Probabilistic Algorithms

Paul Cotofrei

information management institute

PA 2016

Outline

Constraints

- **Constraints** - limitation of the allowed values of the degree of freedom of the system
  - Example: boundaries (localisation constraints)
  - Example: deadline for event occurrence (temporal constraints)
- The constraints may introduce discontinuities or non-analycities for loss function
- Classes of constraints
  - **Hard constraints** - have to met without fail
  - **Soft constraints** - may be violated, but the degree of violation shall be as small as possible

Deal with constraints

Two basic approaches

1. **Incorporation** - integrate the constraint into the configuration and allow only moves (i.e, the selection of a new solution) preserving automatically the feasible configurations
   - advantageous for hard constraints
   - but moves must be powerful enough to reach any feasible solution from any initial configuration in a random walk
   - Example: SIMPLEX algorithm
2. **Only feasible solutions** - start with a feasible solution and accept new solutions only if they are likely feasible
   - if easy to generate a new solution (configuration)
   - reject configurations violating hard constraint
   - usually fail to reach the global solution
   - Example: (Enhanced) Localized random walk

Example - Box Algorithm

Adaptation of non-linear simplex algorithm for constraint problems

Step 0 (Initialisation)

- For each coordinate $\mathbf{d}_i$ of $\mathbf{d} = (\mathbf{d}_1, \mathbf{d}_2, .., \mathbf{d}_p) \in \Theta$ set the bounds $\mathbf{d}_i^{(L)} \leq \mathbf{d}_i \leq \mathbf{d}_i^{(U)}$.
- For each point $\mathbf{d}^{(j)}$, $j = 1...p+1$
  - For $i = 1..p$, generate $\mathbf{d}_i^{(j)}$ uniformly on $(\mathbf{d}_i^{(L)}, \mathbf{d}_i^{(U)})$.
  - If $\mathbf{d}^{(j)}$ is infeasible, calculate the mean $\bar{\mathbf{d}}$ of current generated points and reset $\mathbf{d}^{(j)} = \frac{1}{2}(\bar{\mathbf{d}} + \mathbf{d}^{(j)})$ until $\mathbf{d}^{(j)}$ is feasible

Step 1..3 (Check feasibility) If the new point $\hat{\mathbf{d}}$ (after reflection, expansion or contraction) with $L(\hat{\mathbf{d}}) < L(\mathbf{d}_{max})$ is infeasible, then

(a) If $\hat{\mathbf{d}}_i < \mathbf{d}_i^{(L)}$, set $\hat{\mathbf{d}}_i = \mathbf{d}_i^{(L)}$

(b) If $\hat{\mathbf{d}}_i > \mathbf{d}_i^{(U)}$, set $\hat{\mathbf{d}}_i = \mathbf{d}_i^{(U)}$

(c) If the resulting $\hat{\mathbf{d}}$ is infeasible, set $\hat{\mathbf{d}} = \frac{1}{2}(\bar{\mathbf{d}} + \hat{\mathbf{d}})$ and repeat (a), (b), (c) until $\hat{\mathbf{d}}$ is feasible

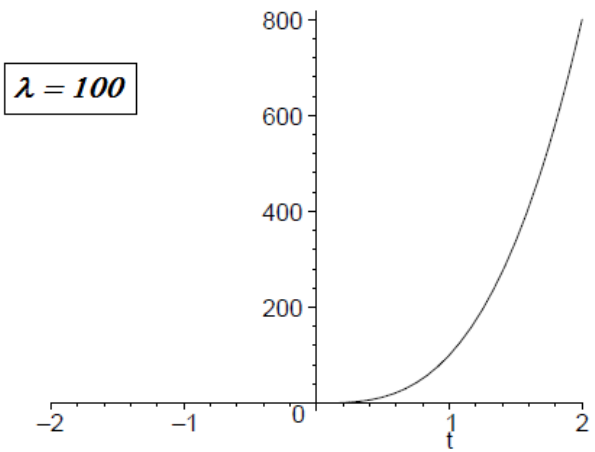
The algorithm is working only for convex feasible regions !

Penalty principle

- **Principle of penalty function** - a solution which violates a constraint is no more rejected, but "punished"
  - the size of the penalty depends of the extend of the violation
  - the non-feasible solutions are lifted-up in the energy landscape
  - a random walk is more likely to end up in a feasible solution in which all constraints are meat
- Additional pseudocosts for L: $\sum_i \lambda_i \times f_i$, where $\lambda_i > 0$ are Lagrange multipliers and f_i are penalty functions
  - λ_i and f_i - chosen so that all f_i are zero at the end of optimization
- Penalty terms - various forms; most common - linear or quadratic with the degree of violation of the constraint (plus an offset)
- A large number of sophisticated penalty functions - difficulties for rnadom walk to find feasible solutions

Penalty functions

- General case: **(P)** minimize $L(\mathbf{d})$ where $\mathbf{d}$ satisfies:
$$\begin{cases} g_i(\mathbf{d}) \leq 0, i = 1..m \\ h_i(\mathbf{d}) = 0, i = 1..m \end{cases}$$
- We denote $\phi(\lambda, t), \lambda > 0, t \in R$ a *penalty function* if:
  1. $\phi()$ continuous
  2. $\phi(\lambda, t) \geq 0, \forall \lambda, t$
  3. $\phi(\lambda, t) = 0$ for $t \leq 0$
  4. $\phi()$ strictly increasing for both $\lambda > 0$ and $t > 0$



- Example: $\phi(\lambda, t) = \begin{cases} 0 & \text{for } t < 0 \\ \lambda^3 t & \text{for } t \geq 0 \end{cases}$

Penalty functions (cont.)

- To solve **(P)** we define a modified loss function

$$\tilde{L}(\mathbf{d}) = L(\mathbf{d}) + \sum_{i=1}^{m} \phi(\alpha_i, g_i(\mathbf{d})) + \sum_{i=1}^{m} (\phi(\beta_i, h_i(\mathbf{d})) + \phi(\beta_i, -h_i(\mathbf{d})))$$

- α_i, β_i - positive constants controlling how strongly the constraints will be enforced
- In the first sum, if a constraint g_i is violated, a penalty ($\phi(\alpha_i, g_i)$) is invoked
- In the second sum, the violation of equality constraints is invoked (both for $h_i < 0$ and $h_i > 0$)
- General principle:
  (a) minimize the unconstraint loss function $\tilde{L}(\mathbf{d})$ for a given set (α_i, β_i); let be $\hat{\mathbf{d}}_n$ the last estimate
  (b) increase α_i and β_i and start again the minimization process starting with $\hat{\mathbf{d}}_n$
  (c) stop when a sufficiently accurate minimum is obtained

Example

- Minimize $L(x, y) = x^2 + y^2$ under the constraints

$$\begin{cases} g_1(x, y) = 6 - x - 2y & \leq 0 \\ h_1(x, y) = 3 - x + y & = 0 \end{cases}$$

- Consider the penalty function $\phi(\lambda, t) = \begin{cases} 0 & \text{for } t < 0 \\ \lambda^3 t & \text{for } t \geq 0 \end{cases}$
- Start with $\alpha_1 = 5$ and $\beta_1 = 5$. The modified loss function is

$$\tilde{L}(x, y) = L(x, y) + \phi(5, g_1(x, y)) + \phi(5, h_1(x, y)) + \phi(5, -h_1(x, y))$$

$$\tilde{L}(x, y) = x^2 + y^2 + \phi(5, 6 - x - y) + \phi(5, 3 - x + y) + \phi(5, -3 + x - y)$$

- A first call of Newton-Raphson algorithm returns $\hat{x}_1 = 3.506$, $\hat{y}_1 = 1.010$. This solution violates the constraints g_1 and h_1.
- A second call, using $\alpha_1 = \beta_1 = 50$ and starting from $(\hat{x}_1, \hat{y}_1)$ returns $\hat{x}_2 = 3.836$ and $\hat{y}_2 = 1.008$;
- A third call, using $\alpha_1 = \beta_1 = 500$ returns $\hat{x}_3 = 3.947$ and $\hat{y}_3 = 1.003$; the optimal solution for $L(x, y)$ is $x = 4$, $y = 1$
- Increasing the penalty parameters does improve the accuracy of the final answer, but it will also slow down the unconstrained algorithm's convergence

Outline

Necessity of heuristics

- Exact optimization algorithms
  - For some complex, huge size problems - large amount of calculation time
  - Years of research to improve and customize the algorithms
  - The problem can not reflect the real-word situation
- Heuristic optimization algorithms
  - Easy and fast to implement
  - More less time to return a solution
  - Return a locally optimum solution (may be the global optimum)
  - For problems accepting a near-optimum, or quasi-optimum solution
  - Easy to adjust parameters for other problems

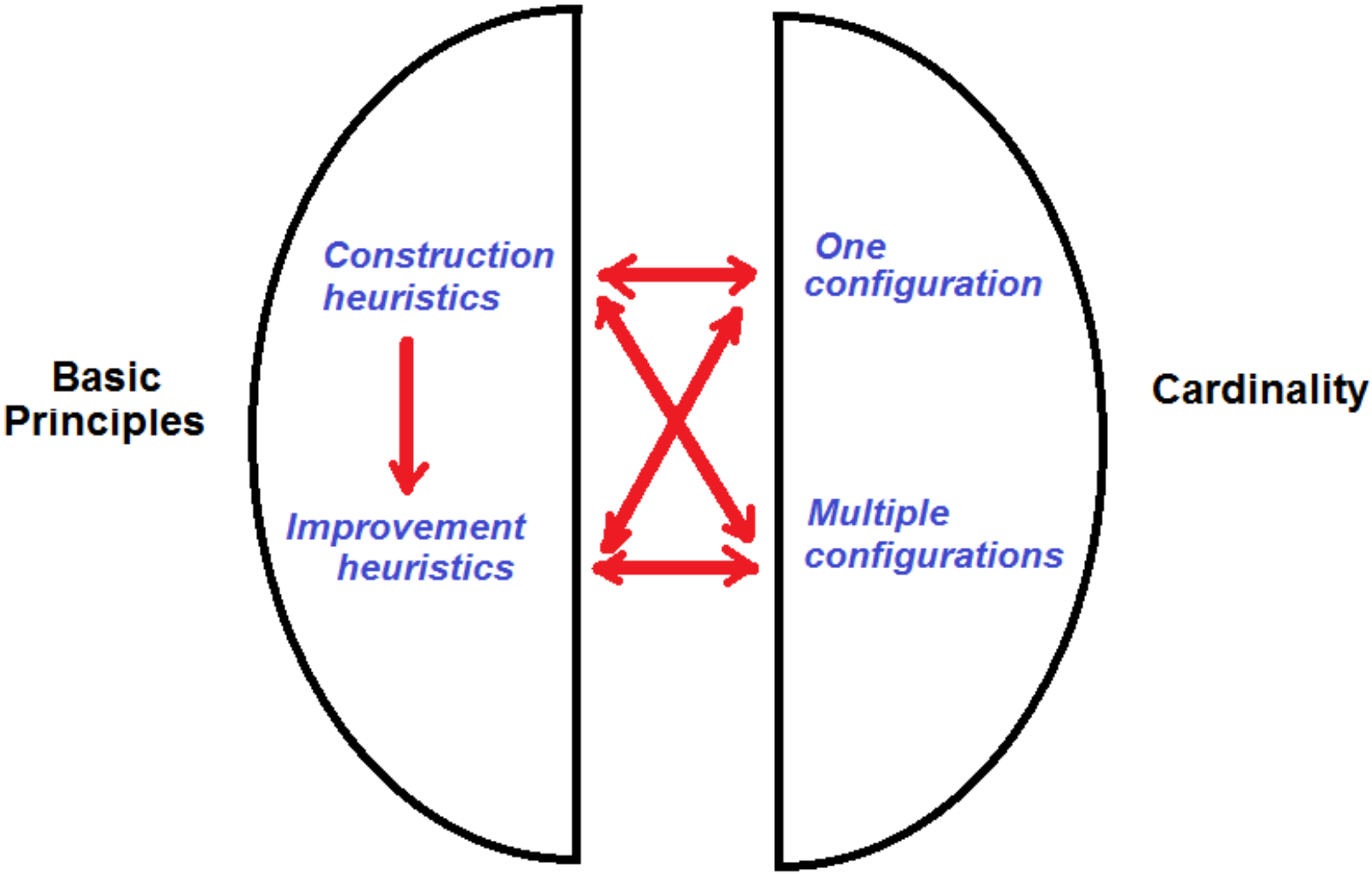
Main approaches for heuristics

- Two natural ways to design an heuristic algorithm
- Construction heuristics
  - The puzzle principle: at each step includes that part of the system in at least locally optimum way
  - This approach does not (or partially) use the synergy effects between different parts of the system
  - Necessity of global approaches to keep the hole system in view
- Markovian improvement heuristics
  - Try to improve the actual configuration by changing it using a sequence of moves
  - Each configuration $\hat{\mathbf{d}}$ has a neighborhood structure $\mathcal{N}(\hat{\mathbf{d}})$ defined as the set of configurations $\mathbf{d}_i$ attainable by applying one possible move
  - Jump from actual configuration $\hat{\mathbf{d}}_k$ to $\hat{\mathbf{d}}_{k+1}$, where $\hat{\mathbf{d}}_{k+1} \in \mathcal{N}(\hat{\mathbf{d}}_k)$
  - The set of moves defines/implies the neighborhood structure $\mathcal{N}(\hat{\mathbf{d}})$
  - The set of configurations + the neighborhood structure = the **search space** of the problem.

Set-based heuristics

- An heuristic may construct/improve, at each step, a single configuration or multiple configurations
- Heuristics using sets of configurations (populations)
  - Construction heuristics: Ant Colony Optimisation (ACO), Particle Swarm Optimisation (PSO)
  - Improvement heuristics: Genetic algorithms (GA), Evolution strategies (ES)
    - New configurations are created (mutation, crossover moves)
    - Darwin's principle: only the configurations with lower loss function value are "surviving"
- Heuristics using sequences of configurations
  - Memory-based search strategy
  - Tabu search, Multicanonical Algorithm, Guided Local Search - store information about previously energies, properties, former configurations, or moves performed.
  - Memory updated by forbidding old visited solutions or structures common to former configurations no longer allowed

General big picture



Outline

General outline

- **Construction/augmentation heuristics** - used to find quickly a reasonable good solution or to provide a a good initial solution for improvement heuristics
- **Initialization phase**
  - use a "tabula rasa"
  - use an optimum solution for a small subsystem of the problem
  - use an extended system (more dimensions), integrating the constraints
- **Selection phase**
  - randomly
  - rule set
- **Placement phase** - the selected item have to be placed somewhere in the system according to some rules
- **Cleaning phase** - the already achieved solution shall be improved according to some rules (for some heuristics)

Insertion heuristics

- **Insertion heuristic** - starts with "tabula rasa", one piece or small subsystem
  - if "tabula rasa" - a piece is selected randomly or according to some rule set
  - if one piece - the next piece is selected using a nearest-neighbor strategy
  - if a small subsystem - a nearest-neighbor strategy, applied for one or a subset of pieces (the strategy may change during the optimization process)
- **Geometric insertion heuristics**
  - from top to bottom, from left to right, from the center to the borders, etc..
- **More intelligent approaches** - add backtracking methods
  - remove an inserted piece, or move it to another place in the system
  - unrestricted backtracking - optimum solution with an exponential cost
  - good balance between the degree of randomness and the set of stiff rules

Construction heuristics for TSP

- Consider a TSP defined by a matrix $d_{n \times n}$ of distances
- Construct a solution (permutation $\sigma(n)$) node by node
- The simplest approach: **Nearest Neighbor Heuristic**

Step 1. Start with a random selected node i.

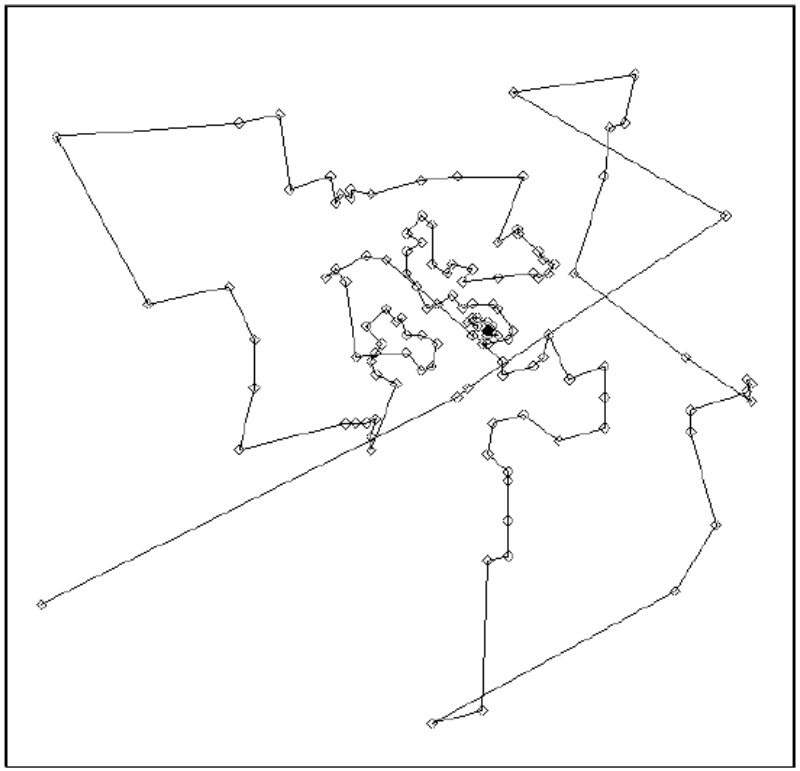
Set the counter c (the number of nodes in the tour) as 1.

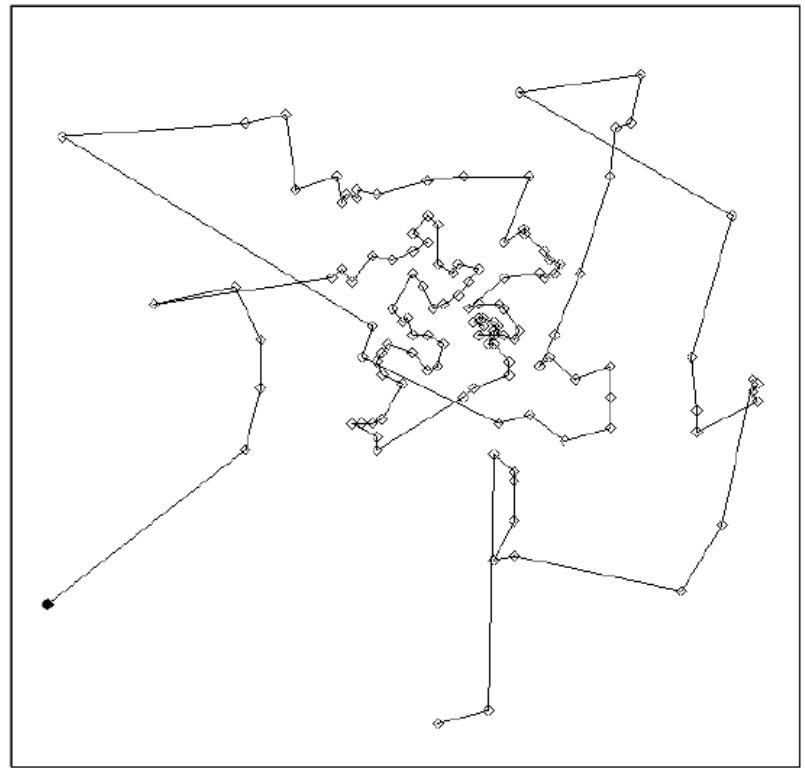
Set $\sigma(c) = i$

Step 2. While $c < n$

1. Let $j = \sigma(c)$.
2. Select i_{new} such that $d(j, i_{new}) = \min_k \{d(j,k) | k \text{ not yet selected}\}$.
3. Set $c = c + 1; \sigma(c) = i_{new}$

Graphical example

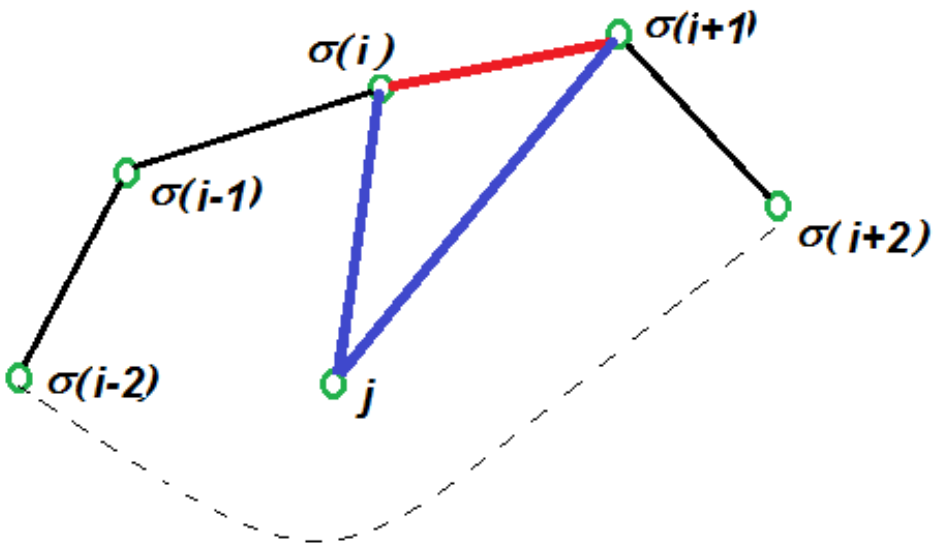




Solution for a TSP instance with 127 nodes (in black the first selected node).
The line connecting the last node with the first one is omitted.

Insertion heuristics

- **Insertion Heuristics** - the new node is inserted somewhere inside the sequence (not appended)
  - How to choose the new node?
  - Where is the optimum place to insert it?
- **Best Insertion** heuristic



$$\Delta = d(\sigma(i), j) + d(j, \sigma(i+1), j) - d(\sigma(i), \sigma(i+1))$$

Best insertion heuristic

Step 1. Select randomly three nodes: $\sigma(1) = i_1, \sigma(2) = i_2, \sigma(3) = i_3$. Set $c = 3$

Step 2. While $c < n$

1. Select randomly a not yet selected node k.
2. Calculate the increase of the path length when inserting k between two successive nodes $\sigma(i)$ and $\sigma(i+1)$:
   $\Delta_{ki} = d(\sigma(i), k) + d(k, \sigma(i+1)) - d(\sigma(i), \sigma(i+1))$,
   $i = 1..(c-1)$
   $\Delta_{kc} = d(\sigma(c), k) + d(k, \sigma(1)) - d(\sigma(c), \sigma(1))$
3. Select the pair for which Δ_{ki} is minimum and apply the insertion. Set $c = c + 1$.

Insertion heuristics

"De-Randomize" the heuristic by selecting the inserting node with a deterministic rule

- **Best-Best insertion** heuristic (or **cheapest** insertion heuristic)

Step 1 Select randomly a node i_1. Set $c = 1$ and $\sigma(c) = i_1$

Step 2 Select the node i_2 such that $d(i_1, i_2) = \min_j\{d(i_1, j)\}$. Set $c = 2$ and $\sigma(c) = i_2$

Step 3 While $c < n$

1. For each node k not yet included in the path, calculate $\delta(k, j_k) = \min_i \Delta_{ki}$, where $(\sigma(j_k), \sigma(j_k + 1))$ is the best place to insert the node k.
2. Select the node $\hat{k}$ such that $\delta(\hat{k}, j_{\hat{k}}) = \min_k \delta(k, j_k)$.
3. Insert the node $\hat{k}$ between $\sigma(j_{\hat{k}})$ and $\sigma(j_{\hat{k}} + 1)$ and set $c = c + 1$

- **Worst-Best insertion** heuristic (or **farthest** insertion heuristic)

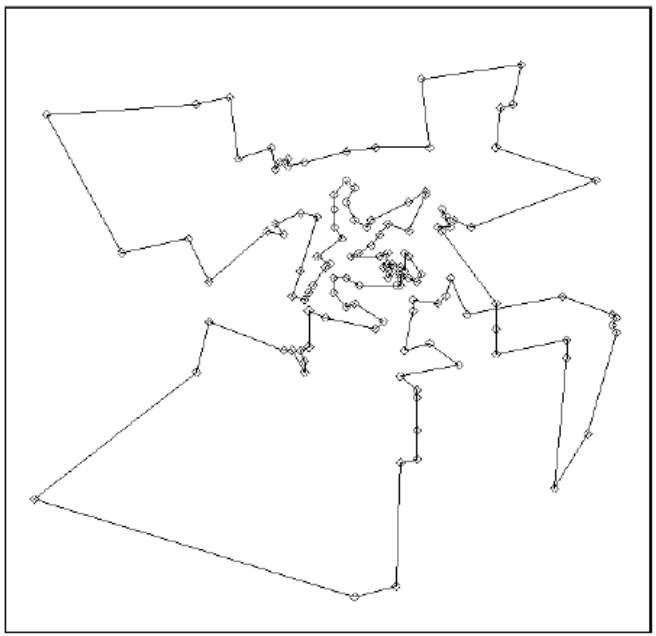
Step 3.2 Select the node k_0 such that $\delta(k_0) = \max_k \delta(k)$

- **Best-Worst insertion** heuristic

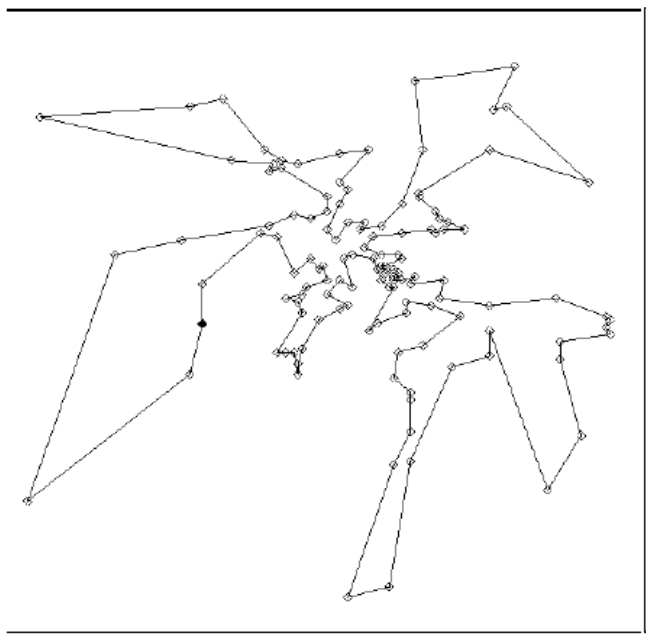
Step 3.1 Calculate $\delta'(k, j_k) = \max_i \Delta_{ki}$

Step 3.2 Select the node $\hat{k}$ such that $\delta'(\hat{k}, j_{\hat{k}}) = \min_k \delta'(k, j_k)$.

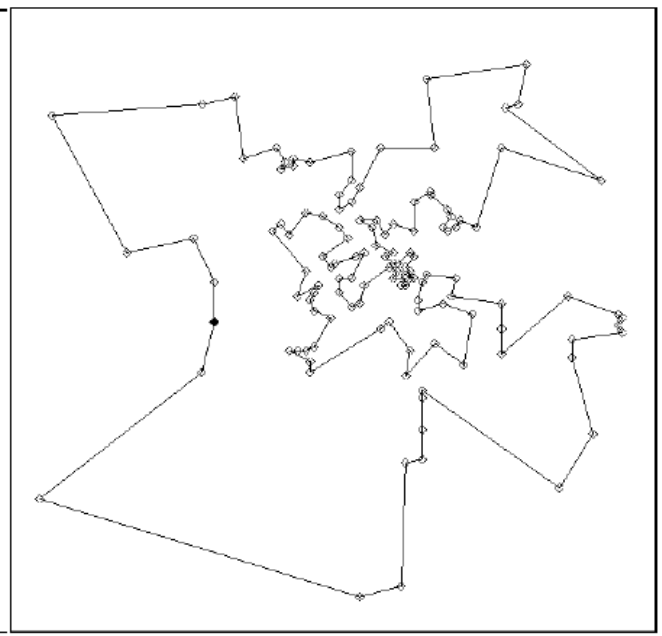
Graphical example



Best Insertion heuristic



Cheapest Insertion heuristic



Farthest Insertion heuristic

Insertion heuristics

- **Nearest insertion** heuristic

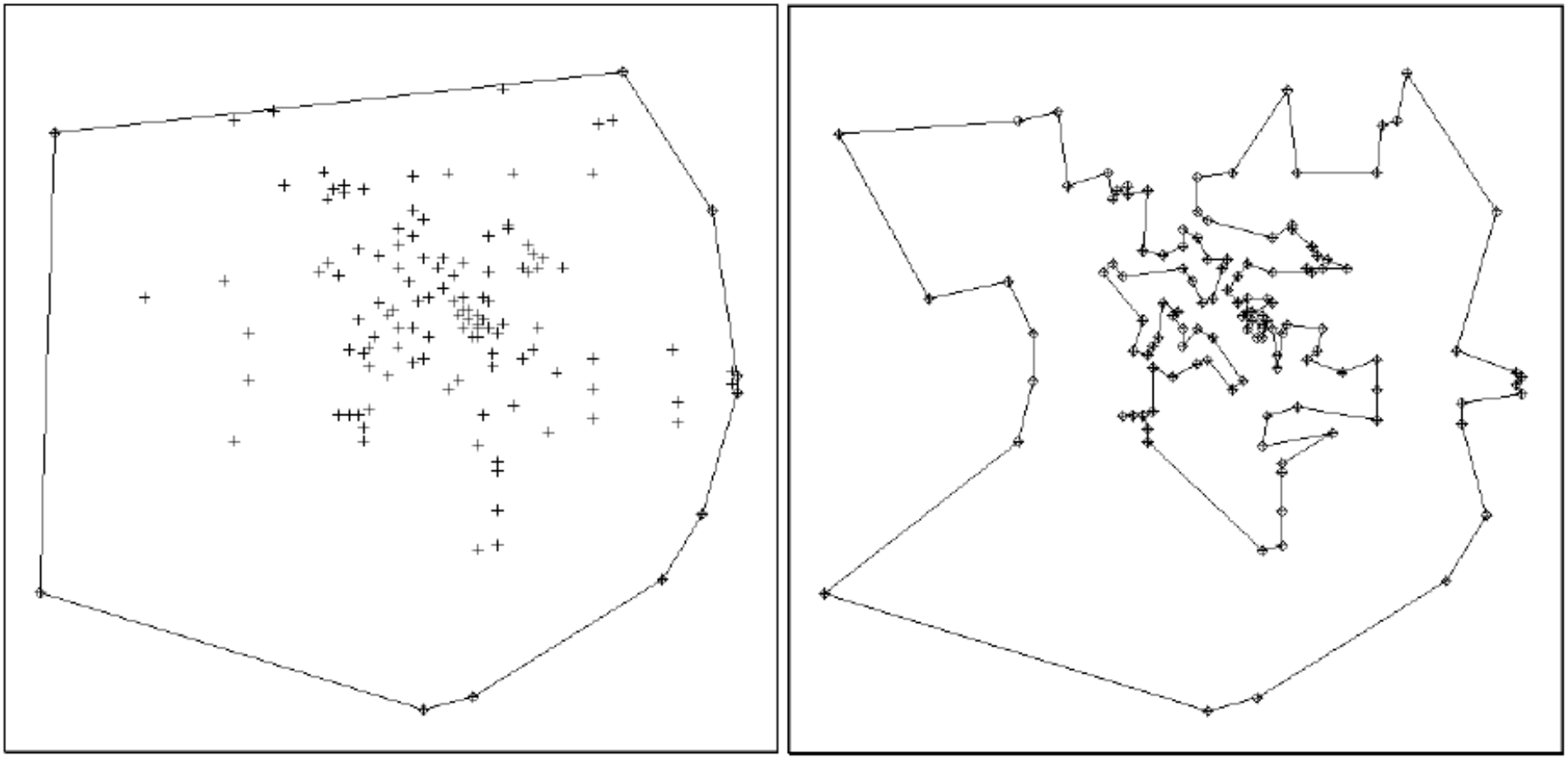
Step 2.1 Select a not yet selected node, having the shortest distance regarding to any node in the tour
or
Step 2.1 Select a not yet selected node for which the minimum distance to one of the node in the tour is maximal

- Convex Hull (CH) - the smallest area covering all nodes
  - Outer nodes - the nodes situated on the perimeter of CH
  - The order in which the outer nodes are arranged is *the same order in which they lie in the optimum solution*

- **Convex hall insertion** heuristic

Step 1. Find the convex hull of all nodes and consider it as initial tour. Set $c =$ *length of convex hall*.

Graphical example



The initial and the final stage of convex hall insertion heuristic

Shortest edge heuristic

- The solution is expressed not as a permutation of n nodes, but as a subset E_H of n edges from the complete set of edges, satisfying a number of constraints.
- Rules for adding an edge to a partial solution
  - An edge may be added to a partial tour as long as:
    1. it does not create a cycle with less than n nodes, and
    2. it does not increase the degree of a node to more than 2

Shortest edge heuristic

Step 1. Sort all edges; Set c = 1;

step 2. Select the shortest, not yet selected edge e, satisfying the constraints 1 and 2;

Step 3. Add the edge e to the tour and set $c = c + 1$

Step 4. Repeat Step 2 until $c = n$

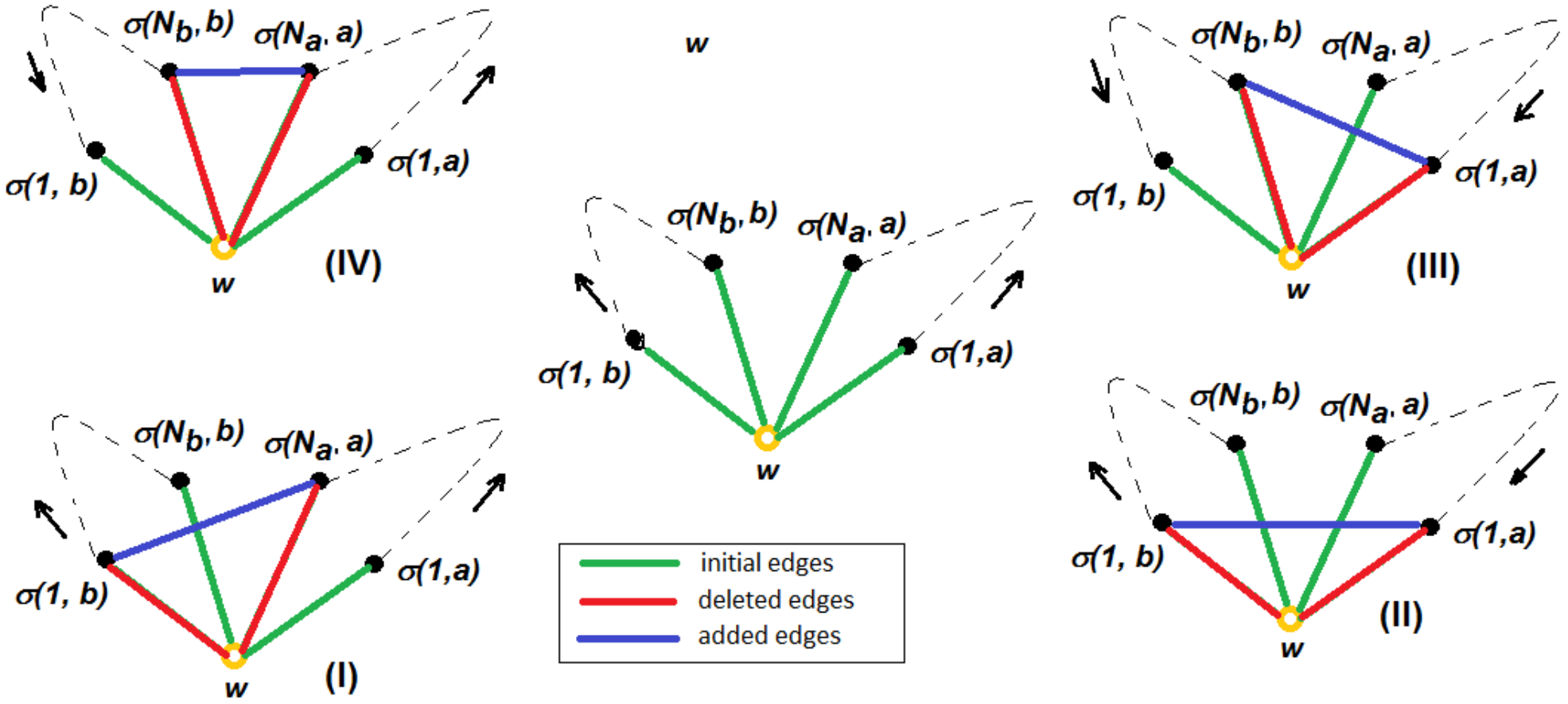
Saving Heuristic

- Insertion heuristics use an "egoistic" approach: each node optimizes only its cost
- An approach "in parallel" could search synergy effects
- **Saving** heuristic

  Init Select an initial node w, denoted "warehouse". Consider $N-1$ agents traveling from w to one of $N-1$ other nodes (no city is visited by more than one agent). Denote $\sigma(i,j)$ the node i in the tour of the j agent. In the initialization step, $\sigma(1,j)$ is the only node visited by agent j (the "warehouse" is not represented, being implicit).

  Saving Consider two tours corresponding to agents a and b. Denote N_a (respectively N_b) the last node of tour a (respectively b). There are four possibilities to merge the tours into a single tour which start and end in d. Calculate the length difference that can be achieved if the two tours are merged.

Graphical example



Four possibilities to merge two partial tours having at least 3 nodes

Saving Heuristic (cont.)

- $$S_1(a,b) = d(\sigma(N_a,a),w) + d(w,\sigma(1,b)) - d(\sigma(N_a,a),\sigma(1,b))$$
  $$S_2(a,b) = d(\sigma(1,a),w) + d(w,\sigma(1,b)) - d(\sigma(1,a),\sigma(1,b))$$
  $$S_3(a,b) = d(\sigma(N_b,b),w) + d(w,\sigma(1,a)) - d(\sigma(N_b,b),\sigma(1,a))$$
  $$S_4(a,b) = d(\sigma(N_a,a),w) + d(w,\sigma(N_b,b)) - d(\sigma(N_a,a),\sigma(N_b,b))$$

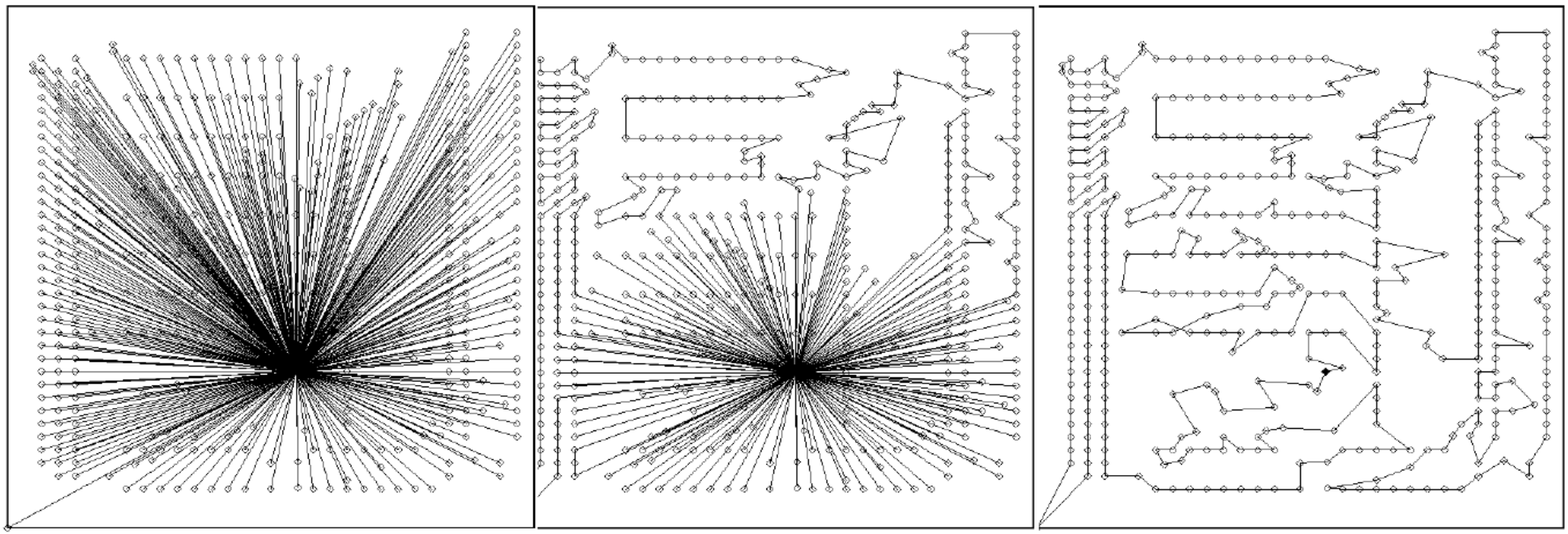
  The savings for merging the tours a and b is given by

  $$S(a,b) = \max\{S_1(a,b), S_2(a,b), S_3(a,b), S_4(a,b)\}$$

Merge Consider the tours $\tilde{a}$ and $\tilde{b}$ for which $S(\tilde{a},\tilde{b})$ is maximal among all possible savings $S(a,b)$.

- If $S(\tilde{a},\tilde{b}) = S_1(\tilde{a},\tilde{b})$ then the tour $\tilde{b}$ is appended at the end of tour $\tilde{a}$
- If $S(\tilde{a},\tilde{b}) = S_2(\tilde{a},\tilde{b})$ then the direction of the tour $\tilde{a}$ is firstly reversed and the tour $\tilde{b}$ is appended at the end of tour $\tilde{a}$
- If $S(\tilde{a},\tilde{b}) = S_3(\tilde{a},\tilde{b})$ then the tour $\tilde{a}$ is appended at the end of tour $\tilde{b}$
- If $S(\tilde{a},\tilde{b}) = S_4(\tilde{a},\tilde{b})$ then the direction of the tour $\tilde{b}$ is firstly reversed and then appended at the end of tour $\tilde{a}$

Graphical example



The initial, an intermediate and the final stage of saving heuristic

Nearest merged heuristic

- Weak points of savings algorithm:
  1. Two tours already merged cannot be split again
  2. One tour cannot be inserted, but only appended to other tour
- **Nearest merger** heuristic

Step 0 Consider N tours, each starting and ending in one of the N nodes. Let be $c = N$ the number of tours;

Step 1 While $c > 1$ do

1 For each pair of tours a, b, calculate the distance between them: $\delta(a,b) = \min_{i \le N_a, j \le N_b} (d(\sigma(i,a), \sigma(j,b))$

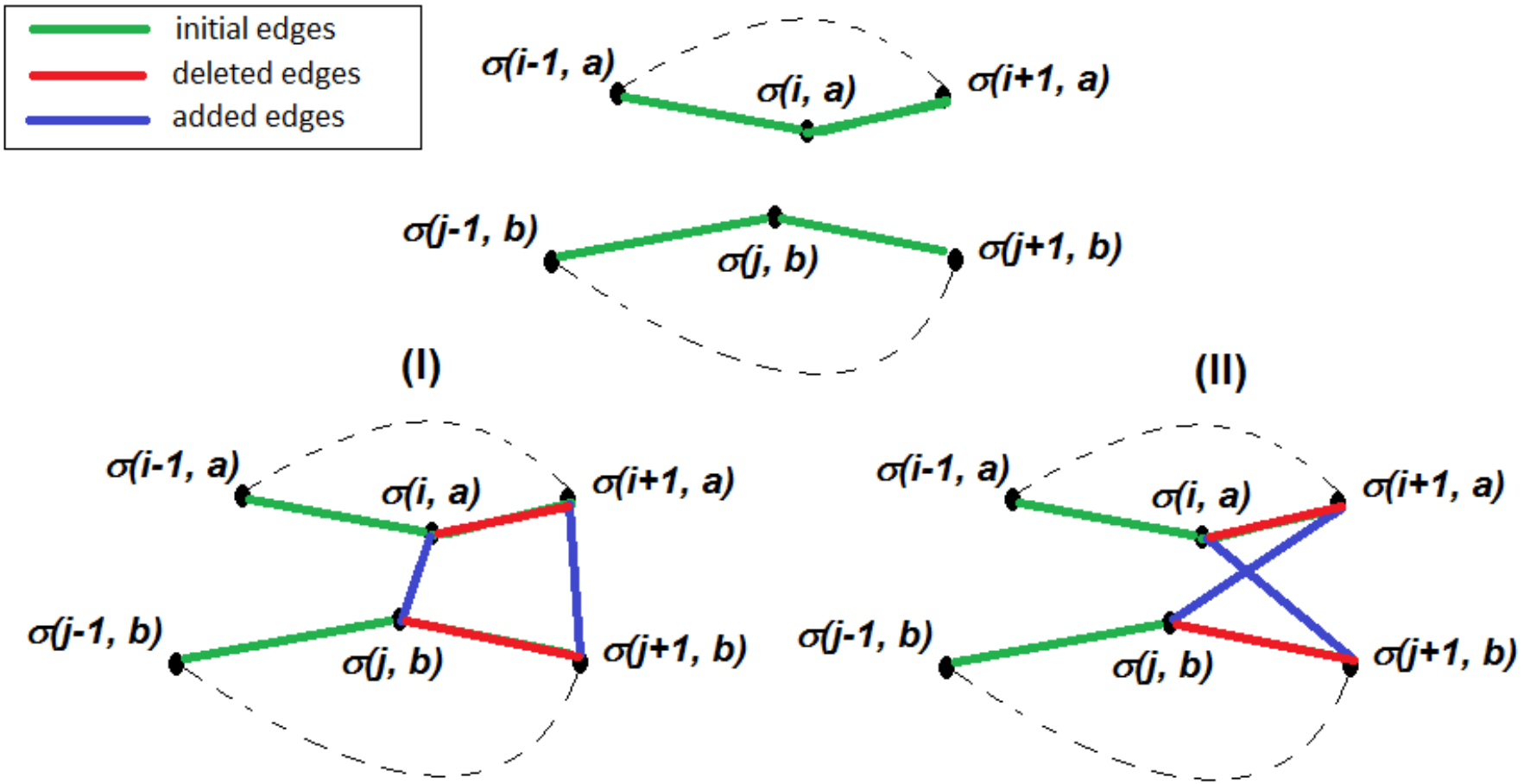
2 Select the tours $\tilde{a}, \tilde{b}$ such that $\delta(\tilde{a}, \tilde{b}) = \min_{a,b}(\delta(a,b))$

3 For each pair (i,j), $1 \le i \le N_{\tilde{a}}$, $1 \le j \le N_{\tilde{b}}$, calculate

$$\begin{aligned} S_1 = \; & d(\sigma(i,\tilde{a}), \sigma(j,\tilde{b})) + d(\sigma(i+1,\tilde{a}), \sigma(j+1,\tilde{b})) \\ & - d(\sigma(i,\tilde{a}), \sigma(i+1,\tilde{a})) - d(\sigma(j,\tilde{b}), \sigma(j+1,\tilde{b})) \\ S_2 = \; & d(\sigma(i,\tilde{a}), \sigma(j+1,\tilde{b})) + d(\sigma(i+1,\tilde{a}), \sigma(j,\tilde{b})) \\ & - d(\sigma(i,\tilde{a}), \sigma(i+1,\tilde{a})) - d(\sigma(j,\tilde{b}), \sigma(j+1,\tilde{b})) \end{aligned}$$

Let $S = \min\{S_1, S_2\}$ and $(\tilde{i}, \tilde{j})$ the pair for which the saving value S is minimum.

Graphical example



The two possibilities to merge two partial tours in nearest merge heuristic

Nearest merged heuristic (cont.)

- Merge the tours $\tilde{a}$ and $\tilde{b}$
  - Cut the edge between $\sigma(\tilde{i}, \tilde{a})$ and $\sigma(\tilde{i}+1, \tilde{a})$ and the edge between $\sigma(\tilde{j}, \tilde{b})$ and $\sigma(\tilde{j}+1, \tilde{b})$
  - If $S_1 < S_2$ add the edge between $\sigma(\tilde{i}, \tilde{a})$ and $\sigma(\tilde{j}, \tilde{b})$ and the edge between $\sigma(\tilde{i}+1, \tilde{a})$ and $\sigma(\tilde{j}+1, \tilde{b})$
  - If $S_2 < S_1$ add the edge between $\sigma(\tilde{i}, \tilde{a})$ and $\sigma(\tilde{j}+1, \tilde{b})$ and the edge between $\sigma(\tilde{i}+1, \tilde{a})$ and $\sigma(\tilde{j}, \tilde{b})$

Remark: the formulae S_1 and S_2 are valid only for tours having at least three nodes; more simpler formulae are available for tours with one or two nodes.

Ruin and Recreate

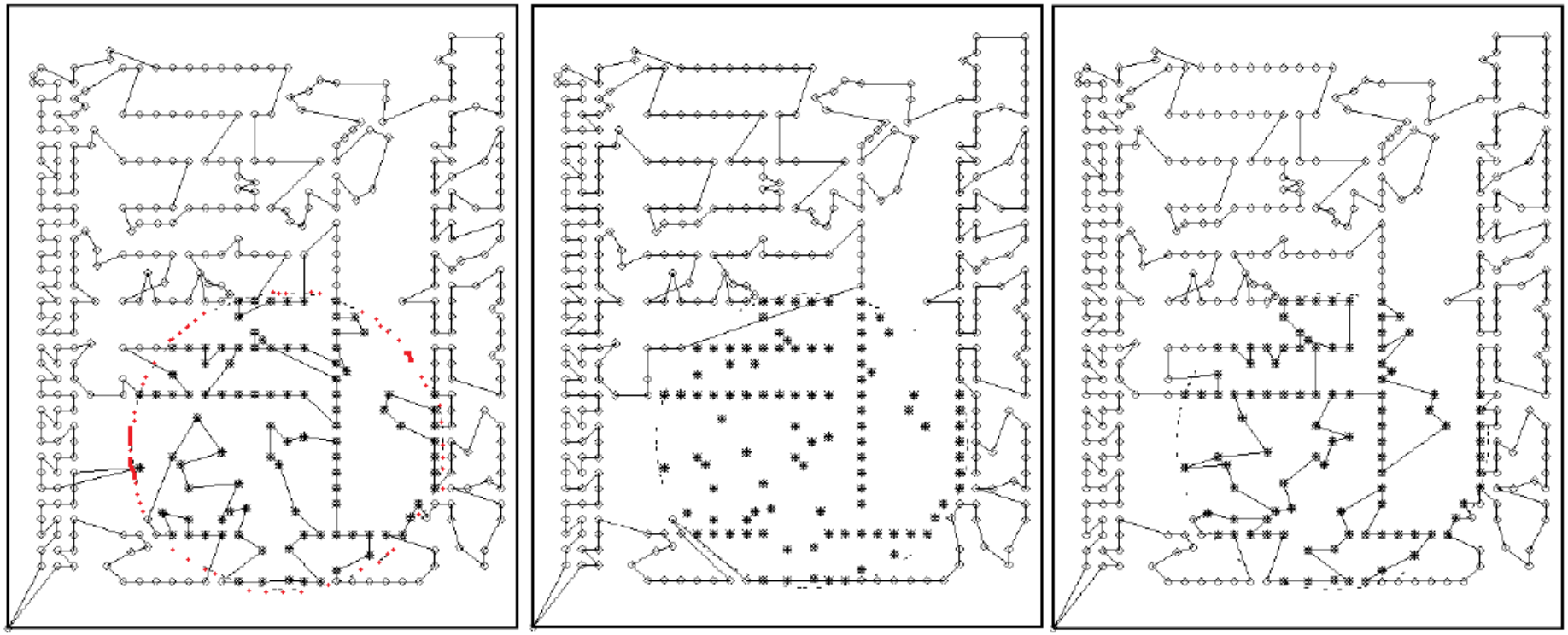
- After founding a solution, destroy a large part of the system (ruin phase)
  - a small part of the system (a skeleton) must remain after the ruin phase
  - is defined by α, the maximum percentage of items to be removed
- Reintroduce the eliminated items (system parts) in a specific way (recreate phase)
  - use construction heuristics
- Ruin phase for TSP: remove nodes or edges
  - Nodes case: removes nodes randomly or use some neighborhood relations
  - To eliminate a node $\sigma(i)$: cut the edges $(\sigma(i-1), \sigma(i))$ and $(\sigma(i), \sigma(i+1))$ and add edge $(\sigma(i-1), \sigma(i+1))$

Ruin & Recreate for TSP

- Three Ruin types
  - *Radial ruin*: select randomly a node c and eliminate from the tour the node c and its closest $k = \alpha n < n - 1$ neighbors;
  - *Sequential ruin*: eliminate $k = \alpha n$ successive nodes in the tour starting from a randomly selected node c; connect the predecessor and the successor of the sequence;
  - *Random ruin*: eliminate $k = \alpha n$ randomly chosen nodes;
- Recreate phase: use the best insertion or other construction heuristic
- The percent α: less than 0.5
- Applying R&R
  - *Random Walk* mode: start with an initial solution; apply m times R&R (m moves)
  - *Greedy* mode: start with an initial solution; repeat m times: accept a R&R move only if conduct to a better solution

Graphical example

Radial Ruin



- LEFT: Nodes to be removed are selected (whose falling on a disk around the central node)
- MIDDLE: Nodes selected are removed
- RIGHT: Removed nodes are re-inserted using best-insertion heuristic

Outline

Improvement heuristics

- Starting from an arbitrary solution (configuration), improve it step by step (using moves) until no further improvement is possible.
- Heuristic move - change (improve) the actual configuration
- Jump from actual configuration $\hat{\mathbf{d}}_k$ to $\hat{\mathbf{d}}_{k+1}$, where $\hat{\mathbf{d}}_{k+1} \in \mathcal{N}(\hat{\mathbf{d}}_k)$ (the neighborhood of configuration $\hat{\mathbf{d}}_k$)
- The neighborhood structure depend on the implemented move set
- Similarly, the set of moves is determined by the neighborhood structure
- The set of configuration + the neighborhood structure = the **search space** of the problem.
- The values of loss function + the search space form the *energy landscape* of the system
- The heuristic search - moves in a random way through the energy landscape

Basic approaches

- Gradient methods
  - the gradient rule - calculate the width of the step and the kind of change allowing the largest possible improvement
  - the steepest descend - maximize the energy difference
  - use of the whole configuration or one of its coordinates
- Greedy methods
  - choose a neighboring configuration at random and accept it if is better or at least as good as the current configuration
- Both approaches lead to a local minimum configuration
- Improvement
  - accept also for deterioration, i.e. configurations worst than current one
  - deform the energy landscape to remove barriers that the classical approaches cannot overcome in the original landscape

Markovian improvement heuristics

- **Markov Chain** - memoryless random process, where the next state depends only on the current state
  - Formally, $P(X_{n+1} = x_{n+1} | X_n = x_n, X_{n-1} = x_{n-1}, ..., X_1 = x_1) = P(X_{n+1} = x_{n+1} | X_n = x_n)$
- **Markov improvement heuristic** - the heuristic does not use any knowledge of formerly visited configurations when jumps from $\hat{\mathbf{d}}_k$ to $\hat{\mathbf{d}}_{k+1}$.
  - start with configuration $\mathbf{d}_0$
  - change $\mathbf{d}_0$ (based on a set of rules) to $\mathbf{d}_1$
  - if the move $\mathbf{d}_0 \rightarrow \mathbf{d}_1$ is accepted, then keep $\mathbf{d}_1$; if not, $\mathbf{d}_1 \equiv \mathbf{d}_0$
  - iterate
  - sequence of configurations $\mathbf{d}_0 \rightarrow \mathbf{d}_1 \rightarrow ...\mathbf{d}_{i-1} \rightarrow \mathbf{d}_i$
- Various move routines (include random effects)
- The **acceptance rule** - defined by the transition probability function $P(\mathbf{d}_k \rightarrow \mathbf{d}_{k+1})$

Acceptance functions

- "**all accept**" function: $P(\mathbf{d}_k \to \mathbf{d}_{k+1}) = 1$.
  - Trivial acceptance functions, used in random walk ($\mathbf{d}_{k+1}$ is chosen randomly)
  - For landscapes with few, isolated "good" configurations
- "**better or equally good**" function: if $\Delta L = L(\mathbf{d}_{k+1}) - L(\mathbf{d}_k)$, then

$$P(\mathbf{d}_k \to \mathbf{d}_{k+1}) = \begin{cases} 1 & \text{if } \Delta L \leq 0 \\ 0 & \text{if not} \end{cases}$$

  - Used by greedy algorithm
  - For landscapes with a single global optimum
- Modified "**b.o.e.g.**" function

$$P(\mathbf{d}_k \to \mathbf{d}_{k+1}) = \begin{cases} 1 & \text{if } \Delta L < 0 \\ 0.5 & \text{if } \Delta L = 0 \\ 0 & \text{if } \Delta L > 0 \end{cases}$$

Control parameter

- For landscapes with many local minima, high "mountains" and "deep" valleys, the "better or equally good" acceptance function conduct to a local optimum
- More elaborate transition probabilities - accepts deteriorations ($\Delta L > 0$), but not with probability 1
  - large deteriorations - small probability
  - small deteriorations - higher probability
  - acceptance probability - function of ΔL
- ΔL may have different orders of magnitude
- The **control parameter** governs the transitions
  - high values of CP - almost all deterioration are accepted (beginning of optimization process)
  - low values of CP - only small deteriorations are accepted (middle phase of optimization process)
  - zero value of CP - no deterioration is accepted (end of optimization process)
- Physical context: CP is the temperature

Heat bath approach

- Standard approach: from current configuration $\mathbf{d}_k$, select randomly a neighbor configuration $\mathbf{d}_{k+1}$; accept or reject $\mathbf{d}_{k+1}$ according to transition probability $P(\mathbf{d}_k \to \mathbf{d}_{k+1})$
- **Heat bath approach**
  - Consider all N neighboring configurations (or at least a specific subset) of $\mathcal{N}(\mathbf{d}_k)$
  - For each $\mathbf{d}_i \in \mathcal{N}(\mathbf{d}_k)$, calculate the transition probability $P(\mathbf{d}_k \to \mathbf{d}_i)$; Calculate $\mathbb{P} = \sum_i^N P(\mathbf{d}_k \to \mathbf{d}_i)$ and generate u uniformly on $(0, 1)$
  - Select configuration $\mathbf{d}_i$ which satisfies

$$\sum_{j=1}^{i-1} P(\mathbf{d} \to \mathbf{d}_j) < u\mathbb{P} < \sum_{j=1}^{i} P(\mathbf{d} \to \mathbf{d}_j)$$

- Advantages - faster dynamic (the system always jumps to a new configuration)
- Disadvantages
  - if system in a local optimum - non-zero probability to get out
  - if many ways to get to a worse configuration and few for a better one - great probability to select the bad choice

Outline

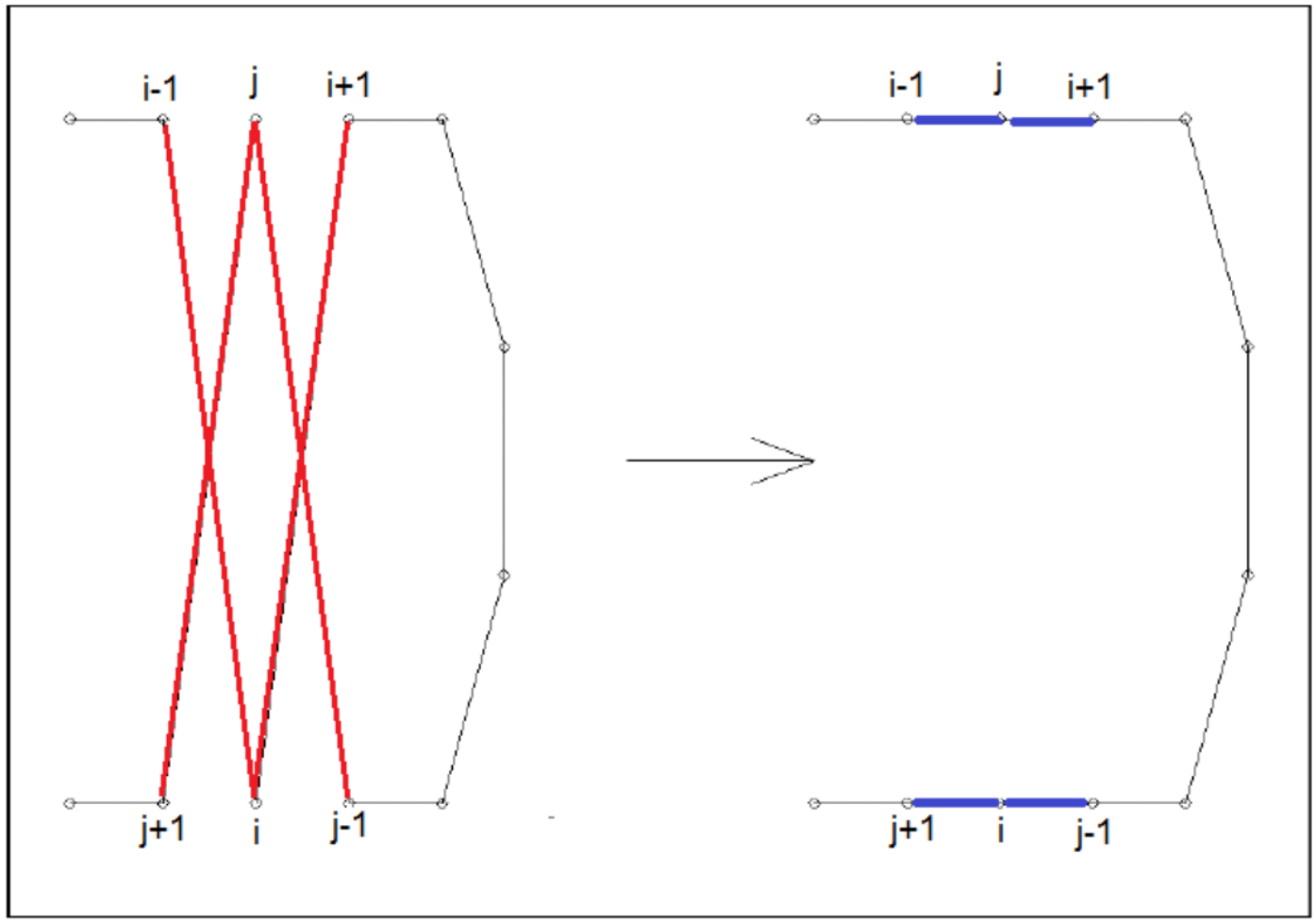
Local Search Approach

- How to define a move between various configurations?
  - Simple jump to any other randomly chosen configuration
    - appropriate for Random Walk
    - for greedy algorithm, the acceptance rate (the number of accepted moves divided by the overall number of move trials) may be very small for larger-size problems
  - Select a configuration rather similar with the actual one
- Two hypotheses:
  - if two configurations are rather similar, then they have roughly the same quality
  - the similar configurations are close, so the search must be done locally
- Small moves: change the actual configuration only slightly.
- Need to define similarity measures
  - Hamming distance between two configurations: the number of coordinates in which they differ (appropriate for discrete, combinatorial problems).

Improvement heuristics for TSP

- For a n size problem, a configuration $\sigma = (\sigma(1), .., \sigma(n))$ is a permutation of the set $\{1, 2, .., n\}$
- Small moves - change slightly the configuration
  - Swap (or transposition, or switch)
  - Translation (or insertion)
  - Inversion
- The loss function for TPS: $L(\sigma) = \sum_{i=1}^{n} d(\sigma(i), \sigma(i+1))$
- The energy difference: $L(\sigma_{new}) - L(\sigma_{curr})$
- The move $\sigma_{curr} \rightarrow \sigma_{new}$ is accepted according to "better or equally good" rule.

Swap move



$$\{..\sigma(i-1), \sigma(i), \sigma(i+1), ..., \sigma(j-1), \sigma(j), \sigma(j+1), ...\}$$

$$\Downarrow$$

$$\{..., \sigma(i-1), \sigma(j), \sigma(i+1), ..., \sigma(j-1), \sigma(i), \sigma(j+1), ...\}$$

Swap move algorithm

Step 1 Select randomly $i \neq j$, $1 \leq i, j \leq n$

Step 2 Define the successor and predecessor of i (resp j) in the cycle σ:

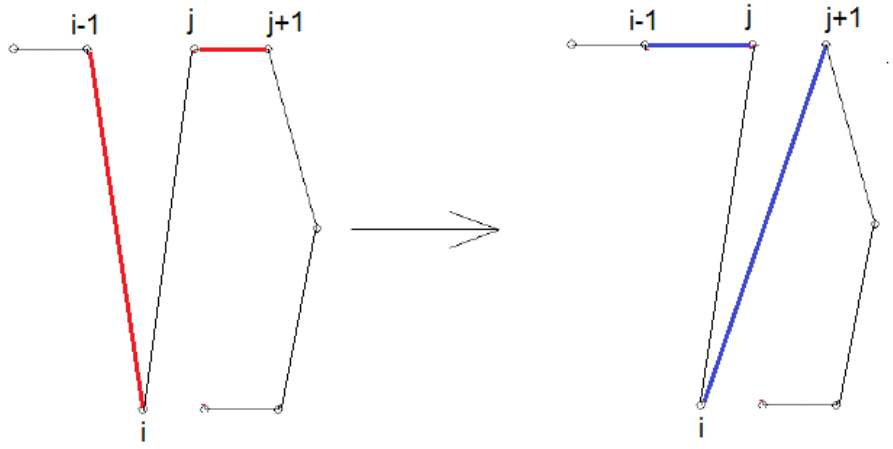
$$\textbf{(1)}\ i_+ = \begin{cases} i+1 & \text{if } 1 \leq i \leq n-1 \\ 1 & \text{if } i = n \end{cases} \qquad \textbf{(2)}\ i_- = \begin{cases} i-1 & \text{if } 2 \leq i \leq n \\ n & \text{if } i = 1 \end{cases}$$

Step 3 If $\tau = swap_{ij}(\sigma)$ is the new cycle, calculate the difference of length $\Delta(\mathcal{L}) = L(\tau) - L(\sigma) =$

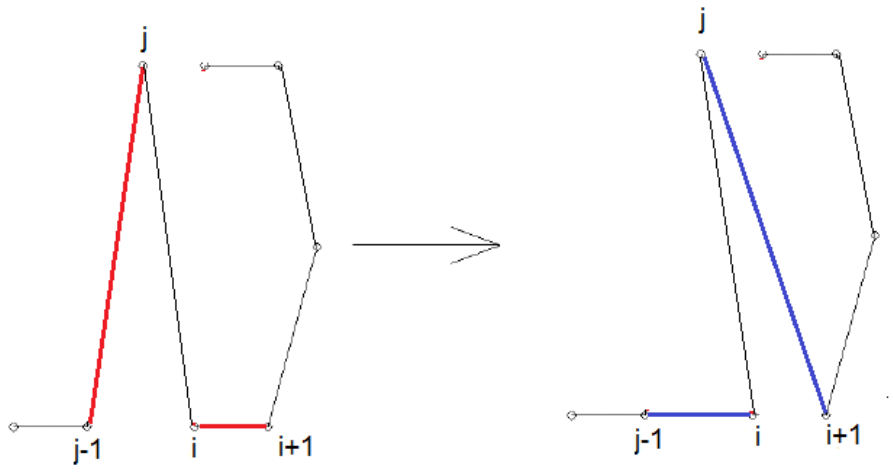
$$= \begin{cases} d(\sigma(i_-),\sigma(j)) + d(\sigma(i),\sigma(j_+)) \\ -d(\sigma(i_-),\sigma(i)) - d(\sigma(j),\sigma(j_+)) & \text{if } i_+ = j \\ \\ d(\sigma(j_-),\sigma(i)) + d(\sigma(j),\sigma(i_+)) \\ -d(\sigma(j_-),\sigma(j)) - d(\sigma(i),\sigma(i_+)) & \text{if } j_+ = i \\ \\ d(\sigma(i_-),\sigma(j)) + d(\sigma(j),\sigma(i_+)) \\ d(\sigma(j_-),\sigma(i)) + d(\sigma(i),\sigma(j_+)) \\ -d(\sigma(i_-),\sigma(i)) - d(\sigma(i),\sigma(i_+)) & \text{otherwise} \\ -d(\sigma(j_-),\sigma(j)) - d(\sigma(j),\sigma(j_+)) \end{cases}$$

Step 4 Accept/reject the new cycle τ according to the acceptance rule

Particular swap moves

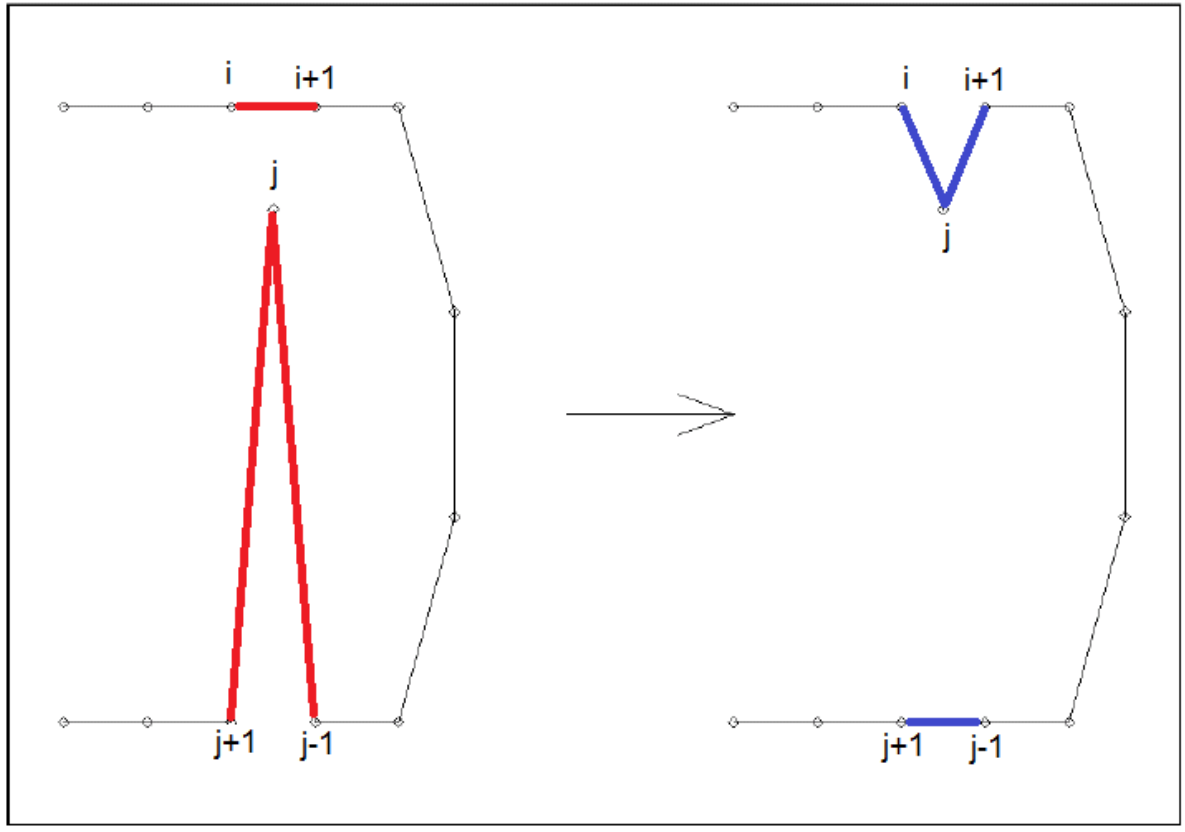


$$i_+ = j$$



$$j_+ = i$$

Translation move



$$\{..., \sigma(i), \sigma(i+1), ..., \sigma(j-1), \sigma(j), \sigma(j+1), ...\}$$

$$\Downarrow$$

$$\{..., \sigma(i), \sigma(j), \sigma(i+1), ..., \sigma(j-1), \sigma(j+1), ...\}$$

Translation move algorithm

Step 1 Select randomly a node i, $1 \leq i \leq n$, and determine i_+

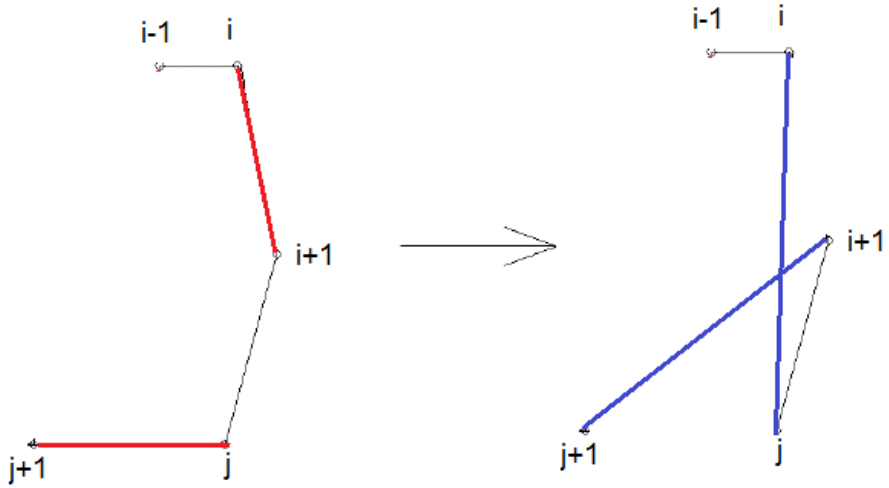
Step 2 Select randomly a node j, $1 \leq j \leq n$, such that $j \neq i$ and $j \neq i_+$. Determine j_- and j_+ (see relations **(1)** and **(2)**)

Step 3 If $\tau = trans_{ij}(\sigma)$ is the new cycle, calculate the difference of length $\Delta(\mathcal{L}) = L(\tau) - L(\sigma) =$

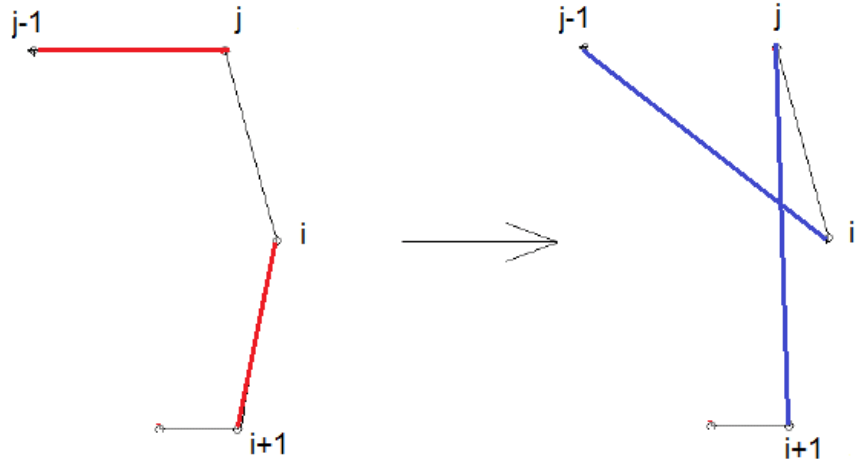
$$= \begin{cases} d(\sigma(i),\sigma(j)) + d(\sigma(i_+),\sigma(j_+)) \\ -d(\sigma(i),\sigma(i_+)) - d(\sigma(j),\sigma(j_+)) & \text{if } i_+ = j_- \\ \\ d(\sigma(j_-),\sigma(i)) + d(\sigma(j),\sigma(i_+)) \\ -d(\sigma(j_-),\sigma(j)) - d(\sigma(i),\sigma(i_+)) & \text{if } j_+ = i \\ \\ d(\sigma(i),\sigma(j)) + d(\sigma(j),\sigma(i_+)) \\ d(\sigma(j_-),\sigma(j_+)) - d(\sigma(i),\sigma(i_+)) & \text{otherwise} \\ -d(\sigma(j_-),\sigma(j)) - d(\sigma(j),\sigma(j_+)) \end{cases}$$

Step 4 Accept/reject the new cycle τ according to the acceptance rule

Particular translation moves

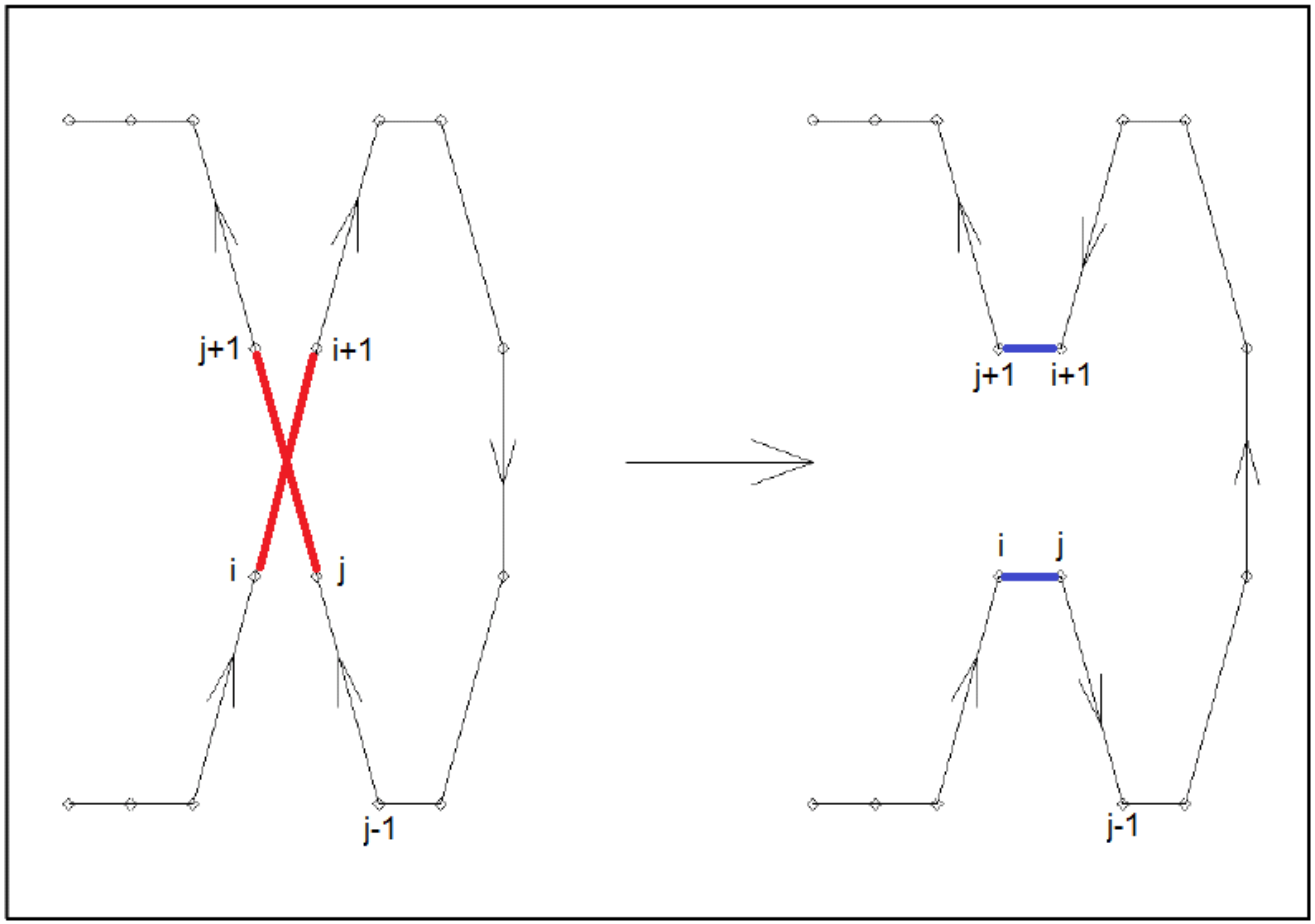


$$i_+ = j_-$$



$$j_+ = i$$

Inversion move



$$\{..., \sigma(i), \sigma(i+1), ..., \sigma(j-1), \sigma(j), \sigma(j+1), ...\}$$

$$\Downarrow$$

$$\{..., , \sigma(i), \sigma(j), \sigma(j-1), \sigma(j-2), ..., \sigma(i+1), \sigma(j+1), ...\}$$

Inversion move algorithm

Step 1 Select randomly a node i, $1 \leq i \leq n$, from the cycle σ

Step 2 Select randomly a node j, $1 \leq j \leq n$ and $j \neq i$

Step 3 Consider the orientation of σ such that $i < j$ and determine i_+ and j_+

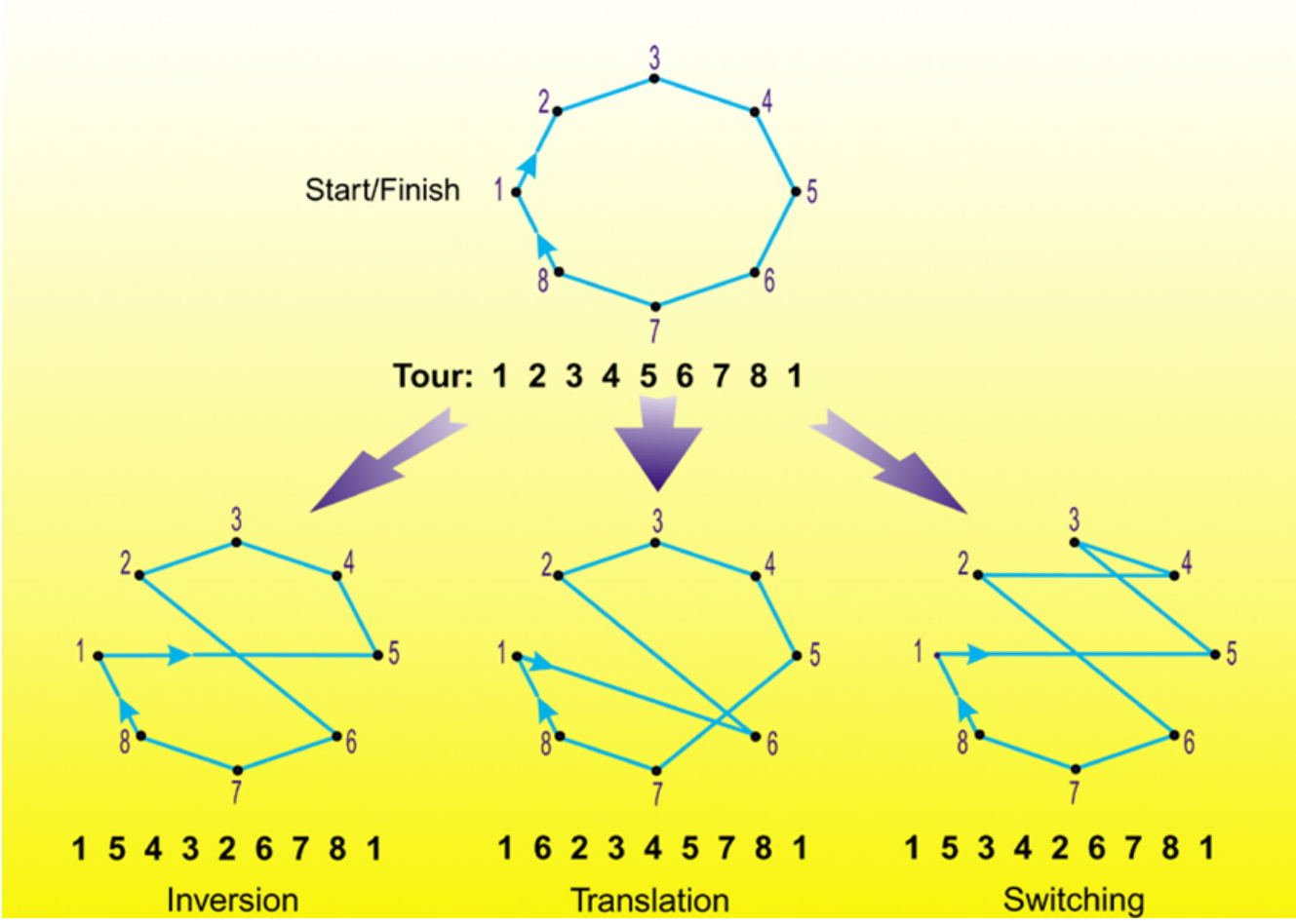
Step 4 If $i_+ = j$ or $j_+ = i$ then return to Step 1

Step 5 If $\tau = inv_{ij}(\sigma)$ is the new cycle, calculate the difference of length $\Delta(\mathcal{L}) = L(\tau) - L(\sigma) =$

$$d(\sigma(i), \sigma(j)) + d(\sigma(i_+), \sigma(j_+)) - d(\sigma(i), \sigma(i_+)) - d(\sigma(j), \sigma(j_+))$$

Step 4 Accept/reject the new cycle τ according to the acceptance rule

Example: 8-City Tour



Inversion reverses order 2-3-4-5;
Translation removes node 6 and insert it after node 1;

Swap interchanges order of 2 and 5.

Moves comparison

- Swap, Translation and Inversion - the smallest possible moves (only two nodes are selected)
- Nodes moved
  - Translation - one node
  - Swap - two nodes
  - Inversion - $|j - i|$ nodes
- Edges cut
  - Inversion - two edges
  - Translation - three edges
  - Swap - four edges
- Neighbors configurations
  - Swap - C_2^n configurations
  - Inversion - $n * (n - 3)/2$ configurations
  - Translation - $O(n^2)$ configurations
- An improvement heuristic may use all three moves, according to a given probability

Greedy algorithm for TSP

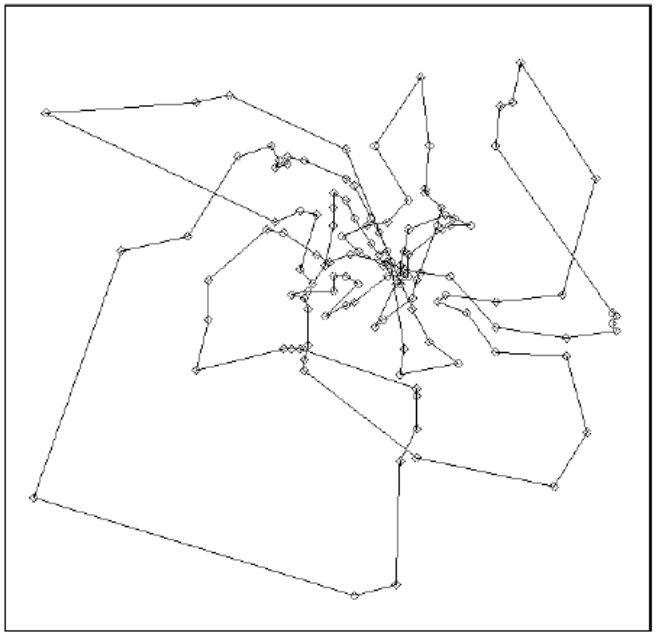
Init Generate an initial configuration $\hat{\mathbf{d}}_0$ (a randomly selected permutation σ or the result of a construction heuristic)

While stopping criterion not satisfied

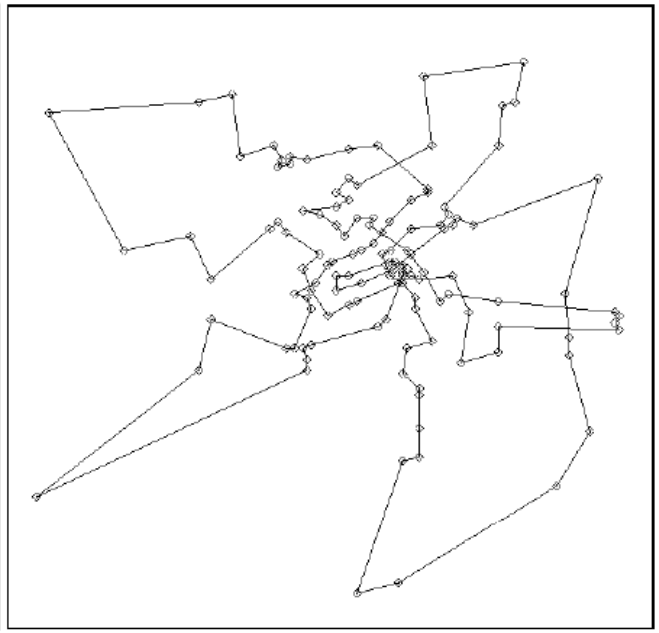
- **Step 1** Let be $\mathcal{N}(\hat{\mathbf{d}}_k)$ the set of all configurations $\{\hat{\mathbf{d}} | \hat{\mathbf{d}} = move(\hat{\mathbf{d}}_k)\}$, where *move* is a fixed TSP small move. Select randomly $\hat{\mathbf{d}}_{new}$ from $\mathcal{N}(\hat{\mathbf{d}}_k)$
- **Step 2** Calculate $\Delta = L(\hat{\mathbf{d}}_{new}) - L(\hat{\mathbf{d}}_k)$
- **Step 3** If $\Delta \leq 0$ then $\hat{\mathbf{d}}_{k+1} = \hat{\mathbf{d}}_{new}$, else $\hat{\mathbf{d}}_{k+1} = \hat{\mathbf{d}}_k$

- Greedy algorithm: the system freezes in a local configuration $\hat{\theta}$ if any neighbor of $\hat{\theta}$ has a higher length.
- If $\mathcal{N}(\hat{\mathbf{d}})$ contains m configurations, how many trials (Step 1) must be done before to select a particular configuration $\hat{\mathbf{d}}_{new}$?
  - In average, the number of trials is m
  - Conclusion: if, after m trials, no better configuration is found, the system may be considered as freezed

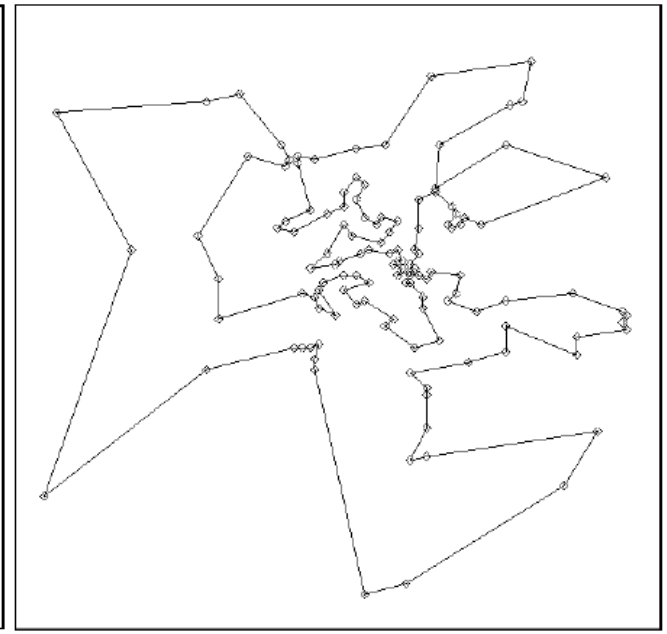
Graphical example



Swap moves



Translation moves



Inversion moves

Larger Mover for TSP

- Extend the Inversion move
  - select randomly three nodes $i < j < k$
  - cut the tour in three parts

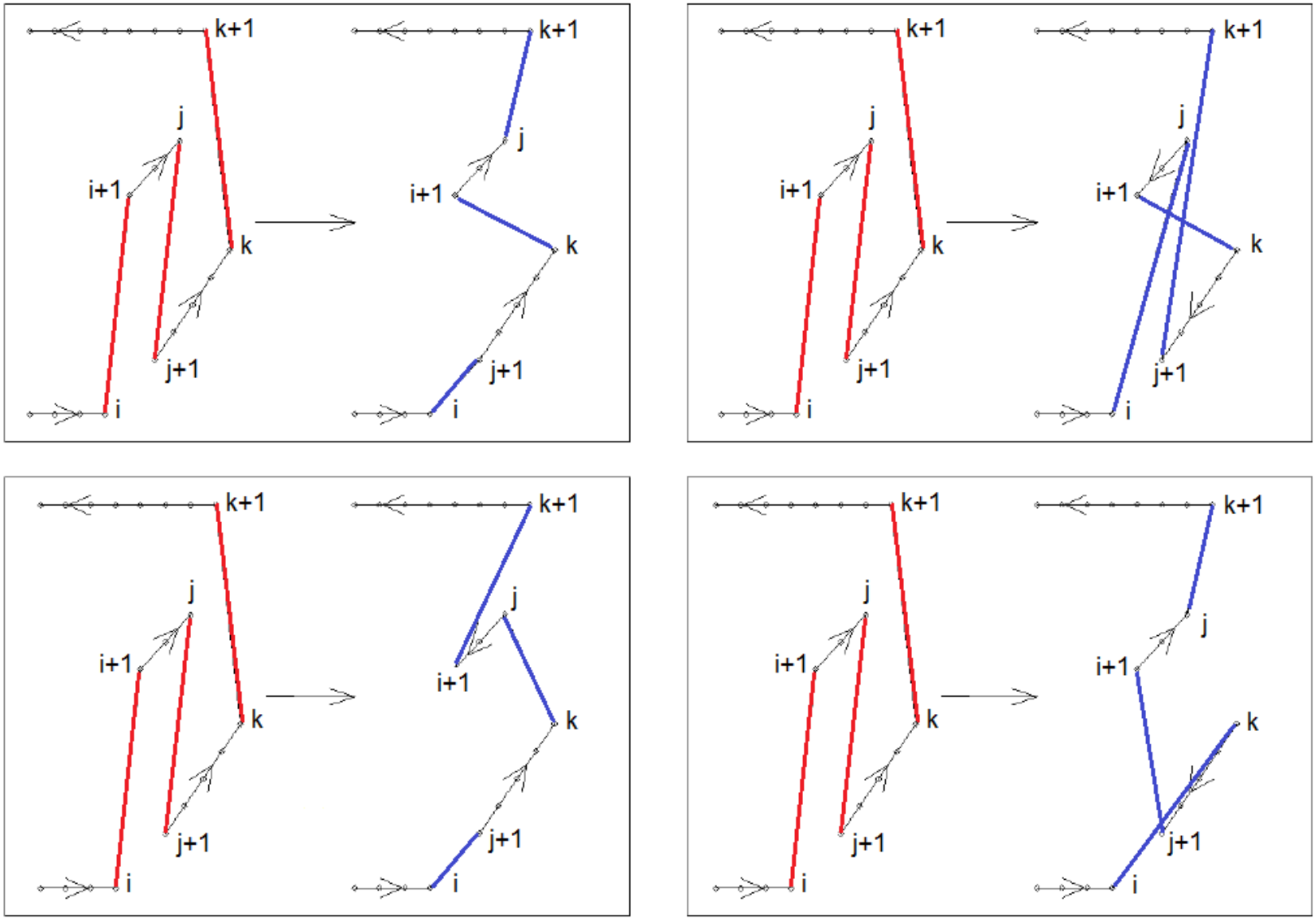
$$\ldots \sigma(i)|\sigma(i_+)\dashrightarrow\sigma(j)|\sigma(j_+)\dashrightarrow\sigma(k)|\sigma(k_+)\ldots$$

  - if $j \neq i_+, k \neq j_+$ and $i \neq k_+$ then there are eight possibilities to reconnect the parts
- Four of these possibilities may be simulated by two successive (small) inversion moves. The remaining four moves are:

L3O1 $\ldots \sigma(i)|\sigma(j_+)\dashrightarrow\sigma(k)|\sigma(i_+)\dashrightarrow\sigma(j)|\sigma(k_+)\ldots$
L3O2 $\ldots \sigma(i)|\sigma(j)\dashleftarrow\sigma(i_+)|\sigma(k)\dashleftarrow\sigma(j_+)|\sigma(k_+)\ldots$
L3O3 $\ldots \sigma(i)|\sigma(j_+)\dashrightarrow\sigma(k)|\sigma(j)\dashleftarrow\sigma(i_+)|\sigma(k_+)\ldots$
L3O4 $\ldots \sigma(i)|\sigma(k)\dashleftarrow\sigma(j_+)|\sigma(i_+)\dashrightarrow\sigma(j)|\sigma(k_+)\ldots$

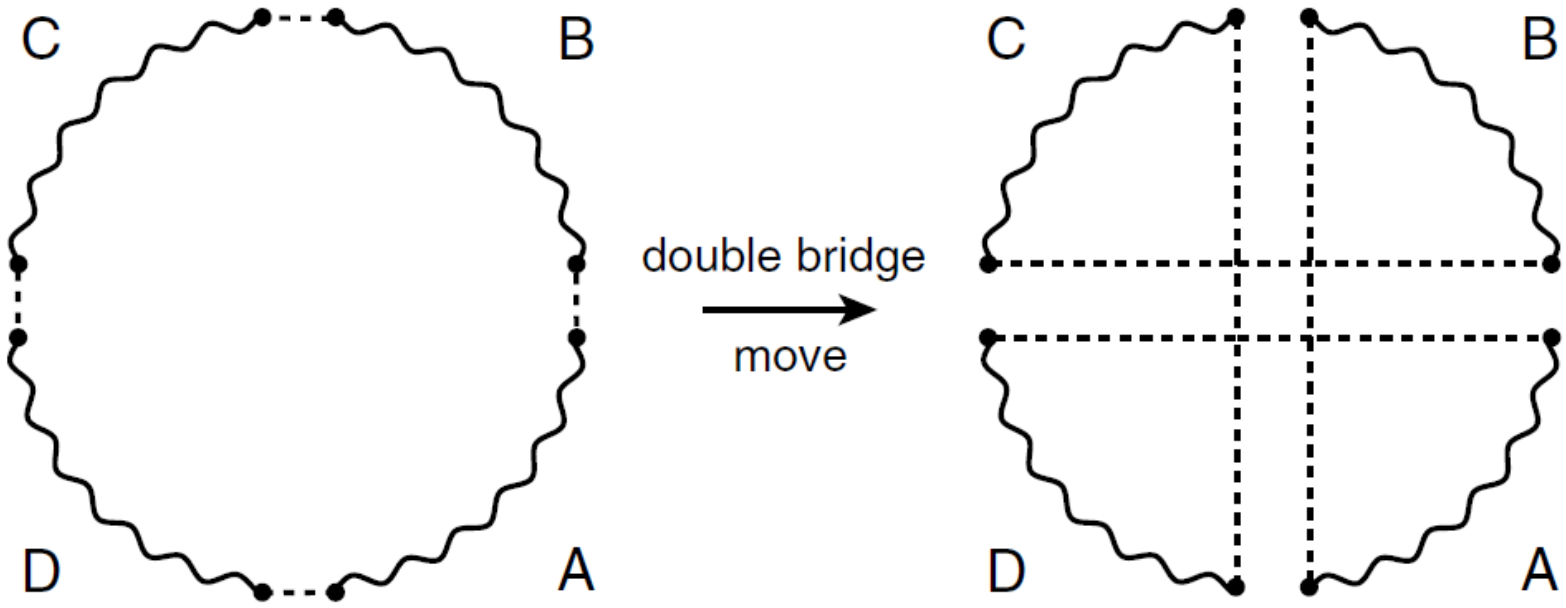
- Denote these moves as Inversion(3) moves (lead to three new edges)

Inversion(3) moves



Four possibilities for Inversion(3) move

Double bridge move



An Inversion(4) move