

Local search (1/2)

Outline (1/2)

Why local search?

Local search

The problems, the assumptions and the starting solutions
Starting solutions

Local search: the scheme

Neighbourhoods for the four problems

KP: flip and swap

BPP: move and swap

TSP: 2-opt and Or-opt

$P \parallel C_{\max}$: move and swap

Summary

Outline (2/2)

First improvement and best improvement

- First improvement

- Best improvement

- Comparison

Evaluating a move incrementally

TSP: 2-opt and Or-opt

- The 2-opt local search

- The Or-opt local search

- First and best improvement on a medium instance

- 3-opt and Lin-Kernighan

Learning objectives

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

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

Prerequisites

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

Why local search?

Introduction: Why local search?

- ▶ The constructive heuristics give a first feasible solution, often far from the optimum
- ▶ They never go back on a choice: nothing in them can repair a bad decision
- ▶ This chapter takes the next step: we start from a solution and we try to improve it

From construction to improvement

- ▶ A constructive heuristic builds a solution one element at a time and never revises a choice once it is made.
- ▶ This chapter studies the next step: **how to improve a solution that we already have**, by modifying it a little at a time.
- ▶ The constructive heuristic gives the starting solution; the local search improves it.

A question to keep in mind

The starting solutions of the constructive heuristics, on the reference instances of the course, and the optimal values:

problem	heuristic	value	$z(I)$
KP	<i>GR</i>	16	17
BPP	<i>FF</i>	4	3
TSP	<i>NN</i>	25	18
$P \parallel C_{\max}$	<i>LPT</i>	11	9

- ▶ Can small modifications of these solutions close the gaps?
- ▶ The chapter answers on each of the four problems.

Summary

From construction to improvement:

- ▶ A constructive heuristic never revises its choices
- ▶ Local search improves a solution that we already have, a little at a time

The question:

- ▶ Can small modifications close the gap between the starting solutions and the optimal values?

Local search

Introduction: Local search

- ▶ In this section we define a local search, the basic improvement heuristic
- ▶ It is built on the notions of neighbourhood, move and local optimum
- ▶ All the algorithms of the chapter are local searches, or combinations of local searches

Definition 1: Local search

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

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

Local search in one sentence

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

Local search:

- ▶ An improvement heuristic: it starts from a feasible solution
- ▶ At every iteration it moves to a better neighbour, if there is one
- ▶ It stops in a local optimum with respect to the neighbourhood N

The problems, the assumptions and the starting solutions



Introduction: The problems, the assumptions and the starting solutions

- ▶ We recall the four problems on which the algorithms are illustrated, and their notation
- ▶ We recall the reference instances and the starting solutions computed by the constructive heuristics
- ▶ We explain how the random starting solutions are generated, so that every experiment can be repeated

The four problems

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

The notation of the four problems

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

Loads and critical machines

- ▶ For $P \parallel C_{\max}$, $L_h = \sum_{j \in J_h} t_j$ is the **load** of machine h and

$$C_{\max} = \max_h L_h.$$

- ▶ For the BPP, $L_b = \sum_{j \in P_b} w_j$ is the **load** of bin b .
- ▶ A machine (or a bin) whose load equals the objective value is called **critical**.

The reference instances

The **reference instances** are those of the chapter «Heuristic algorithms: definitions»:

- ▶ the KP with $p = (10, 7, 6, 4)$, $w = (5, 4, 3, 3)$, $c = 9$;
- ▶ the BPP with $w = (4, 4, 5, 3, 2, 3)$, $c = 7$;
- ▶ the TSP with 5 cities;
- ▶ $P \parallel C_{\max}$ with $t = (5, 5, 4, 4, 3, 3, 3)$ and $m = 3$.

Their optimal values are 17, 3, 18 and 9.

Starting solutions

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

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

Which starting solutions

- ▶ The solution of *FFD* is already optimal, so no local search can improve it: for the BPP we start from the solutions of *NF* and *FF*.
- ▶ To study the behaviour of a local search on many starting solutions we also use **random starting solutions**.

Random starting solutions

- ▶ The random starting solutions are generated with the Park–Miller minimal standard generator: given an integer **seed** $s_0 \in \{1, \dots, 2^{31} - 2\}$, the k -th pseudo-random number is

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

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

- ▶ The sequence is completely determined by the seed: the same seed gives the same starting solutions, in the notes, in the exercises and in the code.

Random tours, assignments and packings

- ▶ A **random tour** keeps city 1 in the first position and shuffles the other cities (Fisher–Yates: for $k = n, n - 1, \dots, 3$, the city in position k is exchanged with the city in a random position among $2, \dots, k$).
- ▶ A **random assignment** of the jobs draws the machine of the jobs $1, 2, \dots, n$ in this order.
- ▶ A **random BPP solution** is computed by FF on a random order of the items (Fisher–Yates on all the positions: for $k = n, n - 1, \dots, 2$, the item in position k is exchanged with the item in a random position among $1, \dots, k$).

Summary

The problems:

- ▶ KP, BPP, TSP and $P \parallel C_{\max}$, with loads L_b , L_h and critical bins and machines
- ▶ Reference instances with optimal values 17, 3, 18 and 9

The starting solutions:

- ▶ The solutions of the constructive heuristics: GR , GR_w , NF , FF , NN , LPT
- ▶ Random solutions from the Park–Miller generator: the same seed gives the same solutions

Local search: the scheme

Introduction: Local search: the scheme

- ▶ We write the pseudo-code of a local search and we list its four ingredients
- ▶ We prove that a local search always stops, in a local optimum
- ▶ We see which local optimum it reaches: the bottom of the valley in which it starts

The pseudo-code of *LS* reads as follows.

The function *Improve* explores the neighbourhood and returns an improving neighbour of x , or *NULL* if there is none.

Algorithm: Local search

Input : a feasible solution x^0 , a neighbourhood N , a rule $\in \{first, best\}$

Output : a local optimum x with respect to N

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

The four ingredients of a local search

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

The function that guides the search

- ▶ The function f that decides whether a neighbour is “better” is usually the objective of the problem.
- ▶ But it does not have to be: the search may be guided by a different function, as long as the solution it returns is feasible.
- ▶ We will use this freedom later in the chapter, with auxiliary objectives and penalties.

Does a local search always stop?
And how many moves can it make?

Proposition 1

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

Proof

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



How many moves?

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

Local search intensifies

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

Summary

The scheme:

- ▶ *Improve* returns an improving neighbour or *NULL*; *LS* moves until *NULL*
- ▶ Four ingredients: starting solution, neighbourhood, acceptance criterion, stopping criterion

Termination:

- ▶ *LS* stops in a local optimum, after at most $f(x^0) - z(I)$ moves for integer f (minimization)
- ▶ In practice few moves: the time goes into exploring the neighbourhoods

Neighbourhoods for the four problems

Introduction: Neighbourhoods for the four problems

- ▶ In this section we define the neighbourhoods used in the chapter: two for each problem, and 3-opt for the TSP
- ▶ For every neighbourhood we fix the order of the moves and we count its elements
- ▶ The size of a neighbourhood is the number of neighbours to evaluate at every iteration

Feasible neighbours, in a fixed order

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

KP: flip and swap

- ▶ The **flip** neighbourhood, defined in the chapter «Heuristic algorithms: definitions», changes one component of x , i.e., adds or removes one item; it has n elements before the feasibility test. The flips are examined for $j = 1, 2, \dots, n$.
- ▶ The **swap** neighbourhood removes one selected item j and adds one unselected item k . If S is the set of the selected items, it has $|S|(n - |S|) \leq n^2/4$ elements; the swaps are examined for $j \in S$ in increasing order and, for each j , for $k \notin S$ in increasing order.
- ▶ The two neighbourhoods are complementary: a flip changes the number of selected items, a swap keeps it. The neighbourhood **flip+swap** is their union, with the flips examined first.

BPP: move

- ▶ The **move** neighbourhood moves one item j from its bin $a = \text{bin}_j$ to another non-empty bin b with $L_b + w_j \leq c$.
- ▶ It has at most $n(nb - 1)$ elements; the moves are examined for $j = 1, 2, \dots, n$ and, for each j , for the bins in increasing order of their label.
- ▶ Moving an item into an empty bin is never useful: it cannot decrease the number of bins.
- ▶ The bins keep the labels of the starting solution; a bin emptied by a move simply disappears.

BPP: swap

- ▶ The **swap** neighbourhood exchanges two items $j < k$ in different bins, provided that both bins remain feasible:

$$L_{bin_j} - w_j + w_k \leq c \quad \text{and} \quad L_{bin_k} - w_k + w_j \leq c.$$

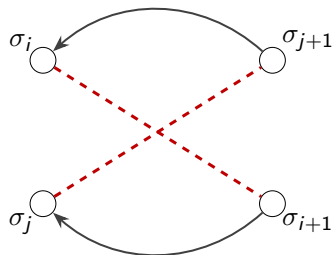
- ▶ Exchanging two items of equal weight does not change any load, and is not considered.
- ▶ It has at most $\frac{n(n-1)}{2}$ elements, examined for $j = 1, \dots, n$ and $k = j + 1, \dots, n$.

TSP: the 2-opt move

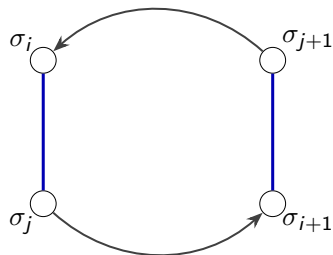
- ▶ A tour is a permutation $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_n)$ of the cities, with $\sigma_{n+1} = \sigma_1$. We keep city 1 in the first position, $\sigma_1 = 1$.
- ▶ The **2-opt** move with positions $i < j$ removes the two edges $\{\sigma_i, \sigma_{i+1}\}$ and $\{\sigma_j, \sigma_{j+1}\}$, and reconnects the tour in the only other possible way: with the edges $\{\sigma_i, \sigma_j\}$ and $\{\sigma_{i+1}, \sigma_{j+1}\}$, travelling the path $\sigma_{i+1}, \dots, \sigma_j$ in the opposite direction (Croes, *Operations Research* 6, 1958):

$$\begin{array}{c} (\sigma_1, \dots, \sigma_i, \underbrace{\sigma_{i+1}, \sigma_{i+2}, \dots, \sigma_j}_{\text{path}}, \sigma_{j+1}, \dots, \sigma_n) \\ \downarrow \\ (\sigma_1, \dots, \sigma_i, \underbrace{\sigma_j, \dots, \sigma_{i+2}, \sigma_{i+1}}_{\text{reversed path}}, \sigma_{j+1}, \dots, \sigma_n). \end{array}$$

TSP: the 2-opt move in a picture



before: the two dashed edges are removed



after: two new edges, the lower path is reversed

TSP: order and size of the 2-opt neighbourhood

- ▶ The moves are examined for $i = 1, \dots, n - 2$ and $j = i + 2, \dots, n$, excluding $i = 1, j = n$ (the two edges would share city σ_1).
- ▶ The 2-opt neighbourhood has

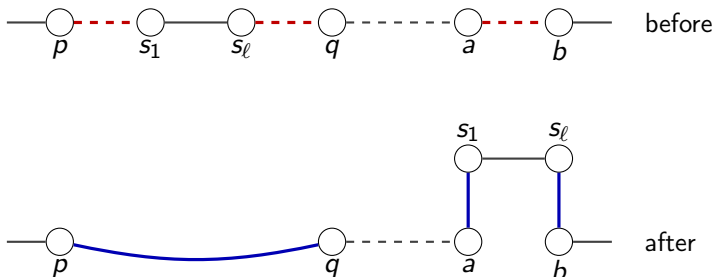
$$\frac{n(n-3)}{2}$$

elements: 5 for the reference instance.

TSP: the Or-opt move

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

TSP: the Or-opt move in a picture



- Dashed red edges are removed, blue edges are added.

TSP: order and size of the Or-opt neighbourhood

- ▶ The moves are examined for $\ell = 1, 2, 3$, then for $i = 2, \dots, n - \ell + 1$, then for $k = 1, \dots, n$.
- ▶ For every ℓ there are $n - \ell$ segments and $n - \ell - 1$ insertion points, so the neighbourhood has

$$\sum_{\ell=1}^3 (n - \ell)(n - \ell - 1)$$

elements, i.e., $O(n^2)$: 170 for $n = 10$.

- ▶ Some of these moves give the same tour (moving a single city forward by one position is the same as moving its successor backward by one position).

TSP: Or-opt and 3-opt

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

$P \parallel C_{\max}$: move and swap

- ▶ The **move** neighbourhood moves one job j from its machine $a = \text{mach}_j$ to another machine $h \neq a$.
- ▶ Every move is feasible, and the neighbourhood has $n(m-1)$ elements, examined for $j = 1, \dots, n$ and, for each j , for $h = 1, \dots, m$.
- ▶ The **swap** neighbourhood exchanges the machines of two jobs $j < k$ on different machines with $t_j \neq t_k$ (otherwise no load changes).
- ▶ It has at most $\frac{n(n-1)}{2}$ elements, examined for $j = 1, \dots, n$ and $k = j+1, \dots, n$.
- ▶ A swap never changes the number of jobs on a machine.

The neighbourhoods at a glance (1/2)

problem	neighbourhood	move	size
KP	flip	add or remove item j	n
KP	swap	item j out, item k in	$ S (n - S) \leq \frac{n^2}{4}$
BPP	move	item j to another bin	$\leq n(nb - 1)$
BPP	swap	exchange items j and k	$\leq \frac{n(n-1)}{2}$

The neighbourhoods at a glance (2/2)

problem	neighbourhood	move	size
TSP	2-opt	remove 2 edges, reverse a path	$\frac{n(n-3)}{2}$
TSP	Or-opt	move a segment of 1, 2 or 3 cities	$\sum_{\ell=1}^3 (n-\ell)(n-\ell-1)$
TSP	3-opt	remove 3 edges, reconnect	$O(n^3)$
$P \parallel C_{\max}$	move	job j to another machine	$n(m-1)$
$P \parallel C_{\max}$	swap	exchange jobs j and k	$\leq \frac{n(n-1)}{2}$

The size of a neighbourhood

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

Summary

The neighbourhoods:

- ▶ KP: flip and swap; BPP and $P \parallel C_{\max}$: move and swap; TSP: 2-opt, Or-opt and 3-opt
- ▶ Only feasible neighbours, examined in a fixed order

The TSP moves:

- ▶ 2-opt removes two edges and reverses a path: $\frac{n(n-3)}{2}$ moves
- ▶ Or-opt moves a segment of 1, 2 or 3 cities: $O(n^2)$ moves, part of 3-opt

The size:

- ▶ The number of neighbours to evaluate at every iteration

First improvement and best improvement

Introduction: First improvement and best improvement

- ▶ In this section we write the two rules that choose the improving neighbour
- ▶ We compare their cost per iteration and the local optima they reach
- ▶ We run both rules by hand on the KP reference instance

Two rules for *Improve*

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

First improvement

The *first improvement* rule examines the neighbours of the current solution in a fixed order and returns the first neighbour that is better than the current solution.

If no neighbour is better, it returns *NULL*.

Algorithm: First improvement

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

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

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

Best improvement

The *best improvement* rule examines all the neighbours of the current solution and returns the best one, if it is better than the current solution; ties are broken in favour of the neighbour examined first.

If no neighbour is better, it returns *NULL*.

Algorithm: Best improvement

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

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

```
1 function BestImprovement( $x, N, f$ )
2    $\bar{y} := \text{NULL}$  // best improving neighbour found so far
3    $\bar{f} := f(x)$  // its value
4   for  $y \in N(x)$ , in the order of the neighbourhood do
5     if  $f(y) < \bar{f}$  then
6        $\bar{y} \leftarrow y$  // better than all the neighbours seen so far
7        $\bar{f} \leftarrow f(y)$ 
8     end
9   end
10  return  $\bar{y}$  // NULL if  $x$  is a local optimum
11 end
```

Comparison (1/2)

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

Comparison (2/2)

- ▶ The two rules follow different paths, so they may stop in **different local optima**.
- ▶ Neither rule is better than the other on all instances: the steepest move now can lead into a worse valley later.
- ▶ With first improvement the result depends on the order in which the neighbours are examined; randomizing that order is a simple way to obtain different local optima from the same starting solution.

Example: first and best improvement on the KP (1/6)

- ▶ We start from the solution of the greedy by weight GR_w on the KP reference instance ($c = 9$), i.e.,

$$x = (0, 0, 1, 1),$$

of weight 6 and profit 10.

- ▶ We use the neighbourhood flip+swap.
- ▶ The neighbours are examined in the order fixed in the previous section: first the flips, then the swaps.

Example: first and best improvement on the KP (2/6)

The neighbours of $x = (0, 0, 1, 1)$:

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

Example: first and best improvement on the KP (3/6)

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

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

Example: first and best improvement on the KP (4/6)

- ▶ $(1, 1, 0, 0)$, of profit 17, is accepted after 5 feasible neighbours.
- ▶ It is the optimal solution, so no neighbour is better: the 6 feasible neighbours are examined and the search stops.
- ▶ In total, **first improvement** makes
2 moves, $3 + 5 + 6 = 14$ feasible neighbours examined.

Example: first and best improvement on the KP (5/6)

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

2 moves, $6 + 6 + 6 = 18$ feasible neighbours examined.

Example: first and best improvement on the KP (6/6)

- ▶ The two rules reach the optimum along different paths:

first improvement: $10 \rightarrow 14 \rightarrow 17$,

best improvement: $10 \rightarrow 16 \rightarrow 17$,

and first improvement examines fewer neighbours.

- ▶ The flip neighbourhood alone would not have moved at all: from $(0, 0, 1, 1)$ every feasible flip removes an item, so $(0, 0, 1, 1)$, of profit 10, is a local optimum for flip.

Summary

First improvement:

- ▶ Returns the first improving neighbour: cheaper iterations, usually more moves
- ▶ The result depends on the order of the neighbours

Best improvement:

- ▶ Examines the whole neighbourhood and returns the best neighbour: the steepest descent
- ▶ Ties go to the neighbour examined first

On the KP:

- ▶ Both rules reach 17 from 10, along different paths, with 14 and 18 neighbours examined

Evaluating a move incrementally

Introduction: Evaluating a move incrementally

- ▶ Most of the running time of a local search is spent evaluating neighbours
- ▶ In this section we compute the change of the objective from the few components that a move changes
- ▶ For the 2-opt move the change is computed in $O(1)$ time instead of $O(n)$

From scratch or incrementally

- ▶ Every iteration of *LS* evaluates many neighbours, and most of the running time is spent there.
- ▶ Computing $f(y)$ **from scratch** costs $O(n)$ for each of the four problems: the profit of a KP solution, the loads of the bins or of the machines, the length of a tour are sums of n terms.
- ▶ A move changes only a few components of the solution.
- ▶ The **incremental evaluation** computes the **change** $\Delta = f(y) - f(x)$ from the components that change, keeping some information about the current solution (its profit and weight, its loads), without recomputing $f(y)$.

Proposition 2

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

$$\Delta = d_{u_1 u_2} + d_{v_1 v_2} - d_{u_1 v_1} - d_{u_2 v_2}.$$

Proof

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



The change of length of a 2-opt move

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

$$\Delta = \underbrace{d_{u_1 u_2} + d_{v_1 v_2}}_{\text{added edges}} - \underbrace{d_{u_1 v_1} + d_{u_2 v_2}}_{\text{removed edges}}$$

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

Example

- ▶ On the tour $(1, 3, 2, 4, 5)$ of the reference instance, of length 25, the move with positions $i = 2, j = 4$ removes $\{3, 2\}$ and $\{4, 5\}$ and adds $\{3, 4\}$ and $\{2, 5\}$:

$$\Delta = d_{34} + d_{25} - d_{32} - d_{45} = 4 + 4 - 4 - 7 = -3.$$

- ▶ The new tour $(1, 3, 4, 2, 5)$ has length $25 - 3 = 22$, as a direct computation confirms:

$$2 + 4 + 3 + 4 + 9 = 22.$$

Why it matters

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

The other neighbourhoods

The same idea applies to the other neighbourhoods:

move	change of the solution	time
KP, flip of j	profit $\pm p_j$, weight $\pm w_j$	$O(1)$
KP, swap j out, k in	profit $+p_k - p_j$, weight $+w_k - w_j$	$O(1)$
BPP, move j from a to b	$L_a - w_j$, $L_b + w_j$; bin a is emptied iff $L_a = w_j$	$O(1)$
TSP, 2-opt	$d_{u_1 u_2} + d_{v_1 v_2} - d_{u_1 v_1} - d_{u_2 v_2}$	$O(1)$
TSP, Or-opt	$d_{pq} + d_{as_1} + d_{s_\ell b} - d_{ps_1} - d_{s_\ell q} - d_{ab}$	$O(1)$
$P \parallel C_{\max}$, move j from a to h	$L_a - t_j$, $L_h + t_j$; $C_{\max} =$ $\max\{L_a - t_j, L_h + t_j, \max_{q \neq a, h} L_q\}$	$O(m)$

The cost of an iteration

- ▶ The Or-opt formula follows as for 2-opt: the move removes the edges $\{p, s_1\}$, $\{s_\ell, q\}$, $\{a, b\}$ and adds $\{p, q\}$, $\{a, s_1\}$, $\{s_\ell, b\}$, and the segment keeps its orientation.
- ▶ The cost of an iteration of *LS* is the size of the neighbourhood times the cost of evaluating a move (plus the cost of applying the accepted move, which is paid once per iteration):

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

Which moves, and how fast?

- ▶ The incremental evaluation does not change the local optima, nor the path followed by the search: it changes only the time needed to follow it.
- ▶ This is why the design of a local search always starts with two questions:

which moves? and how fast can a move be evaluated?

Summary

Incremental evaluation:

- ▶ Compute $\Delta = f(y) - f(x)$ from the components that change, not $f(y)$ from scratch
- ▶ 2-opt: $\Delta = d_{u_1 u_2} + d_{v_1 v_2} - d_{u_1 v_1} - d_{u_2 v_2}$, in $O(1)$; improving iff $\Delta < 0$

Cost of an iteration:

- ▶ Size of the neighbourhood times the cost of evaluating a move
- ▶ Same local optima and same path, in less time

TSP: 2-opt and Or-opt



Introduction: TSP: 2-opt and Or-opt

- ▶ In this section we write the 2-opt and the Or-opt local searches for the TSP, with the incremental evaluation
- ▶ We run them on the reference instance and on a medium instance with 10 cities
- ▶ We see that the two rules, and the two neighbourhoods, can stop in different local optima

The 2-opt local search

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

Algorithm: 2-opt step (1/3)

Input : n cities, distances $d[1..n][1..n]$, tour $\sigma[1..n]$ with $\sigma[1] = 1$, rule $\in \{\textit{first}, \textit{best}\}$

Output : *true* if an improving 2-opt move was applied to σ ,
false otherwise

```
1 function TwoOptStep( $n, d, \sigma, \textit{rule}$ )
2    $\bar{\Delta} := 0$            // best length change found so far
3    $(\bar{i}, \bar{j}) := (0, 0)$        // best move found so far
4   for  $i := 1$  to  $n - 2$  do
5     for  $j := i + 2$  to  $n$  do
6       if  $i = 1$  and  $j = n$  then continue    // the two
       edges are adjacent
```

Algorithm: 2-opt step (2/3)

function *TwoOptStep*($n, d, \sigma, rule$) (continued)

```
    for  $i := 1$  to  $n - 2$  (continued) do
      for  $j := i + 2$  to  $n$  (continued) do
7          $u_1 := \sigma[i]; v_1 := \sigma[i + 1]$       // first removed
           edge
8          $u_2 := \sigma[j]; v_2 := \sigma[j \bmod n + 1]$   // second
           removed edge
9          $\Delta := d[u_1][u_2] + d[v_1][v_2] - d[u_1][v_1] - d[u_2][v_2]$ 
10        if  $\Delta < \bar{\Delta}$  then
11             $\bar{\Delta} \leftarrow \Delta; (\bar{i}, \bar{j}) \leftarrow (i, j)$   // improving move
12            if  $rule = first$  then
13                leave both loops
14            end
15        end
16    end
17 end
```

Algorithm: 2-opt step (3/3)

function *TwoOptStep*($n, d, \sigma, rule$) (continued)

```
18 |   if  $\bar{i} = 0$  then return false           // no improving move:  
    |   local optimum  
19 |   reverse the path  $\sigma[\bar{i} + 1], \dots, \sigma[\bar{j}]$    // apply the move  
20 |   return true  
21 end
```

How 2OPT works

- ▶ In the implementation, *Improve* and the move to the improving neighbour are merged in a single function *Step*:
 - ▶ it explores the neighbourhood with the incremental evaluation;
 - ▶ if it finds an improving move, it applies it to the current solution and returns *true*;
 - ▶ otherwise it returns *false*.
- ▶ *LS* calls *Step* until it returns *false*.
- ▶ 2OPT is *LocalSearch* with *TwoOptStep*; it never changes $\sigma[1] = 1$, since $i \geq 1$ and the reversed path starts at position $i + 1 \geq 2$.
- ▶ An iteration costs $O(n^2)$ time; the reversal of the path costs $O(n)$.

2-opt uncrosses the edges

- ▶ When the distances are Euclidean, a 2-opt local optimum has no crossing edges.
- ▶ If the edges $\{u_1, v_1\}$ and $\{u_2, v_2\}$ cross at a point X , the triangle inequality gives

$$d_{u_1 u_2} \leq d_{u_1 X} + d_{X u_2} \quad \text{and} \quad d_{v_1 v_2} \leq d_{v_1 X} + d_{X v_2},$$

so $\Delta \leq 0$, with $\Delta < 0$ unless the four cities are on a line.

- ▶ The 2-opt move “uncrosses” the two edges.

Example: $2OPT$ on the TSP reference instance (1/6)

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

(i, j)	removed edges	added edges	Δ
$(1, 3)$	$\{1, 3\}, \{2, 4\}$	$\{1, 2\}, \{3, 4\}$	$5 + 4 - 2 - 3 = +4$
$(1, 4)$	$\{1, 3\}, \{4, 5\}$	$\{1, 4\}, \{3, 5\}$	$2 + 7 - 2 - 7 = 0$
$(2, 4)$	$\{3, 2\}, \{4, 5\}$	$\{3, 4\}, \{2, 5\}$	$4 + 4 - 4 - 7 = -3$
$(2, 5)$	$\{3, 2\}, \{5, 1\}$	$\{3, 5\}, \{2, 1\}$	$7 + 5 - 4 - 9 = -1$
$(3, 5)$	$\{2, 4\}, \{5, 1\}$	$\{2, 5\}, \{4, 1\}$	$4 + 2 - 3 - 9 = -6$

Example: 2OPT on the TSP reference instance (2/6)

The new tours given by the 5 moves of $\sigma = (1, 3, 2, 4, 5)$:

(i, j)	Δ	new tour, length
(1, 3)	+4	(1, 2, 3, 4, 5), 29
(1, 4)	0	(1, 4, 2, 3, 5), 25
(2, 4)	-3	(1, 3, 4, 2, 5), 22
(2, 5)	-1	(1, 3, 5, 4, 2), 24
(3, 5)	-6	(1, 3, 2, 5, 4), 19

Example: $2OPT$ on the TSP reference instance (3/6)

First improvement:

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

Example: *2OPT* on the TSP reference instance (4/6)

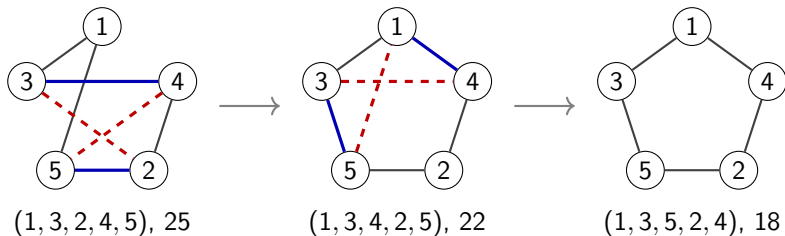
Best improvement:

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

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

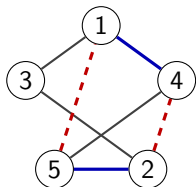
Example: $2OPT$ on the TSP reference instance (5/6)

first improvement

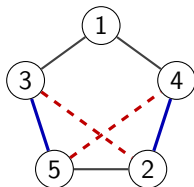


Example: $2OPT$ on the TSP reference instance (6/6)

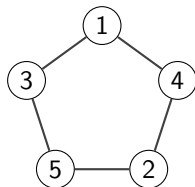
best improvement



(1, 3, 2, 4, 5), 25



(1, 3, 2, 5, 4), 19



(1, 3, 5, 2, 4), 18

The Or-opt local search

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

Algorithm: Or-opt step (1/3)

Input : n cities, distances $d[1..n][1..n]$, tour $\sigma[1..n]$ with $\sigma[1] = 1$, rule $\in \{\text{first}, \text{best}\}$

Output : *true* if an improving Or-opt move was applied to σ ,
false otherwise

```
1 function OrOptStep( $n, d, \sigma, \text{rule}$ )
2    $\bar{\Delta} := 0; (\bar{\ell}, \bar{i}, \bar{k}) := (0, 0, 0)$  // best move found so far
3   for  $\ell := 1$  to 3 do
4     for  $i := 2$  to  $n - \ell + 1$  do
5        $p := \sigma[i - 1]; s_1 := \sigma[i]$  // predecessor, first city of the segment
6        $s_\ell := \sigma[i + \ell - 1]; q := \sigma[(i + \ell - 1) \bmod n + 1]$ 
7       // last city, successor
8       for  $k := 1$  to  $n$  do
9         if  $i - 1 \leq k \leq i + \ell - 1$  then continue
10        // same place, or inside the segment
```

Algorithm: Or-opt step (2/3)

function *OrOptStep*($n, d, \sigma, rule$) (continued)

for $\ell := 1$ **to** 3 (continued) **do**

for $i := 2$ **to** $n - \ell + 1$ (continued) **do**

for $k := 1$ **to** n (continued) **do**

$a := \sigma[k]; b := \sigma[k \bmod n + 1]$ // edge

 where the segment goes

$\Delta := d[p][q] + d[a][s_1] + d[s_\ell][b] - d[p][s_1] -$

$d[s_\ell][q] - d[a][b]$

if $\Delta < \bar{\Delta}$ **then**

$\bar{\Delta} \leftarrow \Delta; (\bar{\ell}, \bar{i}, \bar{k}) \leftarrow (\ell, i, k)$

 // improving move

if $rule = first$ **then**

leave the three loops

end

end

end

9

10

11

12

13

14

15

16

17

Algorithm: Or-opt step (3/3)

function *OrOptStep*($n, d, \sigma, rule$) (continued)

for $\ell := 1$ **to** 3 (continued) **do**

for $i := 2$ **to** $n - \ell + 1$ (continued) **do**

end

end

if $\bar{\ell} = 0$ **then return** *false* // no improving move:

 local optimum

 move $\sigma[\bar{i}], \dots, \sigma[\bar{i} + \bar{\ell} - 1]$ between $\sigma[\bar{k}]$ and

$\sigma[\bar{k} \bmod n + 1]$ // apply

return *true*

end

18

19

20

21

22

23

How *OROPT* works

- ▶ Or-opt moves a short piece of the tour to a better place, without reversing anything.
- ▶ It is the natural neighbourhood when a few cities have been “left behind” by the construction.
- ▶ An iteration costs $O(n^2)$ time.
- ▶ A 2-opt local optimum need not be an Or-opt local optimum, and vice versa: the two neighbourhoods contain different moves.

First and best improvement on a medium instance

- ▶ On 5 cities the 2-opt neighbourhood is small and both rules reach the optimum.
- ▶ To see the difference between the two rules, and the role of Or-opt, we use a medium instance with 10 cities.
- ▶ This instance is not metric:

$$\begin{aligned}d_{68} &= 8 > d_{67} + d_{78} = 4 + 3, \\d_{6,10} &= 11 > d_{67} + d_{7,10} = 4 + 6.\end{aligned}$$

Example: a TSP instance with 10 cities (1/7)

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

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

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

Example: a TSP instance with 10 cities (2/7)

The distances:

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

Example: a TSP instance with 10 cities (3/7)

- ▶ Among its 181 440 distinct tours, the unique optimal one is

$$(1, 10, 2, 3, 9, 4, 5, 6, 7, 8), \quad z(I) = 39.$$

- ▶ The tour of *NN* from city 1 is

$$(1, 10, 2, 3, 4, 9, 7, 8, 5, 6), \quad \text{of length } 49.$$

- ▶ The 2-opt neighbourhood has $\frac{10 \cdot 7}{2} = 35$ moves.

Example: a TSP instance with 10 cities (4/7)

First improvement makes 4 moves:

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

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

Example: a TSP instance with 10 cities (5/7)

Best improvement makes 3 moves:

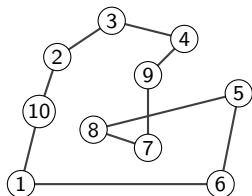
removed edges	new tour	Δ	length
$\{7, 8\}, \{6, 1\}$	(1, 10, 2, 3, 4, 9, 7, 6, 5, 8)	-5	44
$\{9, 7\}, \{5, 8\}$	(1, 10, 2, 3, 4, 9, 5, 6, 7, 8)	-4	40
$\{3, 4\}, \{9, 5\}$	(1, 10, 2, 3, 9, 4, 5, 6, 7, 8)	-1	39

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

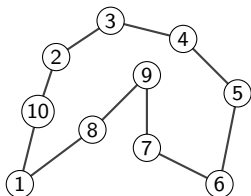
Example: a TSP instance with 10 cities (6/7)

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

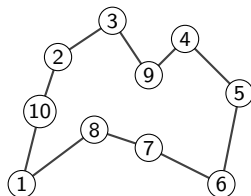
Example: a TSP instance with 10 cities (7/7)



NN: 49



first improvement: 41



best improvement: 39

- The starting tour of NN and the two local optima of 2-opt, with first and best improvement.

Example: *OROPT* from a 2-opt local optimum (1/2)

- ▶ On the instance with 10 cities, first improvement stopped in the 2-opt local optimum

$$\sigma = (1, 10, 2, 3, 4, 5, 6, 7, 9, 8), \quad \text{of length 41.}$$

- ▶ The Or-opt move that takes the segment (9), with $\ell = 1$, between $p = 7$ and $q = 8$, and inserts it between $a = 3$ and $b = 4$ has

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

Example: *OROPT* from a 2-opt local optimum (2/2)

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

3-opt and Lin–Kernighan (1/2)

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

3-opt and Lin–Kernighan (2/2)

- ▶ The **Lin–Kernighan** heuristic (Lin, Kernighan, *Operations Research* 21, 1973) builds a *variable* number k of edge exchanges, one at a time, choosing every exchange greedily and stopping when the total gain cannot become positive.
- ▶ It explores part of the k -opt neighbourhood for every k , in a time close to that of 2-opt, and its implementations are among the best heuristics for the TSP.
- ▶ The sequence 2-opt, 3-opt, k -opt shows the compromise of the chapter «Heuristic algorithms: definitions»: a larger neighbourhood has better local optima, and costs more to explore.

Summary (1/2)

2-opt:

- ▶ *TwoOptStep* evaluates the $\frac{n(n-3)}{2}$ moves in $O(1)$ each and reverses one path
- ▶ In the Euclidean case it uncrosses the edges
- ▶ On the reference instance both rules go from 25 to the optimum 18 in 2 moves

Or-opt:

- ▶ *OrOptStep* moves a segment of 1, 2 or 3 cities, without reversing it
- ▶ Its local optima differ from those of 2-opt

Summary (2/2)

The instance with 10 cities:

- ▶ From NN (49): first improvement stops at 41, best improvement at the optimum 39
- ▶ Or-opt improves the 2-opt local optimum 41 to the optimum 39
- ▶ Only the optimal tour is a local optimum for both neighbourhoods

Larger neighbourhoods:

- ▶ 3-opt and Lin-Kernighan: better local optima, more expensive exploration

Summary of the first part (1/2)

- ▶ 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.
- ▶ Each problem has its neighbourhoods: flip and swap for the KP, move and swap for the BPP and $P \parallel C_{\max}$, 2-opt and Or-opt for the TSP.
- ▶ First improvement accepts the first improving neighbour, best improvement the best one: they can stop in different local optima.

Summary of the first part (2/2)

- ▶ The incremental evaluation computes the change Δ of a move from the components that change: $O(1)$ for a 2-opt move.
- ▶ On the TSP, 2-opt and Or-opt reach the optimum of the reference instance; on the instance with 10 cities the rule and the neighbourhood decide which local optimum is reached.
- ▶ In the second part: $P || C_{\max}$, plateaus and auxiliary objectives, penalties for the KP, variable neighbourhood descent and multi-start.