feasible solution
- In this section we compare two ways of dealing with these vectors: staying feasible, or penalizing the excess weight
- A penalty lets the search walk through infeasible vectors, and reach solutions that a feasible search cannot reach

Two ways of dealing with infeasible vectors

- 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.

The penalized objective

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

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

where $\alpha > 0$ is the **penalty** for each unit of excess weight.

For a feasible x, $g_\alpha(x)$ is the profit of x.

Staying feasible: the flip+swap local search

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

Algorithm: Flip+swap step for the KP (1/4)

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
```

Algorithm: Flip+swap step for the KP (2/4)

function *KPFlipSwapStep*$(n, p, w, c, x, rule)$ (continued)

```
    for j := 1 to n (continued) do
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
```

Algorithm: Flip+swap step for the KP (3/4)

function *KPFlipSwapStep*$(n, p, w, c, x, rule)$ (continued)

```
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
```

Algorithm: Flip+swap step for the KP (4/4)

function *KPFlipSwapStep*$(n, p, w, c, x, rule)$ (continued)

```
24  |  if j̄ = 0 then return false      // no improving move: local optimum
25  |  x[j⃗] ← 1 − x[j⃗]   // 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
30  end
```

How the flip+swap step works

- 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.

Penalizing the excess weight: the penalty flip local search

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

Algorithm: Penalty flip step for the KP (1/2)

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

1 **function** *KPPenaltyFlipStep*$(n, p, w, c, \alpha, x, rule)$

2 $\quad P := \sum_{j=1}^{n} p[j]\,x[j]$; $W := \sum_{j=1}^{n} w[j]\,x[j]$ `// profit and weight of` x

3 $\quad g := P - \alpha \max\{0, W - c\}$ `// value of` x

4 $\quad \bar{\Delta} := 0$; $\bar{j} := 0$ `// best flip found so far`

5 $\quad$ **for** $j := 1$ **to** n **do**

6 $\quad\quad P' := P + (1 - 2x[j])\,p[j]$; $W' := W + (1 - 2x[j])\,w[j]$ `// flip of item` j

7 $\quad\quad \Delta := P' - \alpha \max\{0, W' - c\} - g$ `// change of` g_α

Algorithm: Penalty flip step for the KP (2/2)

function *KPPenaltyFlipStep*$(n, p, w, c, \alpha, x, rule)$ (continued)

```
    for j := 1 to n (continued) do
8       if Δ > Δ̄ then
9           Δ̄ ← Δ; j̄ ← j                    // improving flip
10          if rule = first then
11              leave the loop
12          end
13      end
14  end
15  if j̄ = 0 then return false      // no improving flip:
     local optimum
16  x[j̄] ← 1 − x[j̄]                          // apply the flip
17  return true
18 end
```

The best feasible solution visited

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

Example: penalties on the KP reference instance (1/4)

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

Example: penalties on the KP reference instance (2/4)

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.

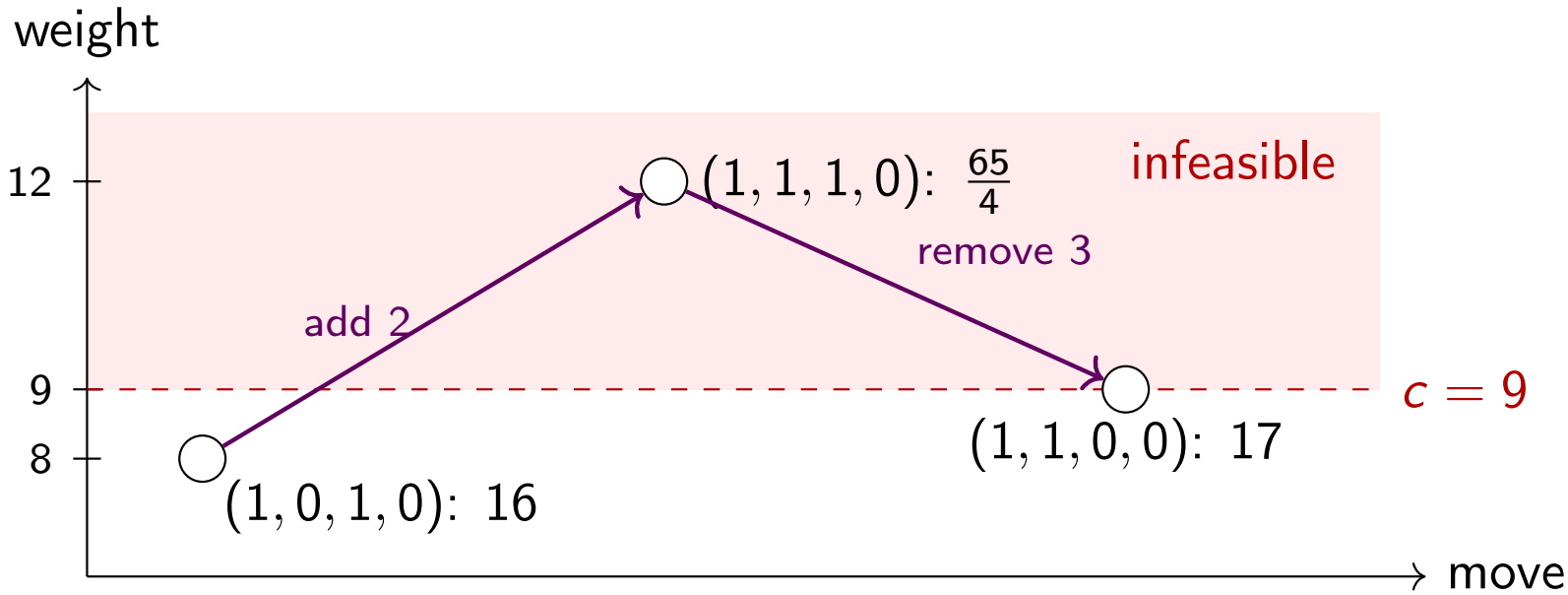
- $\alpha = 2$: the search adds item 2 ($g_2 = 17 > 16$) and stops in $(1, 1, 1, 0)$, infeasible: removing item 3 gives $(1, 1, 0, 0)$ with $g_2 = 17$, not strictly better.

Example: penalties on the KP reference instance (3/4)

- $\alpha = \frac{9}{4}$: 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 on the next slide).

- $\alpha = 3$: no flip improves, and the search stops at once in $(1, 0, 1, 0)$, as the feasible search.

Example: penalties on the KP reference instance (4/4)



- With $\alpha = \frac{9}{4}$ the search reaches the optimum through the infeasible vector $(1, 1, 1, 0)$.

How large should the penalty be?

With a large enough penalty,
the local optima are feasible.

Proposition 4

Let the data be integer and let $\alpha > \max_{j \in \mathscr{J}} p_j$. *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

$$-p_j + \alpha \min\{w_j, e\} \geq -p_j + \alpha > 0.$$

□

Large or small penalty

- 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).
- In practice α is adjusted during the search: increased while the search stays among infeasible vectors, decreased while it stays among feasible ones.

Summary

Staying feasible:

- Flip+swap: only feasible neighbours; the swap performs two flips at once
- The search can be short-sighted

Penalizing:

- $g_\alpha(x)$ = profit minus α times the excess weight; return the best feasible solution visited
- On the reference instance only $\alpha = \frac{9}{4}$ reaches the optimum 17, through $(1, 1, 1, 0)$
- With $\alpha > \max_j p_j$ the local optima are feasible; in practice α is adjusted

Variable neighbourhood descent

Introduction: Variable neighbourhood descent

- A local optimum for one neighbourhood is often not a local optimum for another
- In this section we combine several neighbourhoods in a single search, which stops only when none of them can improve
- We run it on $P \,||\, C_{\max}$ and on the TSP, and on the two medium instances

Several neighbourhoods

- 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** $N_1, \ldots, N_K$, and to stop only when none of them contains an improving neighbour.

Variable neighbourhood descent

- The *variable neighbourhood descent* (denoted by *VND*) explores the neighbourhoods $N_1, \ldots, N_K$ 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: Variable neighbourhood descent

Input : a feasible solution x, the steps $Step_1, \ldots, Step_K$ 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
11 end
```

VND stops in a local optimum for all the neighbourhoods

- *VND* stops after a finite number of moves, by the same argument as for the termination of *LS*: every move strictly improves the objective.
- When it stops, $Step_K$ has failed and, since the last improving move, all of $Step_1, \ldots, Step_{K-1}$ have failed on the same solution.
- The solution returned is a local optimum for every N_k, i.e., for the union $N_1 \cup \cdots \cup N_K$.

The order of the neighbourhoods

- 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: *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: *VND* on the TSP with 2-opt and Or-opt

- On the instance with 10 cities, 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 that 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.

VND on the two medium instances

On the two medium instances of the previous sections, 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

Two improvements that add up

- Compared with the single neighbourhoods, *VND* with the auxiliary objective reaches the optimum
  - in 68 runs out of 100 for $P \,||\, C_{\max}$ (move alone: 4);
  - in 89 for the BPP (move alone: 76).

- Combining neighbourhoods and using the auxiliary objective are independent improvements, and they add up.

Summary

Variable neighbourhood descent:

- Explore $N_1, \ldots, N_K$ in order; after an improving move go back to N_1
- It stops in a local optimum for the union $N_1 \cup \cdots \cup N_K$
- Order the neighbourhoods from the cheapest to the most expensive

Results:

- $P \,||\, C_{\max}$: from LPT to the optimum 9; TSP with 10 cities: from 49 to the optimum 39
- With the auxiliary objective: 68 and 89 optimal runs out of 100

Multi-start local search

Introduction: Multi-start local search

- A local search stops in the valley where it starts
- In this section we restart it from many starting solutions and keep the best local optimum
- We measure on the TSP how often the restarts reach the optimum, and we see the limits of this idea

Start again

- 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;
  - returns the best local optimum found.

Algorithm: Multi-start 2-opt (1/2)

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
```

Algorithm: Multi-start 2-opt (2/2)

function *MultiStartTwoOpt*$(n, d, M, s, rule)$ (continued)

    for $r := 1$ **to** M (continued) **do**
8   **if** $L(\sigma) < \bar{L}$ **then**
9           $\bar{L} \leftarrow L(\sigma)$; $\bar{\sigma} \leftarrow \sigma$ `// new best local optimum`
10   **end**
11 **end**
12 **return** $(\bar{\sigma}, \bar{L})$
13 **end**

Independent runs

- 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: *MSLS* on the TSP with 10 cities (1/2)

On the instance with 10 cities, 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	—

Example: *MSLS* on the TSP with 10 cities (2/2)

- 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.
- *VND* always finds the optimum, the only tour that is a local optimum for both neighbourhoods.

Example: *MSLS* on the TSP with 30 cities (1/4)

- The 30 cities have coordinates

$$(\lfloor 101 u_{2k-1} \rfloor, \lfloor 101 u_{2k} \rfloor), \qquad k = 1, \ldots, 30,$$

  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).

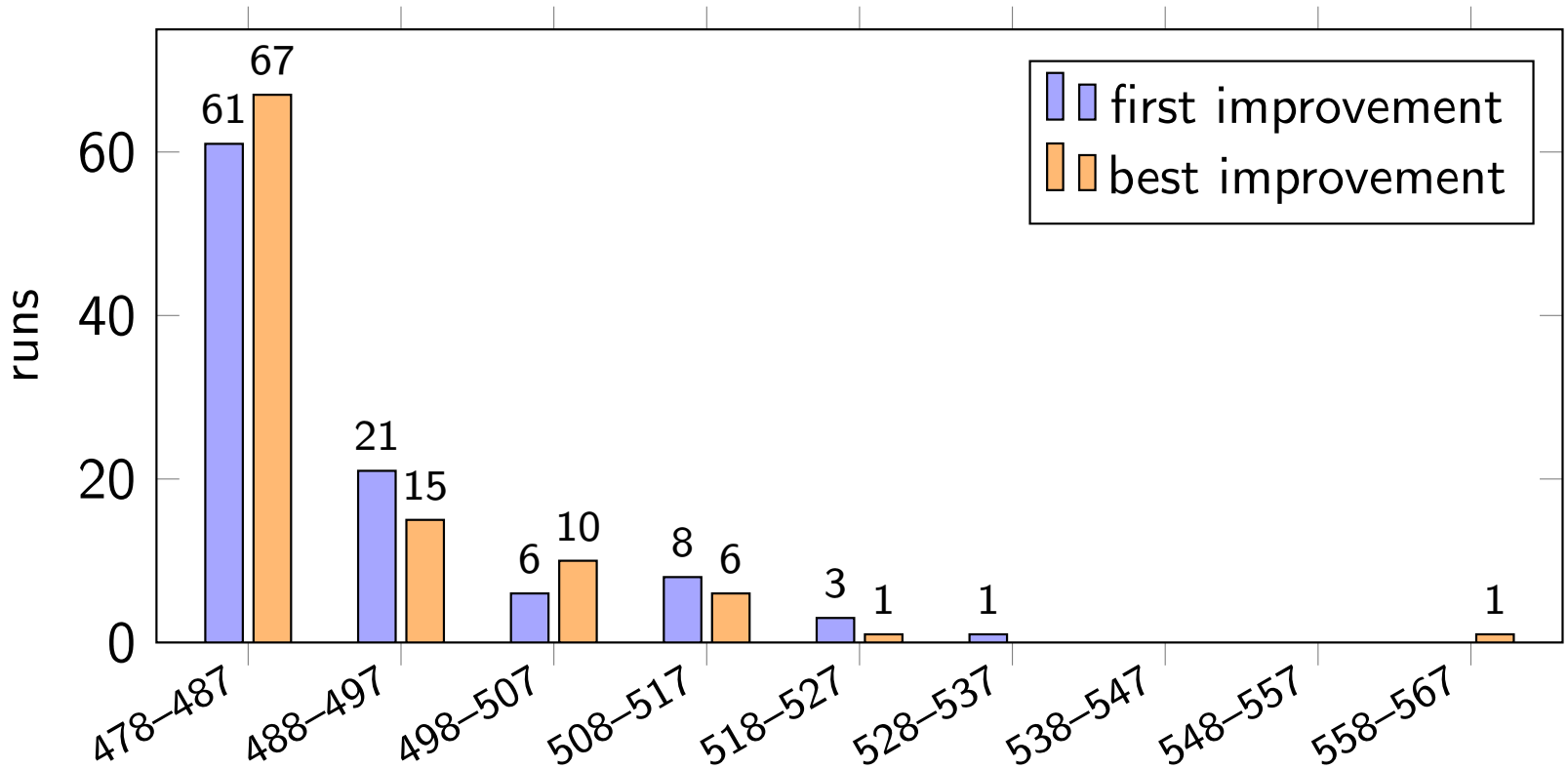
Example: *MSLS* on the TSP with 30 cities (2/4)

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

Example: *MSLS* on the TSP with 30 cities (3/4)

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



Example: *MSLS* on the TSP with 30 cities (4/4)

- 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.

The limits of multi-start (1/2)

- **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.

The limits of multi-start (2/2)

- **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.

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

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.

Summary

Multi-start:

- M independent local searches from random or randomized starting solutions; keep the best
- The simplest form of diversification

On the TSP:

- 10 cities: *VND* reaches the optimum in all the 100 runs
- 30 cities: within 2% of the optimum on average, but only about one run in eight is optimal

The limits:

- No memory, a shrinking probability q, time spent descending; no dual bound

C++ implementation

Introduction: C++ implementation

- The algorithms of the chapter are implemented in C++, one function per file
- We show only short extracts: the generic local search, the variable neighbourhood descent and how they are called
- The complete code is in the folder of the chapter and in its notebooks

The code

- 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 the notes and all the instances of the exercises, and compares the results with those computed by an independent engine.
- The code uses 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

The generic **local search** receives the step as a function without arguments, which acts on the current solution:

```
1 int LocalSearch(const std::function<bool()>& Step)
2 {
3     int moves = 0; // number of moves performed
4     while (Step()) // an improving neighbour was found and accepted
5     {
6         moves = moves + 1;
7     }
8     return moves; // the current solution is a local optimum
9 }
```

The variable neighbourhood descent

The heart of `VND`, which receives the steps `Steps[0]`, ..., `Steps[K-1]` of the K neighbourhoods:

```
1  while (k < K)
2  {
3      if (Steps[k]())  // improving neighbour found in N_k, and accepted
4      {
5          moves = moves + 1;
6          k = 0;  // back to the first neighbourhood
7      }
8      else
9      {
10         k = k + 1;  // local optimum for N_k: try the next neighbourhood
11     }
12 }
```

Calling the local searches

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

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

- The other files contain the steps of the chapter (`TwoOptStep`, `OrOptStep`, `PCmaxMoveStep`, `PCmaxSwapStep`, `BPPMoveStep`, `BPPSwapStep`, `KPFlipSwapStep`, `KPPenaltyFlipStep`), the multi-start 2-opt, the Park–Miller generator and the random tours.

Summary

The code:

- One function per file in `LOCAL_SEARCH_CODE/src/`; `make test` checks all the examples
- 0-based indexing, same algorithmic logic

The design:

- A step explores a neighbourhood and applies an improving move, returning `true`, or returns `false`
- `LocalSearch` and `VND` only call the steps

Typical mistakes

Introduction: Typical mistakes

- We collect the mistakes that are made most often with local search
- They concern the neighbourhood, the objective, the order of the moves, the 2-opt move and the meaning of a local optimum
- Check them before running a local search by hand

Typical mistakes (1/3)

Observation 1: Forgetting that a local optimum depends on the neighbourhood

The LPT solution is a local optimum for move, but not for swap.

Observation 2: 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.

Typical mistakes (2/3)

Observation 3: Ignoring the order of examination

The moves are examined in a fixed order and ties go to the first one examined (in the example of the solution of FF, item 3 comes before item 5).

Observation 4: 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.

Typical mistakes (3/3)

Observation 5: Concluding that a local optimum is optimal

Only a dual bound can say so.

Summary

Neighbourhood and objective:

- A local optimum depends on the neighbourhood
- The number of bins is blind: use the auxiliary objective

Moves:

- Fixed order of examination, ties to the first one
- 2-opt: only the two removed and the two added edges count

Optimality:

- A local optimum is not proved optimal without a dual bound

In summary

Introduction: In summary

- We collect the key ideas of the chapter on local search
- The dashboard gathers the results on the reference instances

Summary (1/2)

- 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.

Summary (2/2)

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, $= \mathrm{LB}(I)$)	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.