Metaheuristics (2/2)

Outline

- Variable neighbourhood search
  - Example
- GRASP
  - GRASP for the TSP, and the effect of α
- Large neighbourhood search
  - Large neighbourhood search for the BPP: heuristic and exact repair
  - Adaptive large neighbourhood search
- Population-based methods: genetic and memetic algorithms
- Comparison and conclusions
  - The methods on the medium instances
  - Intensification and diversification
  - Parameters and tuning
  - What these methods do not use
- Typical mistakes
- C++ implementation
- In summary

Where we were (1/2)

- A **metaheuristic** is a general strategy that guides a local search beyond the first local optimum, to explore other regions of the set of feasible solutions.
- The methods are illustrated on the KP, the BPP, the TSP and $P \,||\, C_{\max}$, with the local searches TwoOptLS, MoveDescent and KPLocalSearch of the chapter «Local search».
- The random numbers are produced by the minimal standard generator of Park and Miller: with $seed = 1$ the first ones are 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789.
- The reference instances have optimal values 17 (KP), 3 (BPP), 18 (TSP) and 9 ($P \,||\, C_{\max}$); the medium instances TSP20, KP20, PCM15 and BPP24 have optimal values 291, 521, 69 and 9.

Where we were (2/2)

- **Simulated annealing** accepts a worse random neighbour with probability $e^{-\delta/T}$, and lowers the temperature T gradually.
- **Tabu search** moves to the best non-tabu neighbour, even if it is worse, and forbids the moves that would undo the recent ones for *tenure* iterations.
- **Iterated local search** perturbs the current local optimum with a kick, descends again, and accepts the new local optimum if it is better.

This second part covers VNS, GRASP, large neighbourhood search, genetic and memetic algorithms,
and compares all the methods on the medium instances.

Variable neighbourhood search

Introduction: Variable neighbourhood search

- Variable neighbourhood search escapes from a local optimum by *changing the neighbourhood*: when the search is stuck, it samples a larger one
- We see its pseudo-code for $P \,||\, C_{\max}$, the role of the shaking, and the difference with the variable neighbourhood descent of the chapter «Local search»
- Then we run it by hand on the reference instance and over ten seeds on PCM15

The idea of variable neighbourhood search

- *Variable neighbourhood search* (denoted by *VNS*) uses a sequence of neighbourhoods $N_1, N_2, \ldots, N_{k_{\max}}$ of increasing size.
- From the current local optimum it draws a random solution in N_k (*shaking*), applies a local search to it, and
  - moves to the new local optimum if it is better, restarting from $k = 1$;
  - otherwise it tries the next, larger neighbourhood N_{k+1}.

Next: the pseudo-code of *VNS* for $P \,||\, C_{\max}$, where N_k contains the assignments obtained with k moves and the local search is MoveDescent.

Algorithm: Variable neighbourhood search for $P \,||\, C_{\max}$ (1/3)

Input : times $t[1..n]$, m machines, starting assignment $mach[1..n]$, $k_{\max}$, $maxIter$

Output : an assignment $mach$ and its makespan C

1 **function** *VNS*$(t, m, mach, k_{\max}, maxIter)$

2 $\quad$ LB $:= \max\{\max_j t[j],\ \lceil \sum_j t[j]/m \rceil\}$

3 $\quad$ $mach :=$ MoveDescent$(mach)$; $L :=$ loads of $mach$; $C := \max_h L[h]$

4 $\quad$ $it := 0$; $k := 1$

Algorithm: Variable neighbourhood search for $P \,||\, C_{\max}$ (2/3)

function *VNS*$(t, m, mach, k_{\max}, maxIter)$ (continued)

```
5   while it < maxIter and C > LB do
6       it := it + 1
7       nw := Shake(mach, k)        // random solution in N_k
8       nw := MoveDescent(nw); Lnew := loads of nw
            // local search
9       if Lnew is lexicographically better than L then
10          mach := nw; L := Lnew; k := 1; C := max_h L[h]
                // move
11      else
12          if k < k_max then k := k + 1 else k := 1
13      end
14  end
15  return mach, C
16 end
```

Algorithm: Variable neighbourhood search for $P \,||\, C_{\max}$ (3/3)

```
17 function Shake(mach, k)
18     for s := 1 to k do
19         j := 1 + ⌊RAND() · n⌋;  g := 1 + ⌊RAND() · (m − 1)⌋
20         mach[j] := the g-th machine different from mach[j]
21     end
22     return mach
23 end
```

Three observations

- *VNS* was proposed by Mladenović and Hansen (1997). It rests on three observations:
  - a local optimum for one neighbourhood need not be a local optimum for another;
  - a global optimum is a local optimum for every neighbourhood;
  - and the local optima of different neighbourhoods are often close to each other.

- **Shaking** diversifies, and the size k controls how much: small k explores the solutions close to the current local optimum, larger k goes farther only when the closer ones have failed.

- The **local search** intensifies. Returning to $k = 1$ after every improvement concentrates the search near the new, better solution.

VND and VNS

- The variable neighbourhood descent (*VND*) of the chapter «Local search» also uses several neighbourhoods, but *deterministically*: it explores them completely, one after the other, and stops at a solution that is a local optimum for all of them.
- *VNS* explores the large neighbourhoods only by *random samples*, which is what makes very large neighbourhoods usable; the local search inside *VNS* can itself be a *VND*.
- Here N_k contains the assignments obtained by moving k jobs (or fewer, if the same job is drawn twice).
- Since a swap of two jobs is the combination of two moves, N_2 contains the swap neighbourhood: shaking with $k = 2$ can reproduce the swaps that the move neighbourhood cannot see.

Example: *VNS* on the $P \,||\, C_{\max}$ reference instance (1/3)

- We start from the LPT solution, with loads $(11, 8, 8)$, and use $k_{\max} = 3$ and $seed = 1$ (random numbers 0.0000, 0.1315, 0.7556, 0.4587, 0.5328, 0.2190, 0.0470, 0.6789).
- MoveDescent does not change the LPT solution, which is a local optimum for the move neighbourhood (as seen in the example of tabu search).
- Each random move uses two numbers: $j = 1 + \lfloor 7u \rfloor$ and $g = 1 + \lfloor 2u \rfloor$.

Example: *VNS* on the $P \,||\, C_{\max}$ reference instance (2/3)

- **Iteration 1**, $k = 1$: job $j = 1 + \lfloor 7 \cdot 0.0000 \rfloor = 1$ moves from M_1 to the first other machine ($g = 1 + \lfloor 2 \cdot 0.1315 \rfloor = 1$), M_2: loads $(6, 13, 8)$.
  MoveDescent moves job 6 from M_2 to M_1: loads $(9, 10, 8)$, $C_{\max} = 10$, better than 11. The new solution $J_1 = \{5, 6, 7\}$, $J_2 = \{1, 2\}$, $J_3 = \{3, 4\}$ is accepted, and $k = 1$.

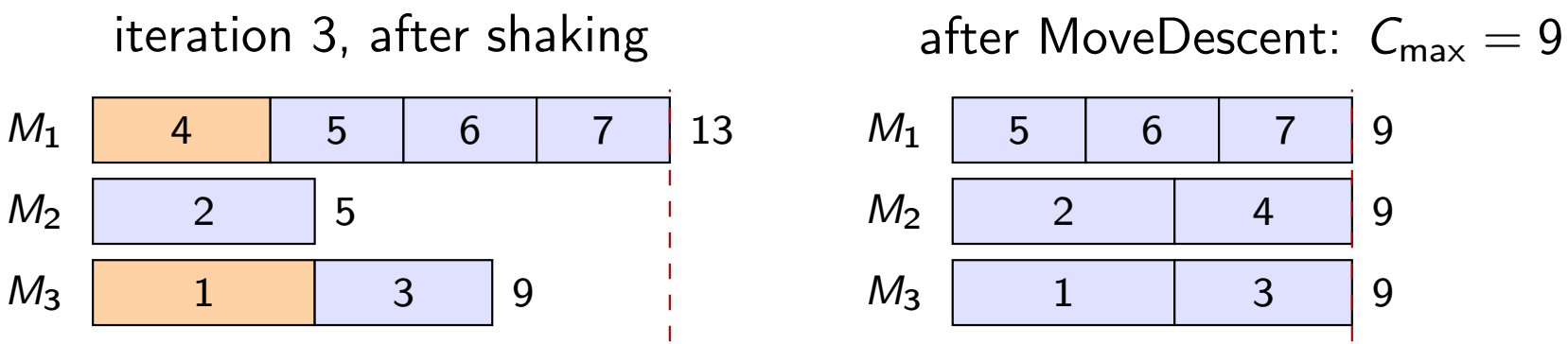
- **Iteration 2**, $k = 1$: job $j = 1 + \lfloor 7 \cdot 0.7556 \rfloor = 6$ moves from M_1 to M_2 ($g = 1 + \lfloor 2 \cdot 0.4587 \rfloor = 1$): loads $(6, 13, 8)$.
  MoveDescent moves job 6 back to M_1 and returns the current solution, which is not better: $k = 2$.

Example: *VNS* on the $P \,||\, C_{\max}$ reference instance (3/3)

- **Iteration 3**, $k = 2$: job $j = 1 + \lfloor 7 \cdot 0.5328 \rfloor = 4$ moves from M_3 to the first other machine M_1 ($g = 1 + \lfloor 2 \cdot 0.2190 \rfloor = 1$); then job $j = 1 + \lfloor 7 \cdot 0.0470 \rfloor = 1$ moves from M_2 to the second other machine M_3 ($g = 1 + \lfloor 2 \cdot 0.6789 \rfloor = 2$): loads $(13, 5, 9)$. MoveDescent moves job 4 from M_1 to M_2: loads $(9, 9, 9)$. The solution is accepted, and since $C_{\max} = 9 = \mathrm{LB}$ the search stops: $VNS(I) = 9 = z(I)$.

- In terms of the current solution of iteration 3, the two random moves and the descent have exchanged jobs 1 and 4: a swap, which the move neighbourhood of MoveDescent cannot perform.

Iteration 3 of *VNS*



- The two jobs moved by the shaking are in orange: job 4 on M_1 and job 1 on M_3.
- MoveDescent then moves job 4 to M_2, and the three machines have load 9.

VNS on PCM15 over ten seeds

- On PCM15, from the LPT solution (73) and with at most 300 iterations, *VNS* gives the following makespans for $seed = 1, \ldots, 10$, and the number of iterations performed:

seed	1	2	3	4	5	6	7	8	9	10
$k_{max} = 1$: C_{max}	69	70	70	70	69	69	69	69	70	69
iterations	27	300	300	300	48	59	20	43	300	13
$k_{max} = 2$: C_{max}	70	69	69	69	69	70	69	70	70	69
iterations	300	41	126	285	33	300	31	300	300	26
$k_{max} = 3$: C_{max}	70	69	69	69	69	69	69	70	70	69
iterations	300	32	38	29	26	11	25	300	300	25

What the seeds show

- With $k_{\max} = 3$, *VNS* reaches the optimum 69 in 7 runs out of 10, usually in a few tens of iterations; the best tabu search on PCM15 needed a carefully tuned tenure.
- When *VNS* fails, the search is stuck at 70: the shaking explores at most three moves away, and the optimum is farther.
- Larger $k_{\max}$, or restarts, can be added at the price of more iterations.

Summary

Variable neighbourhood search:

- shaking in N_k, local search, move if better and return to $k = 1$, otherwise $k + 1$
- shaking diversifies, the local search intensifies
- unlike *VND*, it samples the large neighbourhoods at random

On $P \,||\, C_{\max}$:

- N_2 contains the swaps that the move neighbourhood cannot see
- reference instance: optimum in three iterations; PCM15 with $k_{\max} = 3$: optimum in 7 runs out of 10

GRASP

Introduction: GRASP

- GRASP escapes from a local optimum by *restarting* from a new solution, built by a randomized greedy algorithm
- We see its pseudo-code for the KP, the restricted candidate list and the role of the parameter α
- Then we run two iterations by hand on the KP, and we measure the effect of α on TSP20 and KP20

The idea of GRASP

- The *greedy randomized adaptive search procedure* (denoted by *GRASP*) is a multi-start method.
- At every iteration it builds a solution with a *randomized* greedy algorithm, which chooses the next element at random among the best candidates (the *restricted candidate list*, RCL) instead of always taking the best one,
- and improves the solution with a local search.
- The best local optimum found is returned.

Next: the pseudo-code of *GRASP* for the KP, with the randomized greedy algorithm by efficiency and KPLocalSearch.

Algorithm: GRASP for the KP (1/2)

Input : KP instance $p[1..n]$, $w[1..n]$, c; parameter $\alpha \in [0,1]$; number of iterations *maxIter*

Output : best solution found

```
1 function GRASP(p, w, c, α, maxIter)
2     zbest := −1
3     for it := 1 to maxIter do
4         x := RandomizedGreedy(p, w, c, α)  // construction
5         x := KPLocalSearch(x)              // local search
6         if Σ_j p[j]x[j] > zbest then best := x;  zbest := Σ_j p[j]x[j]
7     end
8     return best
9 end
```

Algorithm: GRASP for the KP (2/2)

```
10 function RandomizedGreedy(p, w, c, α)
11     x := (0, ..., 0);  rc := c      // residual capacity
12     while some item with x[j] = 0 has w[j] ≤ rc do
13         e_max, e_min := maximum and minimum of
            p[j]/w[j] over these candidate items
14         RCL := candidates with
            p[j]/w[j] ≥ e_max − α (e_max − e_min), by
            non-increasing p[j]/w[j]
15         j := the element of RCL in position
            1 + ⌊RAND() · |RCL|⌋
16         x[j] := 1;  rc := rc − w[j]
17     end
18     return x
19 end
```

The restricted candidate list

- *GRASP* was introduced by Feo and Resende (1989) for the set covering problem and presented as a general metaheuristic in Feo and Resende (1995). The randomized construction with a restricted candidate list is the one of the chapter «Constructive heuristics».
- The RCL is defined by a threshold on the greedy score, controlled by α.

KP (maximize the efficiency p_j/w_j of the items that fit):

$$RCL = \left\{ j \text{ candidate} : \frac{p_j}{w_j} \geq e_{\max} - \alpha\,(e_{\max} - e_{\min}) \right\}.$$

TSP (minimize the distance from the last city ℓ of the partial tour):

$$RCL = \left\{ j \text{ unvisited} : d_{\ell j} \leq d_{\min} + \alpha\,(d_{\max} - d_{\min}) \right\}.$$

The parameter α

- With $\alpha = 0$ the RCL contains only the best candidates, and the construction is the greedy algorithm: every iteration builds the same solution (up to ties), and *GRASP* is a single local search.
- With $\alpha = 1$ the RCL contains all the candidates, and the construction is random.
- Intermediate values give solutions that are **different** (diversification) and **good** (a good start for the local search, which then intensifies).
- The iterations of *GRASP* are **independent**: nothing is learnt from one to the next, and the iterations can be run in parallel.
- Extensions add memory: *reactive* GRASP adapts α to the results, and *path relinking* explores the solutions between two good local optima.

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

- With $\alpha = \frac{1}{2}$ and $seed = 2$ (random numbers 0.0000, 0.2631, 0.5112, 0.9173) we perform two iterations.
- The efficiencies are $\frac{10}{5} = 2$, $\frac{7}{4} = 1.75$, $\frac{6}{3} = 2$, $\frac{4}{3} \approx 1.33$.

Iteration 1.

- With $rc = 9$ all items fit: $e_{\max} = 2$, $e_{\min} = \frac{4}{3}$, the threshold is $2 - \frac{1}{2} \cdot \frac{2}{3} = \frac{5}{3} \approx 1.67$, and $RCL = (1, 3, 2)$.
- With $u = 0.0000$ the first element is chosen: item 1, $rc = 4$.
- The candidates are items 2, 3, 4, the threshold is again $\frac{5}{3}$, $RCL = (3, 2)$, and $u = 0.2631$ chooses position $1 + \lfloor 2 \cdot 0.2631 \rfloor = 1$: item 3, $rc = 1$, and no item fits.
- The construction gives $\{1, 3\}$, of profit 16; KPLocalSearch swaps item 3 out and item 2 in: $\{1, 2\}$, profit 17.

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

Iteration 2.

- $RCL = (1, 3, 2)$ and $u = 0.5112$ chooses position $1 + \lfloor 3 \cdot 0.5112 \rfloor = 2$: item 3, $rc = 6$.
- The candidates are items 1, 2, 4, the threshold is $\frac{5}{3}$, $RCL = (1, 2)$, and $u = 0.9173$ chooses position 2: item 2, $rc = 2$.
- The construction gives $\{2, 3\}$, of profit 13, a *different* and worse solution than the greedy one.
- The local search swaps item 3 out and item 1 in: $\{1, 2\}$, profit 17.

Example: *GRASP* on the KP reference instance (3/3)

$$GRASP(I) = 17 = z(I).$$

- On this small instance every start leads to the optimum.
- On larger instances different starts lead to different local optima.

GRASP for the TSP

- For the TSP the randomized construction is the nearest neighbour heuristic with an RCL: from the last city ℓ of the partial tour, the next city is chosen at random among the unvisited cities whose distance from ℓ is at most $d_{\text{min}} + \alpha(d_{\text{max}} - d_{\text{min}})$, and the local search is $\textsc{TwoOptLS}$.

- The table on the next slide reports the results of *GRASP* with 20 iterations on TSP20 and on KP20, for 10 runs ($seed = 1, \ldots, 10$):
  - minimum, average and maximum value of the best solution,
  - and number of runs that reach the optimum.

The effect of α

α	TSP20 ($z(I) = 291$) min	average	max	optimal	KP20 ($z(I) = 521$) min	average	max	optimal
0	319	319.0	319	0	514	514.0	514	0
0.1	291	291.0	291	10	521	521.0	521	10
0.25	291	291.0	291	10	521	521.0	521	10
0.5	291	291.0	291	10	521	521.0	521	10
1	291	291.0	291	10	519	520.8	521	9

- With $\alpha = 0$ all the iterations are the same: the result is the local optimum reached from the greedy solution (319 for TSP20, 514 for KP20), in every run.
- A little randomization is enough to reach the optimum in every run on these instances; with $\alpha = 1$ the constructions are random, and on KP20 one run misses the optimum.

Summary

GRASP:

- multi-start: randomized greedy construction, then local search; the best local optimum is returned
- the RCL contains the candidates within a threshold set by α
- $\alpha = 0$: greedy, a single local search; $\alpha = 1$: random construction

Results:

- the iterations are independent and can be run in parallel
- with $0 < \alpha < 1$, *GRASP* reaches the optimum of TSP20 and KP20 in every run

Large neighbourhood search

Introduction: Large neighbourhood search

- Large neighbourhood search escapes from a local optimum by *destroying* a large part of the solution and *repairing* it
- We see its pseudo-code for the TSP and for the BPP, with a heuristic or an exact repair
- Then we see its adaptive version, which learns which destroy method works best

The idea of large neighbourhood search

- *Large neighbourhood search* (denoted by *LNS*) improves a solution by *destroying* a part of it and *repairing* it:
  - a destroy method removes some elements (items, cities, jobs) from the current solution,
  - a repair method reinserts them,
  - and the new solution replaces the current one if an acceptance criterion is satisfied.
- The neighbourhood of a solution is the set of all the solutions that destroy and repair can produce, and it is *large*.

Next: the pseudo-code of *LNS* for the TSP, which removes q random cities and reinserts them by cheapest insertion.

Algorithm: Large neighbourhood search for the TSP (1/2)

Input : distances d, starting tour σ, number q of cities to remove, number of iterations *maxIter*

Output : the current tour at the end

```
1  function LNS(d, σ, q, maxIter)
2      cur := σ;  lcur := L(cur)
3      for it := 1 to maxIter do
4          nw := cur
5          removed := RandomRemoval(nw, q)          // destroy
6          CheapestInsertionRepair(nw, removed)     // repair
7          if L(nw) < lcur then cur := nw;  lcur := L(nw)
               // acceptance
8      end
9      return cur
10 end
```

Algorithm: Large neighbourhood search for the TSP (2/2)

```
11 function RandomRemoval(σ, q)
12     for s := 1 to q do
13         k := 2 + ⌊RAND() · (|σ| − 1)⌋  // city 1 is never
             removed
14         append σ[k] to removed, and delete it from σ
15     end
16     return removed
17 end
18 function CheapestInsertionRepair(σ, removed)
19     foreach city v of removed, in the order of removal do
20         insert v between the consecutive cities a, b of σ of
             minimum d_av + d_vb − d_ab
21     end
22 end
```

A large neighbourhood, explored implicitly

- *LNS* was introduced by Shaw (1998) for vehicle routing problems, where the repair was done by constraint programming.
- The **size** of the neighbourhood grows very quickly with the number q of removed elements: for the TSP there are $\binom{n-1}{q}$ ways to choose the removed cities, and many ways to reinsert them.
- Such a neighbourhood cannot be listed and evaluated move by move, as in a local search: it is explored **implicitly**, by the repair method.

The repair method

- The repair method is a **sub-algorithm**: a constructive heuristic (cheapest insertion, FFD), or an *exact* algorithm for the subproblem left by the destroy method.
- The stronger the repair, the better the best neighbour found; the larger q, the larger the neighbourhood, and the more expensive its exploration.

Destroy and repair

- The destroy method **diversifies**: it is usually random, and q controls how far the search goes.
- The repair method **intensifies**: it rebuilds the destroyed part in the best way it can.
- Removing *related* elements (cities close to each other, items of the same bins) makes the repair more effective than removing unrelated ones, which can only be put back where they were.

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

- From the nearest neighbour tour $(1, 3, 2, 4, 5)$ of length 25, with $q = 2$ and $seed = 1$.
- The first random number 0.0000 removes the city in position $2 + \lfloor 4 \cdot 0.0000 \rfloor = 2$, city 3; the partial tour is $(1, 2, 4, 5)$.
- 0.1315 removes the city in position $2 + \lfloor 3 \cdot 0.1315 \rfloor = 2$, city 2.
- The partial tour $(1, 4, 5)$ has length $2 + 7 + 9 = 18$.

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

Cheapest insertion of city 3:

$$
\begin{aligned}
(1,4): &\ d_{13} + d_{34} - d_{14} = 2 + 4 - 2 = 4, \\
(4,5): &\ 4 + 7 - 7 = 4, \\
(5,1): &\ 7 + 2 - 9 = 0,
\end{aligned}
$$

so city 3 goes between 5 and 1: $(1, 4, 5, 3)$.

Example: *LNS* on the TSP reference instance (3/3)

Cheapest insertion of city 2:

$$(1,4): \ 5+3-2=6, \qquad (4,5): \ 3+4-7=0,$$
$$(5,3): \ 4+4-7=1, \qquad (3,1): \ 4+5-2=7,$$

so city 2 goes between 4 and 5: $(1,4,2,5,3)$, of length 18, accepted.

- It is the optimal tour, traversed in the opposite direction: one destroy-and-repair step has changed four edges of the tour.

Large neighbourhood search for the BPP

- For the BPP the natural destroy method **empties bins**: the least loaded bin, which is the one we would like to get rid of, and $q - 1$ other bins chosen at random.
- The repair method repacks the removed items into new bins, while the other bins are kept as they are.
- The new packing is accepted if it uses fewer bins, or the same number of bins and a value of the auxiliary objective $\sum_b (\sum_{j \in P_b} w_j)^2$ not smaller: on the plateaus of the flat objective, the search moves toward fuller bins.

Large neighbourhood search for the BPP: the repair

- The repair solves a **BPP subproblem**, on the removed items only, and can do it in two ways:
  - with FFD,
  - or **exactly**, by a branch-and-bound that tries every bin for every item and keeps the packing with the fewest bins.
- The subproblem has only the items of q bins, so an exact repair is affordable.

Algorithm: Large neighbourhood search for the BPP (1/2)

Input : weights $w[1..n]$, capacity c, packing $bin[1..n]$ with nb bins, q, *exact*, *maxIter*

Output : a packing bin and its number of bins nb

```
1 function LNS-BPP(w, c, bin, nb, q, exact, maxIter)
2 |   LB := ⌈∑_j w[j]/c⌉;  it := 0
3 |   while it < maxIter and nb > LB do
4 |   |   it := it + 1
5 |   |   empty := the least loaded bin and q − 1 other bins at
  |   |    random                                          // destroy
6 |   |   items := the items packed in the bins of empty
7 |   |   newbin := the other bins, renumbered 1, ..., nb − q
```

Algorithm: Large neighbourhood search for the BPP (2/2)

function *LNS-BPP*$(w, c, bin, nb, q, exact, maxIter)$
(continued)

```
    while it < maxIter and nb > LB (continued) do
8       if exact then pack items optimally in k new bins else
          pack items by FFD in k new bins                    // repair
9       nb' := nb − q + k
10      if nb' < nb or (nb' = nb and
          ∑_b(load of b in newbin)^2 ≥ ∑_b(load of b in bin)^2)
          then
11          bin := newbin;  nb := nb'                        // acceptance
12      end
13  end
14  return bin, nb
15 end
```

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

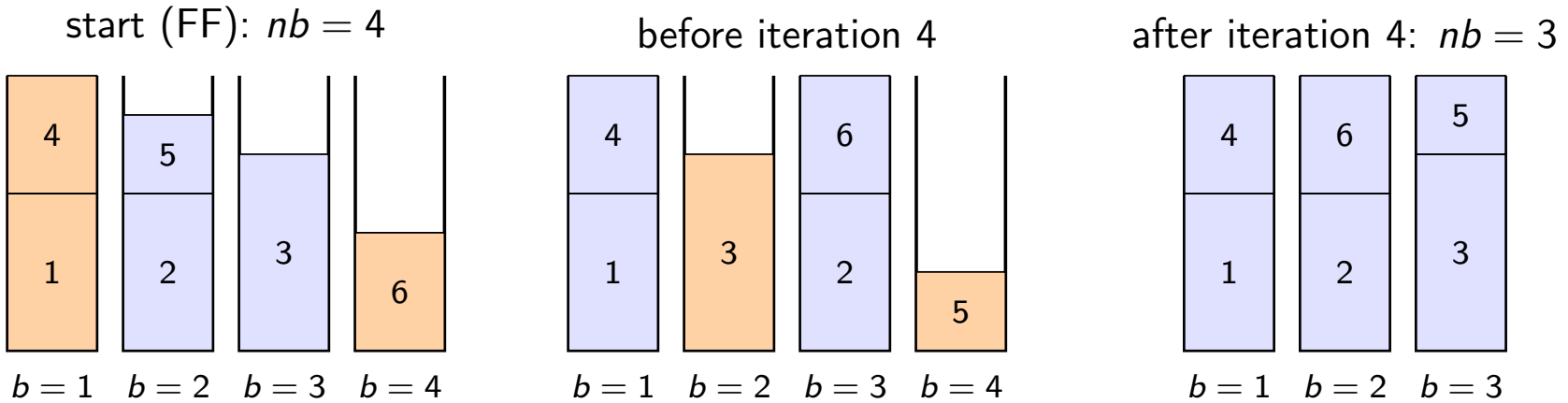
- We start from the packing computed by First Fit, $P_1 = \{1, 4\}$, $P_2 = \{2, 5\}$, $P_3 = \{3\}$, $P_4 = \{6\}$, with loads $(7, 6, 5, 3)$.
- We use $q = 2$, repair by FFD and $seed = 3$ (random numbers 0.0000, 0.3946, 0.2668, 0.3760).
- At every iteration the least loaded bin is emptied, together with the bin in position $1 + \lfloor 3u \rfloor$ among the other three.

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

it	loads	emptied	items	FFD	nb'	$\sum_b(\cdot)^2$
1	$(7,6,5,3)$	P_4, P_1	$\{1,4,6\}$	$\{1,4\},\{6\}$	4	$119 \to 119$
2	$(6,5,7,3)$	P_4, P_2	$\{3,6\}$	$\{3\},\{6\}$	4	$119 \to 119$
3	$(6,7,5,3)$	P_4, P_1	$\{2,5,6\}$	$\{2,6\},\{5\}$	4	$119 \to 127$
4	$(7,5,7,2)$	P_4, P_2	$\{3,5\}$	$\{3,5\}$	3	$127 \to 147$

- Iterations 1 and 2 only reshuffle the items and keep the same loads (the auxiliary objective stays 119, and the new packing is accepted because it is not worse).
- Iteration 3 builds the full bin $\{2,6\}$, raising the auxiliary objective to 127.
- Iteration 4 empties the bins $\{5\}$ and $\{3\}$, and FFD packs items 3 and 5 together: 3 bins, the optimum, and the search stops since $nb = \mathrm{LB}$.

Example: *LNS* on the BPP reference instance (3/3)



- The packings at the start (the bins emptied at iteration 1 in orange), before iteration 4 (the bins it empties in orange) and after iteration 4.

LNS on BPP24: heuristic or exact repair

- On BPP24, starting from the FFD packing (10 bins), with 200 iterations and $seed = 1, \ldots, 10$, the result depends on the repair:

q	repair	bins at the end (10 seeds)	iterations to reach 9
2	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	never
2	exact	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	never
3	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	never
3	exact	9, 9, 9, 9, 9, 9, 9, 9, 9, 9	4, 4, 12, 38, 3, 20, 5, 2, 7, 5
4	FFD	10, 10, 10, 10, 10, 10, 10, 10, 10, 10	never
4	exact	9, 9, 9, 9, 9, 9, 9, 9, 9, 9	2, 3, 9, 12, 4, 5, 1, 3, 10, 2

The power of the repair

- With the FFD repair, *LNS* **never** improves the FFD packing: the items removed from an FFD packing are repacked by FFD in the same way.
- With the exact repair and $q \geq 3$ it reaches the optimum in every run, in a few iterations: it finds three bins whose items fit into two, which is what FFD cannot see.
- With $q = 2$ even the exact repair fails, because the items of two bins of an FFD packing never fit into one.

A large neighbourhood is explored by solving a **subproblem**: the part of the solution left free by the destroy method is re-optimized, while the rest is fixed.

The more powerful the method that solves the subproblem, the more powerful the neighbourhood.

Adaptive large neighbourhood search

- Which destroy method works best depends on the instance, and on the phase of the search.
- **Adaptive** LNS (ALNS), introduced by Ropke and Pisinger (2006), uses several destroy (and repair) methods, chooses one at random at every iteration, and **learns** which ones work, by giving to each method o a weight w_o:

$$P(\text{choose } o) = \frac{w_o}{\sum_{o'} w_{o'}}, \qquad w_o \leftarrow (1-\rho)\, w_o + \rho\, \psi,$$

- $\rho \in [0, 1]$ is the *reaction factor*;
- the score ψ rewards the outcome of the iteration in which o was used: $\psi = 3$ if the new solution is the best found so far, 2 if it is better than the current one, 1 if it is as good, 0 if it is worse.

Algorithm: Adaptive large neighbourhood search for the TSP (1/2)

Input : distances d, starting tour σ, q, *maxIter*, reaction factor ρ

Output : best tour found

```
1 function ALNS(d, σ, q, maxIter, ρ)
2     cur := σ;  best := σ;  w1 := 1;  w2 := 1
3     for it := 1 to maxIter do
4         if Rand() · (w1 + w2) < w1 then o := 1 else o := 2
            // roulette wheel
5         nw := cur
6         if o = 1 then removed := RandomRemoval(nw, q)
           else removed := WorstRemoval(nw, q)
7         CheapestInsertionRepair(nw, removed)
```

Algorithm: Adaptive large neighbourhood search for the TSP (2/2)

function *ALNS*$(d, \sigma, q, maxIter, \rho)$ (continued)

    for $it := 1$ **to** *maxIter* (continued) **do**
8       $\psi :=$ 3, 2, 1 or 0 if $L(nw) < L(best)$, $L(nw) < L(cur)$, $L(nw) = L(cur)$, $L(nw) > L(cur)$
9   **if** $L(nw) \leq L(cur)$ **then** $cur := nw$
10   **if** $L(cur) < L(best)$ **then** $best := cur$
11      $w_o := (1 - \rho)\, w_o + \rho\, \psi$ `// adaptive weights`
12 **end**
13 **return** *best*
14 **end**

What ALNS learns

- WorstRemoval removes, q times, the city whose removal shortens the tour the most: the city v, between a and b, of maximum $d_{av} + d_{vb} - d_{ab}$. It is a deterministic, greedy destroy method: it removes the cities that seem badly placed.
- On TSP20, from the nearest neighbour tour, with $q = 5$, 200 iterations and $\rho = 0.1$, ALNS reaches the optimum in 7 runs out of 10 (average 299.4).
- At the end the weight of the random removal is between 0.77 and 0.98, the weight of the worst removal between 0.03 and 0.08: the method has learnt that, from a good tour, removing the worst-placed cities and reinserting them by cheapest insertion almost always rebuilds a tour that is not better, and it has almost stopped using it.
- In these runs plain *LNS* with random removal ($q = 5$) does slightly better: adaptivity pays off when there are many methods, of unknown quality, and long runs.

Summary (1/2)

Large neighbourhood search:

- destroy a part of the solution, repair it, accept if better (or not worse)
- the neighbourhood is large and explored implicitly by the repair method
- destroy diversifies (q), repair intensifies

On the reference instances:

- TSP: one destroy-and-repair step with $q = 2$ reaches the optimal tour
- BPP: emptying the least loaded bin and repacking by FFD reaches 3 bins in four iterations

Summary (2/2)

The repair:

- on BPP24 the FFD repair never improves the FFD packing
- the exact repair with $q \geq 3$ reaches the optimum 9 in every run
- the more powerful the method that solves the subproblem, the more powerful the neighbourhood

Adaptive LNS:

- chooses the destroy method by roulette wheel, with weights updated by the scores ψ
- pays off with many methods of unknown quality and long runs

Population-based methods: genetic and memetic algorithms

Introduction: Population-based methods

- Genetic and memetic algorithms keep a *population* of solutions, and combine the parts of good solutions into new ones
- We see a steady-state genetic algorithm for the KP, its components (encoding, selection, crossover, mutation, repair, replacement), and the memetic variant with a local search
- Then we run three iterations by hand on the KP, and compare the two variants on KP20

The idea of genetic and memetic algorithms

- A *genetic algorithm* (denoted by *GA*) keeps a *population* of solutions, encoded as strings (*chromosomes*). At every iteration it
  - *selects* two parents, preferring the fitter ones,
  - *recombines* them into a child (*crossover*),
  - changes a few random genes of the child (*mutation*),
  - *repairs* it if it is infeasible,
  - and inserts it into the population in place of a worse individual.
- A *memetic algorithm* also applies a local search to every child.

Next: the pseudo-code of a steady-state *GA* for the KP with binary encoding;
with *memetic* $= \mathrm{true}$ it is a memetic algorithm.

Algorithm: Genetic and memetic algorithm for the KP (1/4)

Input : KP instance $p[1..n]$, $w[1..n]$, c; population size P; mutation probability p_m; *maxIter*; *memetic*

Output : best individual of the final population

```
1 function GeneticKP(p, w, c, P, p_m, maxIter, memetic)
2     for s := 1 to P do    // random initial population
3         for j := 1 to n do pop[s][j] := 1 if RAND() < 0.5,
           else 0
4         REPAIRKP(pop[s]); if memetic then
           KPLOCALSEARCH(pop[s])
5         fit[s] := Σ_j p[j] pop[s][j]
6     end
```

Algorithm: Genetic and memetic algorithm for the KP (2/4)

function *GeneticKP*$(p, w, c, P, p_m, maxIter, memetic)$ (continued)

```
7   |   for it := 1 to maxIter do
8   |   |   a := Tournament(); b := Tournament()
    |   |    // selection
9   |   |   cut := 1 + ⌊Rand() · (n − 1)⌋          // one-point
    |   |    crossover
10  |   |   child := (pop[a][1..cut], pop[b][cut+1..n])
11  |   |   for j := 1 to n do if Rand() < p_m then
    |   |    child[j] := 1 − child[j]
12  |   |                                          // mutation
```

Algorithm: Genetic and memetic algorithm for the KP (3/4)

function *GeneticKP*$(p, w, c, P, p_m, maxIter, memetic)$
(continued)

```
    for it := 1 to maxIter (continued) do
12      RepairKP(child); if memetic then
          KPLocalSearch(child)
13      worst := the last index s of minimum fit[s]
14      if child ∉ pop and Σ_j p[j] child[j] > fit[worst] then
          // replacement
15        pop[worst] := child; fit[worst] := Σ_j p[j] child[j]
16      end
17  end
18  return the individual of maximum fit
19 end
```

Algorithm: Genetic and memetic algorithm for the KP (4/4)

```
20 function Tournament()
21 |   a := 1 + ⌊RAND() · P⌋;  b := 1 + ⌊RAND() · P⌋
22 |   if fit[a] ≥ fit[b] then return a else return b
23 end
```

Encoding, selection, crossover

- Genetic algorithms were introduced by Holland (1975) as a model of natural evolution, and made popular as optimization methods by Goldberg (1989). Memetic algorithms, in which every individual is improved by a local search, were proposed by Moscato (1989).
- **Encoding.** For the KP the binary vector x is a natural chromosome: the gene j says whether item j is selected. The **fitness** of an individual is its profit.
- **Selection.** A binary tournament draws two individuals at random and keeps the fitter one: good individuals are more likely to become parents, but every individual has a chance.
- **Crossover.** One-point crossover takes the genes $1, \ldots, cut$ from the first parent and the genes $cut + 1, \ldots, n$ from the second one. The idea is that good solutions share good *building blocks*, and recombining two good parents can put the good blocks of both into one child.

Mutation, repair, replacement

- **Mutation.** Each gene is flipped with a small probability p_m, typically $\frac{1}{n}$: it brings back into the population genes that selection may have lost.
- **Repair.** The child can be infeasible. RepairKP removes the selected item of minimum efficiency until the capacity is respected (ties: the item of largest index). Alternatives are penalties, as in simulated annealing with a penalty, or encodings in which every chromosome is feasible.
- **Replacement.** In the steady-state version of the pseudo-code, the child replaces the worst individual if it is better and not already in the population; duplicates would make the population lose its diversity.

Intensification and diversification in a *GA*

- Selection and replacement **intensify** (the population concentrates on good solutions); crossover and mutation **diversify**. Without local search, a *GA* is good at finding promising regions but slow at reaching their local optima: the memetic algorithm adds the missing intensification.

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

- With $P = 4$, $p_m = \frac{1}{4}$ and $seed = 36$. The first 16 random numbers build the initial population (gene 1 if the number is smaller than 0.5):

$$
\begin{array}{lllll}
pop[1]: & 0.0003, & 0.7354, & 0.2018, & 0.5114 \quad \rightarrow (1,0,1,0) \\
pop[2]: & 0.1796, & 0.8825, & 0.6936, & 0.4391 \quad \rightarrow (1,0,0,1) \\
pop[3]: & 0.4547, & 0.6489, & 0.8061, & 0.6990 \quad \rightarrow (1,0,0,0) \\
pop[4]: & 0.9148, & 0.2446, & 0.9246, & 0.0692 \quad \rightarrow (0,1,0,1)
\end{array}
$$

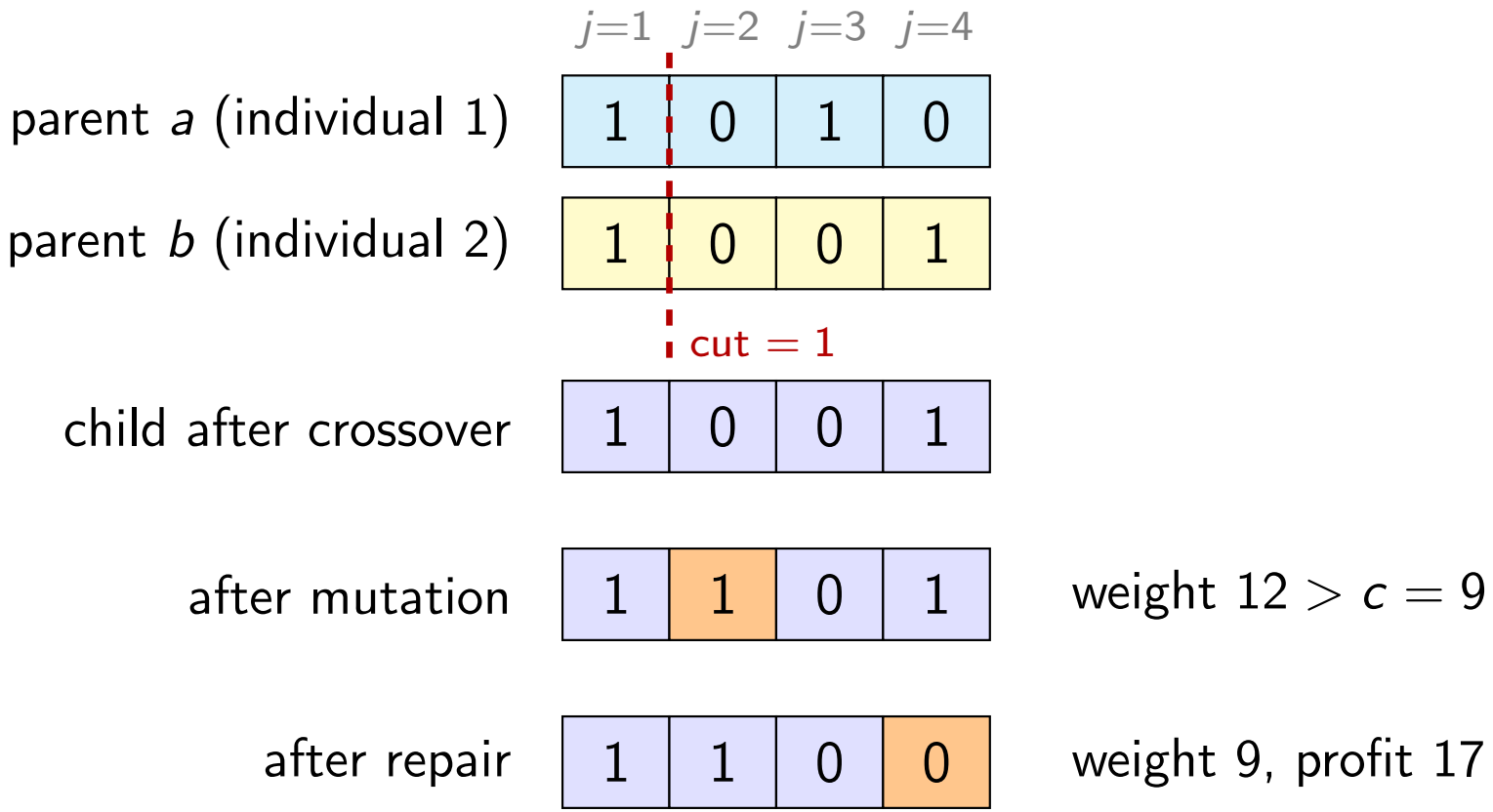
- All four individuals are feasible: $pop[1] = (1,0,1,0)$, $pop[2] = (1,0,0,1)$, $pop[3] = (1,0,0,0)$, $pop[4] = (0,1,0,1)$, with fitness $(16, 14, 10, 11)$.

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

Iteration 1 (random numbers 0.1614, 0.2771, 0.8030, 0.4063, 0.0295, 0.7238, 0.2032, 0.4957, 0.4620).

- The first tournament draws individuals $1 + \lfloor 4 \cdot 0.1614 \rfloor = 1$ and $1 + \lfloor 4 \cdot 0.2771 \rfloor = 2$, and selects $a = 1$ (fitness $16 > 14$).
- The second draws 4 and 2 and selects $b = 2$ (fitness $14 > 11$).
- The cut is $1 + \lfloor 3 \cdot 0.0295 \rfloor = 1$: the child takes gene 1 from $pop[1]$ and genes 2, 3, 4 from $pop[2]$, $child = (1, 0, 0, 1)$.
- Only the second mutation number, 0.2032, is smaller than $p_m = 0.25$: gene 2 flips, $child = (1, 1, 0, 1)$, of weight $12 > 9$.
- The repair removes item 4, the selected item of minimum efficiency: $child = (1, 1, 0, 0)$, of weight 9 and profit 17. It replaces the worst individual, $pop[3]$ of fitness 10.

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



- Iteration 1: crossover, mutation of gene 2, and repair (item 4 removed).

Example: *GA* on the KP reference instance (4/4)

- **Iteration 2** selects $a = 2$ and $b = 3$, cuts after gene 1 and builds $(1, 1, 0, 0)$, with no mutation: it is already in the population, and it is discarded.
- **Iteration 3** selects $a = 2$ and $b = 1$, cuts after gene 3, builds $(1, 0, 0, 0)$, and the mutation flips genes 1 and 4: $(0, 0, 0, 1)$, of profit 4, worse than every individual, discarded.

The best individual is $(1, 1, 0, 0)$: $GA(I) = 17 = z(I)$,
found by crossover, mutation and repair together.

GA and memetic algorithm on KP20

- On KP20, with $P = 10$, $p_m = 0.05$ and 300 iterations, for $seed = 1, \ldots, 10$:

	min	average	max	optimal
GA	508	516.9	521	4
memetic	521	521.0	521	10

	best profit per seed
GA	521, 519, 521, 508, 514, 521, 510, 519, 521, 515
memetic	521, 521, 521, 521, 521, 521, 521, 521, 521, 521

- The plain *GA* reaches the optimum 521 in 4 runs out of 10; the memetic algorithm in every run.
- The local search makes every individual a local optimum, and the crossover then recombines local optima instead of arbitrary solutions.

Summary

Genetic algorithm:

- a population of chromosomes; tournament selection, one-point crossover, mutation, repair, replacement of the worst individual
- selection and replacement intensify, crossover and mutation diversify

Memetic algorithm:

- a local search on every child: it recombines local optima
- on KP20 it reaches the optimum in every run, the plain *GA* in 4 runs out of 10

Comparison and conclusions

Introduction: Comparison and conclusions

- We collect the results of all the experiments of the chapter on the medium instances
- We summarize which component of each method intensifies and which diversifies, and how the parameters are tuned
- Finally we see what these methods do not use, and what they cannot give: a dual bound

The methods on the medium instances (1/3)

- For each method, the best value found in 10 runs ($seed = 1, \ldots, 10$), as minimum, average and maximum, and the number of runs that reach the optimum. Deterministic methods give the same value in every run.

instance	method	parameters	min	average	max	optimal
TSP20	NN	—	326	326	326	0
	NN + 2-opt	—	319	319	319	0
	SA	$T_0 = 20$, $\alpha = 0.95$	291	294.3	298	5
	SA	$T_0 = 20$, $\alpha = 0.99$	291	291.0	291	10
	TS	*tenure* $= 5$, 200 it.	291	291	291	10
	ILS	100 it.	291	291.0	291	10
	GRASP	$\alpha = 0.25$, 20 it.	291	291.0	291	10
	LNS	$q = 5$, 200 it.	291	293.8	319	9
	ALNS	$q = 5$, 200 it.	291	299.4	319	7

The methods on the medium instances (2/3)

instance	method	parameters	min	average	max	optimal
KP20	greedy	—	490	490	490	0
	greedy + local search	—	514	514	514	0
	GRASP	$\alpha = 0.25$, 20 it.	521	521.0	521	10
	GA	$P = 10$, 300 it.	508	516.9	521	4
	memetic	$P = 10$, 300 it.	521	521.0	521	10

The methods on the medium instances (3/3)

instance	method	parameters	min	average	max	optimal
PCM15	LPT	—	73	73	73	0
	TS	*tenure* $= 15$, 300 it.	69	69	69	10
	VNS	$k_{\max} = 3$, 300 it.	69	69.3	70	7
BPP24	FFD	—	10	10	10	0
	LNS, FFD repair	$q = 3$, 200 it.	10	10.0	10	0
	LNS, exact repair	$q = 3$, 200 it.	9	9.0	9	10

What the comparison shows

- Every metaheuristic improves on the local search it is built on, and with suitable parameters several of them reach the optimum in every run.
- The table, however, also shows how much the result depends
  - on the parameters (the tenure of TS, the strength q of the destruction, the cooling of SA)
  - and on the components (the repair of LNS, the local search of the memetic algorithm).
- On four small instances it does not rank the methods: such a ranking needs many instances and equal running times, as discussed in the chapter «Heuristic algorithms: definitions».

Intensification and diversification (1/2)

- Every metaheuristic of this chapter combines a component that **intensifies** (searches around the good solutions already found) and a component that **diversifies** (moves the search to unexplored regions), and has parameters that set the balance between the two.

method	intensification	diversification	balance set by
SA	accepting only improving moves at low temperature	accepting worse moves at high temperature	T_0, cooling factor α, L
TS	moving to the best neighbour; restarting from elite solutions	tabu list; penalties on frequent attributes	tenure, aspiration criterion

Intensification and diversification (2/2)

method	intensification	diversification	balance set by
ILS	local search after every kick; accepting only better local optima	the kick	strength of the kick, acceptance criterion
VNS	local search; return to $k = 1$ after an improvement	shaking in N_k with growing k	$k_{\max}$, the neighbour-hoods N_k
GRASP	local search on every constructed solution	randomized construction	α
LNS / ALNS	repair (heuristic or exact), acceptance	random destroy	q, choice of the methods, weights
GA / memetic	selection, replacement, local search (memetic)	crossover, mutation	population size, p_m

Parameters and tuning

- A metaheuristic is an algorithm only after its parameters are fixed, and the experiments of this chapter show that the parameters can make the difference between finding the optimum in every run and never finding it. Choosing them is called **tuning**.
- The parameters must be tuned on **training** instances and evaluated on *different* ones, otherwise the picture is optimistic.
- Tuning one parameter at a time ignores their interactions (for *SA*, the best T_0 depends on α); tools such as `irace` (López-Ibáñez et al., 2016) tune all of them together.
- A method that works well for a wide range of parameters is preferable to one that works very well only for a narrow range.

What these methods do not use (1/2)

- None of the metaheuristics of this chapter uses a **relaxation** of the problem or a **MIP solver**. They manipulate feasible (or penalized) solutions only, and they produce only *primal* bounds.
- As a consequence, they cannot tell how good their solutions are. On the medium instances we could measure the quality only because the optimal values had been computed separately, by exact algorithms.

What these methods do not use (2/2)

- The two exceptions confirm the rule: for $P \,||\, C_{\max}$ and the BPP, *TS*, *VNS* and *LNS* stopped as soon as they reached a simple lower bound, which is a *dual* bound that certifies optimality.
- When the optimum is above the bound, no such certificate exists, and the search goes on until the iteration limit.
- The exact repair of *LNS* for the BPP, finally, showed that re-optimizing a part of the solution exactly, with the rest fixed, gives very powerful neighbourhoods.

Metaheuristics escape from local optima, but they give only one side of the gap: the primal bound.

Summary

Comparison:

- every metaheuristic improves on its local search; several reach the optimum in every run
- the results depend strongly on the parameters and on the components
- a ranking needs many instances and equal running times

Tuning and limits:

- tune on training instances, evaluate on different ones
- no relaxation, no MIP solver: only primal bounds, no certificate of quality

Typical mistakes

Introduction: Typical mistakes

- We collect the mistakes that are made most often when running and evaluating a metaheuristic
- Two concern the experiments, two the algorithms, one the execution by hand
- Check them before reporting a result

Typical mistakes (1/3)

Observation 1: Reporting one run

The results depend on the seed: report minimum, average and maximum over several runs.

Observation 2: Tuning and evaluating on the same instances

The picture is optimistic; use training instances for the parameters and different ones for the evaluation.

Typical mistakes (2/3)

Observation 3: Losing the best solution

In simulated annealing and tabu search the current solution can be worse than the best one found: the best is stored apart.

Observation 4: Choosing the tenure without testing it

With tenure 0 *tabu search is a local search; a tenure that is too small cycles, one that is too large forbids almost every move.*

Typical mistakes (3/3)

Observation 5: Miscounting the random numbers

Every iteration consumes a fixed number of them (ten in the genetic algorithm of the exercises, with $n = 5$); a wrong count changes every later step.

Summary

Experiments:

- report minimum, average and maximum over several seeds
- tune and evaluate on different instances

Algorithms and execution by hand:

- store the best solution apart from the current one
- test the tenure of tabu search
- count the random numbers consumed by every iteration

C++ implementation

Introduction: C++ implementation

- Every algorithm of the chapter is implemented in C++, with the same logic as the pseudo-codes
- We see how the code is organized, how the examples are checked, and three short extracts: the random number generator, the core of simulated annealing and the double-bridge kick
- The complete code is in the notes of the chapter

The organization of the code

- All the functions are in the folder `METAHEUR_CODE/src`, one function per file, and are declared in the header `Metaheuristics.h`.
- The items, cities and jobs are indexed $0, 1, \ldots, n-1$, the machines $0, \ldots, m-1$ and the bins $0, \ldots, nb-1$: the algorithmic logic is the one of the pseudo-codes, with 0-based indices.
- Vectors are `Vec` (`vector<int>`) and matrices `Matrix` (`vector<Vec>`).
- The command `make test` compiles the code and runs every example of this chapter and every instance of the exercises, checking the value found against the expected one; `make data` writes the data of the figure of the *SA* experiment on TSP20.

The random number generator

- The random numbers are produced by `Rand`, the minimal standard generator. The structure `Random` contains the seed and, optionally, a list of given numbers, which are returned instead: the exercises give their random numbers explicitly.

```
1  double Rand(Random& r)
2  {
3      double u;
4      if (r.given.size() > 0)
5      {
6          u = r.given[r.used];
7      }
8      else
9      {
10         r.seed = (16807 * r.seed) % 2147483647;
11         u = (double) r.seed / 2147483647.0;
12     }
13     r.used = r.used + 1;
14     return u;
15 }
```

The core of simulated annealing for the TSP

```
while (T > Tmin)
{
    for (int l = 0; l < L; ++l)  // L moves at temperature T
    {
        int k = (int) (Rand(r) * M);  // random neighbour
        int delta = TwoOptDelta(cur, d, mi[k], mj[k]);
        double v = Rand(r);
        if (delta <= 0 || v < exp(-delta / T))  // Metropolis acceptance rule
        {
            ApplyTwoOpt(cur, mi[k], mj[k]);
            fcur = fcur + delta;
            if (fcur < fbest) { best = cur; fbest = fcur; }
        }
    }
    T = alpha * T;  // cooling
}
```

The double-bridge kick

```
1  Vec DoubleBridge(const Vec& pi, Random& r)
2  {
3      int n = pi.size();
4      int p1 = 1 + (int) (Rand(r) * (n - 3));
5      int p2 = p1 + 1 + (int) (Rand(r) * (n - 2 - p1));
6      int p3 = p2 + 1 + (int) (Rand(r) * (n - 1 - p2));
7      Vec kick;
8      for (int k = 0; k < p1; ++k) kick.push_back(pi[k]);  // A
9      for (int k = p2; k < p3; ++k) kick.push_back(pi[k]);  // C
10     for (int k = p1; k < p2; ++k) kick.push_back(pi[k]);  // B
11     for (int k = p3; k < n; ++k) kick.push_back(pi[k]);  // D
12     return kick;
13 }
```

The files of the chapter (1/2)

what	files in `METAHEUR_CODE/src`
random numbers	`Rand`
2-opt local search for the TSP	`TourLength`, `TwoOptDelta`, `ApplyTwoOpt`, `TwoOptLS`, `NearestNeighbour`
move descent for $P \mid\mid C_{\max}$	`Loads`, `LexLess`, `MoveDescent`, `LPT`
local search for the KP	`Profit`, `Weight`, `KPLocalSearch`
simulated annealing	`SimulatedAnnealingTSP`, `SimulatedAnnealingKP`
tabu search	`TabuSearchPCmax`, `TabuSearchTSP`

The files of the chapter (2/2)

what	files in `METAHEUR_CODE/src`
iterated local search	`DoubleBridge`, `IteratedLocalSearchTSP`
variable neighbourhood search	`Shake`, `VNSPCmax`
GRASP	`RandomizedGreedyKP`, `GraspKP`, `RandomizedNN`, `GraspTSP`
LNS and ALNS for the TSP	`RandomRemoval`, `WorstRemoval`, `CheapestInsertionRepair`, `LnsTSP`, `AlnsTSP`
LNS for the BPP	`FirstFit`, `DestroyBins`, `RepairFFD`, `RepairExact`, `LnsBPP`
genetic and memetic algorithm	`RepairKP`, `Tournament`, `GeneticKP`

Summary

The code:

- one function per file in `METAHEUR_CODE/src`, declared in `Metaheuristics.h`
- the logic of the pseudo-codes, with 0-based indices

Checks:

- `make test` runs every example and every exercise against the expected value
- `Rand` can return given numbers, as in the exercises

References (1/3)

- V. Černý, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, *Journal of Optimization Theory and Applications* 45 (1985) 41–51.
- T.A. Feo, M.G.C. Resende, A probabilistic heuristic for a computationally difficult set covering problem, *Operations Research Letters* 8 (1989) 67–71.
- T.A. Feo, M.G.C. Resende, Greedy randomized adaptive search procedures, *Journal of Global Optimization* 6 (1995) 109–133.
- F. Glover, Future paths for integer programming and links to artificial intelligence, *Computers & Operations Research* 13 (1986) 533–549.
- F. Glover, M. Laguna, *Tabu Search*, Kluwer, 1997.
- D.E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley, 1989.
- B. Hajek, Cooling schedules for optimal annealing, *Mathematics of Operations Research* 13 (1988) 311–329.

References (2/3)

- J.H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, 1975.
- S. Kirkpatrick, C.D. Gelatt, M.P. Vecchi, Optimization by simulated annealing, *Science* 220 (1983) 671–680.
- M. López-Ibáñez, J. Dubois-Lacoste, L. Pérez Cáceres, M. Birattari, T. Stützle, The irace package: iterated racing for automatic algorithm configuration, *Operations Research Perspectives* 3 (2016) 43–58.
- H.R. Lourenço, O.C. Martin, T. Stützle, Iterated local search, in F. Glover, G.A. Kochenberger (eds.), *Handbook of Metaheuristics*, Kluwer, 2003, 321–353.
- O. Martin, S.W. Otto, E.W. Felten, Large-step Markov chains for the traveling salesman problem, *Complex Systems* 5 (1991) 299–326.
- N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller, Equation of state calculations by fast computing machines, *The Journal of Chemical Physics* 21 (1953) 1087–1092.

References (3/3)

- N. Mladenović, P. Hansen, Variable neighborhood search, *Computers & Operations Research* 24 (1997) 1097–1100.
- P. Moscato, On evolution, search, optimization, genetic algorithms and martial arts: towards memetic algorithms, Technical Report C3P 826, Caltech, 1989.
- S.K. Park, K.W. Miller, Random number generators: good ones are hard to find, *Communications of the ACM* 31 (1988) 1192–1201.
- S. Ropke, D. Pisinger, An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows, *Transportation Science* 40 (2006) 455–472.
- P. Shaw, Using constraint programming and local search methods to solve vehicle routing problems, in *Principles and Practice of Constraint Programming – CP98*, Lecture Notes in Computer Science 1520, Springer, 1998, 417–431.
- É. Taillard, Robust taboo search for the quadratic assignment problem, *Parallel Computing* 17 (1991) 443–455.

In summary

Introduction: In summary

- We collect the key ideas of the whole chapter on metaheuristics, both parts
- The dashboard compares, on the medium instances, the constructive heuristics, the local searches, the best metaheuristic and the optimum

Summary (1/2)

- A metaheuristic escapes from local optima by accepting worse solutions (SA, TS), by perturbing (ILS, VNS), by randomizing the construction (GRASP), by destroying and repairing (LNS) or by recombining a population (GA).
- Each method balances intensification and diversification through a few parameters, and the results depend strongly on them.
- They use no relaxation: they produce primal bounds only, and cannot tell how good their solutions are.

Summary (2/2)

Dashboard of the medium instances. Best value of the methods against the optimum (the optimum computed by an exact algorithm):

instance	constructive	local search	best metaheuristic	optimum
TSP20 (min)	326 (*NN*)	319 (2-opt)	291	291
KP20 (max)	490 (greedy)	514	521	521
PCM15 (min)	73 (*LPT*)	—	69	69
BPP24 (min)	10 (FFD)	—	9	9