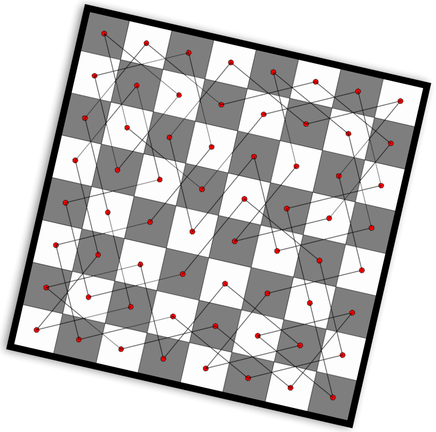
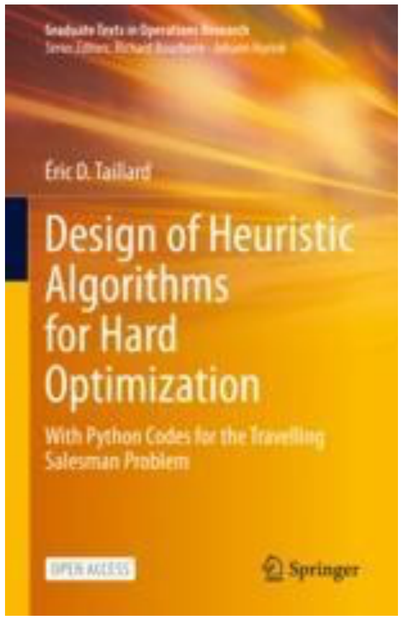
Advanced Algorithms for Optimization

Local Search

Pierre Schaus



I use many figures and slides from this book dedicated to the subject of LS



Greedy Algorithms

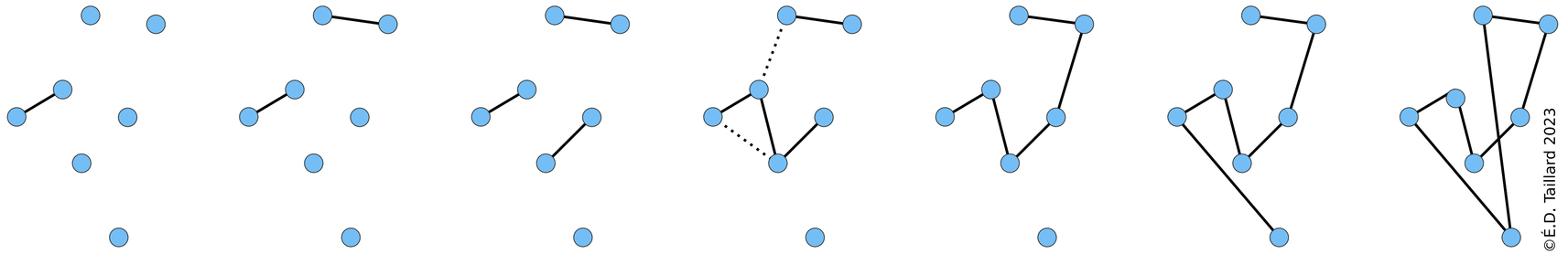
- Construct a solution, element by element
- Add the element that seems most appropriate at each step
- Do not question a choice

Input: A trivial partial solution s (generally $\varnothing$); set E of elements constituting a solution; incremental cost function $c(s, e)$
Result: Complete solution s
$R \leftarrow E$ `// Elements that can be added to` s
while $R \neq \varnothing$ **do**
  $\forall e \in R$, compute $c(s, e)$
  Choose e' optimizing $c(s, e')$
  $s \leftarrow s \cup e'$ `// Include` e' `in the partial solution` s
  Remove from R the elements that cannot be added any more to s

---

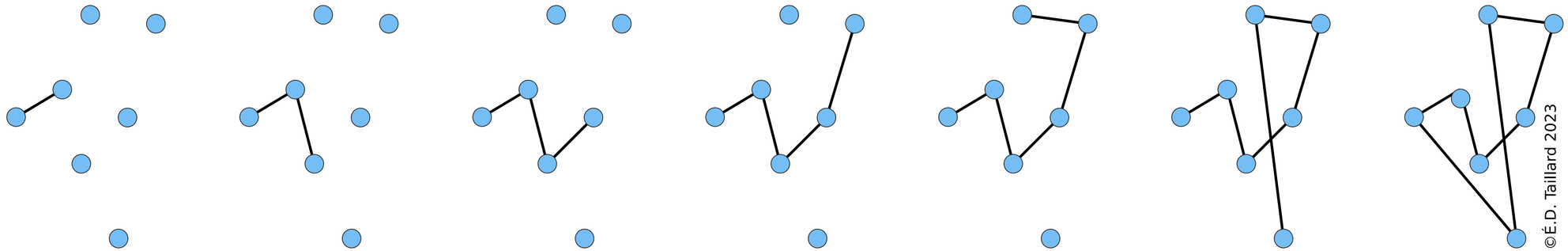
Greedy on the edge Weight

- Element e to be added: an edge
- Incremental cost: edge cost e
- Elements to remove: do not create degree 3 vertices or sub-cycles
- Choose the cheapest edge (not creating a sub-tour nor degree 3)



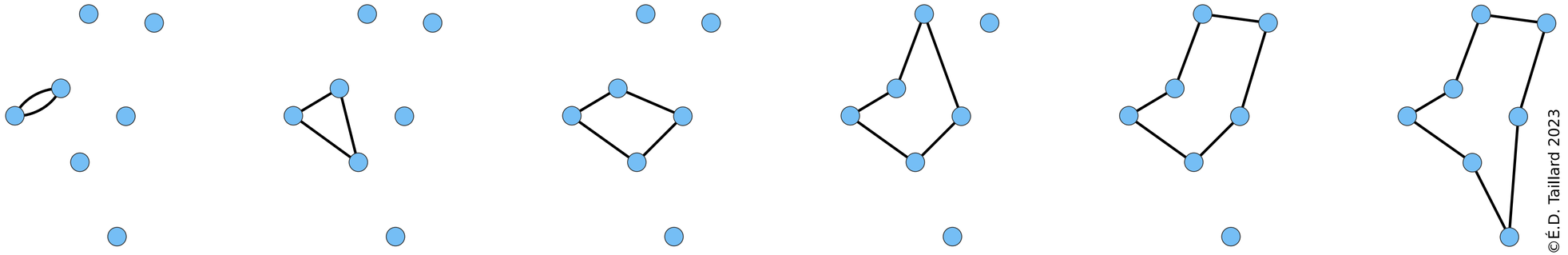
Greedy: Nearest Neighbor

- Start from any city
- Element e to be added: a city
- Incremental cost: edge cost to reach e from the last added city
- Choose the city the nearest to the last city of the partial tour



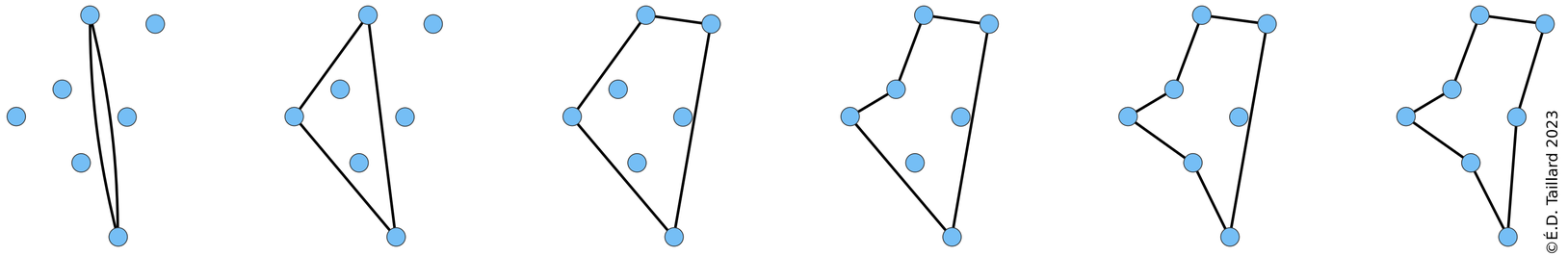
Greedy: Cheapest Insertion

- Start from a 2-city tour
- Element e to be added: a city
- Incremental cost: cost of a minimum detour to add e to the partial tour
- Choose the city with the lowest incremental cost



Greedy: Farthest Insertion

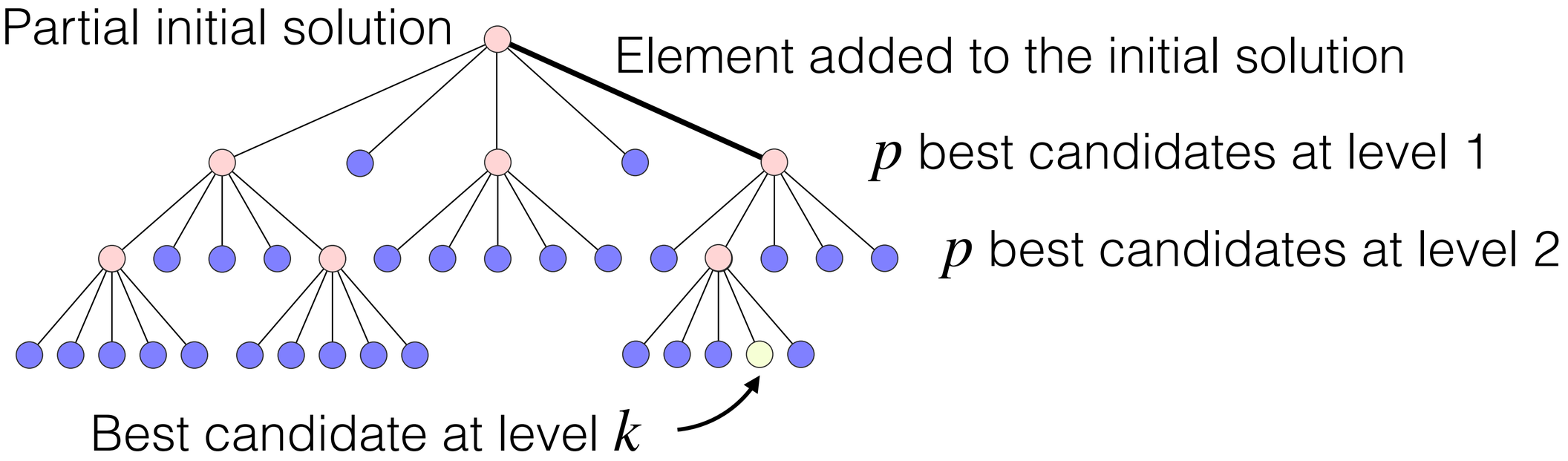
- Start from a 2-city tour
- Element e to be added: a city
- Incremental cost: cost of a minimum detour to add e to the partial tour
- Choose the city with the largest incremental cost



Beam Search (make greedy, less greedy and less myopic)

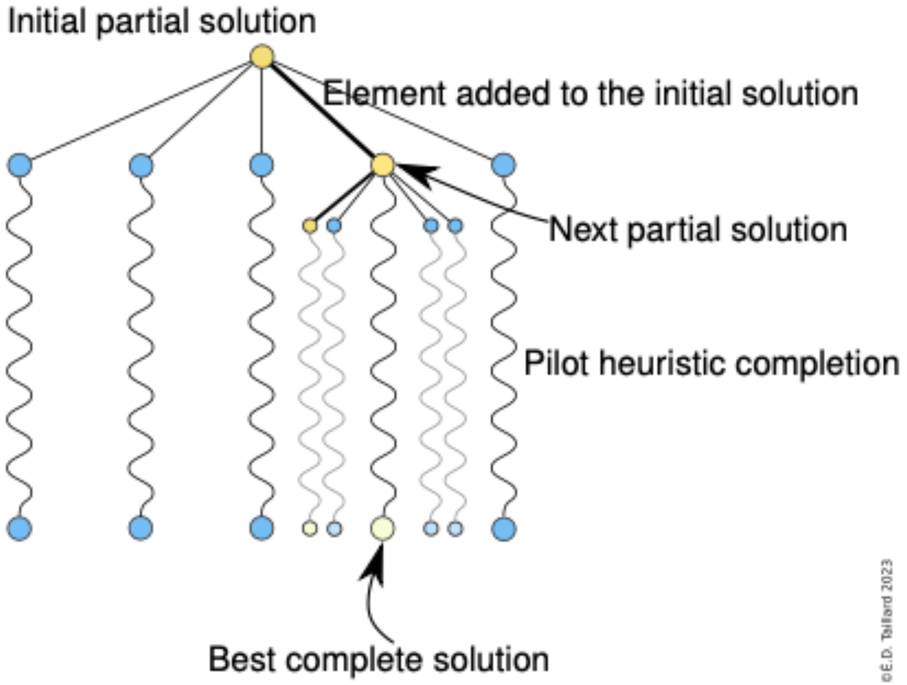
- Two parameters: p (beam width) and k (lookahead depth)

- Sort of BFS controlling the exponential growth
  - Select only the p best candidates at each level
  - Stop k levels below the last added element
  - Add the element that results in the best partial solution at the last level evaluated



Pilot Method

- Try all possibilities to complete a partial solution with one element, then launch a heuristic method (pilot)
  - The pilot heuristic starts with various partial solutions
  - Run the pilot heuristic until complete solutions are obtained
  - The element to be added to the partial solution is the one that led to the best complete solution



Pilot Algorithm

Input: s_p trivial partial solution; set E of elements constituting a solution; pilot heuristic $h(s_e)$ for completing a partial solution s_e; fitness function $f(s)$

Result: Complete solution $s*$

$R \leftarrow E$ `// Elements that can be added to` s

while $R \neq \varnothing$ **do**

  $v \leftarrow \infty$

  forall $e \in R$ **do**

    Complete s_p with e to get s_e

    Apply $h(s_e)$ to get a complete solution s

    if $f(s) \leqslant v$ **then**

      $v \leftarrow f(s)$

      $s_r \leftarrow s_e$

      if s *is better than* $s*$ **then** Store the improved solution

        $s* \leftarrow s$

      end

    end

  end

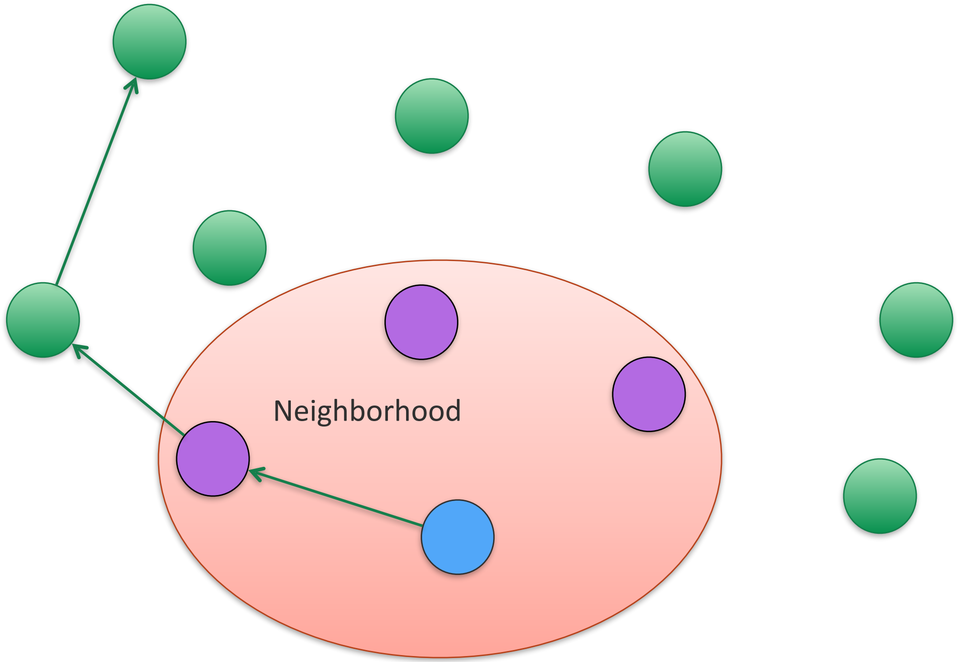
  $s_p \leftarrow s_r$ `// Add an element to the partial solution` s_p

  Remove from R the elements that cannot properly be added to s_p

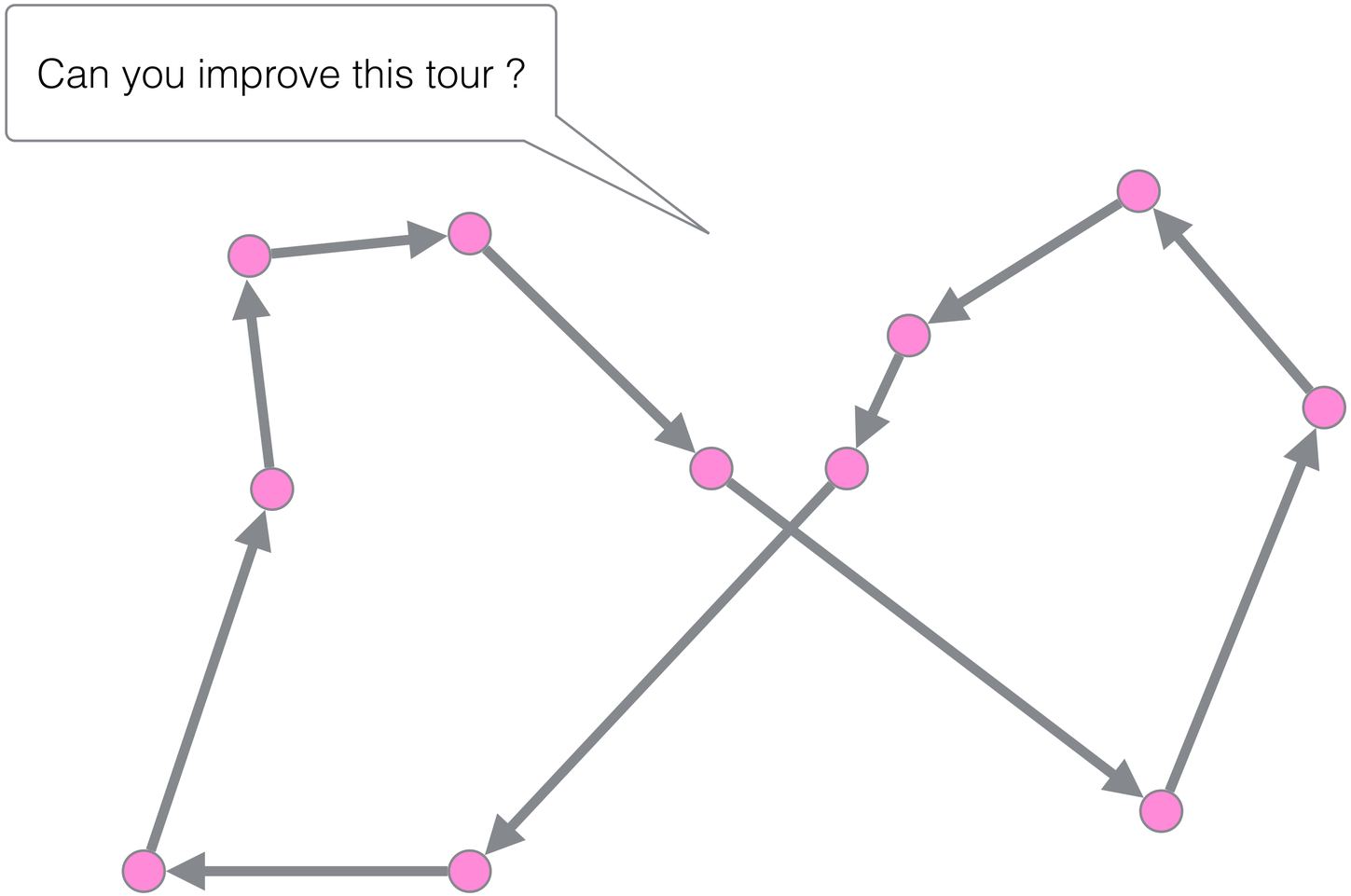
end

Improving the solution = exploring a neighborhood

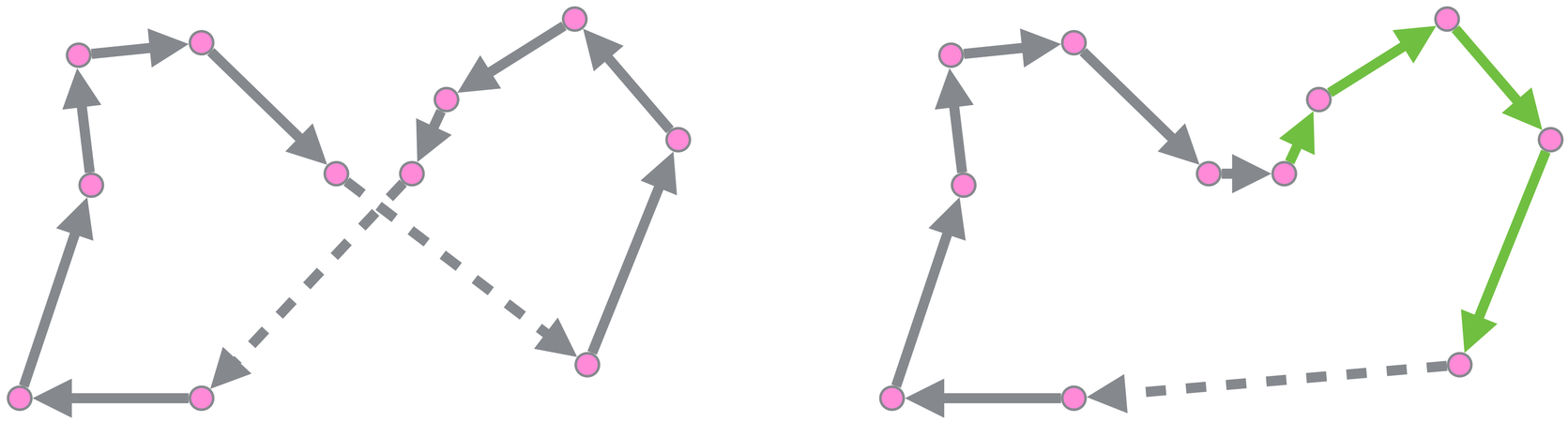
keep a single current state and move to neighbouring states to improve it



TSP Neighborhood: Intuition first



Echange two edges = 2 Opt Move



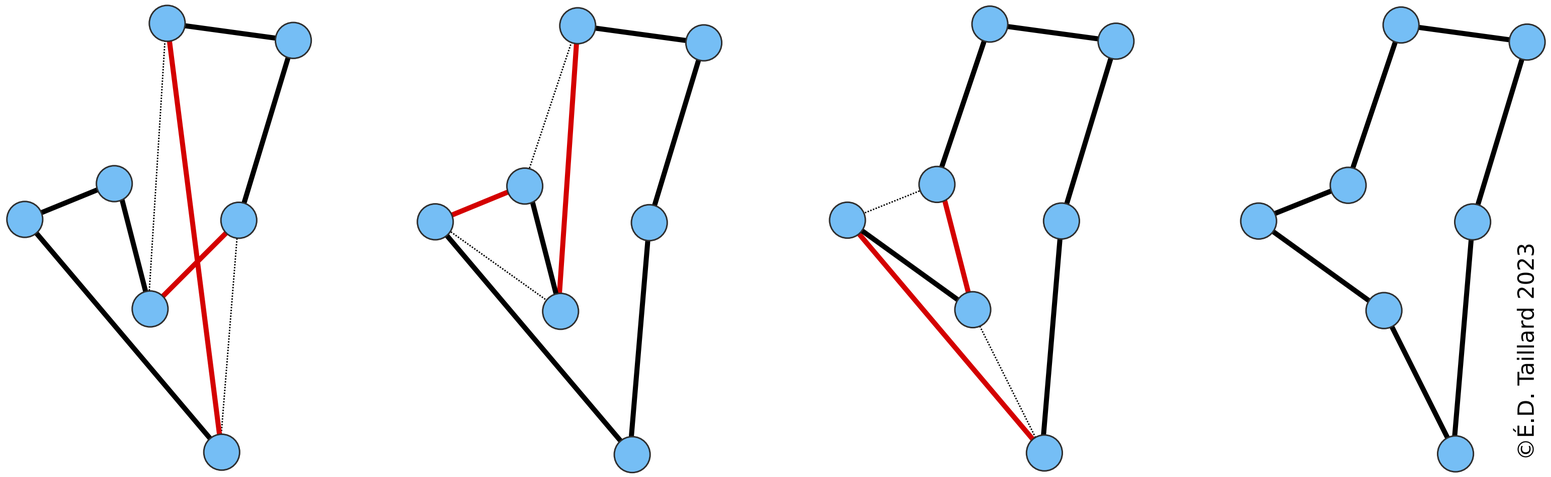
Disconnect by removing two edges, and re-construct the tour.

Formally: $N(s) = \{[i,j] \mid i,j \in s, i \neq j, j \neq s_i, i \neq s_j\}$

Observation: In an Euclidian TSP the optimal solution cannot have crossing edges. When no 2-Opt improving moves are possible, you are in a situation with no crossing edges (already not trivial solution to improve by human).

Local Search is

- Start from a solution obtained with constructive method, then
- Modify it locally to improve it
- Example:



Local Search

Input: Solution s, method modifying a solution
Result: Improved solution s
repeat
  if *there is a modification of* s *into* s' *improving* s **then**
    $s \leftarrow s'$
until *no improvement of* s *is found*

Two simple strategies for computing improving moves

- First improving move
- Best improving move

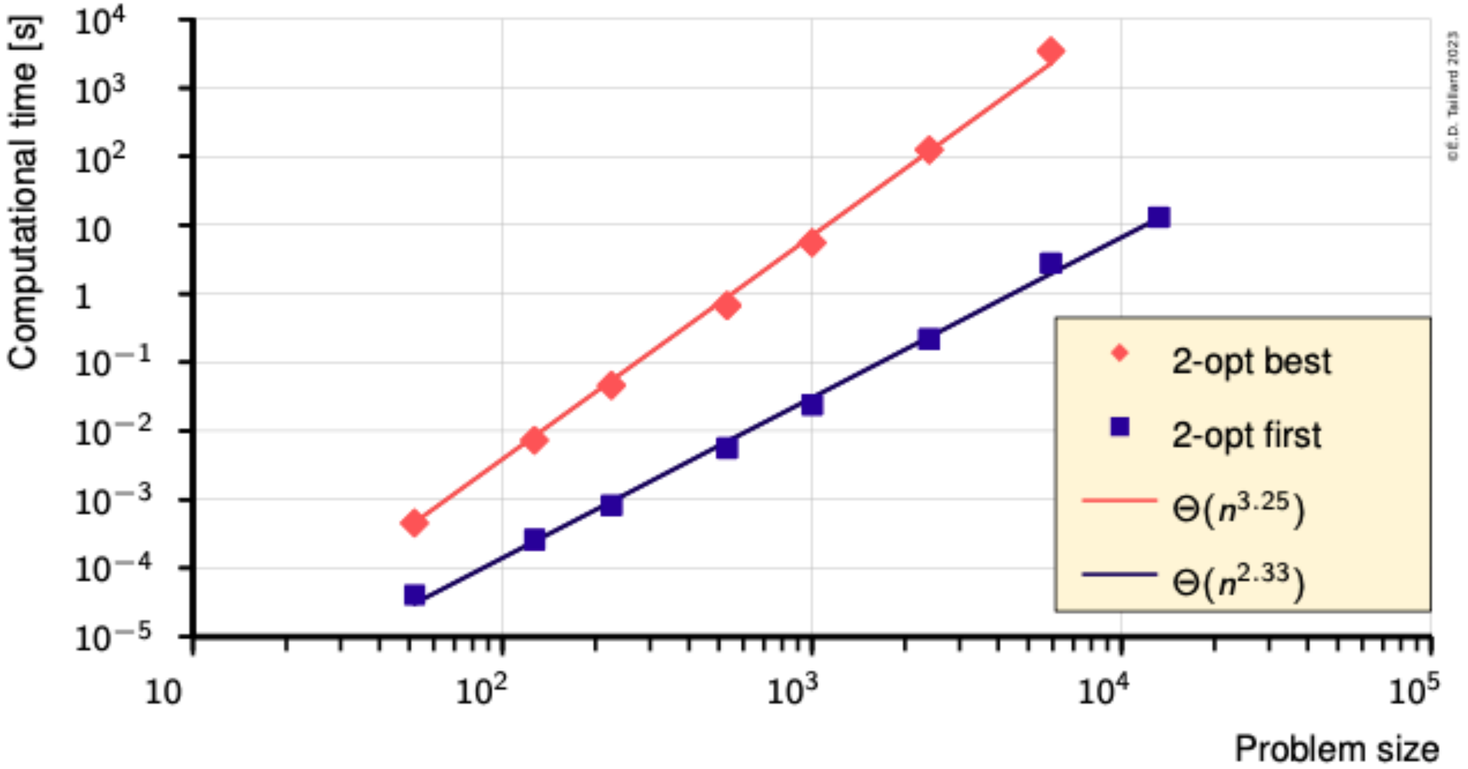
First Improving Move

Input: Solution s, neighbourhood specification $N(\cdot)$, fitness function $f(\cdot)$ to minimize.
Result: Improved solution s
forall $s' \in N(s)$ **do**
  if $f(s') < f(s)$ **then** Move to s', break the loop and initiate the next one
    $s \leftarrow s'$

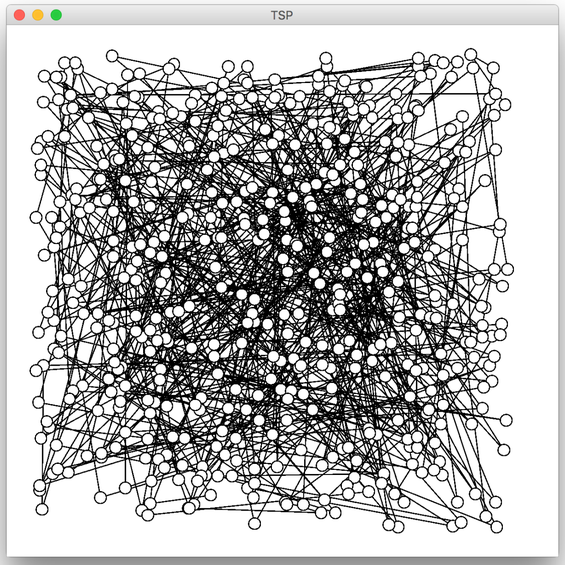
Best Improving Move

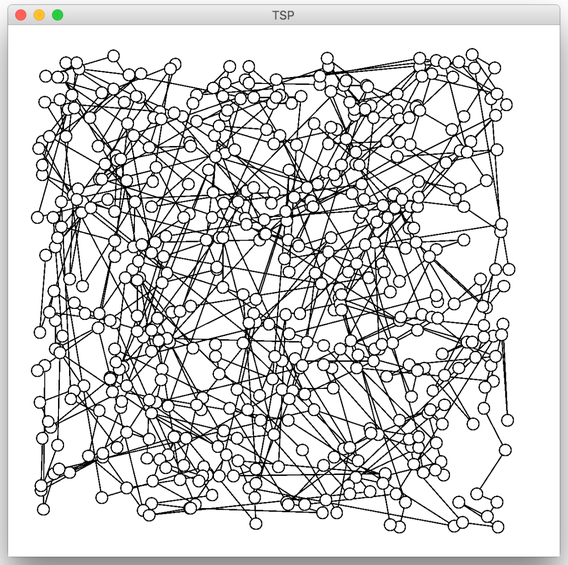
Input: Solution s, neighbourhood specification $N(\cdot)$, fitness function $f(\cdot)$ to minimize.
Result: Improved solution s
repeat
  $end \leftarrow$ true
  $best\_neighbour\_value \leftarrow \infty$
  forall $s' \in N(s)$ **do**
    if $f(s') < best\_neighbour\_value$ **then** A better neighbour is found
      $best\_neighbour\_value \leftarrow f(s')$
      $best\_neighbour \leftarrow s'$
  if $best\_neighbour\_value < f(s)$ **then** Move to the improved solution
    $s \leftarrow best\_neighbour$
    $end \leftarrow$ false
until end

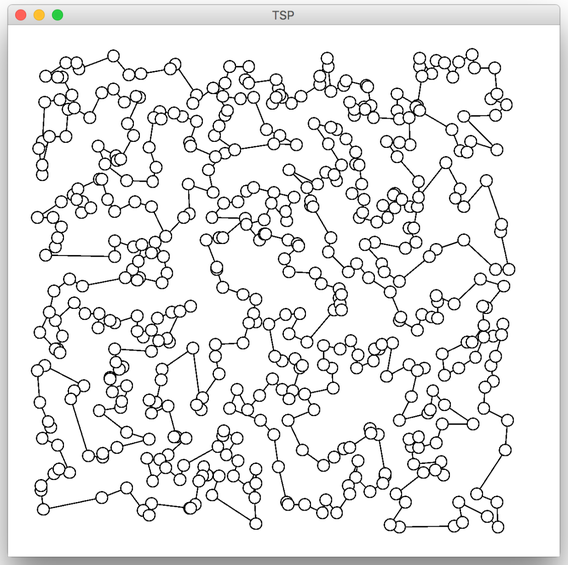
Empirical Complexity of 2-Opt for the two strategies



2-Opt Evolution on Random 2D Euclidian TSP







Connected-Neighborhood

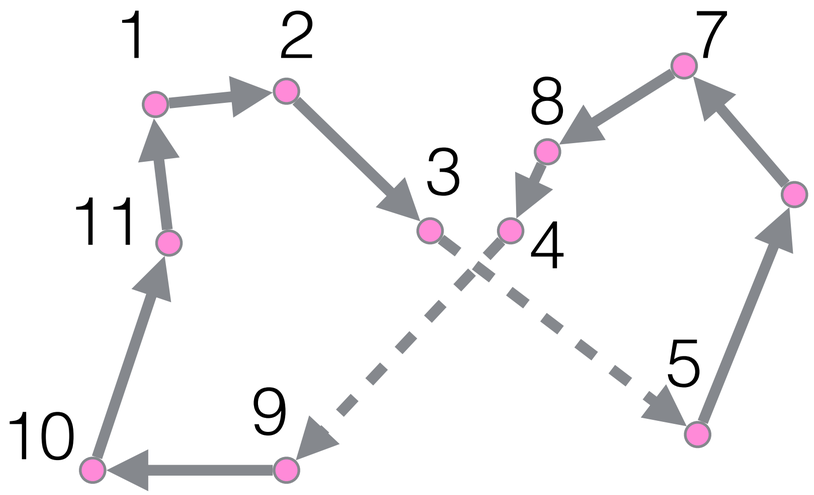
- A neighborhood is connected if and only if for each solution s, there exists a path to an optimal solution s*.

- Two advantages:
  - You don’t necessarily need a restarting strategy
  - Randomized heuristics where there is a non zero probability of accepting a neighbor $k \in N(s)$ for each solution s, may be guaranteed to reach a global optimum (example: simulated annealing).

- To prove a neighborhood is connected, you must provide an algorithm to transform any solution s1 into a solution s2 by selecting the moves allowed by the neighborhood.

Is 2-Opt a connected neighborhood?

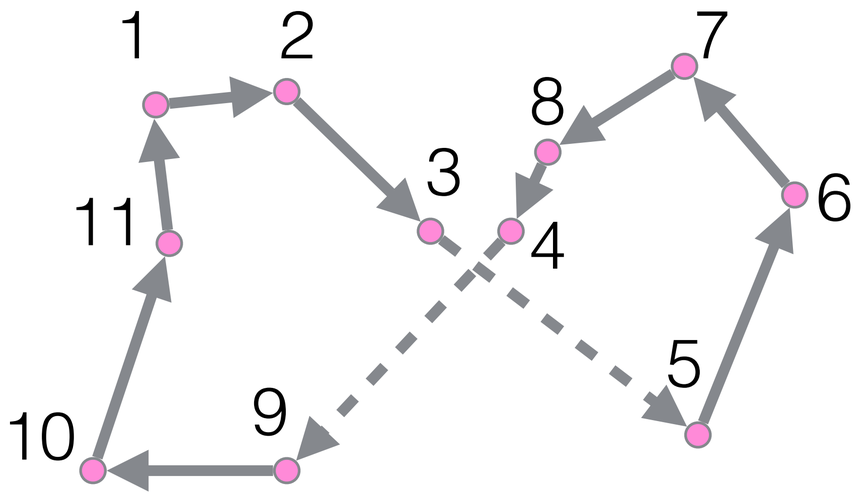
- Is the 2Opt move a connected neighborhood for the TSP?

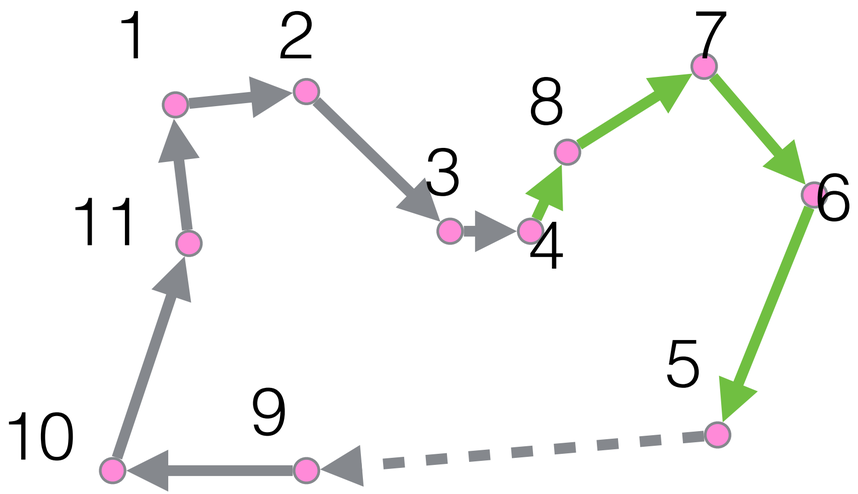


1,2,3,5,6,7,8,4,9,10,11

Implementation: The candidate solution is represented as a permutation array

2-Opt and connectivity





1,2,3,5,6,7,8,4,9,10,11

1,2,3,4,8,7,6,5,9,10,11

A 2-Opt move amounts at « reversing » a subsequence of the tour

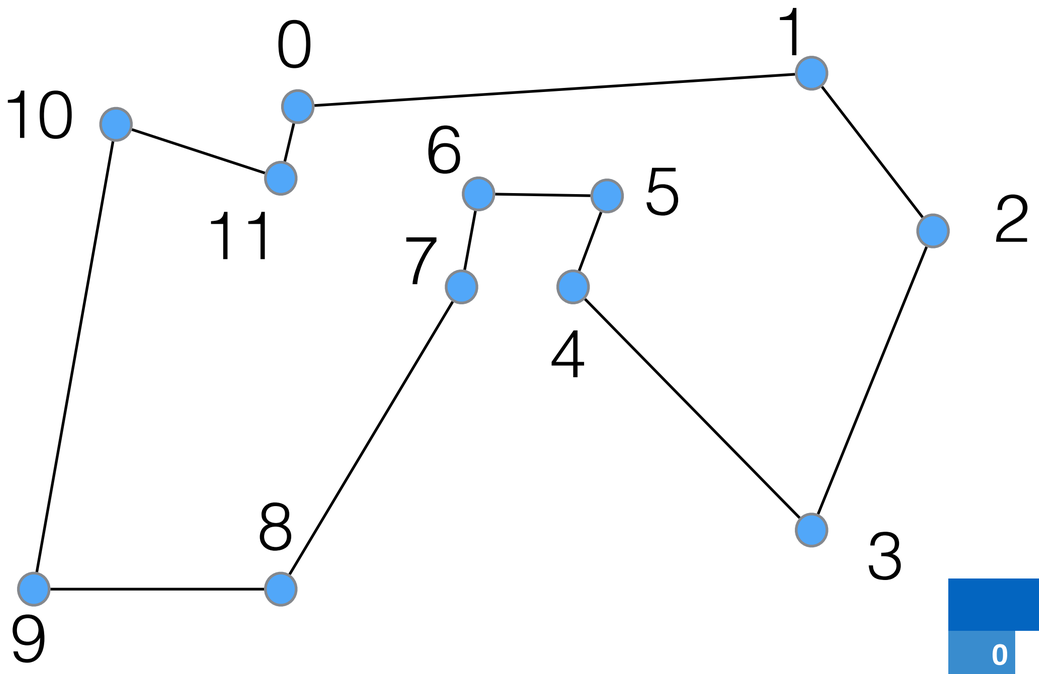
3,1,2,4,7,6,5,8,9,11,10

assume this is the optimal solution s*, given a current solution s, can you find a sequence of 2-opt move to transform s into s* ?

Other properties to consider when designing a neighborhood

- Connectivity: Any feasible solution can reach at least one globally optimal solution
- Low Diameter: Not too many steps are needed to connect any 2 solutions ($O(n)$ for 2-Opt)
- Low Ruggedness: Fitness function value does not change too much from one solution to its neighbor, few local optima (difficult for asymmetric TSP, because you reverse a subtler)
- Small Size: Not too many neighbor solutions should be evaluated ($O(n^2)$ for 2-Opt)
- Fast Evaluation: Fitness function evaluation of a neighbor solution must be fast (($O(1)$ for for getting the delta but $O(n)$ applying the move, although some special data-structures exits to do in $O(1)$)
- Unfortunately the ideal properties of a neighborhood are often conflicting

Working example



totalCost = 2056

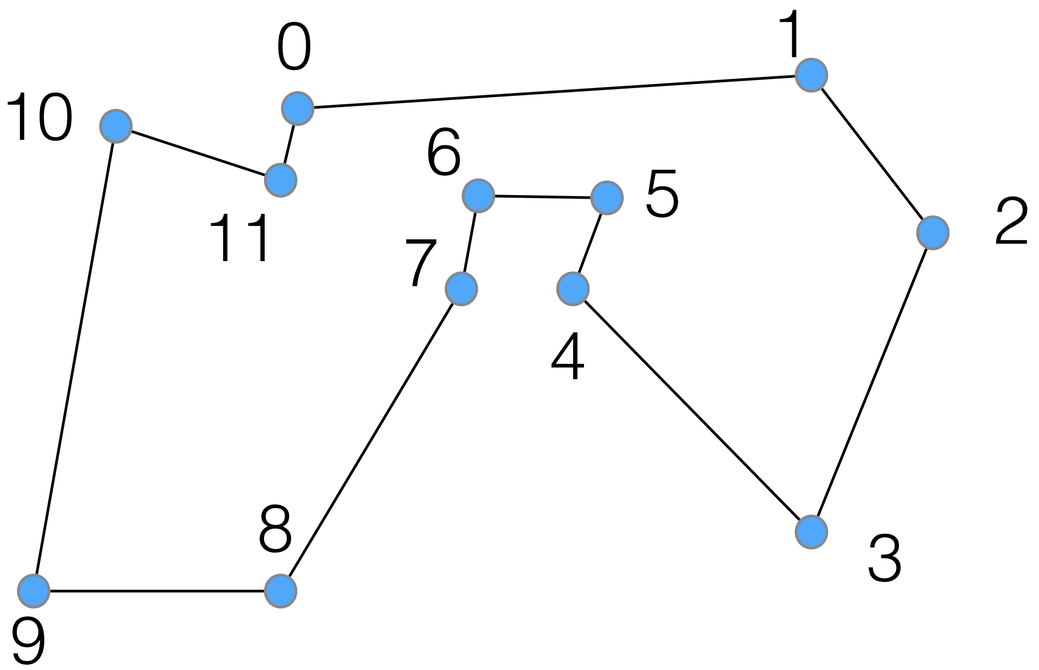
Symmetric
Distance Matrix

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	355	444	453	225	221	136	166	326	372	118	45
1	355	0	135	311	212	160	242	278	509	636	473	365
2	444	135	0	219	243	223	312	321	511	658	559	441
3	453	311	219	0	232	269	324	288	367	527	546	429
4	225	212	243	232	0	63	92	78	297	426	330	210
5	221	160	223	269	63	0	90	118	356	476	336	219
6	136	242	312	324	92	90	0	66	306	407	247	129
7	166	278	321	288	78	118	66	0	244	358	260	141
8	326	509	511	367	297	356	306	244	0	160	335	281
9	372	636	658	527	426	476	407	358	160	0	327	331
10	118	473	559	546	330	336	247	260	335	327	0	120
11	45	365	441	429	210	219	129	141	281	331	120	0

Exercise

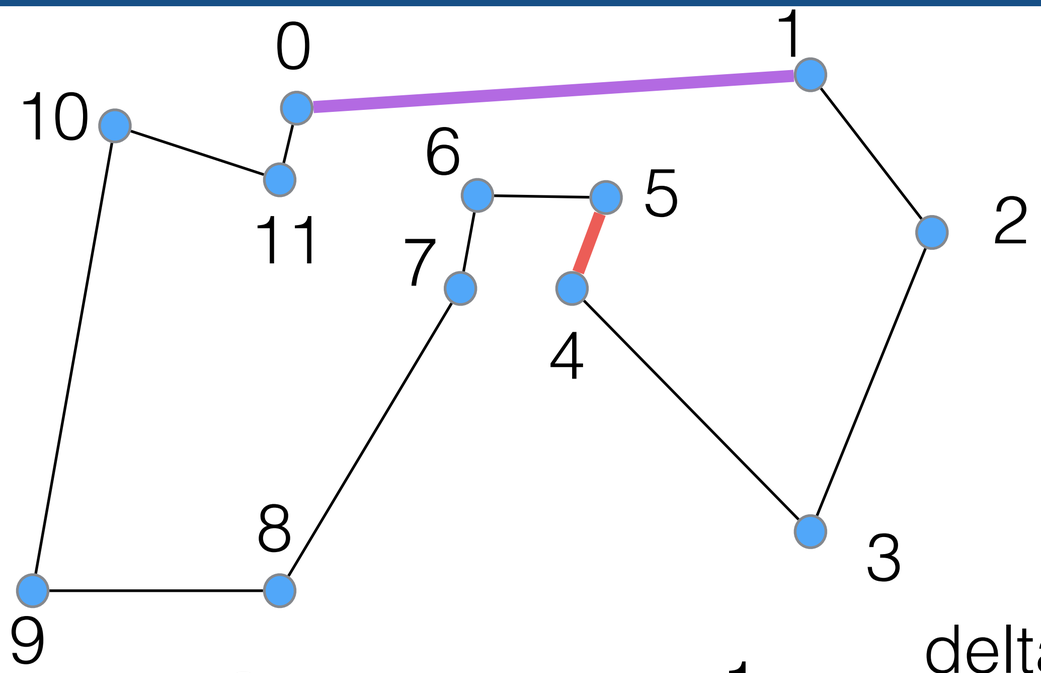
- In a Euclidien TSP, if the current solution has two crossing edges, it can be improved with a single 2-OPT move.

- But if the current solution has no crossing edges, does it mean that it can't be improve by a 2-OPT move?



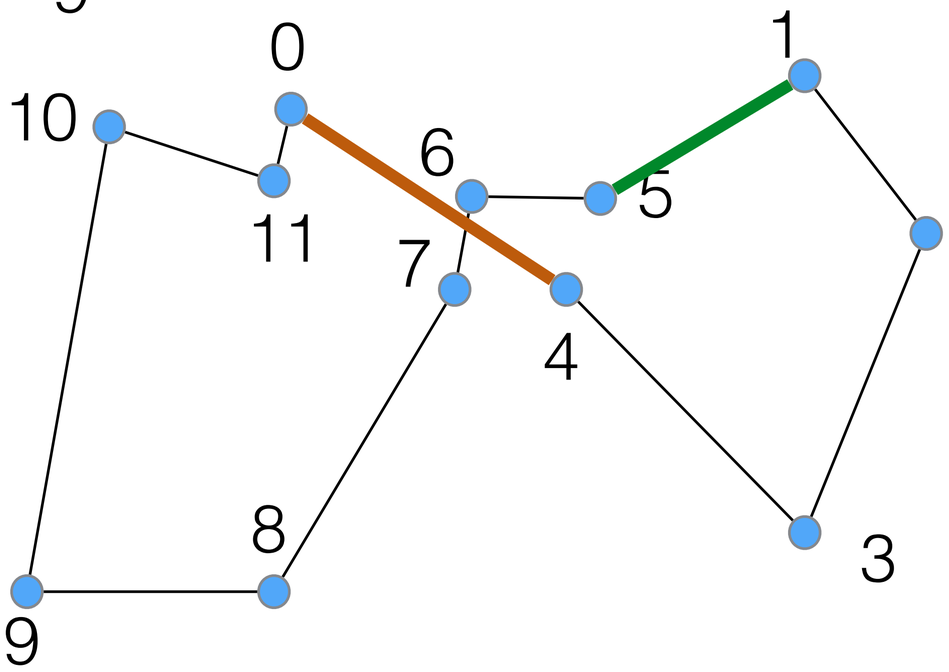
totalCost = 2056

Answer



totalCost = 2056

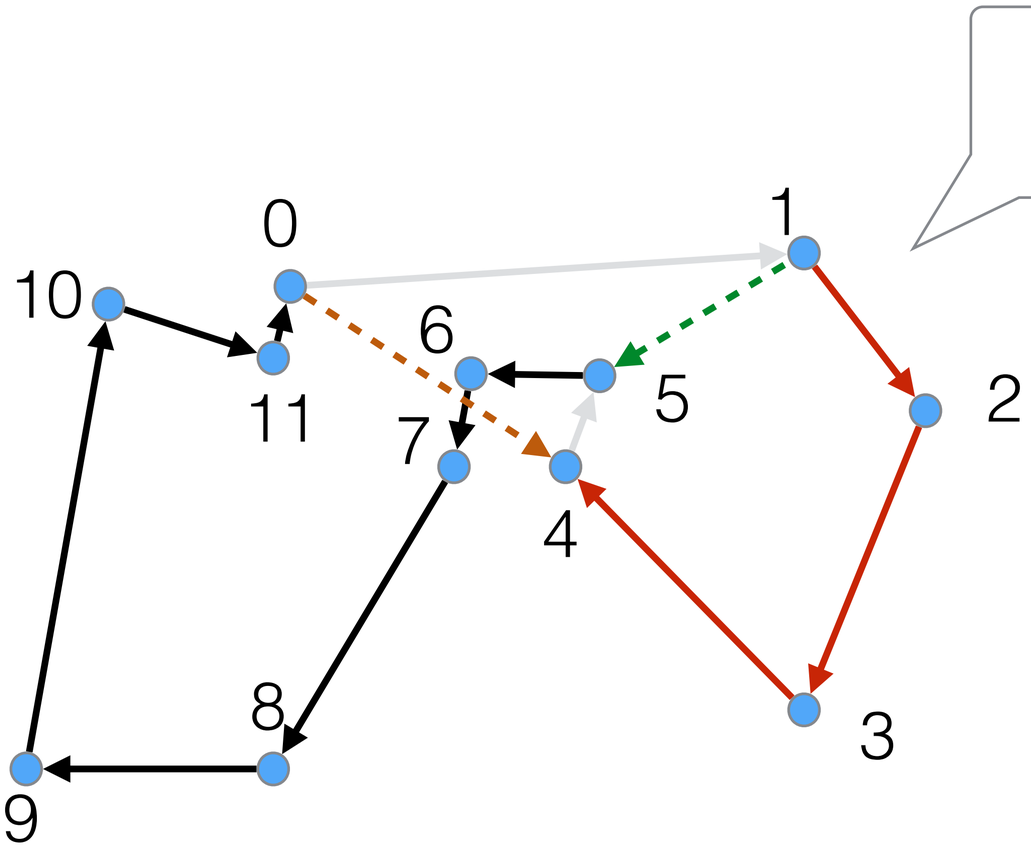
delta = **-355-63+225+160**



totalCost = 2023

twoOpt(left,right) = inverting a sub-sequence

- Our tour representation is (artificially) « oriented »



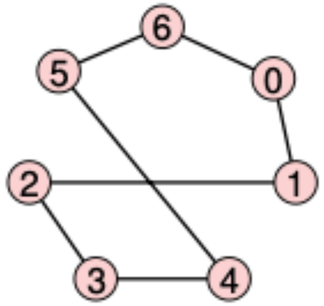
The red path must be flipped

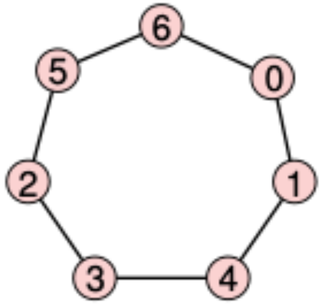
0,**1,2,3,4**,5,6,7,8,9,10,11,0

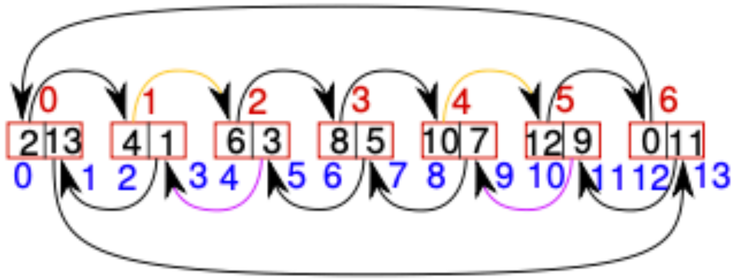
0,**4,3,2,1**,5,6,7,8,9,10,11,0

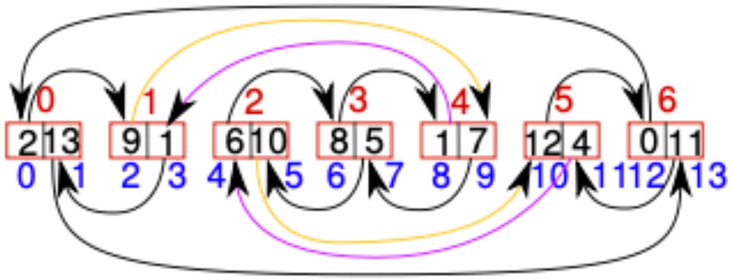
Data-Structure to do a 2-Opt Move in O(1)

A portion of the tour must be reversed in constant time



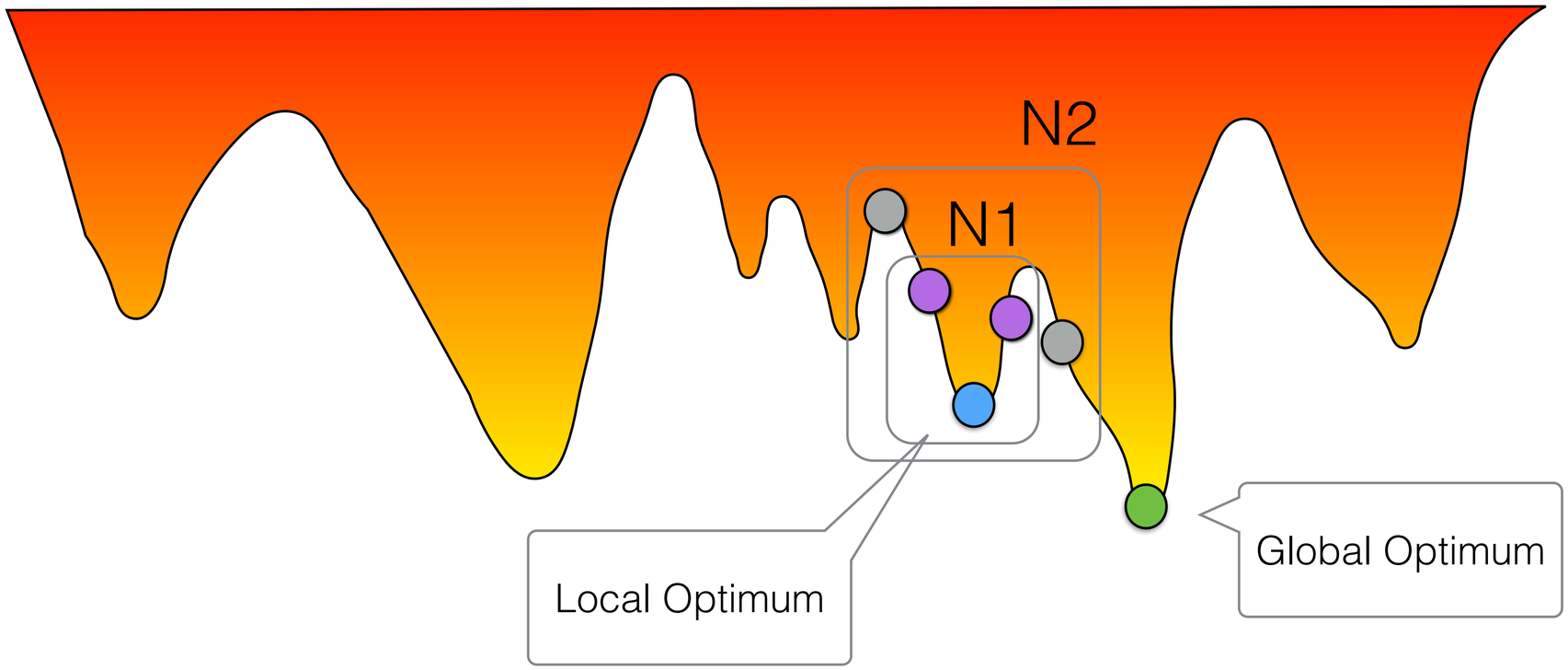






Doubly linked list implemented with an array of size 2n, giving the two neighbors of a node i: tab[2i]/2 and tab[2i+1]/2

The problem of Local Minima



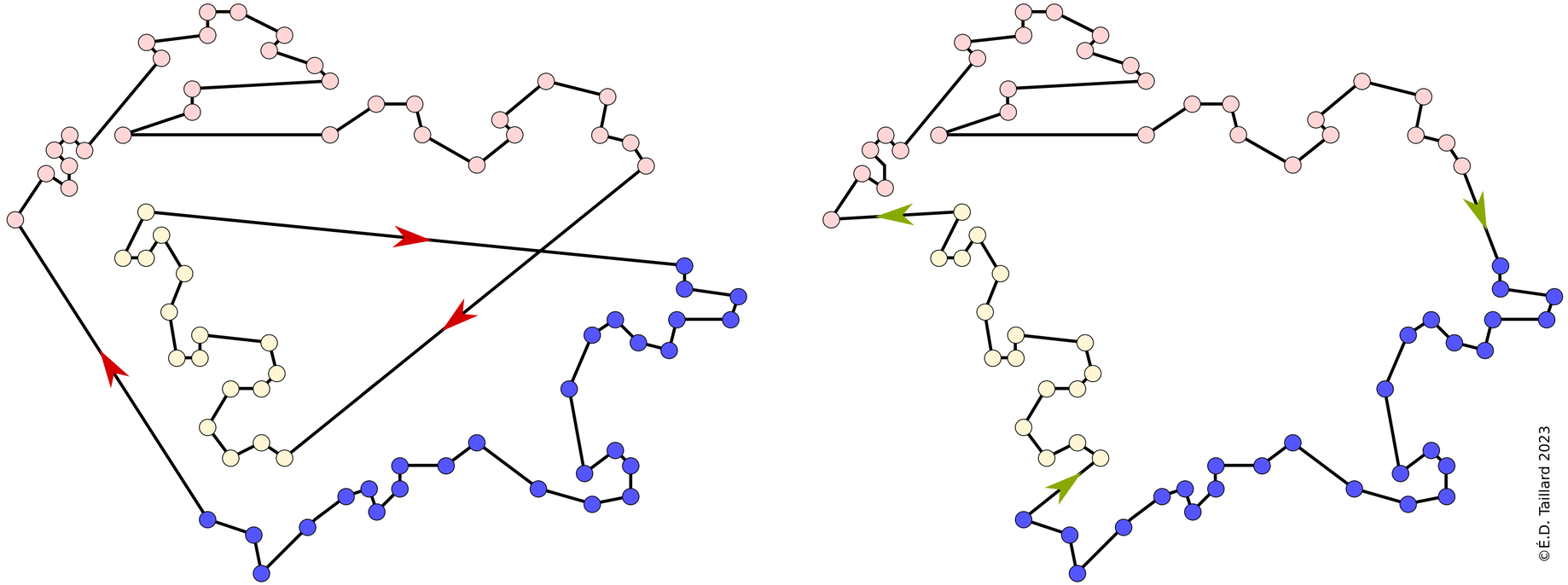
Two solutions

1. Enlarge the neighborhood

2. Accept to degrade the solution (meta-heuristics)

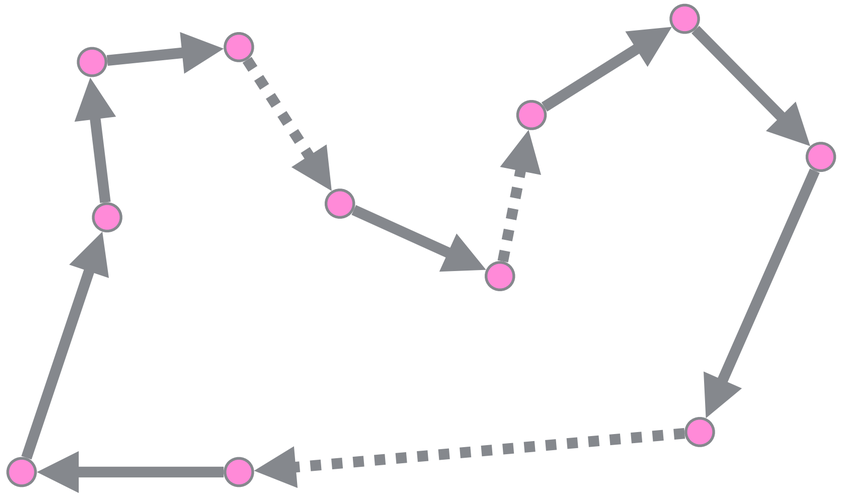
3 Opt Move

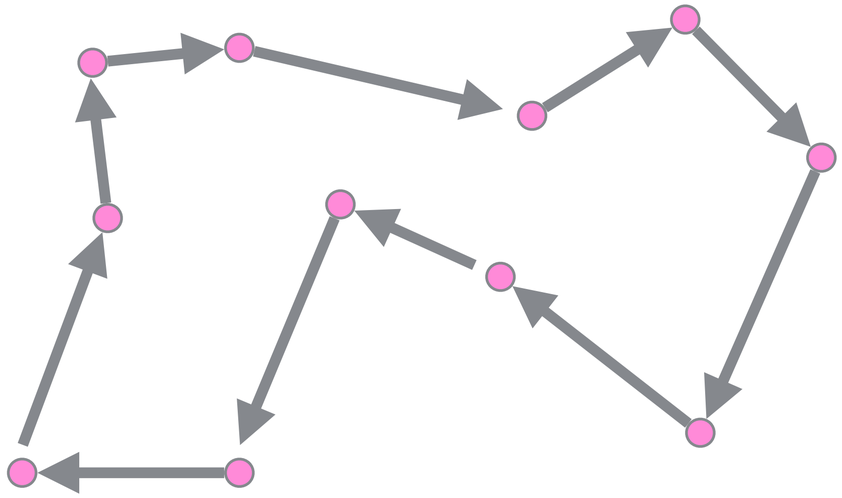
- Replace 3 edges by 3 others, for example
- $(i \to s_i), (j \to s_j), (k \to s_k)$ replaced by: $(i \to s_j), (j \to s_k), (k \to s_i)$
- Respects the visiting order of the cities on the other arcs, unlike 2-opt



3-Opt Move,

- Size of the neighborhood is $O(n^3)$, it was $O(n^2)$ for 2-Opt



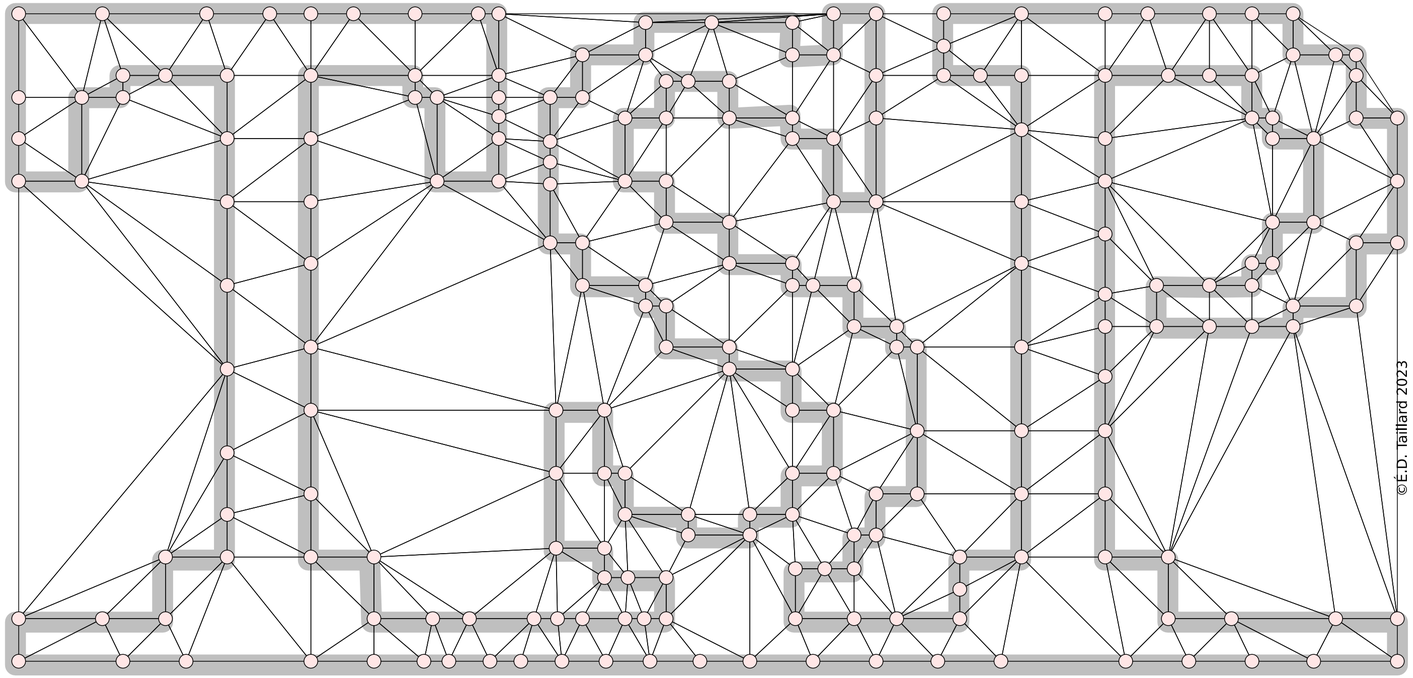


While 3-Opt can still be useful, 4-Opt almost never pays off.

Neighborhood Restriction

- Idea: do not consider all neighbor solutions, but only a potentially interesting subset

- One possibility to do that is to exclude some “long edges”. For an Euclidean TSP you can for instance keep the edges of Delaunay triangulation.

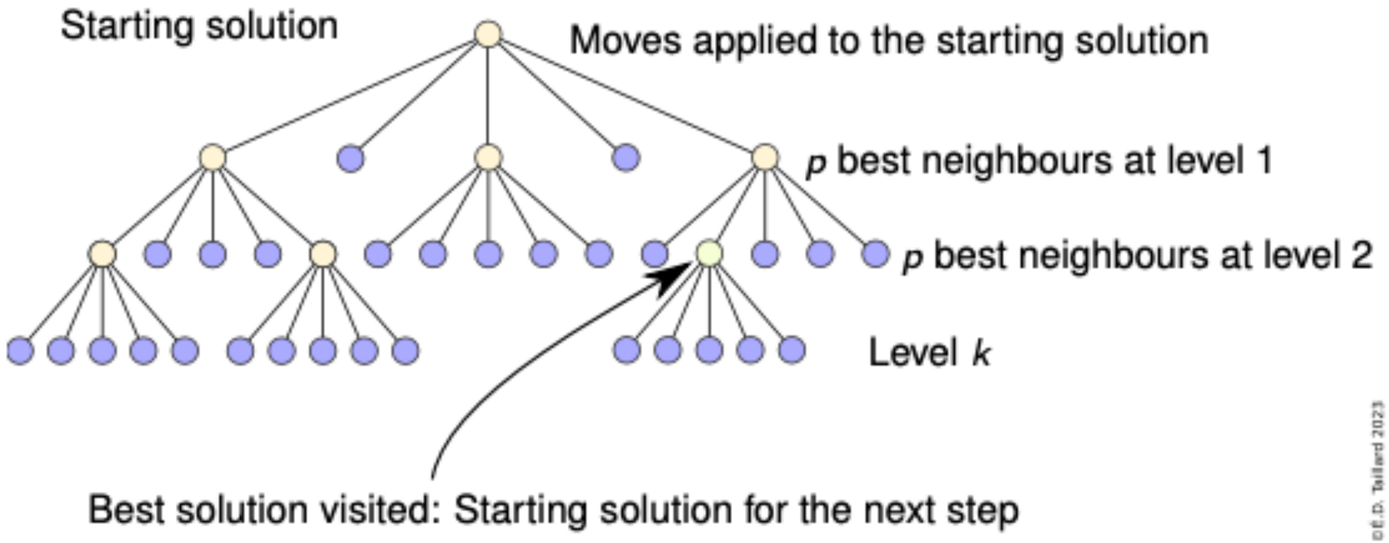


Neighborhood Extension: Filter and Fan

- Construct a neighborhood composed of k basic moves (e.g. 2-Opt)
  - $N_e(s) = \{s' \mid s' = s \oplus m_1 \oplus \ldots \oplus m_k, m_1, \ldots, m_k \in M(s)\}$ with
  - $N(s) = \{[i,j] \mid i,j \in s, i \neq j, j \neq s_i, i \neq s_j\}$ (set of 2-Opt)

- Problem: $|N_e(s)|$ grows exponentially with k

- Idea: Use Beam search to select it and control the exponential growth



Let us dream …

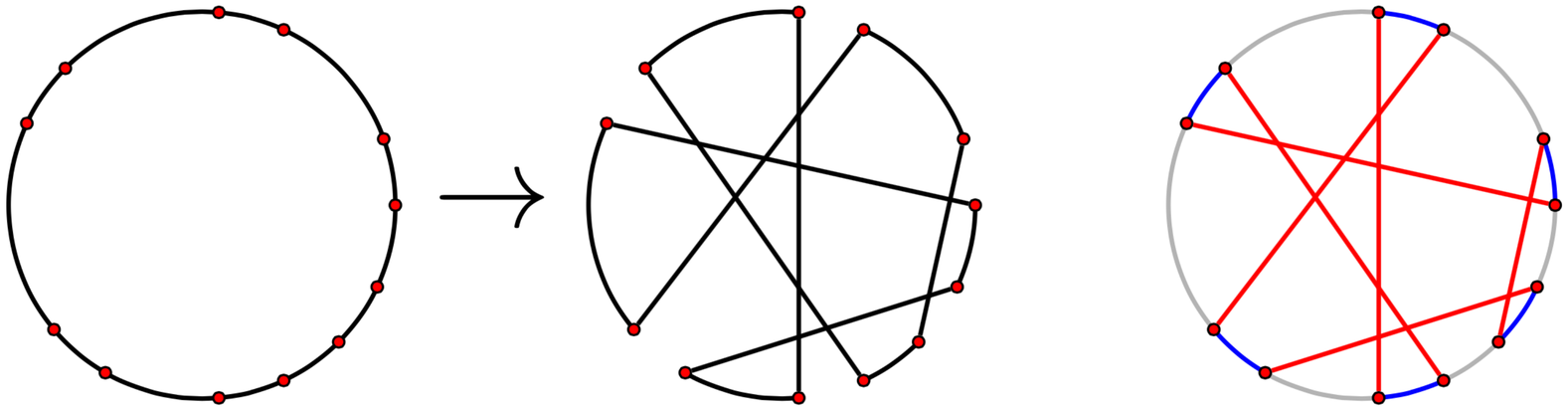
- What about a k-Opt without specifying the k ?

- Would it be tractable?

Efficient k-Opt = Lin-Kernighan (aka Ejection Chains)

- General k-Opt is computationally too intensive
- But we can limit our-self to a particular form of k-Opt moves: ***Sequential k-Opt moves***
- A k-Opt move is called sequential if it can be described by a **path alternating between deleted and added edges.**

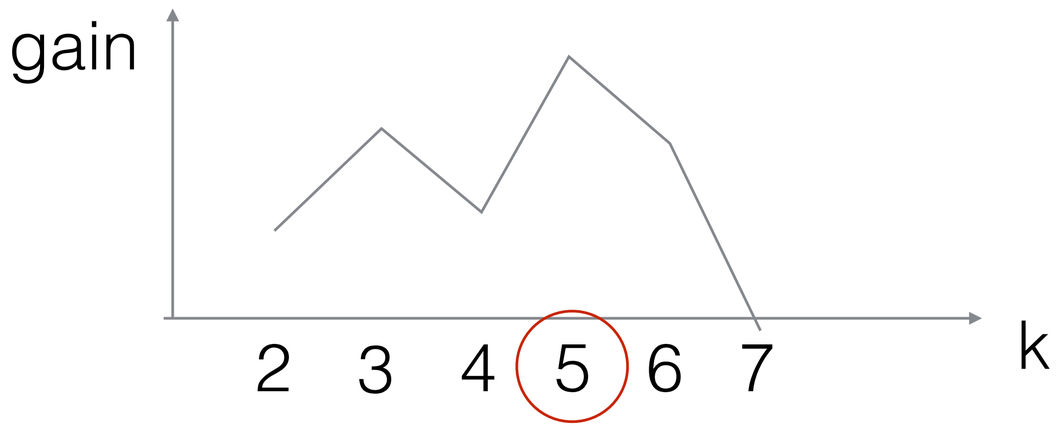
Example: sequential 6-Opt



Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, *21*(2), 498-516.

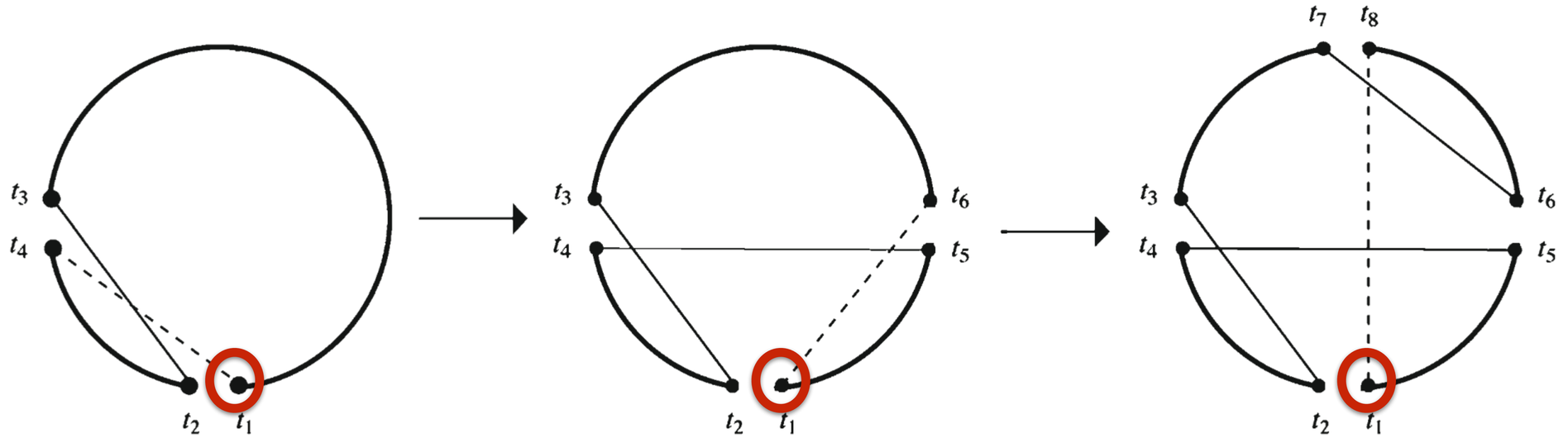
Lin-Kernighan

- Sequential k-Opt moves: What values for k ?
- The idea is to build greedily a k-Opt move
- At each step we compute the « gain » of applying it.
  - The (k+1)-sequential move is an extension of the k-sequential move, etc
- Then we select (retrospectively) the k that gives the best « gain »



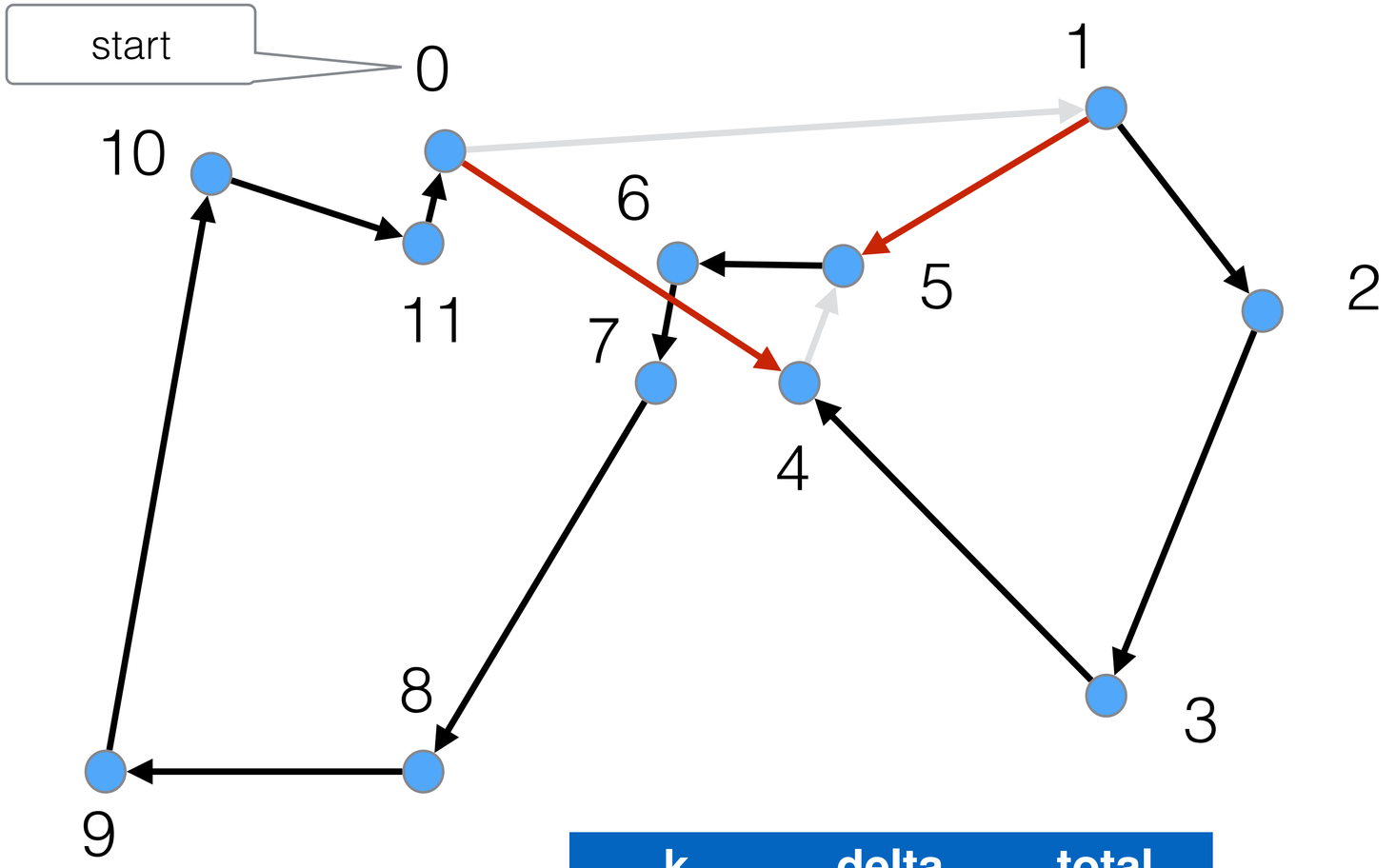
Observation:

- A sequential k-Opt move has the same effect as k-1 2Opt Moves.
- Example: sequential 4-Opt = 2Opt + 2Opt + 2Opt



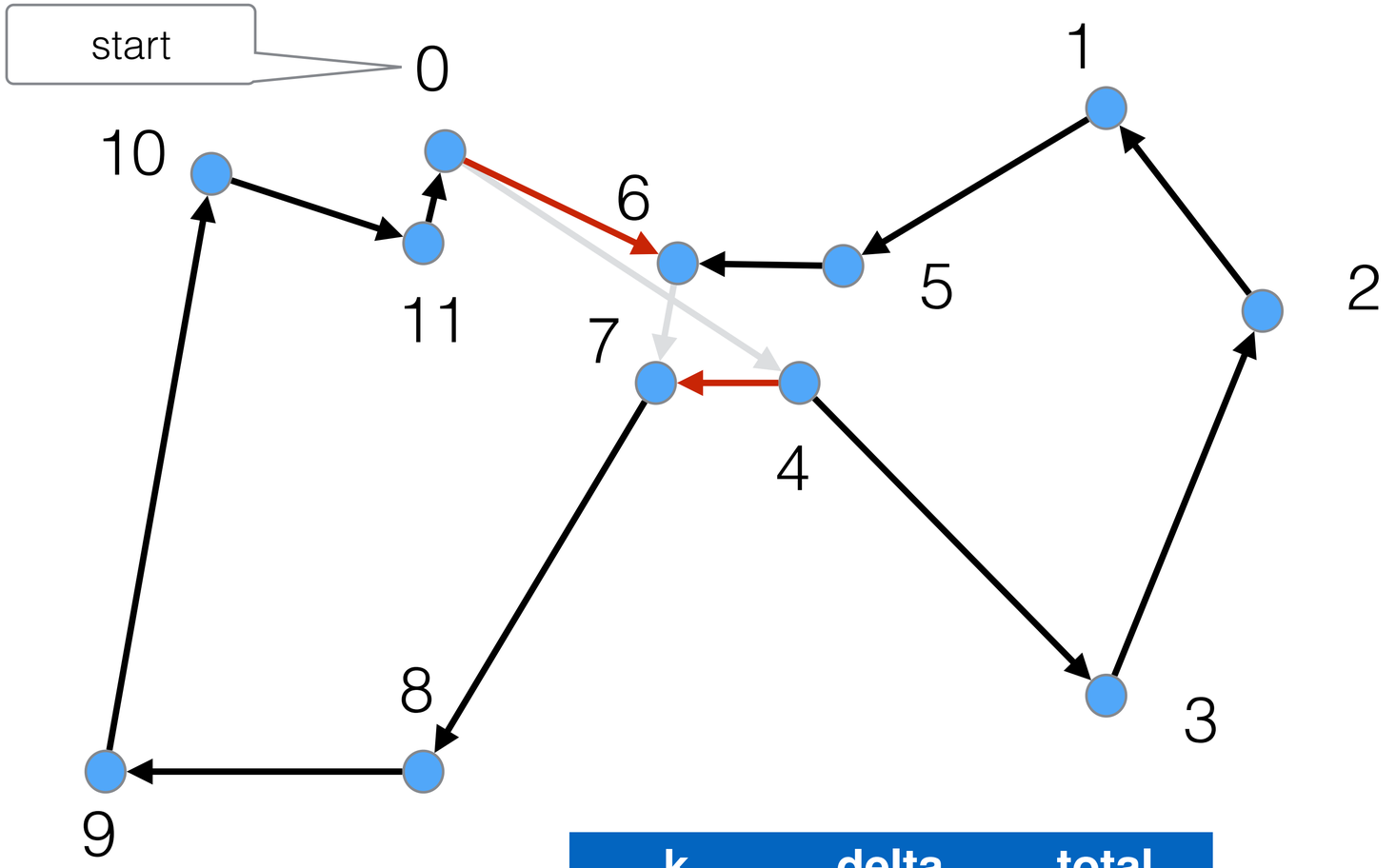
Sequential 4-opt move performed by three 2-opt moves. Close-up edges are shown by *dashed lines*

K-Opt starting at 0



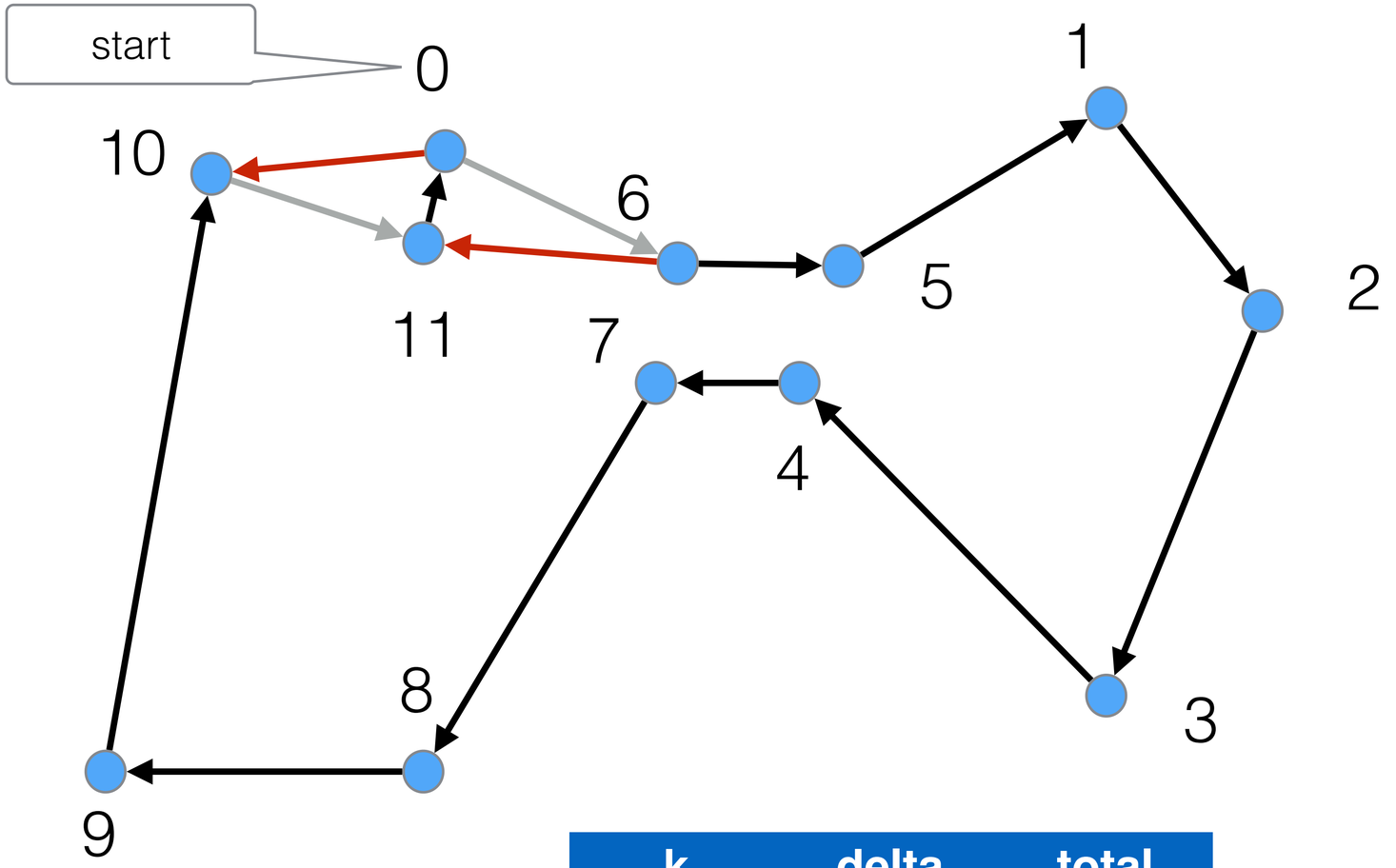
k	delta	total
2	-33	-33

K-Opt starting at 0



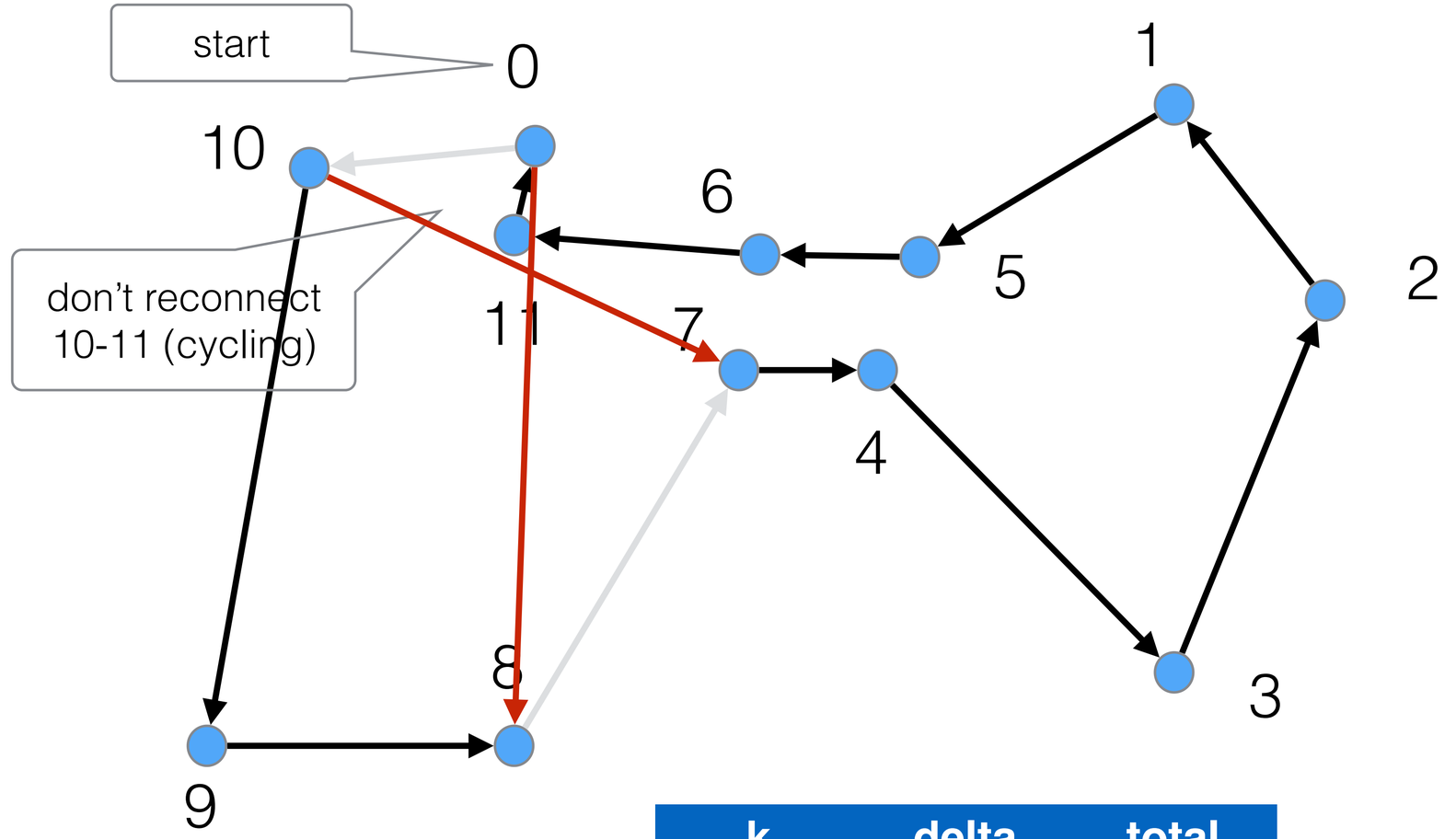
k	delta	total
2	-33	-33
3	-77	-110

K-Opt starting at 0



k	delta	total
2	-33	-33
3	-77	-110
4	-9	-119

K-Opt starting at 0

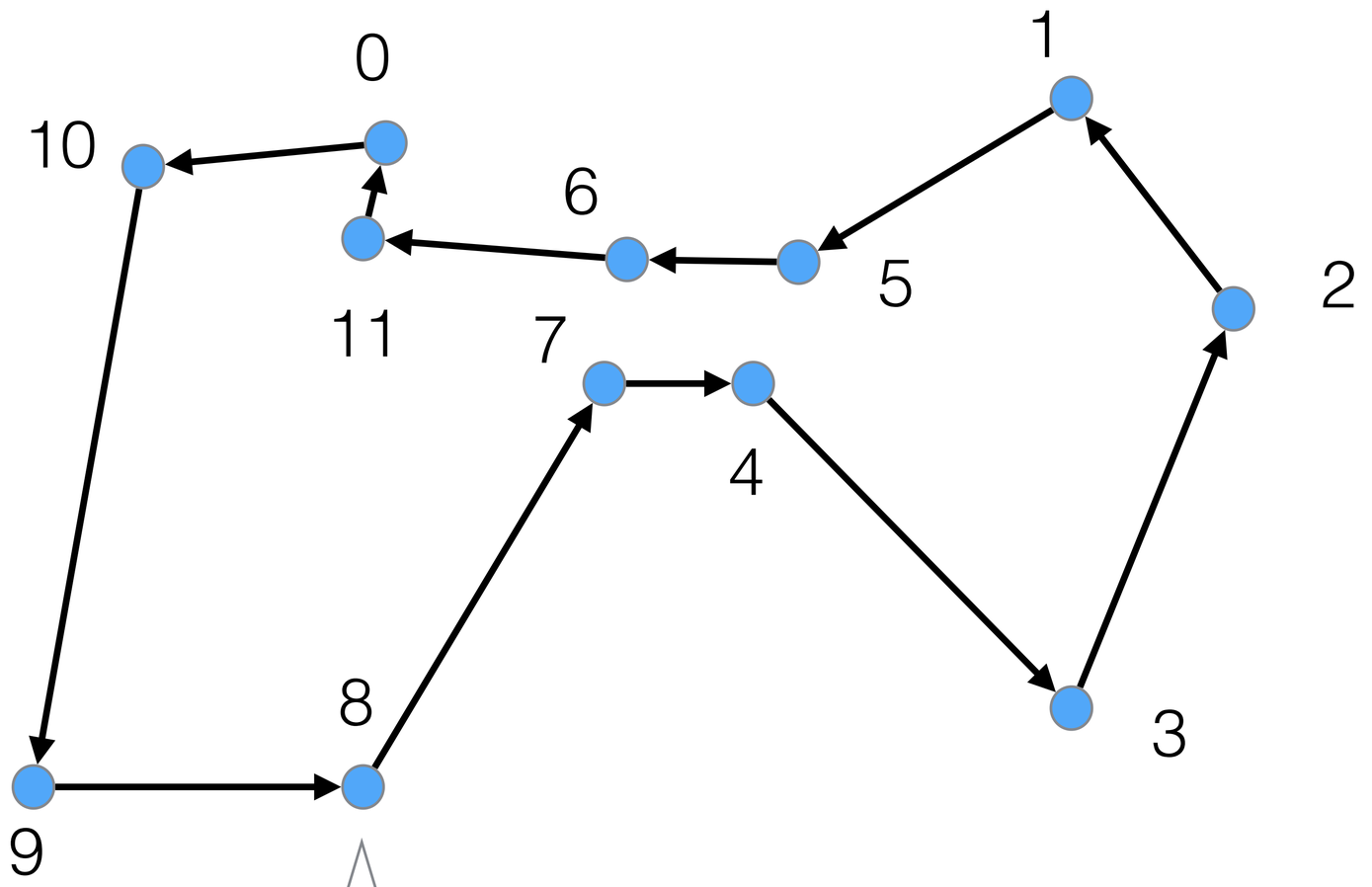


k	delta	total
2	-33	-33
3	-77	-110
4	-9	-119
5	224	105

restore this solution

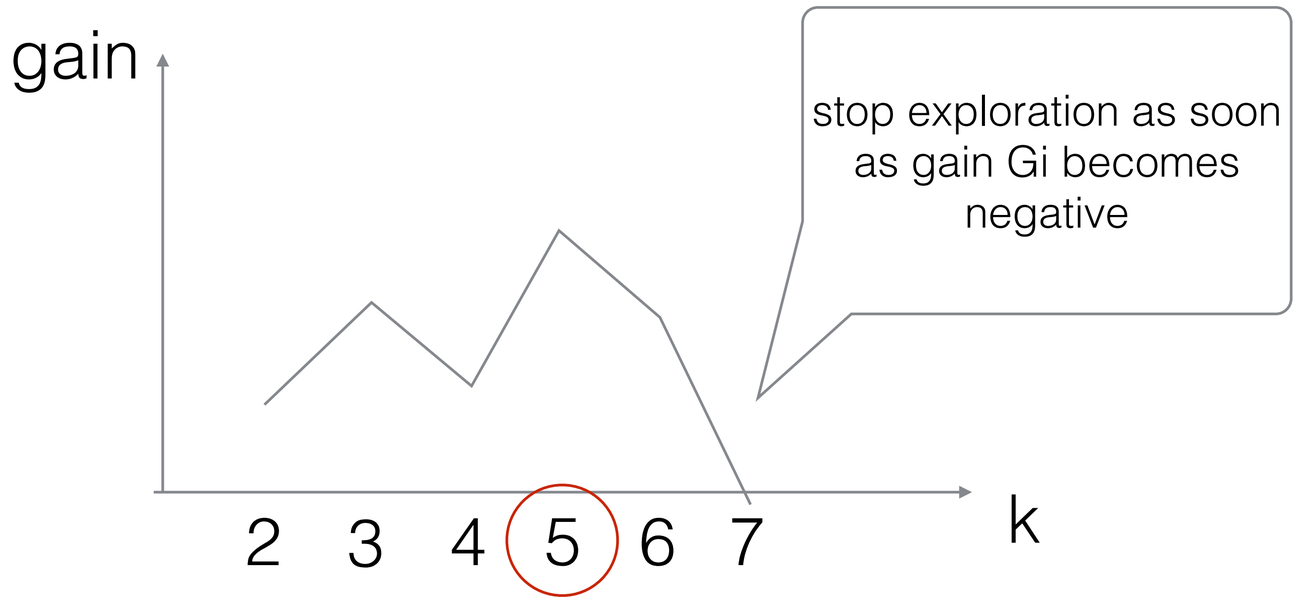
stop!

K-Opt starting at 8



Do a K-Opt iteration starting from 8, do at-least 2, you should see something very nice after two iterations (that 2-Opt alone cannot see)

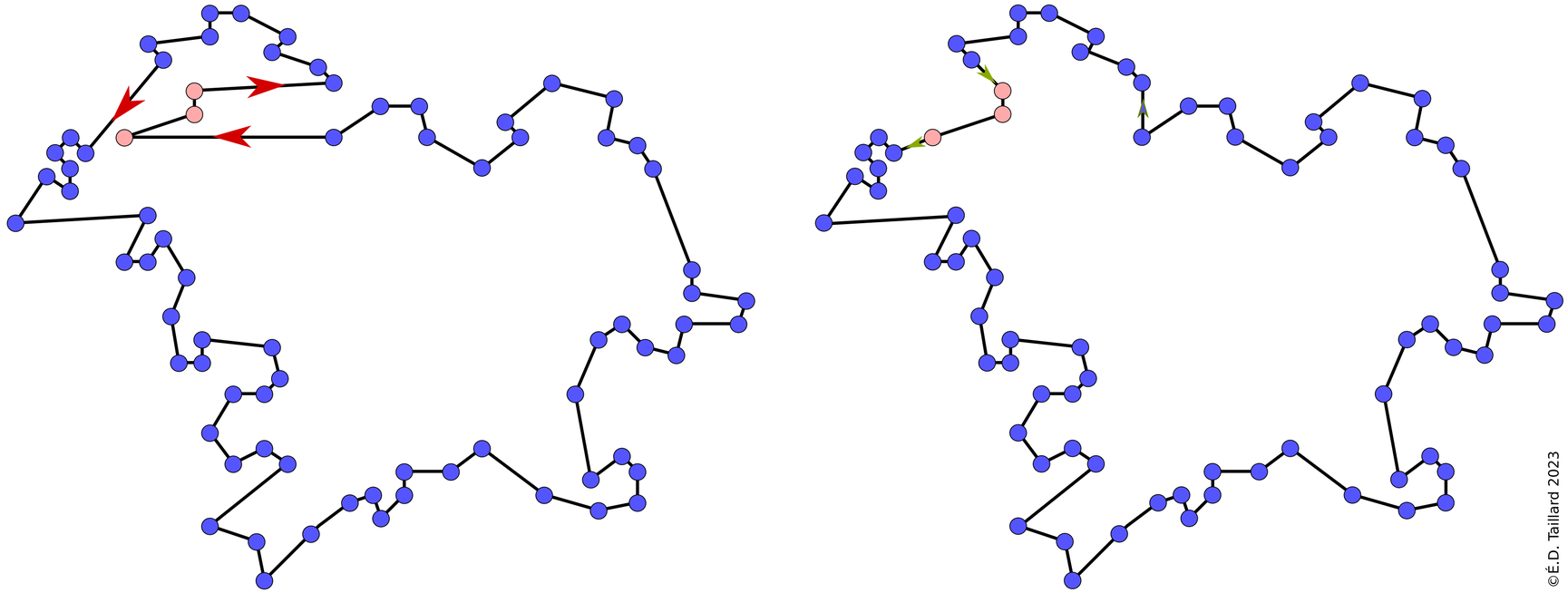
Positive gain stoping criterion for one iter



Or Neighborhood (another not 2-Opt based Neighborhood)

- Move / Reinsert sub-sequence of 3,2 or 1 cities

- Do it gradually: when not possible to improve with 3, try 2, then 1

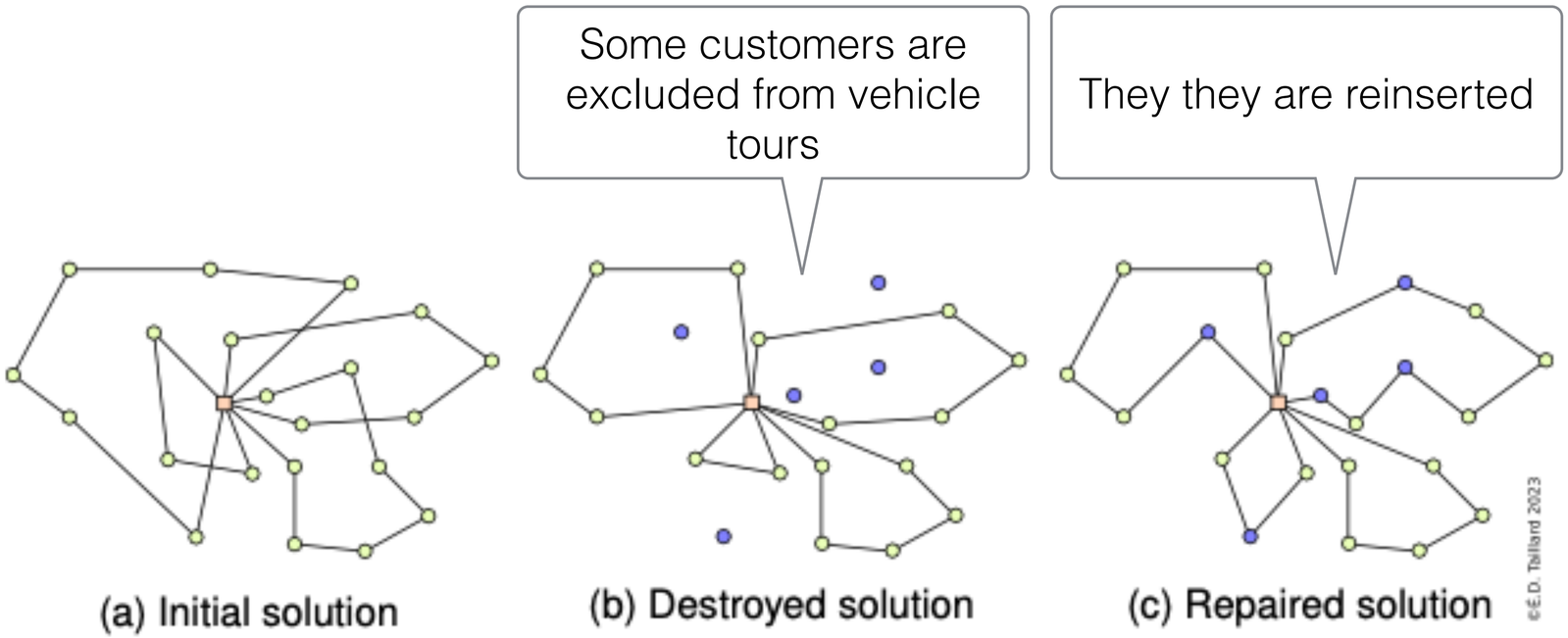


Or, I. (1976). Traveling salesman-type combinatorial problems and their relation to the logistics of blood banking. *PhD thesis (Department of Industrial Engineering and Management Science, Northwestern University)*.

Large Neighborhood Search (LNS)

- Push the idea of enlarging the neighborhood to an extreme
  - Destroy part of the solution
  - Rebuild it by trying to do better

- Illustration on a VRP problem (same as TSP but with one depot and several vehicles)



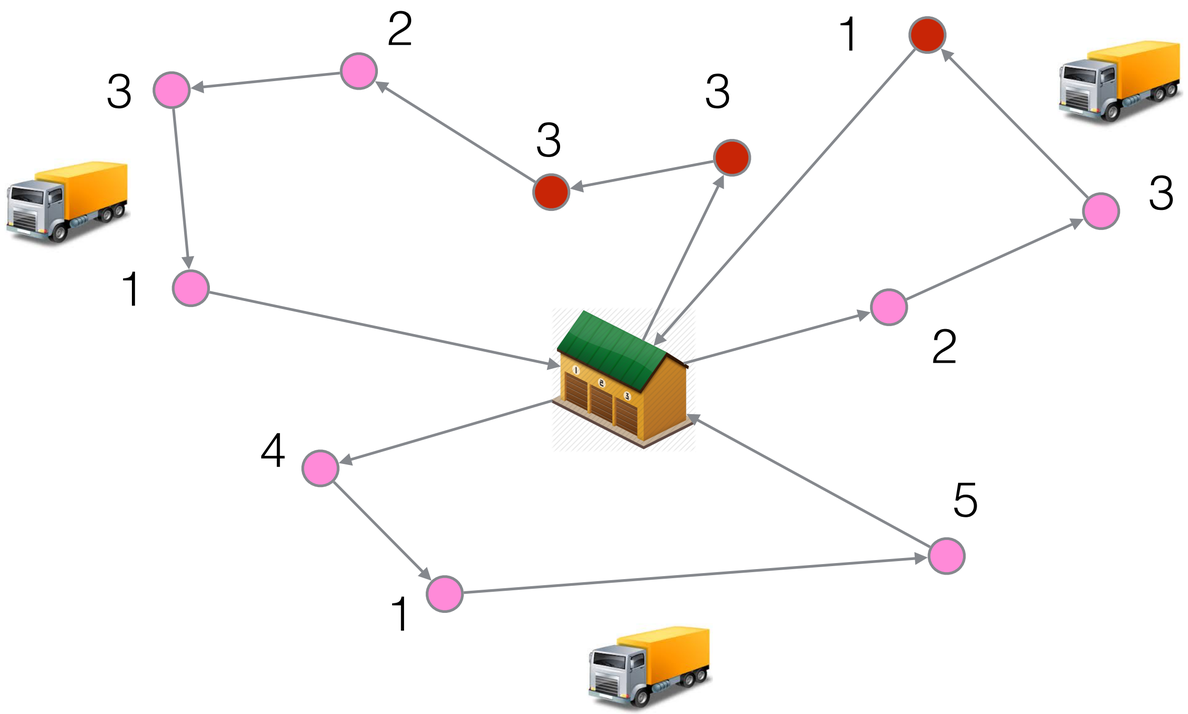
LNS

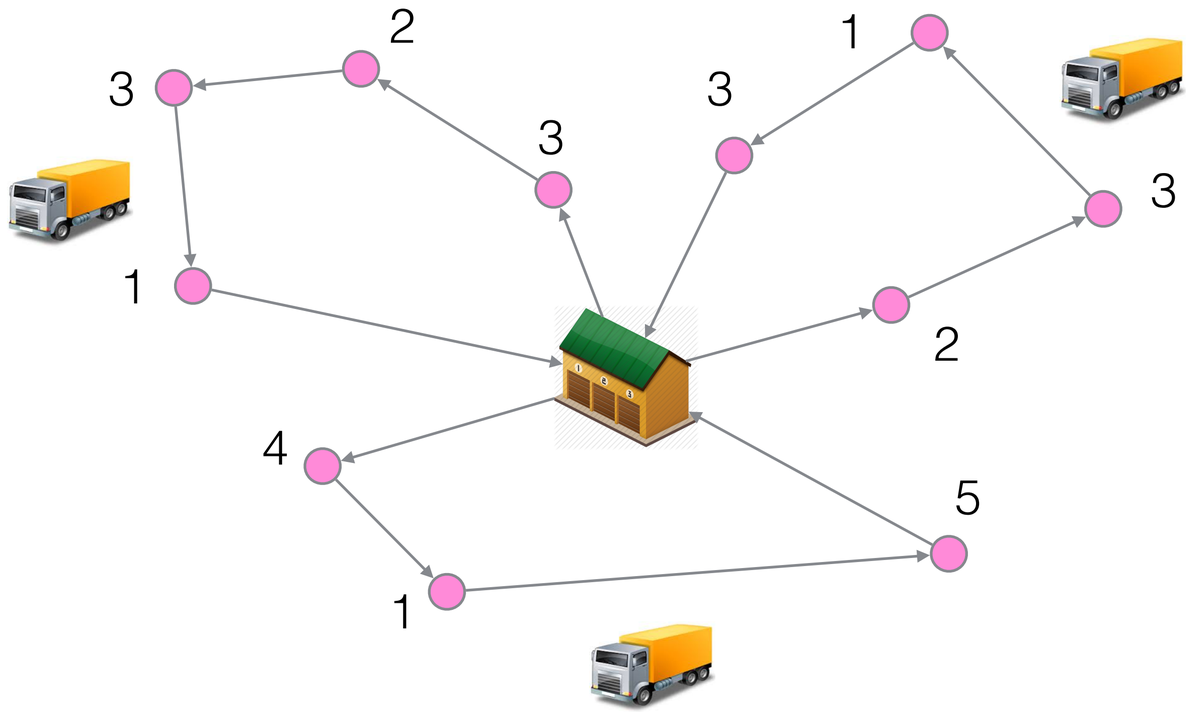
Input: Solution s, destroy method $d(\cdot)$, repair method $r(\cdot)$, acceptance criterion $a(\cdot, \cdot)$
Result: Improved solution s^*
$s^* \leftarrow s$
repeat
  $s' \leftarrow r(d(s))$
  if $a(s, s')$ **then**
    $s \leftarrow s'$
  end
  if s' *better than* s^* **then**
    $s^* \leftarrow s'$
  end
until *a stopping criterion is satisfied*

LNS in practice

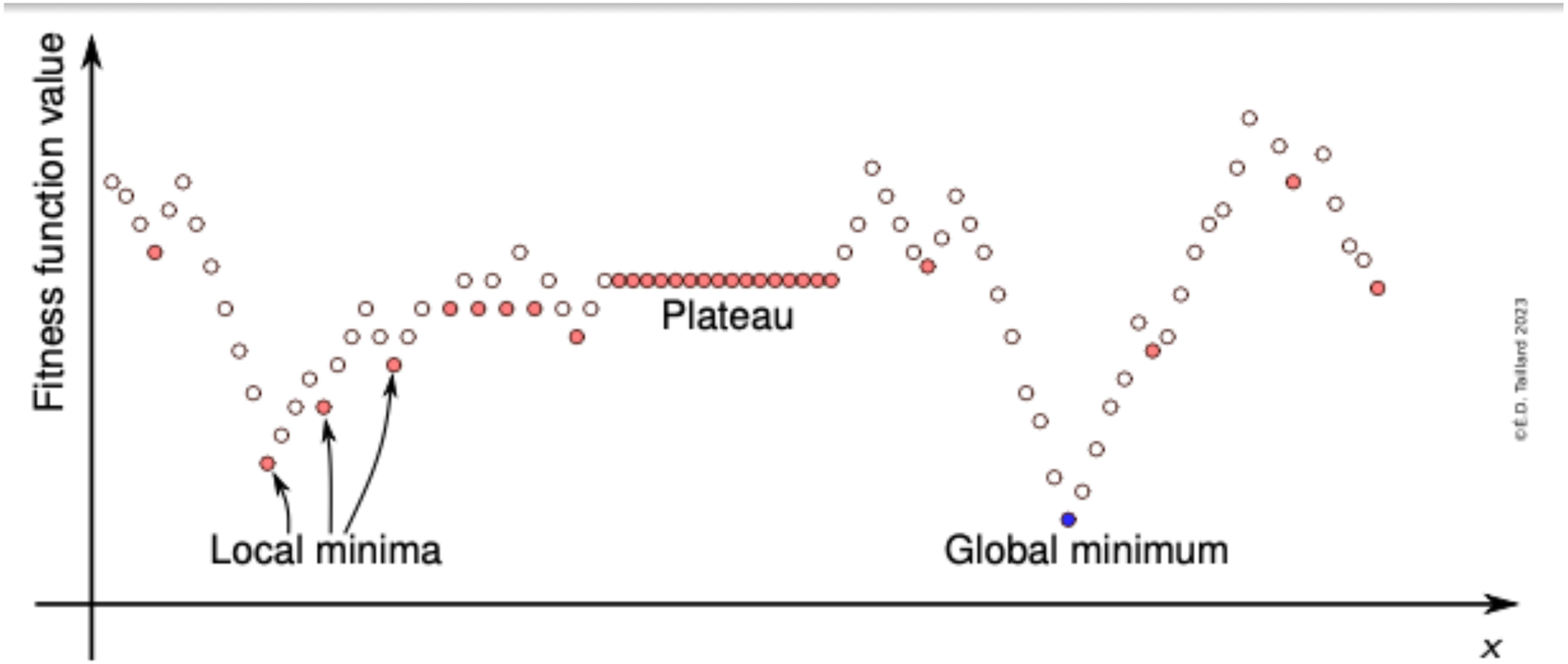
- For the destroy part, don’t choose cities not to far appart. Very little chance to bring benefits.

- Instead consider the geometry of the problem:
  - Choose cities/customers not to far appart and in different vehicles





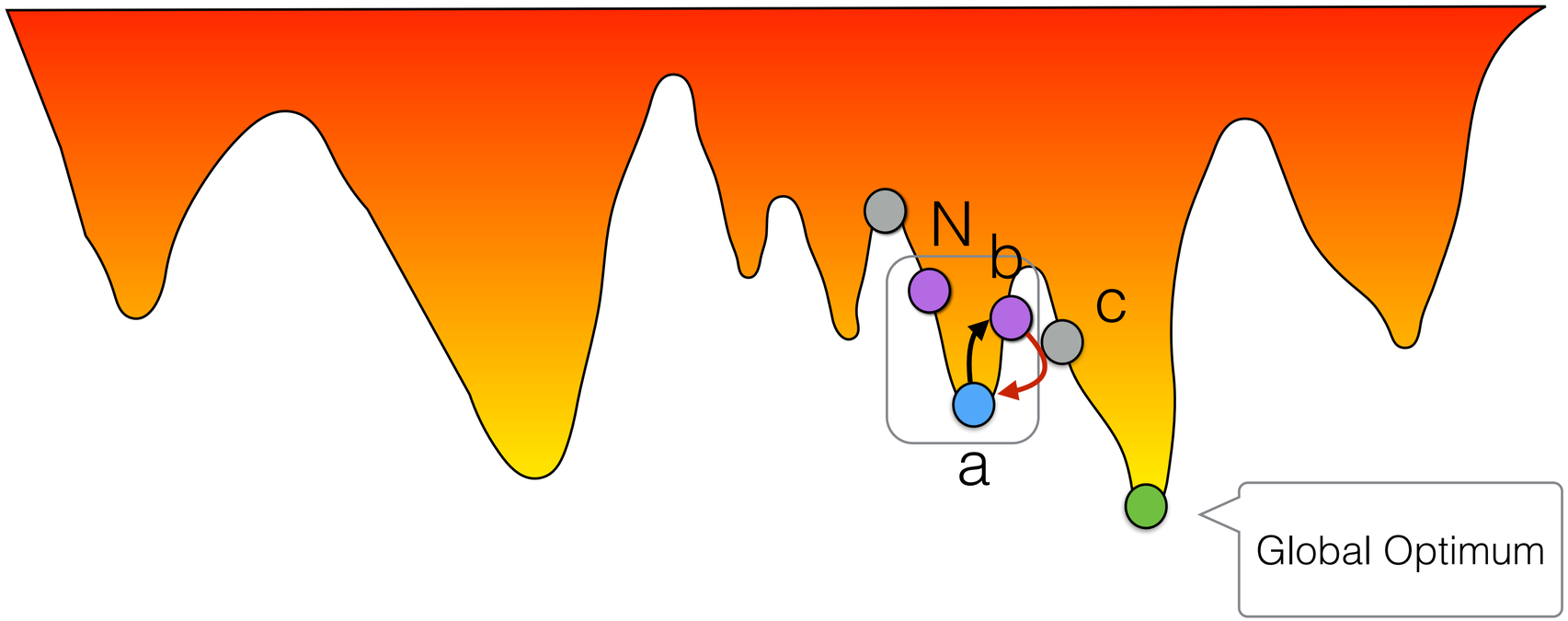
The problem of Local Minima



Two solutions

1. Enlarge the neighborhood
2. Accept to degrade the solution (meta-heuristics)

Best Move: The cycling Problem



- From **a** you move to **b** (no better choice in neighboring N)
- But from **b**, the best move is to return back to **a**
- Solution: make the **the reverse move becomes tabu**, you don't want to come to **a** for a while (duration = tabu tenure).
- Of course, doing so, you accept that you can degrade the solution

Tabu Algorithm

Input: Solution s, set M of moves, fitness function $f(\cdot)$ to minimize, parameters I_{max}, d
Result: Improved solution s^*
$s^* \leftarrow s$
for I_{max} *iterations* **do**
  $best\_neighbour\_value \leftarrow \infty$
  forall $m \in M$ *(such that* m *(or* $s \oplus m$*) is not marked as taboo)* **do**
    if $f(s \oplus m) < best\_neighbour\_value$ **then**
      $best\_neighbour\_value \leftarrow f(s \oplus m)$
      $m^* \leftarrow m$
    end
  end
  if $best\_neighbour\_value < \infty$ **then**
    Mark $(m^*)^{-1}$ (or s) as taboo for the next d iterations
    $s \leftarrow s \oplus m^*$
    if $f(s) < f(s^*)$ **then**
      $s^* \leftarrow s$
    end
  else
    Error message: d too large: no move allowed!
  end
end

Tabu search for the Knapsack

- Simple move: take one item, and change its selected status but do not violate the capacity

$$\begin{aligned} \max r = \quad & 12s_1 + 10s_2 + 9s_3 + 7s_4 + 4s_5 + 8s_6 + 11s_7 + 6s_8 + 13s_9 \\ \text{S.t.} \quad & 10s_1 + 12s_2 + 8s_3 + 7s_4 + 5s_5 + 13s_6 + 9s_7 + 6s_8 + 14s_9 \leqslant 45 \\ & s_i \in \{0,1\} \quad (i = 1, \ldots, 9) \end{aligned}$$

Iter.	Var. Modif.	Solution	r	v	Taboo status (duration $d = 3$)
0	—	(0, 0, 0, 0, 0, 0, 0, 0, 0)	0	0	(0, 0, 0, 0, 0, 0, 0, 0, 0)
1	s_9	(0, 0, 0, 0, 0, 0, 0, 0, 1)	13	14	(0, 0, 0, 0, 0, 0, 0, 0, 4)
2	s_1	(1, 0, 0, 0, 0, 0, 0, 0, 1)	25	24	(5, 0, 0, 0, 0, 0, 0, 0, 4)
3	s_7	(1, 0, 0, 0, 0, 0, 1, 0, 1)	36	33	(5, 0, 0, 0, 0, 0, 6, 0, 4)
4	s_2	(1, 1, 0, 0, 0, 0, 1, 0, 1)	46	45	(5, 7, 0, 0, 0, 0, 6, 0, 4)
5	s_9	(1, 1, 0, 0, 0, 0, 1, 0, 0)	33	31	(5, 7, 0, 0, 0, 0, 6, 0, 8)
6	s_3	(1, 1, 1, 0, 0, 0, 1, 0, 0)	42	39	(5, 7, 9, 0, 0, 0, 6, 0, 8)
7	s_8	(1, 1, 1, 0, 0, 0, 1, 1, 0)	48	45	(5, 7, 9, 0, 0, 0, 6, 10, 8)
8	s_2	(1, 0, 1, 0, 0, 0, 1, 1, 0)	38	33	(5, 11, 9, 0, 0, 0, 6, 10, 8)
9	s_4	(1, 0, 1, 1, 0, 0, 1, 1, 0)	45	40	(5, 11, 9, 12, 0, 0, 6, 10, 8)
10	s_5	(1, 0, 1, 1, 1, 0, 1, 1, 0)	49	45	(5, 11, 9, 12, 13, 0, 6, 10, 8)

Objective

Size

For each item, the next iteration when it is available again

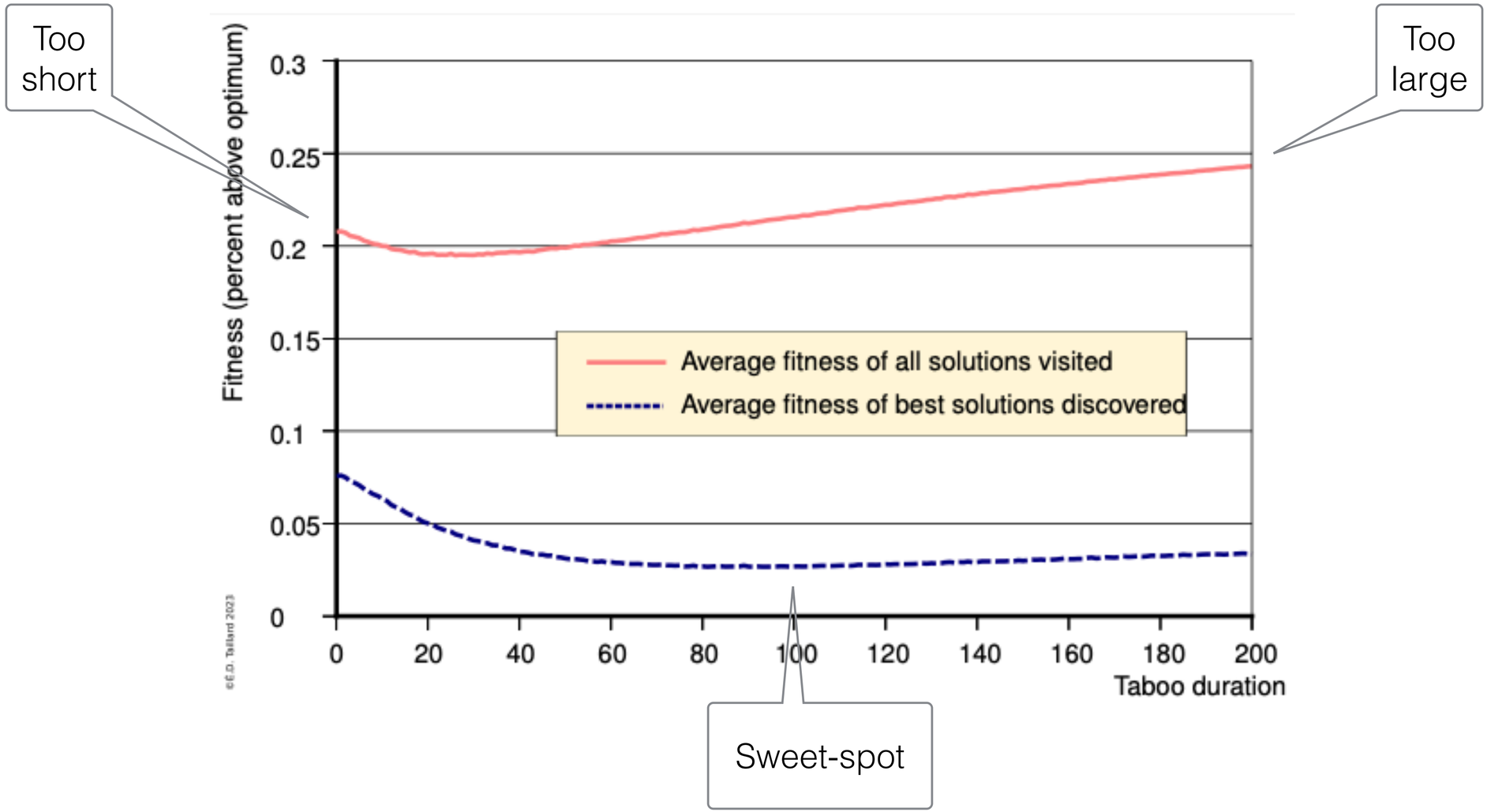
Tabu for the TSP

- You can set an edge tabu.

- When an edge is removed (by a 2-Opt move), make it tabu for a number of iterations.

Impact of tabu duration for Euclidian TSP 100 Cities

- Rule of thumb: chose tabu duration = $\sqrt{|N(s)|}$

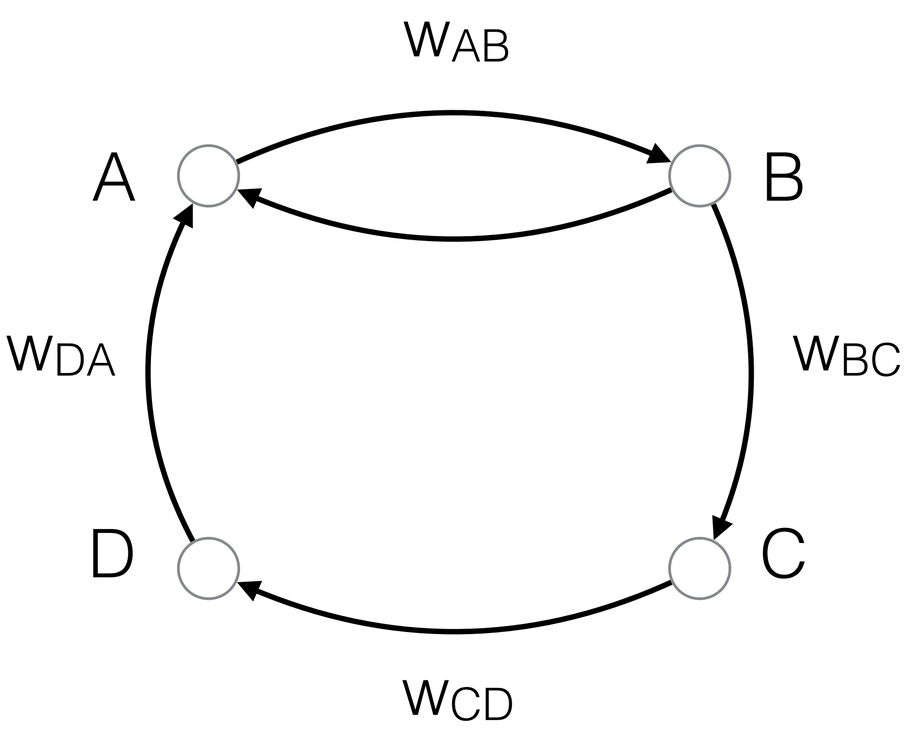


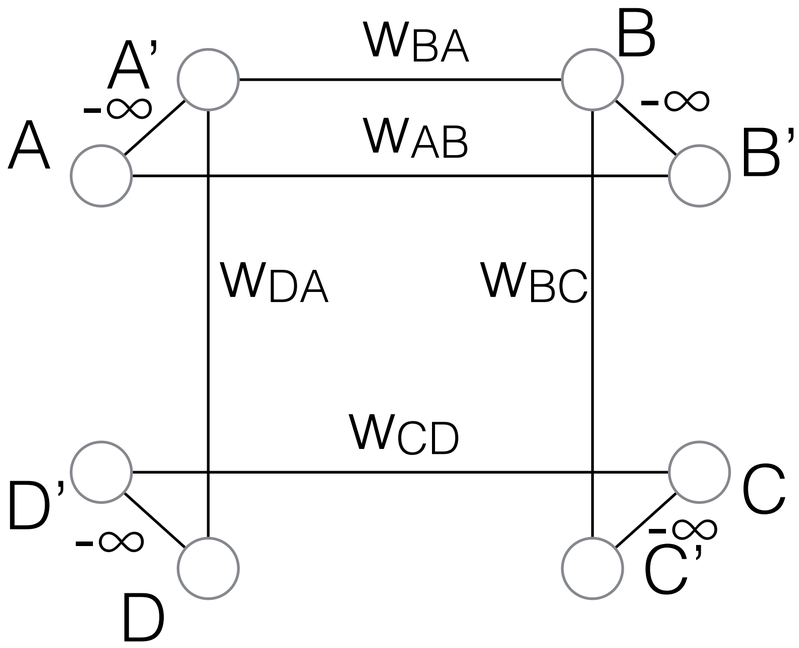
Some ideas to improving tabu search

- Aspiration: even if a move is tabu, if it improves the best-so-far solution, select it
- Oscillation: chose and vary the length of the tabu duration. Even choosing it random in some range works well.
- Restart: From time to time, restart (just perfume a bunch of random moves to explore a completely different region).

ATSP reduction to TSP

- The distance matrix is not always symmetric (city: one-way roads, etc).
- Possible to transform an ATSP into a TSP by doubling the number of nodes.



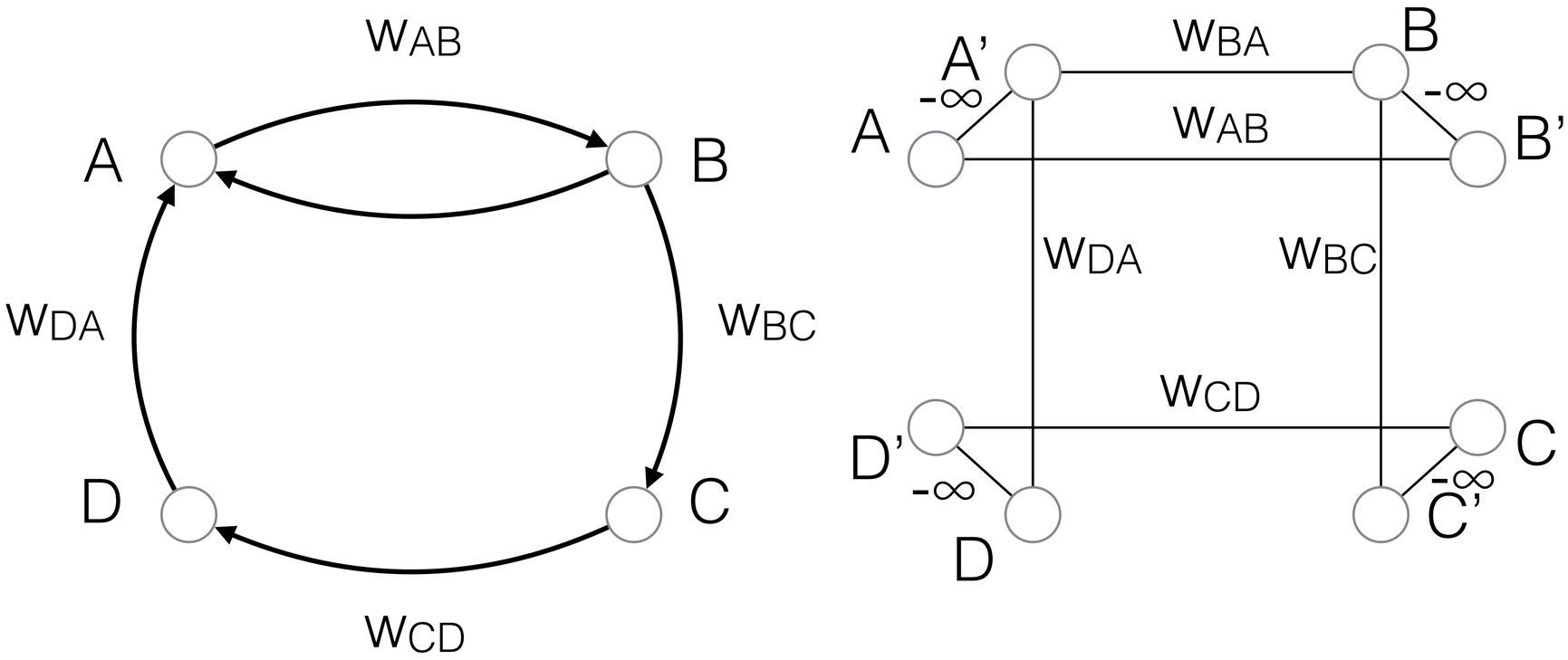


1 to 1 correspondance

Jonker, Roy; Volgenant, Ton. "Transforming asymmetric into symmetric traveling salesman problems". *Operations Research Letters*, 1983

ATSP reduction to TSP

- Idea: N-N’ are so attractive (large negative number) that they are part of any optimal TSP. Hence A’B and AB’ cannot be both selected otherwise there would be a sub-tour (not hamiltonian circuit).



1 to 1 correspondance

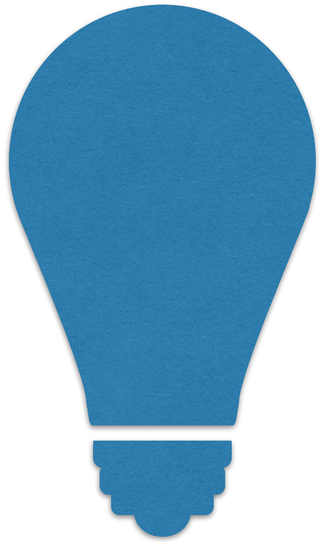
Jonker, Roy; Volgenant, Ton. "Transforming asymmetric into symmetric traveling salesman problems". *Operations Research Letters*, 1983

Case Study: Sudoku

- Local Search is not very good for this problem.
- Do you know why ?

Sudou: Model Idea 1

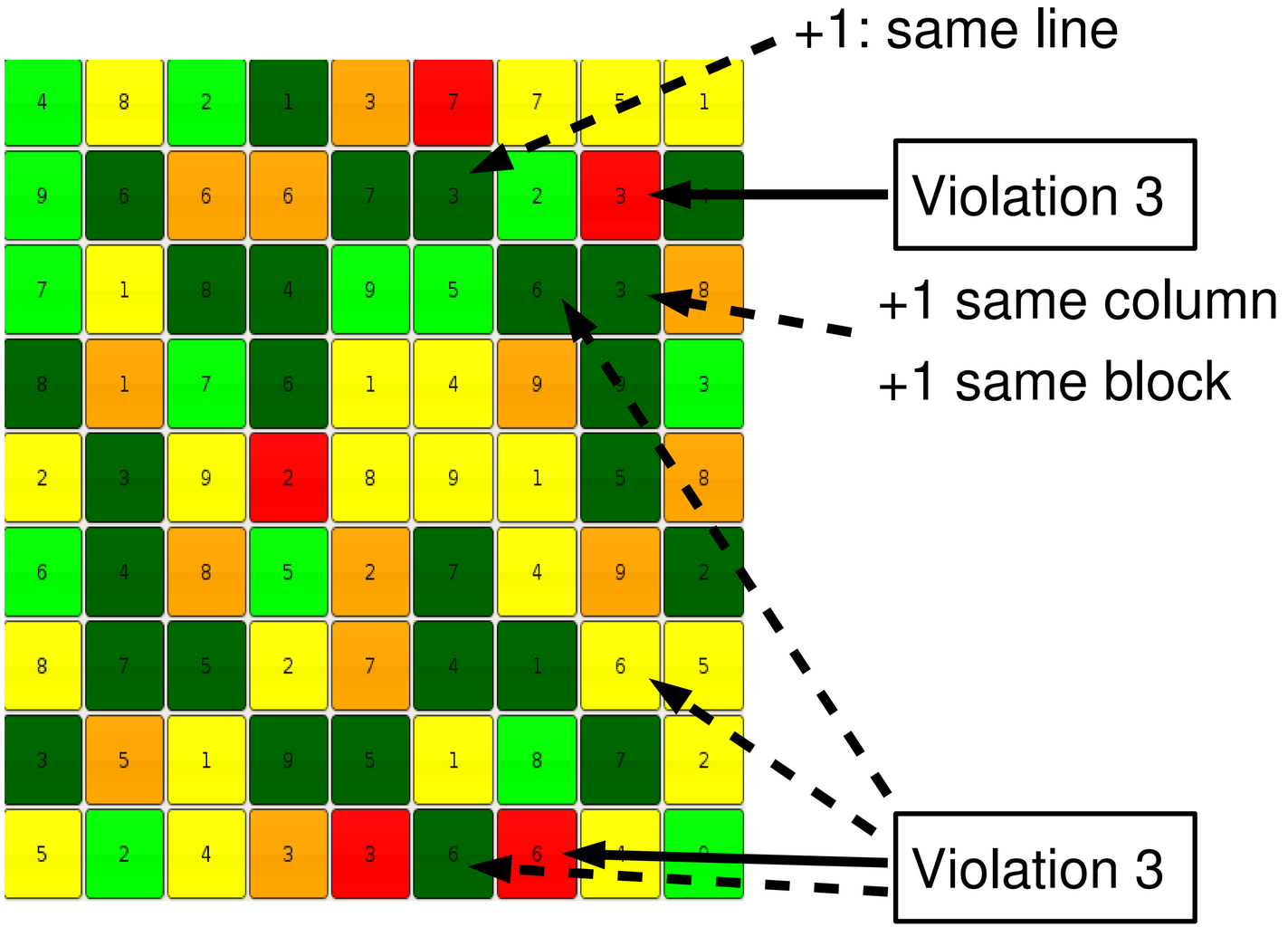
- Initialize the cells with a random number but the number of occurrences of each is OK (9x1, 9x2, …, 9x9) so that we can do Swaps



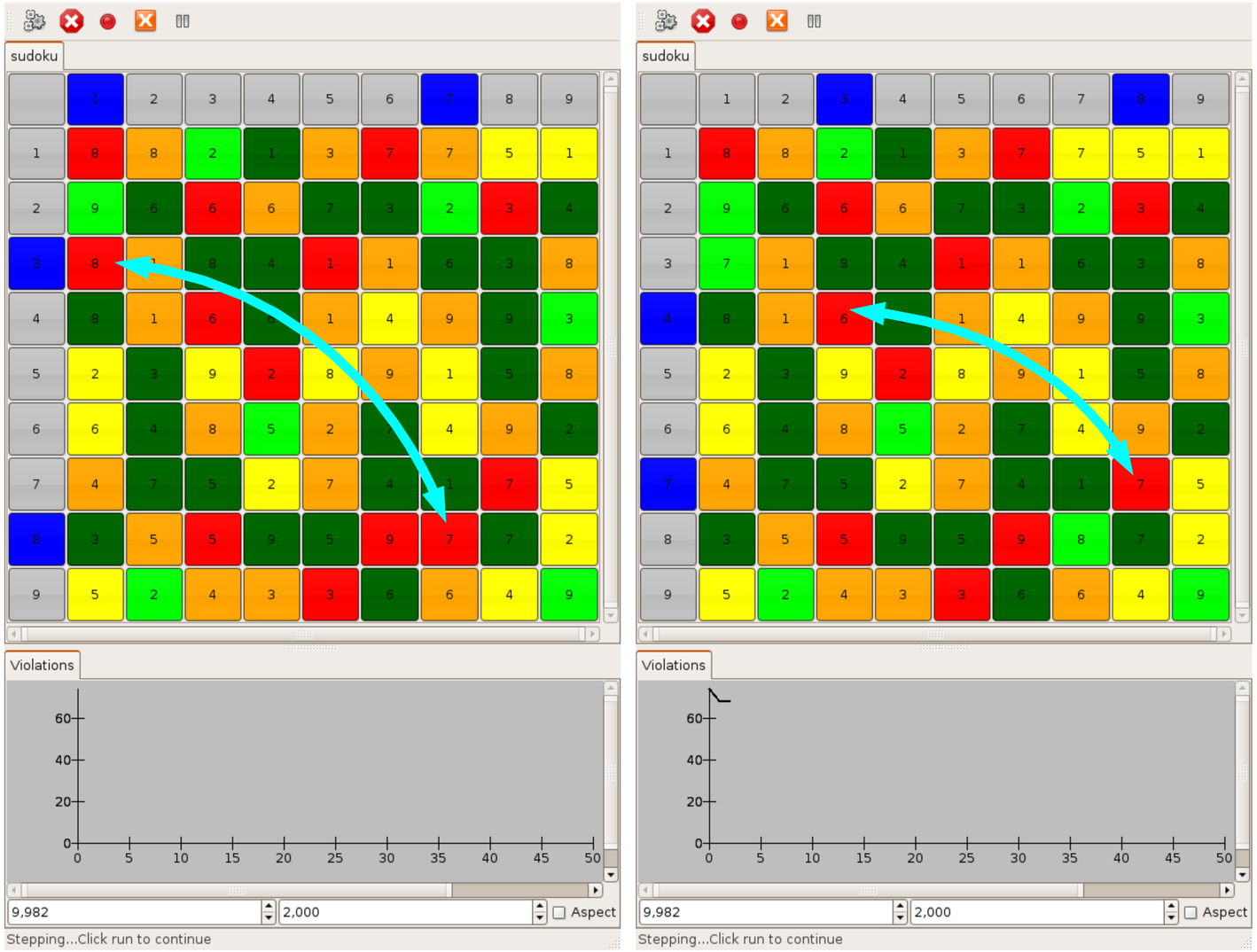
Sudoku Hard and Soft Constraint

- Hard Constraints (cannot be violated)
  - Each number in {1..9} occurs 9x
- Soft Constraints (can be violated)
  - Each number in {1..9} occurs 1x in a row
  - Each number in {1..9} occurs 1x in a column
  - Each number in {1..9} occurs 1x in a 3x3 block

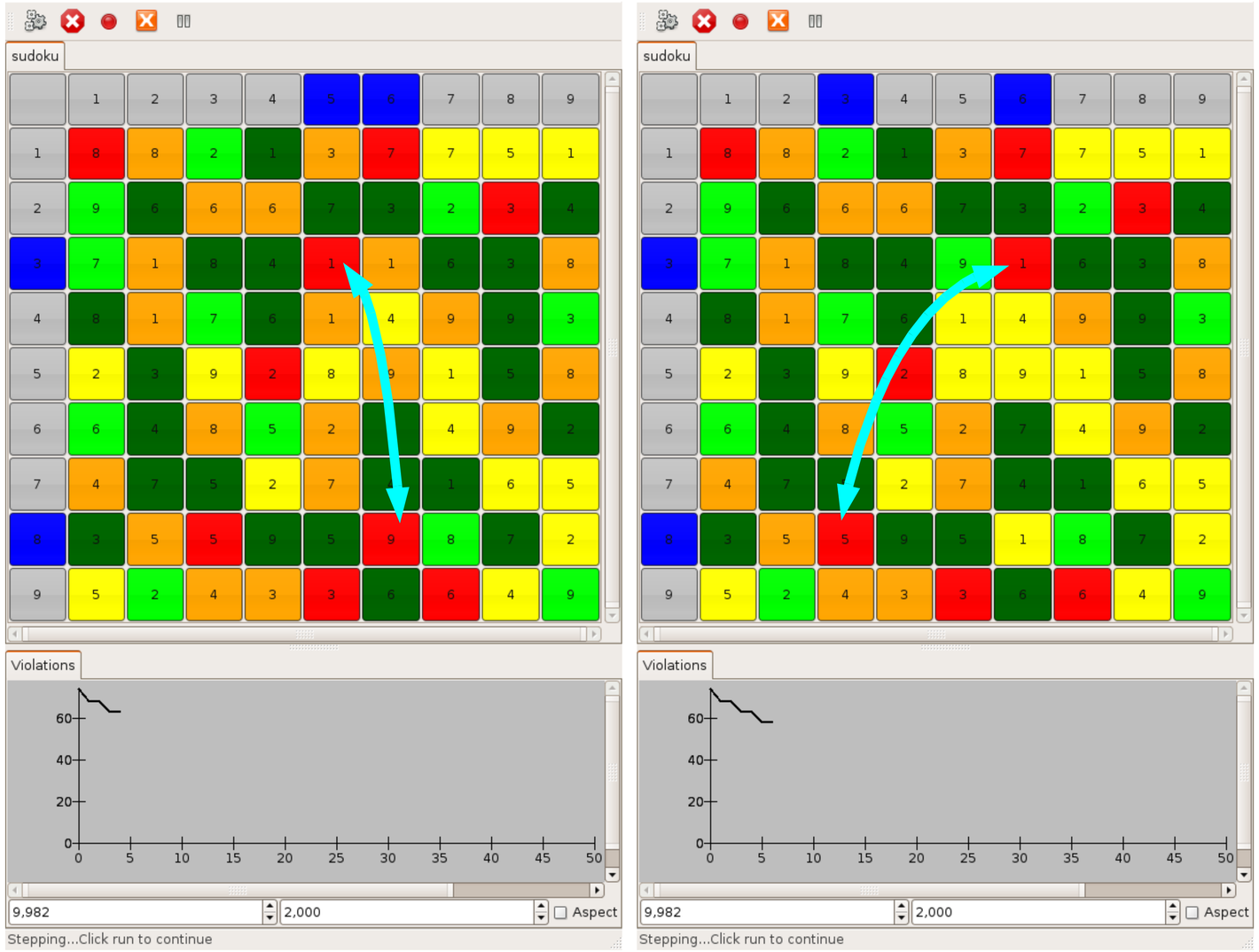
Sudoku: Computation of violation



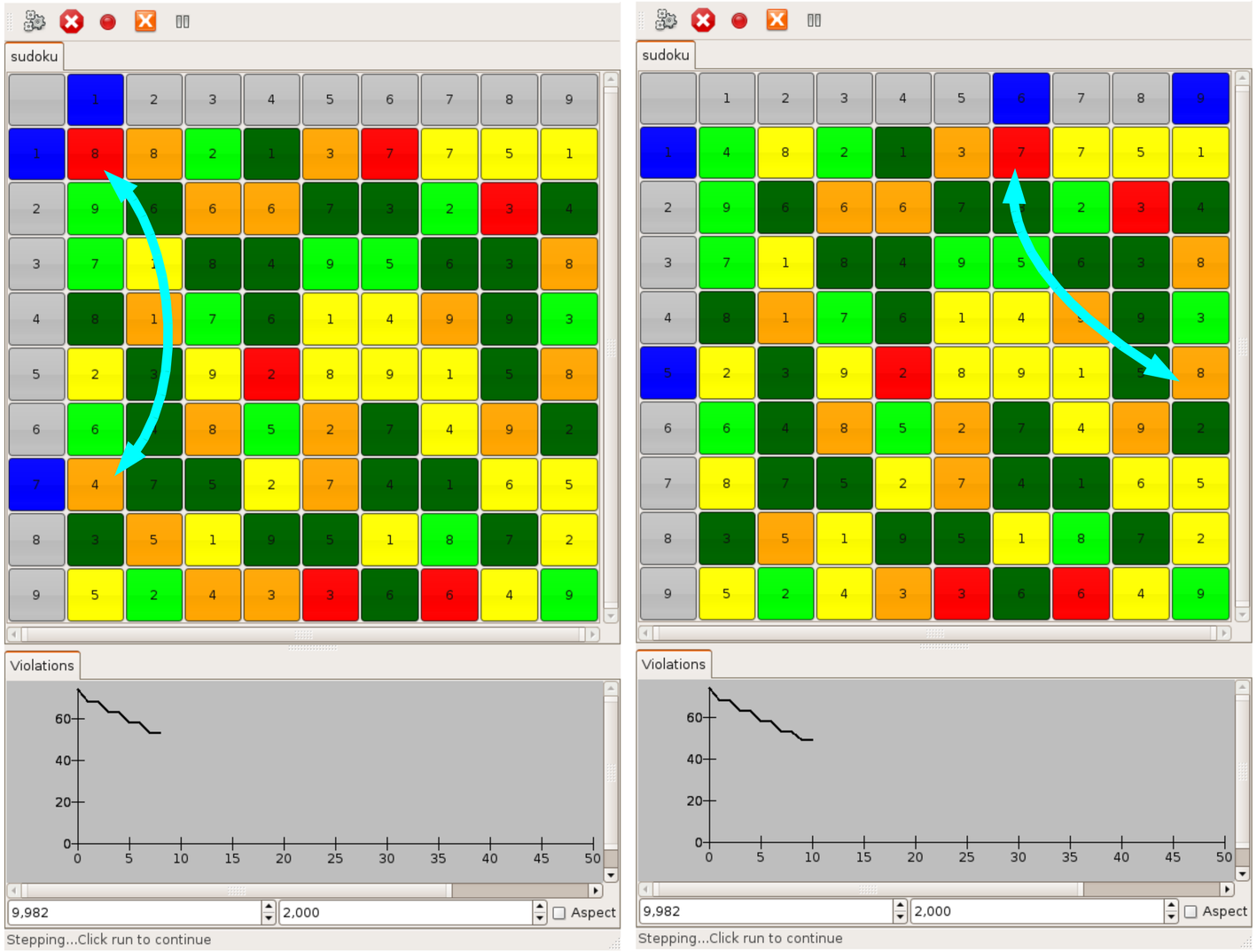
Sudoku: Swap Moves 1



Sudoku: Swap Moves 2

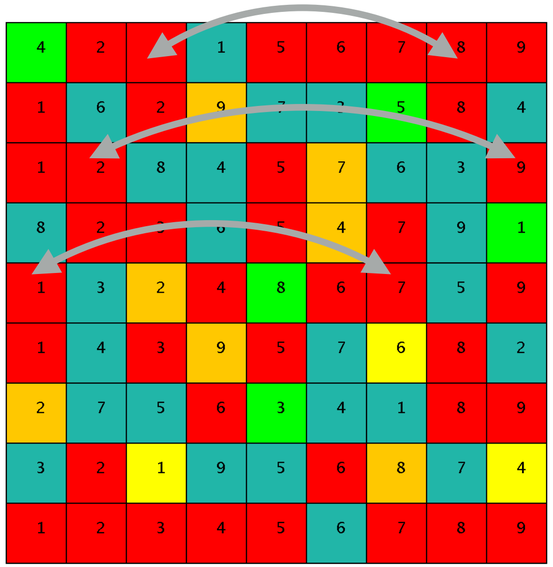


Sudoku: Swap Moves 3



Sudoku Idea 2: Augment then set of hard constraints

- Hard Constraints (cannot be violated)
  - Each number in {1..9} occurs 9x
  - Each number in {1..9} occurs 1x in a row
- Soft Constraints (can be violated)
  - Each number in {1..9} occurs 1x in a block
  - Each number in {1..9} occurs 1x in a column



Hard constraints are enforced by:
1. initialization
2. moves = swaps of two cells on a same row

Implementation

- Using a Constrained Based Local Search (Library)

- Facilitates the development of Local Search algorithms
  - you can add your constraints and objective functions
  - it will compute for you the violations
  - it will compute for you the delta's (what if I exchange the values of two variables)
  - …

- You can focus on the interesting part: moves and meta-heuristics. We will come back to that later, let's first understand the magic

CBLS Sudoku Model

```
int[] instance1 = new int[]{
        0, 0, 0, 1, 0, 0, 0, 0, 0,
        0, 6, 0, 0, 7, 3, 0, 0, 4,
        0, 0, 8, 4, 0, 0, 6, 3, 0,
        8, 0, 0, 6, 0, 0, 0, 9, 0,
        0, 3, 0, 0, 0, 0, 0, 5, 0,
        0, 4, 0, 0, 0, 7, 0, 0, 2,
        0, 7, 5, 0, 0, 4, 1, 0, 0,
        3, 0, 0, 9, 5, 0, 0, 7, 0,
        0, 0, 0, 0, 0, 6, 0, 0, 0};

IntVarLS [] grid;
IntVarLS violation;
ConstraintSystem constraintSystem;
ArrayList<Pair> possibleSwaps;

SolverLS ls = makeSolver();


grid = makeIntVarArray(9*9, i -> makeIntVar(ls,init[i]));


ArrayList<Constraint> constraints = new ArrayList<>();
for (int k = 0; k < 9; k++) {
    final int i = k;
    Constraint allDiffCol = new AllDifferent(makeIntVarArray(n, j -> grid[j * 9 + i]));
    Constraint allDiffBlock = new AllDifferent(makeIntVarArray(n, j -> grid[blocks.get(i).get(j)]));
    constraints.add(allDiffBlock);
    constraints.add(allDiffCol);
}

constraintSystem = new ConstraintSystem(constraints.toArray(new
violation = constraintSystem.violation();
```

4 2 3 1 5 6 7 8 9
1 6 2 9 7 3 5 8 4
1 2 8 4 5 7 6 3 9
8 2 3 6 5 4 7 9 1
1 3 2 4 8 6 7 5 9
1 4 3 9 5 7 6 8 2
2 7 5 6 3 4 1 8 9
3 2 1 9 5 6 8 7 4
1 2 3 4 5 6 7 8 9

decision variables: value in each cell

Constraint for rows and blocks

An object responsible to compute the total violation

CBLS Sudoku Model

```
// swap two cells on the same line
possibleSwaps = new ArrayList<>();
for (int l = 0; l < 9; l++) {
    for (int i = 0; i < 9; i++) {
        for (int j = i+1; j < 9; j++) {
            int v1 = l*9+i;
            int v2 = l*9+j;
            if (problem[v1] == 0 && problem[v2] == 0) {
                possibleSwaps.add(new Pair(v1,v2));
            }
        }
    }
}
public int swapDelta(int a, int b) {
    int before = violation.value();
    swap(a,b);
    int after = violation.value();
    swap(a,b);
    return after-before;
}

public void swap(int a, int b) {
    int va = grid[a].value();
    int vb = grid[b].value();
    grid[a].setValue(vb);
    grid[b].setValue(va);
}
```

Pre-compute all the possible swap position not involving hint position

Compute the delta if exchanging values in position a and b

Exchange values in position a and b

5	8	1	7	3	4	6	9	8
2	9	7	6	8	1	2	5	4
6	3	4	5	9	2	7	1	3
3	4	2	9	5	8	1	6	7
1	7	5	4	6	3	9	8	2
8	6	9	2	1	7	3	4	5
9	1	3	8	2	5	4	7	6
7	5	6	3	4	9	8	2	1
4	2	8	1	7	6	5	3	9

CBLS Sudoku Greedy Search

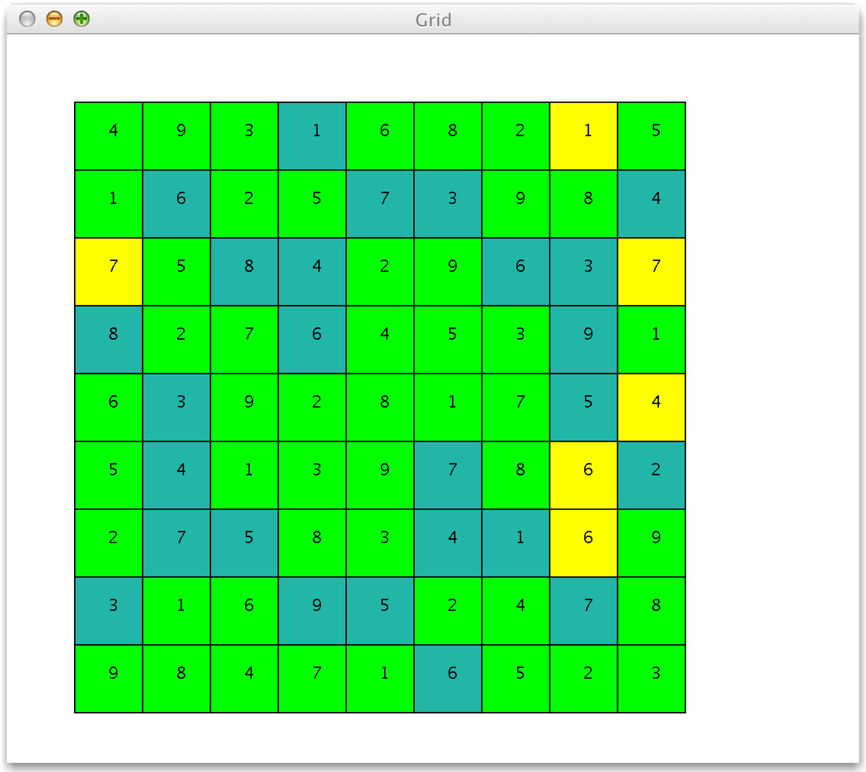
```
public void solve() {
    int iter = 1;
    while (constraintSystem.violation().value() > 0){
        iterationGreedy(iter++);
    }
}
public void iterationGreedy(int iter) {
    Pair bestSwap = bestSwap();
    grid[bestSwap.a].swap(grid[bestSwap.b]);
    notifyAllObservers(iter,bestSwap);
}

public Pair bestSwap() {
    Pair bestSwap = null;
    int bestDelta = Integer.MAX_VALUE;
    for (Pair p : possibleSwaps) {
        int delta = violation.getSwapDelta(grid[p.a],grid[p.b]);
        if (delta < bestDelta) {
            bestDelta = delta;
            bestSwap = p;
        }
    }
    return bestSwap;
}
```

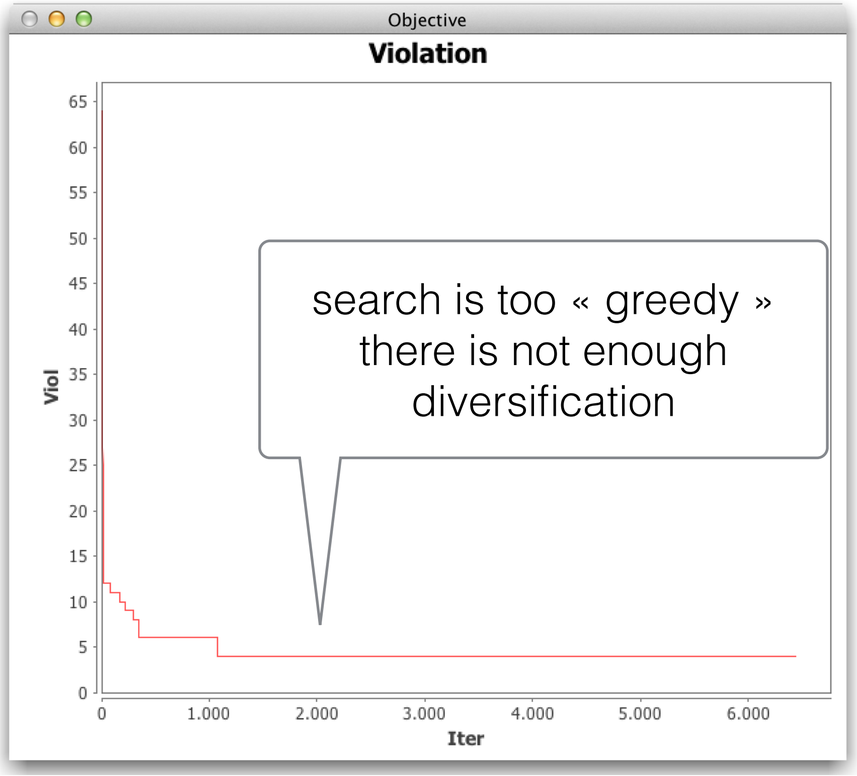
Solve until feasible solution found (zero violation)

One iteration

Find the best exchange

Problem if too greedy





Sudoku Search with Tabu

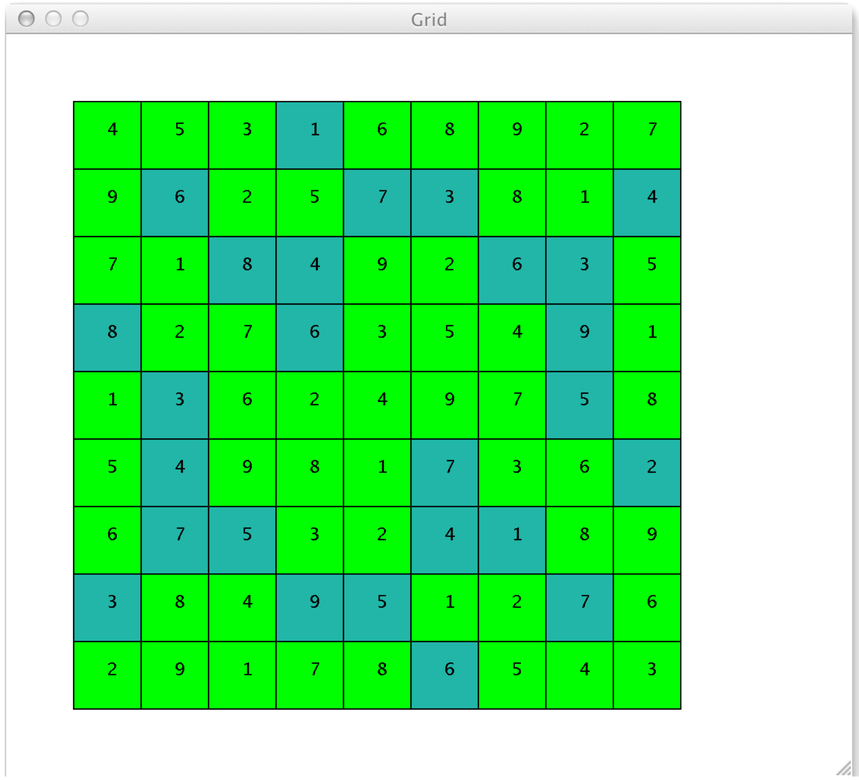
```
public void solve() {
    int iter = 1;
    int tabu = 20;
    while (constraintSystem.violation().value() > 0){
        iterationTabu(iter++,tabu);
        return;
    }
}

public void iterationTabu(int iter, int tabu) {
    Pair bestSwap = bestSwapNonTabu(iter);
    grid[bestSwap.a].swap(grid[bestSwap.b]);
    bestSwap.iter = iter + rand.nextInt(tabu);
    notifyAllObservers(iter,bestSwap);
}

public Pair bestSwapNonTabu(int iter) {
    Pair bestSwap = null;
    int bestDelta = Integer.MAX_VALUE;
    for (Pair p : possibleSwaps) {
        if (p.iter < iter) {
            int delta = violation.getSwapDelta(grid[p.a],grid[p.b]);
            if (delta < bestDelta) {
                bestDelta = delta;
                bestSwap = p;
            }
        }
    }
    return  bestSwap;
}
```

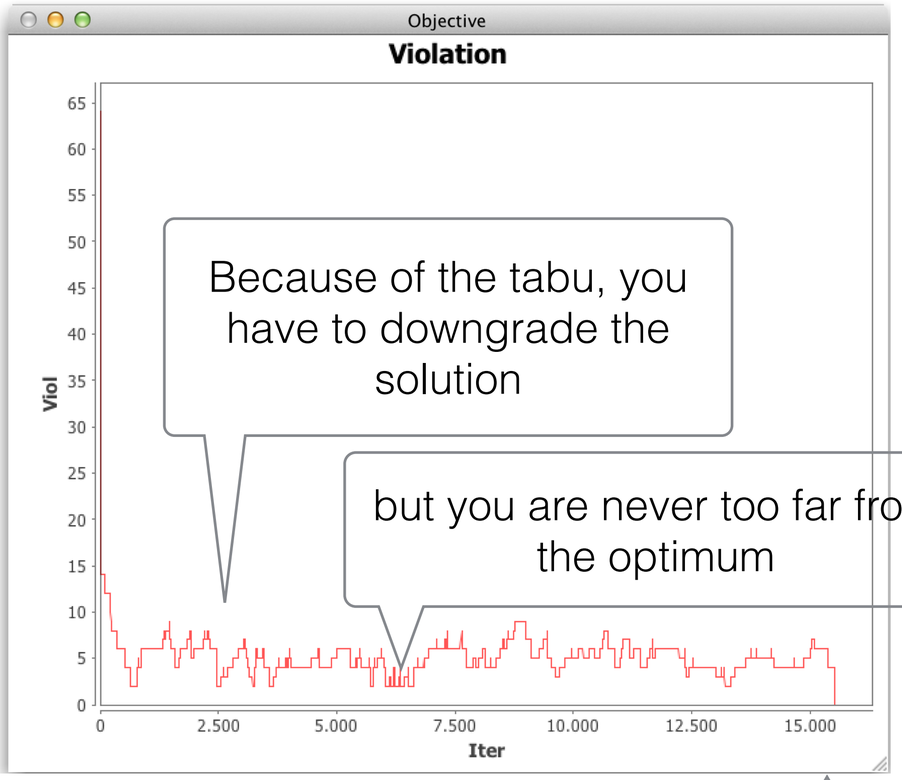
Solve until feasible solution found (zero violation)

One iteration: bestSwap and set this swap tabu for a random number of iterations

Find the best exchange that is not tabu

Sudoku with Tabu Search

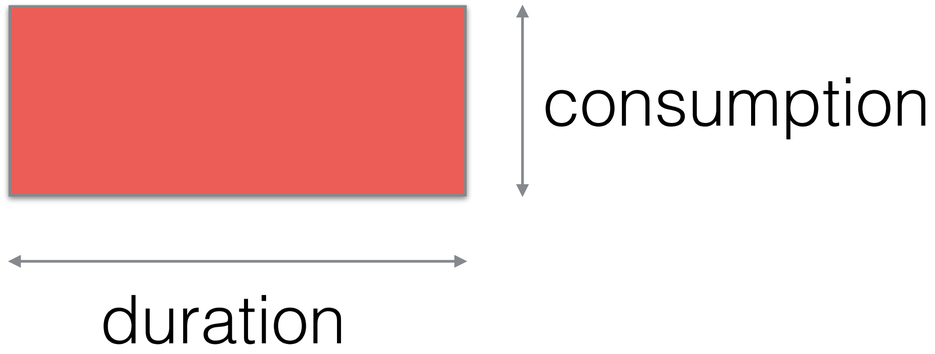


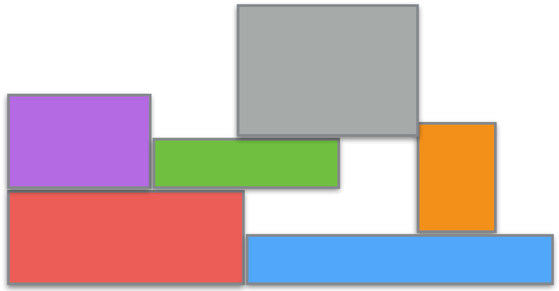


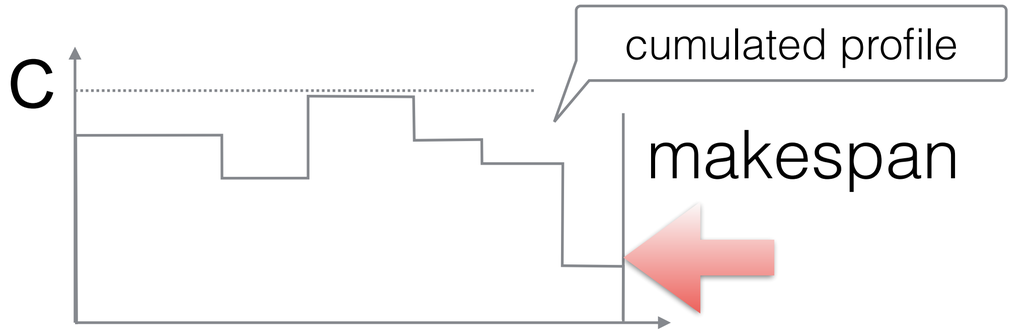
Able to reach 0 violation.
Tabu offers a good Diversification/ Intensification tradeoff

Local Search for Cumulative Scheduling

- Given a set of non preemptive activities (cannot be interrupted)
- A resource with capa C
- How to schedule them to minimize the total duration (makespan) without exceeding the capacity of the resource?







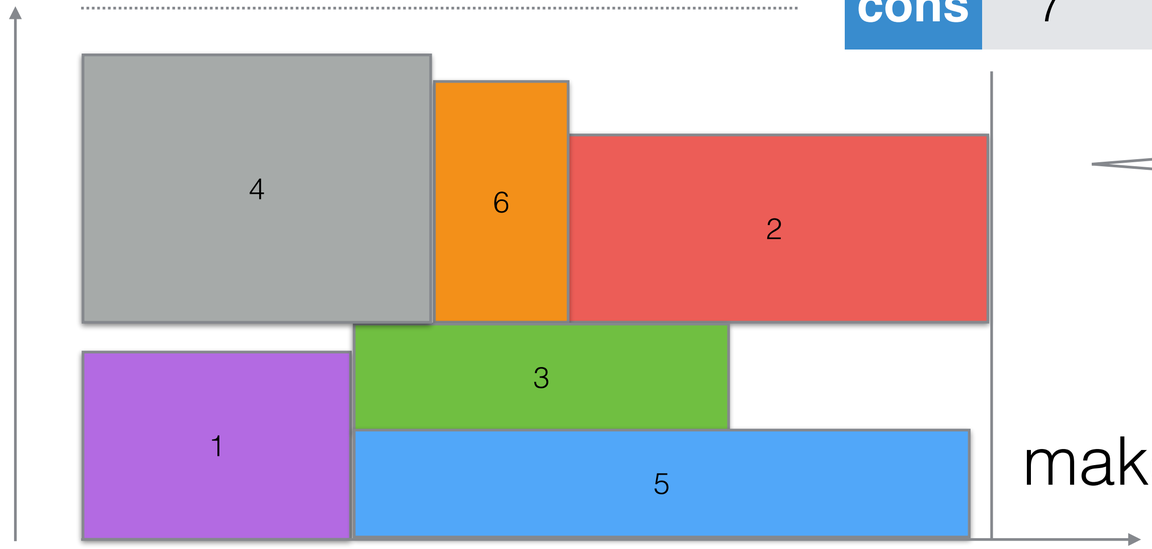
Scheduling with capa: IFlat-IRelax algorithm

- Iterate between two steps:
  1. flatten = add strong precedences constraints until the capacity constraint is satisfied (assuming each activity starts as soon as possible while satisfying the precedence constraints)
  2. relax = remove some precedences randomly on the critical path

Example

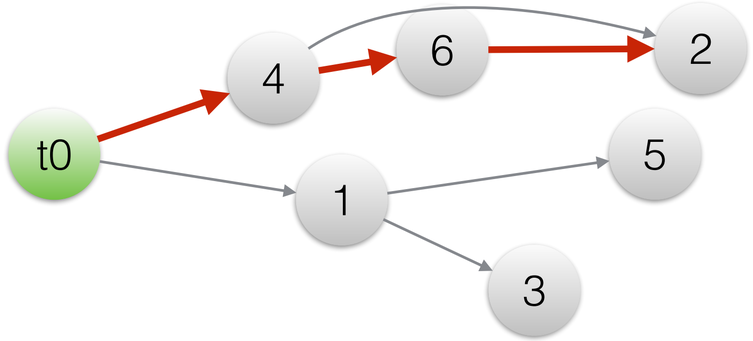
	1	2	3	4	5	6
dur	10	16	14	13	21	5
cons	7	7	4	10	4	9

C=20



each activity scheduled at its earliest start while satisfying the precedences (dynamic programming)

current precedence graph

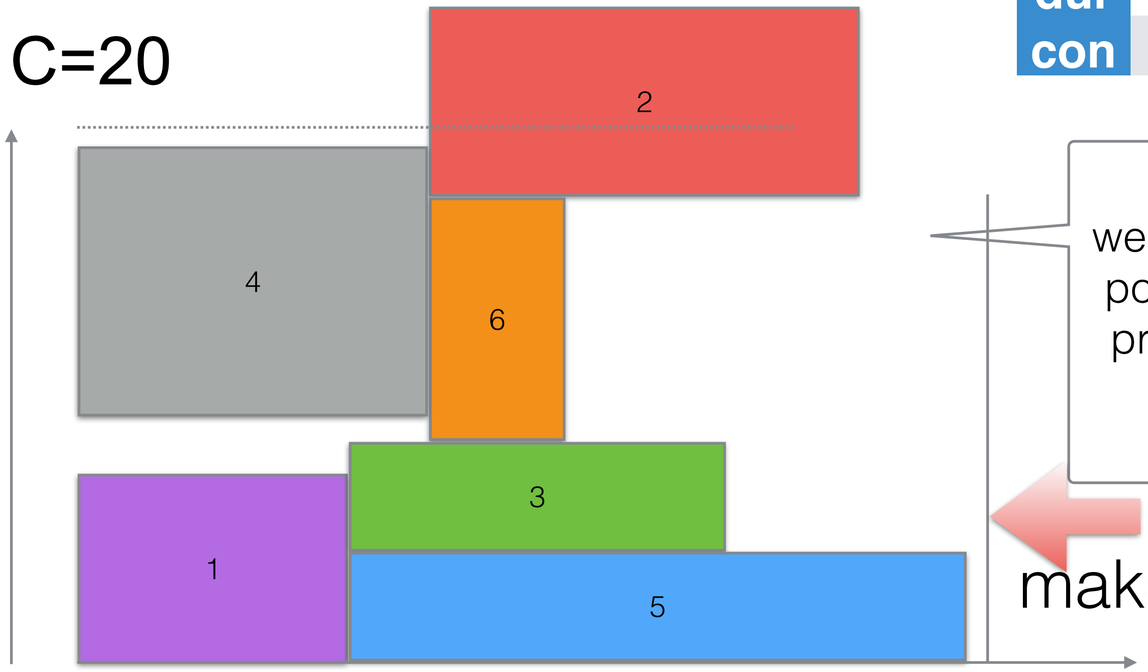


critical path = the path that causes the makespan value

to have a chance to decrease the makespan we have to relax precedences on the critical path (say we remove 6->2)

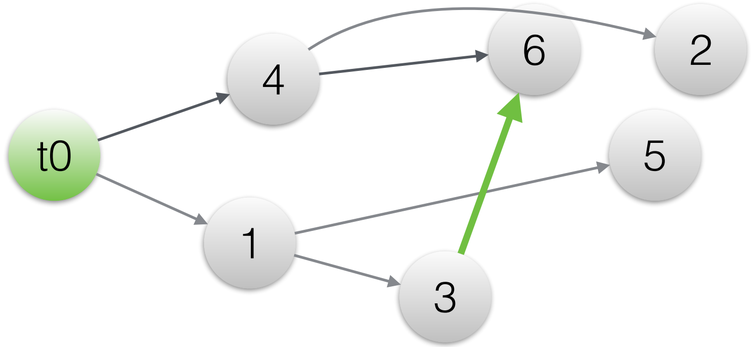
Example

	1	2	3	4	5	6
dur	10	16	14	13	21	5
con	7	7	4	10	4	9



we now exceed the capa at some point. We must flatten by adding precedences between activities where we exceed

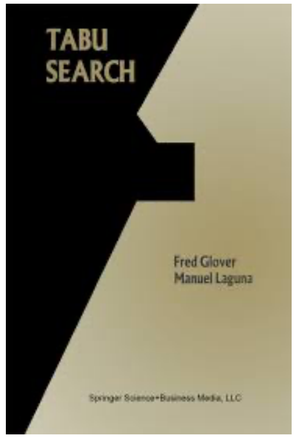
current precedence graph

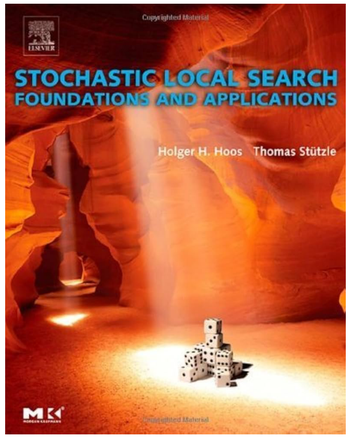


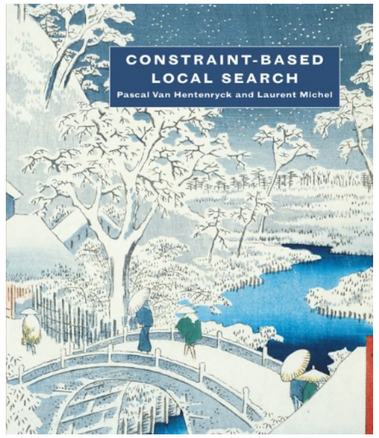
critical path = the path that causes the makespan value

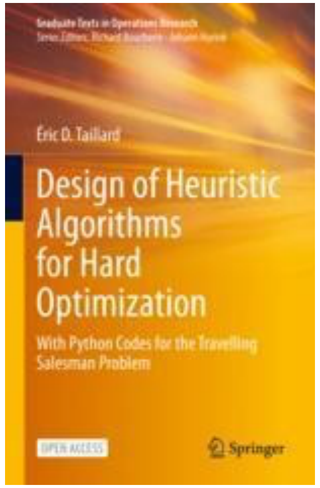
We just scratched the surface of Local Search Topic

- Learning can be added on top of constructive search: Ant Colony Optimization or Greedy Random Adaptive Search methods
- Many other meta-heuristics exists to escape from Local Optima: Simulated Annealing, Variable Objective Search, Iterated Local Search
- Population based meta-heuristics: Genetic and Memetic Algorithms
- Local Search Libraries (Comet, OscaR, OR-Tools, etc)



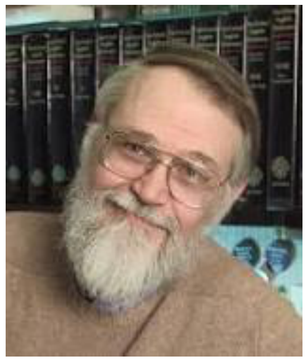






Some Important Inventors

Brian Wilson Kernighan



1942

also coauthor of the AWK and AMPL programming languages

Fred Glover

1937

Tabu Search meta-heuristic

Advanced Algorithms for Optimization

Local Search

Pierre Schaus