Operations Research (Master's Degree Course)

10. Approximation and Heuristic Algorithms

Silvano Martello
DEI "Guglielmo Marconi", Università di Bologna, Italy

Introduction

- For few optimization problems we know **polynomial time** exact algorithms;
- for most optimization problems we only know exact algorithms which, in the worst case, can take **exponential time**.
- Industrial problems usually belong to the latter category, and an exponential time is unacceptable for real world applications.
- **Approximation** (or **heuristic**) **algorithm** = method who tries to determine, within an **acceptable running time**, a "**good**" feasible solution, without being able to guarantee its optimality, and sometimes without being able to find a feasible solution.
- For such algorithms we need information on the performance (running time, average error, maximum error, ...). Main methodologies:
  - **experimental analysis**;
  - **probabilistic analysis**;
  - **worst-case analysis**;
- Terminology:
  - **Approximation algorithm** if we also have theoretical results (e.g., worst-case error);
  - **Heuristic algorithm** if we are mostly interested in the practical performance.
  - **Metaheuristic algorithm:** the most recent evolution of heuristics.

Experimental analysis

- implement the algorithm;
- implement an exact algorithm (or a method to compute a bound);
- make experiments on a computer
  - by generating many random instances of various sizes, using different probability distributions, and/or
  - using known instances from the literature (real world instances, artificial instances)
- **Pros:**
  - easy to implement;
  - provides useful information;
  - frequently also used to evaluate exact algorithms.
- **Cons:**
  - lack of theoretical rigor;
  - unsure extendability to real world situations;

Probabilistic analysis

- Advanced theoretical tools from probability theory.
- **Average instance** of a problem = probability distribution over all possible instances.
- Running time and solution value = **random variables**.
- Study the limit of time and value as the instance dimension tends to infinity.
- **Pros:**
  - theoretical rigor.
- **Cons:**
  - tough, only possible for very simple algorithms;
  - unsure extendability to real world situations.

Worst-case analysis

- Instance I of a maximization problem P;
- $OPT(I) =$ optimal solution value;
- $A(I)$ value of the solution provided by approximation algorithm A.
- Find the maximum relative deviation between $OPT(I)$ and $A(I)$ over all instances of P:

$$R_A = \inf_{I \in P} \left\{ \frac{A(I)}{OPT(I)} \right\} .$$

- $R_A \leq 1$.
- For minimization problems, $R_A = \sup_{I \in P}\{A(I)/OPT(I)\} (\geq 1.)$
- $R_A =$ **worst-case performance ratio (WCPR)**
- **Pros:**
  - theoretical rigor.
- **Cons:**
  - only possible for simple algorithms;
  - pessimistic with respect to real world situations.

Two problems we will use as examples

- **0–1 Knapsack Problem (KP01)** :

  given n elements, each having a profit p_j and a weight w_j $(j = 1, \ldots, n)$,
  and a container of capacity c $(c \geq w_j \ \forall \ j)$,
  select a subset of elements having maximum total profit
  and total weight not greater than c:

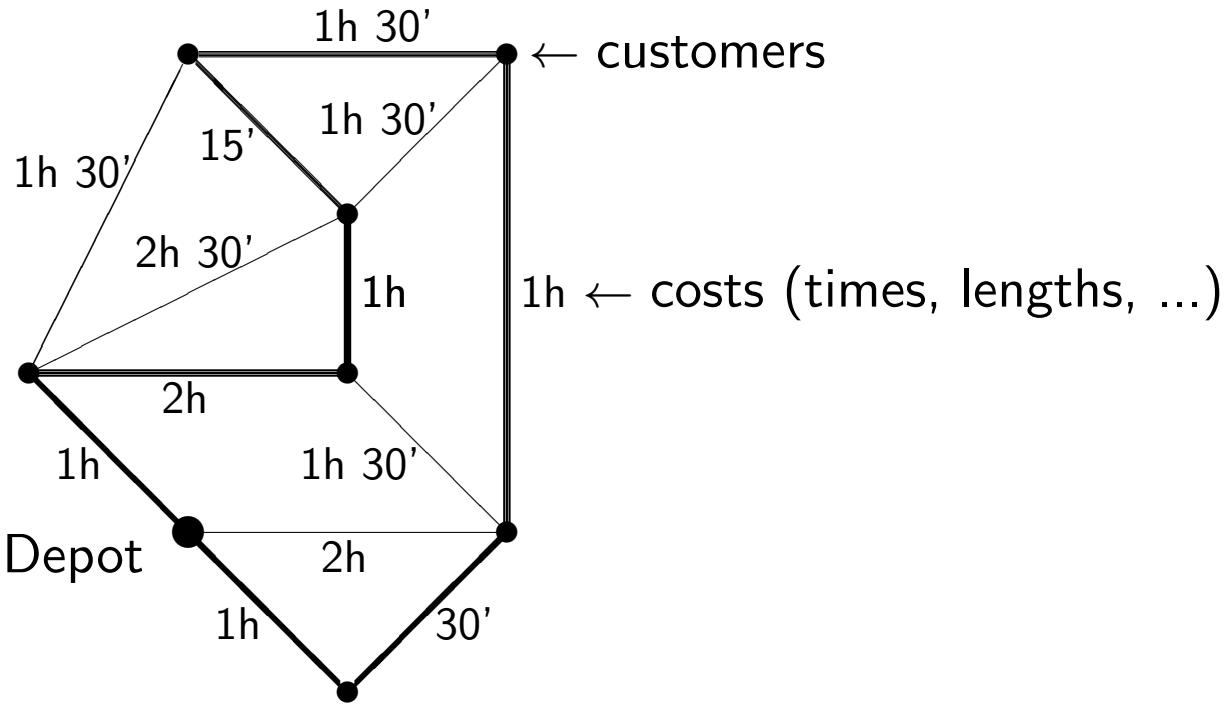
$$x_j = \begin{cases} 1 & \text{if element } j \text{ is selected} \\ 0 & \text{otherwise} \end{cases} \qquad (j = 1, \ldots, n)$$

$$\begin{aligned} \max \ & \sum_{j=1}^{n} p_j x_j \\ & \sum_{j=1}^{n} w_j x_j \quad \leq \quad c \\ & x_j \quad \in \quad \{0, 1\} \quad j = 1, \ldots, n \end{aligned}$$

Two problems we will use as examples (cont'd)

- **Traveling Salesman Problem (TSP)** :

  Given a graph $G = (V, E)$ having a cost c_{ij} associated with each edge (i, j), find the minimum cost circuit that passes through each vertex exactly once.



If the graph is **non oriented** (edge $(i, j) \equiv$ edge (j, i)), $\Rightarrow$ **Symmetric TSP, STSP**

If the graph is **oriented** (arc $(i, j) \not\equiv$ arc (j, i)), $\Rightarrow$ **Asymmetric TSP, ATSP**

Approximation algorithms, KP01

- Greedy algorithm :
  z^g = current profit;
  $\overline{c}$ = current residual capacity
- **procedure Greedy:**
  begin
  sort the items by non-increasing p_j/w_j values;
  $\overline{c} = c$; $z^g = 0$;
  for $j := 1$ **to** n **do**
  if $w_j \leq \overline{c}$ **then** $x_j := 1$, $\overline{c} := \overline{c} - w_j$, $z^g := z^g + p_j$
  else $x_j := 0$
  end.
- Time complexity $O(n \log n)$ (for sorting).
- Example:

$(p_j) =$	100	60	70	45	45	4	4	4	15	
$(w_j) =$	10	10	12	8	8	1	1	1	4	$c = 26$

Greedy:

$(x_j) =$	1	1	0	0	0	1	1	1	0	

$z = 172$ $\overline{c} = 3$

Approximation algorithms, KP01 (cont'd)

- **Experimental analysis:** very effective for large-size instances.
- **Probabilistic analysis:** it can be proved that, if capacities, profits and weights are uniformly random real numbers, then $\lim_{n\to\infty} \mathbb{P}(z^g \text{ is optimal}) = 1$.
- **Worst-case analysis:**
  - Greedy can be arbitrarily bad, i.e., its worst-case ratio can be arbitrarily close to 0:
  - $$\begin{array}{rcl} (p_j) & = & (\ 2, \quad M\ ), \\ (w_j) & = & (\ 1, \quad M\ ), \\ c & = & M \quad > 2; \end{array}$$

    $$OPT(I) = M, \ \ G(I)(\text{Greedy solution}) = 2 \ \Rightarrow \ \frac{G(I)}{OPT(I)} \xrightarrow[M \to \infty]{} 0.$$
  - Improved algorithm: $\overline{G}(I) = \max\left(G(I), \max_j\{p_j\}\right)$.
  - **To prove that R_A is the WCPR of an algorithm A for problem P,**
    1. prove that, for all instances I of P, we have $A(I)/OPT(I) \geq R_A$.
    2. find an instance $\overline{I}$ for which $A(\overline{I})/OPT(\overline{I}) = R_A$, or
       a series of instances $\overline{I}$ for which $A(\overline{I})/OPT(\overline{I}) \to R_A$).

Approximation algorithms, WCPR of Greedy

- **Theorem** *The WCPR of algorithm* $\overline{G}$ *is* $R_{\overline{G}} = \frac{1}{2}$.

  Proof

  1. Let s be the critical item of an instance I. We have

  $$OPT(I) \leq \sum_{j=1}^{s-1} p_j + p_s \,.$$

  From $\overline{G}$ we have

  $$\overline{G}(I) \geq \sum_{j=1}^{s-1} p_j \quad \text{and} \quad \overline{G}(I) \geq p_s \,,$$

  from which $OPT(I) \leq 2\overline{G}(I)\ \forall I$.

  2. Series of instances I with $n = 3$, $(p_j) = (2, M, M)$, $(w_j) = (1, M, M)$, $c = 2M > 2$:

  $OPT(I) = 2\,M$, $\overline{G}(I) = \max(M + 2, M) = M + 2$,

  $\Rightarrow \overline{G}(I)/OPT(I)$ is arbitrarily close to $\frac{1}{2}$ for M sufficiently large. ($\frac{1}{2}$ is *tight*.) □

- **Next question:** Is it possible to prefix the worst-case behavior and have an algorithm that guarantees it?

  Answer: Yes: Polynomial-Time Approximation Scheme, **PTAS:**

Approximation algorithms, PTAS for KP01

- **PTAS** = family of approximation algorithms which produces, in polynomial time, a prefixed worst-case behavior.

  procedure S(k) (**comment:** PTAS for KP01; k is a prefixed positive integer):
  begin
  $z := 0$;
  for each $T \subset \{1, \ldots, n\}$ such that $|T| \leq k$ **and** $\sum_{j \in T} w_j \leq c$ **do**
  impose the elements of T to the solution;
  execute **Greedy** on the elements of $\{1, \ldots, n\} \setminus T$ with capacity $c - \sum_{j \in T} w_j$;
  if $z^g + \sum_{j \in T} p_j > z$ **then** $z := z^g + \sum_{j \in T} p_j$ (and store the solution)
  endfor
  end.

- Time complexity $O(n^{k+1})$ ($\Leftarrow$ $|T|$ is $O(n^k)$, and sorting is performed only once).
- It can be proved that the worst-case behavior of S(k) is $R_{S(k)} = \frac{k}{k+1}$.
- The running time is polynomial for any prefixed k, but it grows exponentially with k, i.e., with the inverse of the relative error $\varepsilon = 1 - \frac{S(I)}{OPT(I)} \leq 1 - \frac{k}{k+1} = \frac{1}{k+1}$
- $\exists$ **Fully polynomial-time approximation schemes (FPTAS)** for which the running time grows polynomially with the inverse of the relative error (based on dynamic programming).

Approximation algorithms, TSP

- Can any NP-hard problem be approximated in some way? **Bad news:**
- **Theorem** *If there exists a polynomial-time algorithm A for the TSP, and a constant R $(1 \leq R < \infty)$ such that, for any instance I*

$$A(I) \leq R \cdot OPT(I)$$

then $P = NP$.
- **Proof** We show that A would solve in polynomial time the problem of deciding if a graph $G = (V, E)$ has a Hamiltonian cycle.

  Define a weighted graph $\overline{G} = (V, \overline{E})$ with weights

$$c_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \text{ (the edges of the given graph } G); \\ R \cdot n & \text{otherwise (the edges that do not exist in } G) \end{cases} \quad (i, j = 1, \ldots, n),$$

  and execute algorithm A.

  If G has an HC, the cost of the optimal TSP is n.

  $\Rightarrow A$ must find a solution of value $\leq R \cdot n$.

  $\Rightarrow A$ cannot use any edge having cost $R \cdot n$.

  The HC problem would then be solved in polynomial time by A: G has an HC if and only if the solution produced by A has value n (and such solution would be the required HC). $\square$

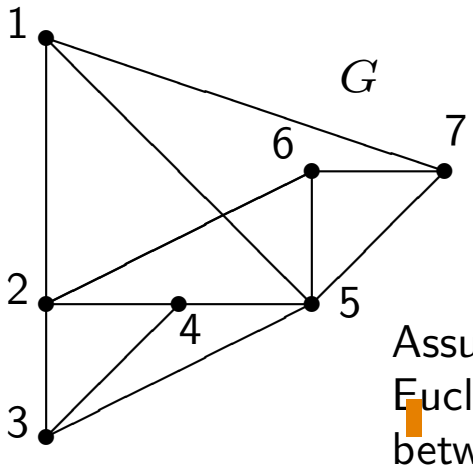
Approximation algorithms, better news for a special case of the STSP

- Let us assume that the **triangularity condition** holds, i.e., that

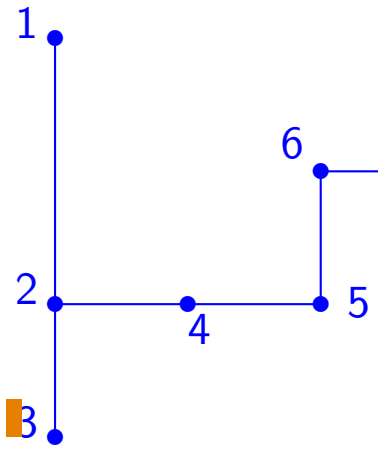
$$c_{ij} + c_{jk} \geq c_{ik} \quad \forall\ i, j, k$$

- **If it does not hold**, replace each c_{ij} with the cost of the shortest path from i to j.

  Note: in this way, if the graph is connected, we can always assume that $(i, j) \in E\ \forall\ i, j$.
- **Spanning Tree** of a graph G (n vertices) = connected graph containing $n - 1$ edges of G.
- **Shortest Spanning Tree** of a graph G = Spanning Tree having minimum total cost:



Assume wlog c_{ij} = Euclidean distance between i and j

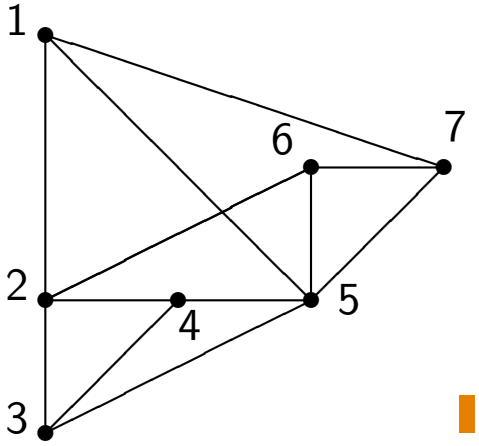


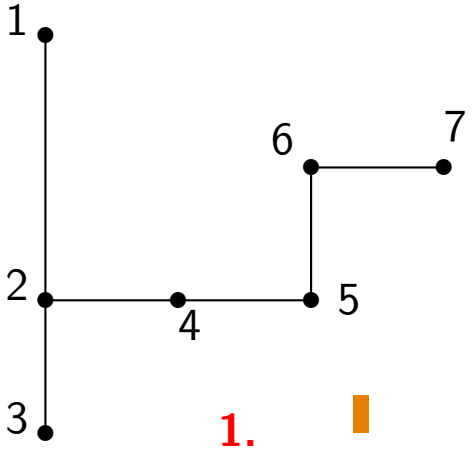
shortest spanning tree of G (can be found in $O(n^2)$ time)

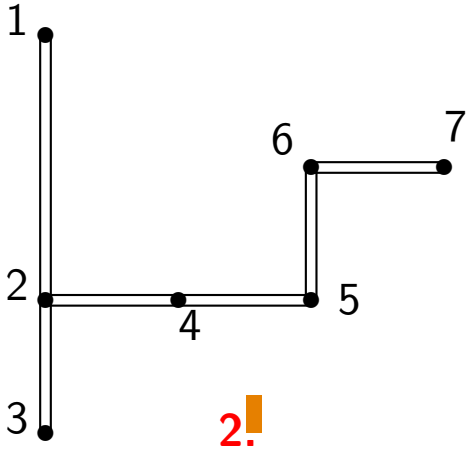
- **Observation:** the cost of the shortest spanning tree is less that the cost of the TSP ($\Leftarrow$ by removing an edge from the circuit one has a ST).

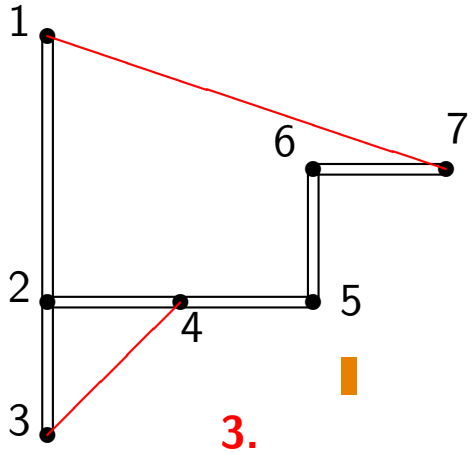
Approximation algorithm for the STSP

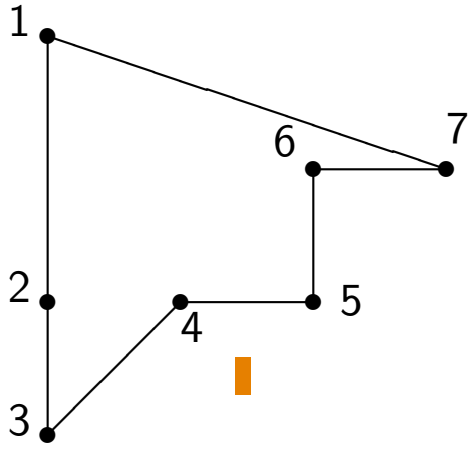
- **procedure** TREE:
  begin
  1. find the *shortest spanning tree* T of the graph;
  2. create a *multiple graph* G' using two copies of each edge of T;
  3. build a circuit in G' using "shortcuts" given from the triangularity condition
  end.





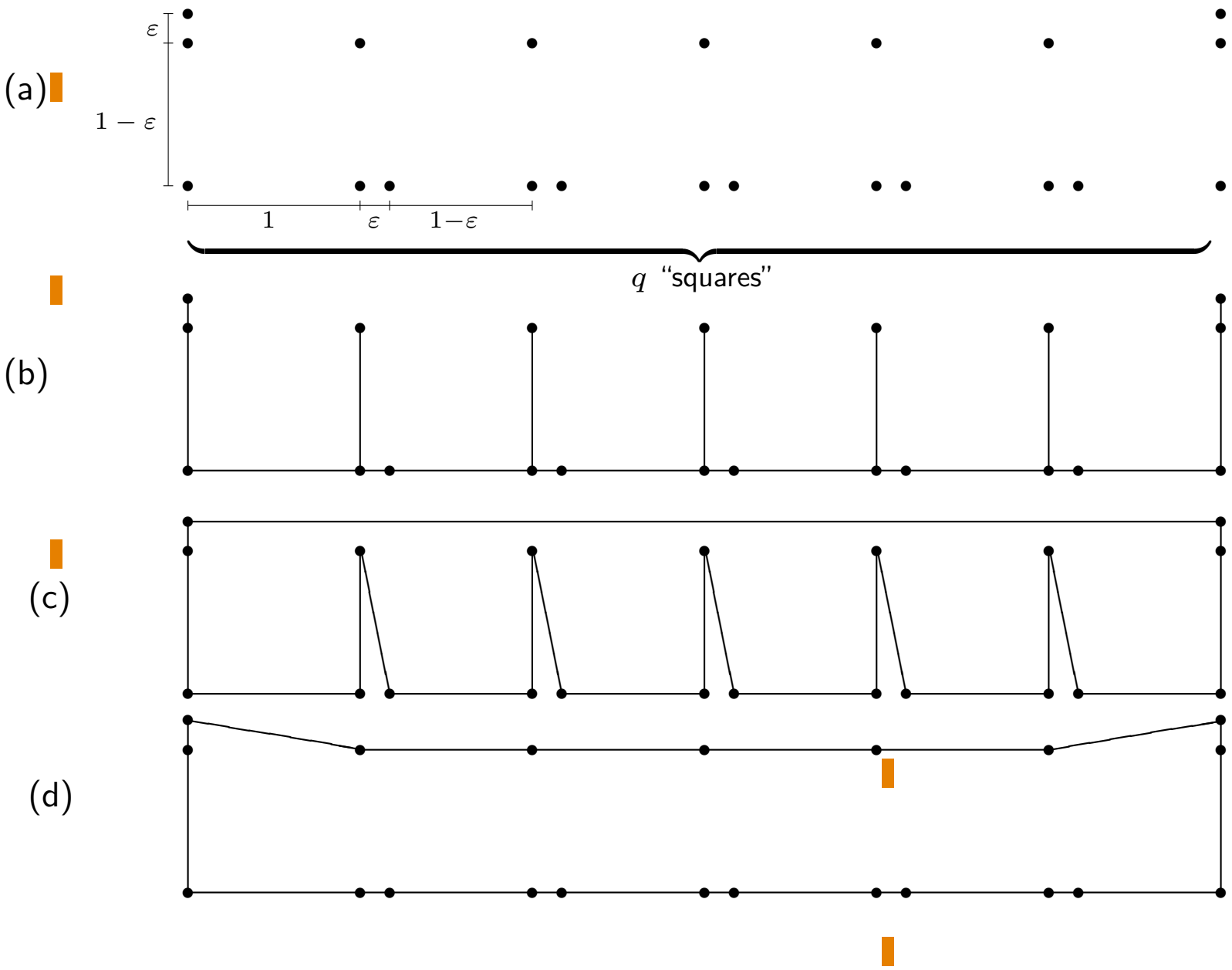






- **cost $<2\cdot$(optimal TSP cost);**

The worst-case bound is tight



Euclidean graph (all edges)

SST: $q + (q+1)(1-\varepsilon) + 2\varepsilon$

Approximate solution $\sim 2q + 2q(1-\varepsilon)$

Optimal solution $\sim 2q + 2$

Approximability status of the TSP

1. The **general TSP** (symmetric or asymmetric) **cannot be approximated** within a constant factor.

2. The **symmetric TSP with triangularity condition (Metric TSP)**
   - can be approximated with worst-case performance ratio $= 2$ by the SST algorithm;
   - the SST algorithm can be improved with a more careful construction of the approximate tour from the SST;
   - the time complexity grows to $O(n^3)$, but
     the resulting algorithm has worst-case performance ratio $= \frac{3}{2}$ (**Christofides, 1976**).
   - **After over 40 years, no better algorithm is known!**
   - **October 2020:** Karlin, Klein, and Gharan announce a new heuristic for which:
     For some absolute constant $\varepsilon > 10^{-36}$, the algorithm outputs a tour with expected cost at most $\frac{3}{2} - \varepsilon$ times the cost of the optimum solution.
   - **Proof** About 80 pages.

3. For the **asymmetric TSP with triangularity condition**
   - **no algorithm** with guaranteed worst-case performance ratio is known.

Heuristic algorithms

- Classification of the classical heuristic algorithms:
  - **Greedy algorithm**:
    - find a solution through a simple scan of the input data

      (very fast, limited accuracy).
  - **Local search algorithm**:
    - start from an initial solution (usually greedy);
    - recursively generate a series of solutions obtained from the current solution through small improvements;
    - terminate when no further improvement is possible.

Heuristic algorithms, local search for KP01

- **Procedure Local Search:**
  - iteratively exchange an item that is in the current solution with one of the items that follow it and is not in the current solution provided the exchange is feasible and improves the solution.
- **procedure** Local Search KP:
  begin
  call **Greedy** (z^g = solution value, $\overline{c}$ = residual capacity);
  $z := z^g$;
  for $i := 1$ **to** n **do**
  if $x_i = 1$ **then**
  for $j := i + 1$ **to** n **do**
  if $x_j = 0$ **and** $\overline{c} + w_i \geq w_j$ **and** $p_j > p_i$ **then**
  $x_i := 0$, $x_j := 1$;
  $z := z - p_i + p_j$;
  $\overline{c} := \overline{c} + w_i - w_j$
  endif
  end.
- Time complexity $O(n^2)$.
- The operation of modifying the current solution is called a **move**.

Example:

$(p_j) =$ 100 60 70 45 45 4 4 4 15

$(w_j) =$ 10 10 12 8 8 1 1 1 4 $\quad c = 26$

Greedy:

$(x_j) =$ 1 1 0 0 0 1 1 1 0

$z = z^g = 172 \quad \overline{c} = 3$

Local Search:

$i = 2,\ j = 3$:

$(x_j) =$ 1 0 1 0 0 1 1 1 0

$z = 182 \quad \overline{c} = 1$

No further move is possible

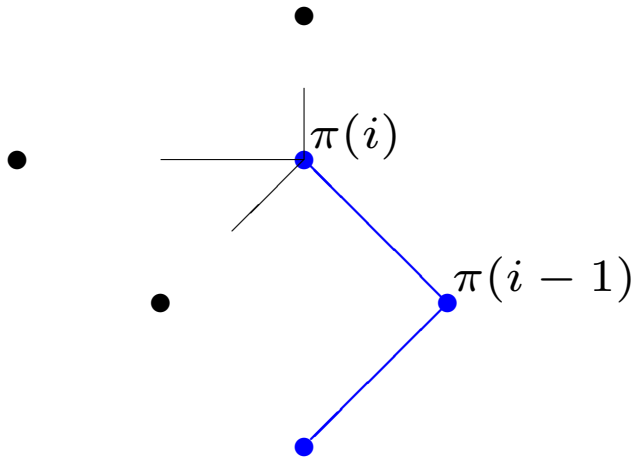
More complex algorithms: exchange items with item pairs, pairs with pairs, ... but

Note: the optimal solution is $(x_j) = (1\ 0\ 0\ 1\ 1\ 0\ 0\ 0\ 0)$;

to obtain it one should exchange quadruplets with pairs (impractical).

Heuristic algorithms, greedy algorithms for the TSP

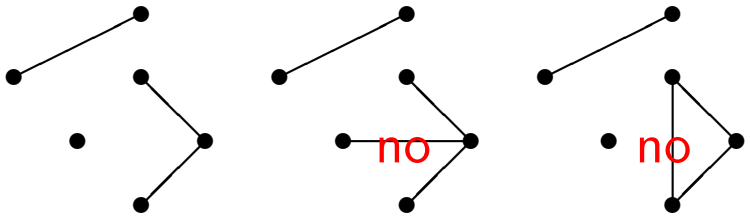
- Descriptions for the STSP. Immediate extension to the ATSP.
- **Nearest Neighbor**: iterative extension of a path through the shortest emanating edge.
- $\pi(1), \pi(2), \ldots$ = sequence of already selected **vertices**.
- **procedure Nearest Neighbor**:
  begin
    $\pi_1 := 1$ (**comment:** starting vertex v_1);
    for $i := 1$ **to** $n - 1$ **do**
      find the vertex v_k that minimizes $\{c_{\pi_i,k} : k \neq \pi_j \text{ for } 1 \leq j \leq i\}$;
      $\pi_{i+1} := k$
    endfor
  end.



- Time complexity $O(n^2)$.

Heuristic algorithms, greedy algorithms for the TSP (cont'd)

- **Multi-fragment:** iterative addition of the shortest non-forbidden edge.
- $\{\sigma(1), \sigma(2), \dots\}$ = set of the already selected **edges**.
- **procedure Multi-Fragment**:
  begin
  sort the edges by non-decreasing costs;
  $\sigma_1 :=$ first edge, $\sigma_2 :=$ second edge, $i := 2$;
  repeat
  let e be the next edge in the sequence;
  if $\{\sigma_1, \dots, \sigma_i\} \cup \{e\}$ does not contain circuits or vertices of degree 3

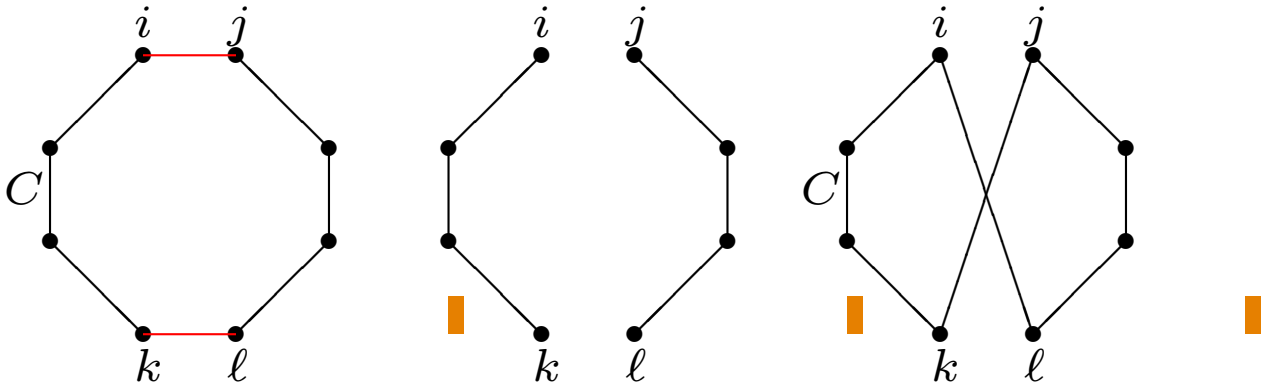


  then $i := i + 1$, $\sigma_i := e$
  until $i = n - 1$;
  $\sigma_n :=$ unique edge that closes the current circuit
  end.

- Time complexity $O(n^2 \log n)$.

Heuristic algorithms, local search for the STSP

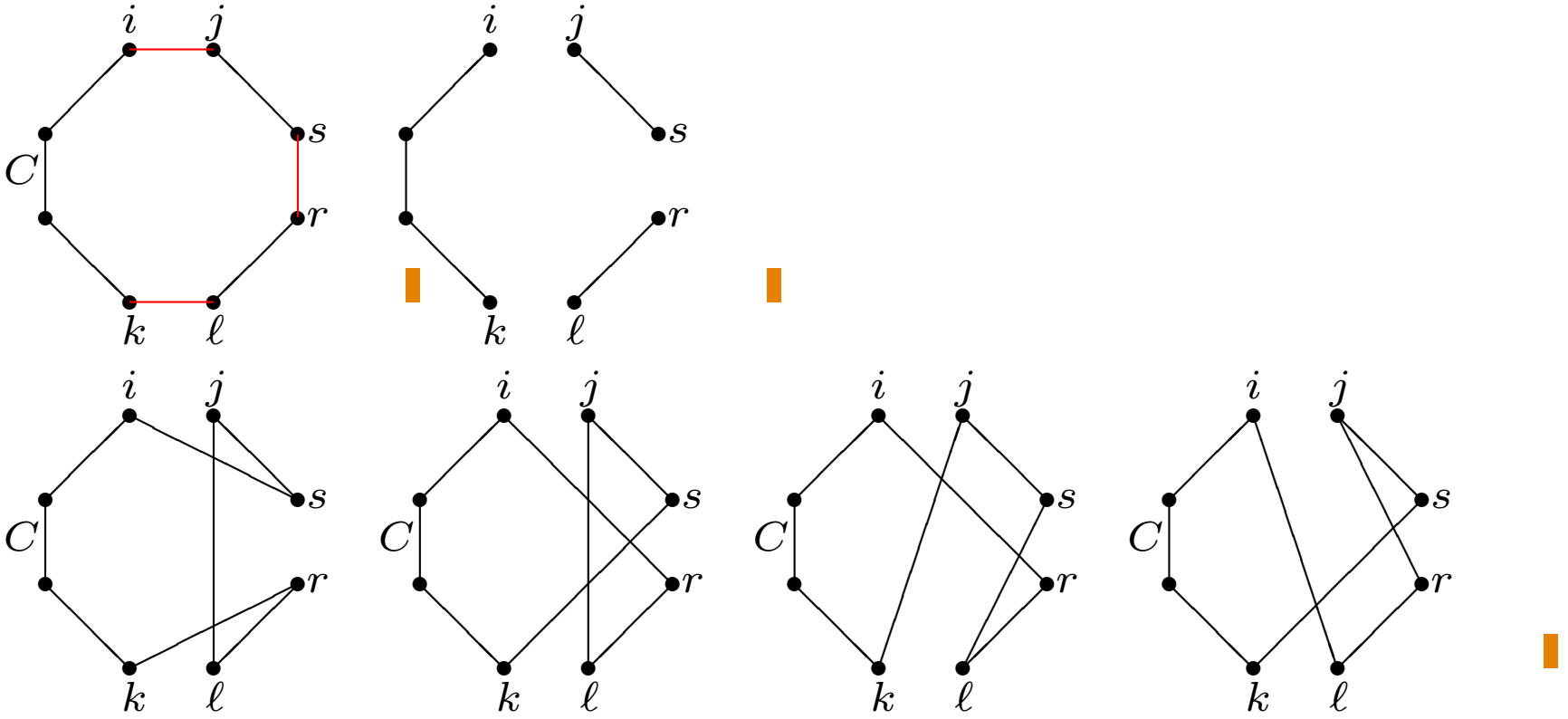
- $C =$ set of the edges of the current circuit.
- **procedure Two-Opt**:
  begin
    while $\exists\ (i,j),\ (k,\ell) \in C\ :\ c_{ij} + c_{k\ell} > c_{i\ell} + c_{kj}$ **do**
      $C := C \setminus \{(i,j),(k,\ell)\} \cup \{(i,\ell),(k,j)\}$
    endwhile
  end.



- The number of moves can be exponential.
- **General algorithm k-Opt:** remove k edges and interconnect the resulting paths.
- Practical applications: Two-Opt and Three-Opt.

Heuristic algorithms, local search for the STSP (cont'd)

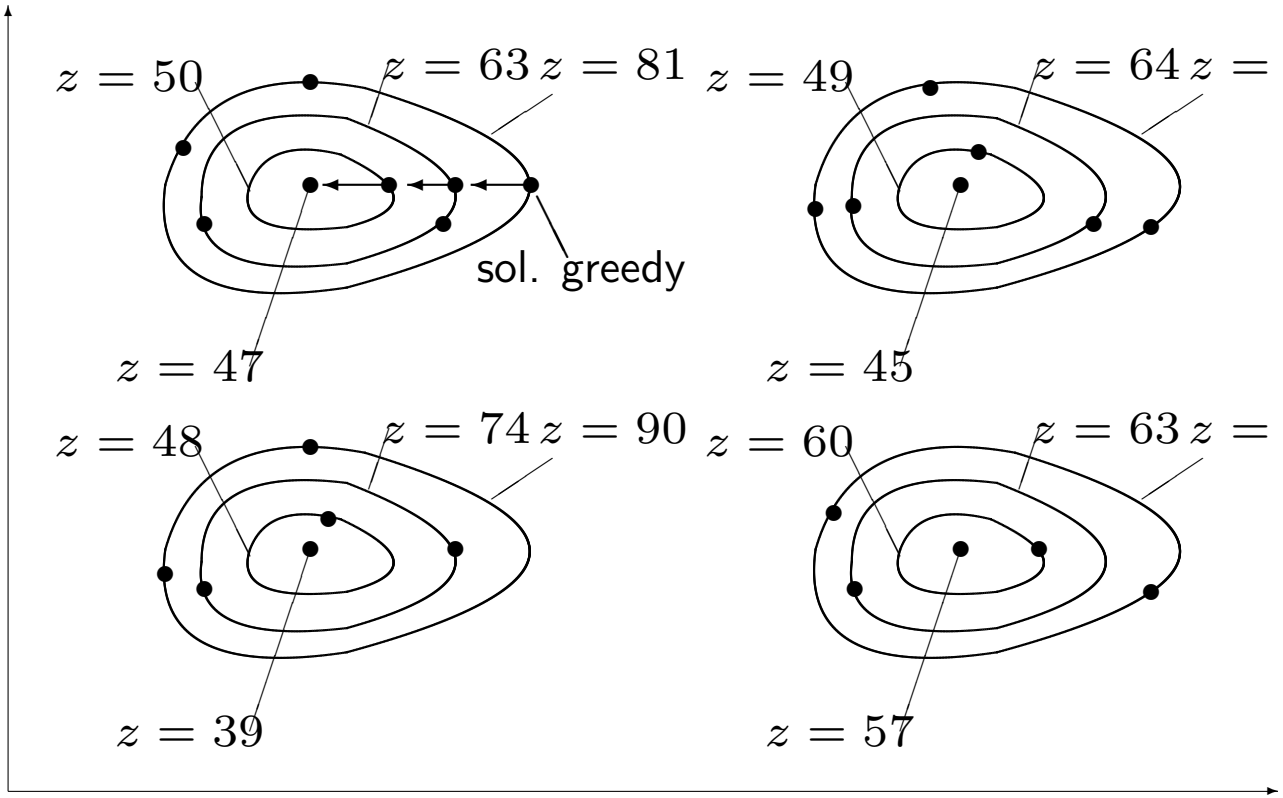
- **procedure Three-Opt**
  begin
    while $\exists\ (i,j),\ (k,\ell),\ (r,s) \in C\ :\ c_{ij} + c_{k\ell} + c_{rs} > \ \ldots$ **do**
      $C := C \setminus \{(i,j),(k,\ell),(r,s)\} \cup\ \ldots$
    endwhile
  end.



In the **course web page**: **applets** for executing all heuristic algorithms for the TSP.

Metaeuristic algorithms

- A local search starts from a feasible solution and explores a **neighborhood** of feasible solutions of increasing quality, terminating when no further improvement is possible.
- Main drawback: it can be trapped in a local minimum.
- Example: solution space and isocost lines (minimization problem):



- $=$ feasible solutions

- Various algorithms $\longleftrightarrow$ different methods (**paradigms**) to handle this drawback.
- Metaheuristics are nowadays the most widely used techniques for the practical solution of difficult optimization problems.

Metaheuristic algorithms (cont'd)

Basic definitions:

- **Metaheuristic** = generic scheme (**template**) for organizing a search in the solution space of an optimization problem in order to find good solutions.
- The trajectory followed by a solution during the search is often guided by a **neighborhood function** $\mathcal{N}$;
- $\mathcal{N}$ maps a solution s to a portion $\mathcal{N}(s)$ of the solution space containing solutions "close" to s;
- two solutions s and s' are close if s' can be obtained by applying some "simple" operator to s;
- **move** = transformation of s into s'.

Classification of Metaheuristic algorithms:

- **Single solution methods:**
  - Randomized algorithms;
  - Tabu Search;
  - Simulated Annealing, ...
- **Population based methods:**
  - Genetic Algorithms;
  - Scatter Search;
  - Ant Colony Optimization, ...
- Obviously non-monotone: the best encountered solution is stored.

Randomization

- Randomization is the simplest metaheuristic technique:
  - a greedy algorithm is "randomized" so that it can generate different solutions;
  - at each iteration a local search (possibly randomized) is used to improve the generated solution.
  - Best paradigm: **Greedy Randomized Adaptive Search Procedure (GRASP)**:

    At each iteration

    * define a **sorted list of the best candidates** for the next move;
    * randomly select the move;
    * update the list
- The improvement is generally limited:
  - the possibility that different local optima are "close" to each other is not exploited ;
  - it is preferable to start a new search "close" to the last local optimum found;

Tabu Search

- General strategy: the best move is **always** executed,

  even if it produces a solution worse than the current one (**uphill move**).
- In practice, the algorithm alternates between:
  - local search for finding a local optimum, and, once this has been found,
  - selection of the best move to a neighboring solution, which is then used as starting solution for a new local search.
- Should this be all that is done, the best move from the best neighbor of the local optimum

  could produce the local optimum we just left. Hence:
- **Tabu:** We save information on the most recent moves in one or more **Tabu lists**,

  that are used to **prohibit** new moves that would undo the progress obtained in recent moves.
- A Tabu Search algorithm includes other features. Mainly:
  - aspiration;
  - diversification;
  - intensification.

Tabu Search (cont'd)

Main components of a Tabu Search algorithm

1. Algorithm to generate a **starting solution** (e.g., Greedy).
2. Definition of the **neighborhood**, i.e., definition of the **move** that leads from a solution to a neighbor (e.g., Two-opt).
3. Definition of the **Tabu list**.
   - A Tabu Search algorithm is effective if it can explore a huge number (millions) of solutions;
   - $\Rightarrow$ each iteration must require a very short CPU time;
   - $\Rightarrow$ it would be inefficient to store **all** the **complete solutions** explored.
   - Example: for **KP** or **TSP** a solution consists of **n** values;

     if the Tabu list contains **t** solutions, testing a move requires **O(nt)** time (excessive).
   - **Two fundamental decisions:**

     A. stored information: usual techniques store one or more **attributes** of a move, e.g.,

     Two-opt for TSP: we store the shortest edge eliminated by the move;

     Local search for KP: we store the indices of the two exchanged items;

     a move is tabu if it inserts an edge (exchanges two items) from the tabu list.

Tabu Search (cont'd)

. . .

3. Definition of the **Tabu list**.

   . . .

   - **Two fundamental decisions:**

     . . .

     B. Tabu list length (**Tabu tenure**): a Tabu list stores a maximum number **t** of moves

     when it is full, the next stored move eliminates the oldest one;

     usual tenures are between 5 and 10 ("magic" number: 7).

4. **Aspiration criteria**: cases where Tabu can be violated,

   e.g., if the new solution is better that than the incumbent the move is accepted even if tabu.

5. **Diversification**: when the current region is "poor" of good solutions

   (e.g., no improvement since many iterations)

   we drastically change the current solution (e.g., by starting from a new greedy solution).

6. **Intensification**: when the current region is "rich" of good solutions

   (e.g., several improvement in recent iterations)

   we force the local search to remain close to the recent solutions.

Simulated Annealing

- Simulated annealing first appeared in 1983, before Tabu Search was invented (1986).
- **Main similarities**:
  - we move from a solution to a neighboring solution;
  - uphill moves (to a worse solution) are allowed.
- **Main differences**:
  - in **Tabu search** uphill moves are only allowed from a local optimum, and

    are not based on randomization.
  - in **Simulated annealing** uphill moves are always allowed, and

    are heavily based on randomization:

    the algorithm examines the neighboring solutions in random order, and performs the first move that

    a. is better than the current solution, **or**

    b. passes a special randomized test.
  - There are analogies with annihilation processes in thermodynamics:

    the basic ideas come from a 1953 algorithm on the simulation of annealing

    (controlled heating and cooling) in metallurgy.

Origins of Simulated Annealing

- From **Statistical mechanics:** a system in which
  - x is a state ($\longleftrightarrow$ a feasible solution);
  - $f(x)$ is the energy of state x ($\longleftrightarrow$ the solution value);
  - T is the system temperature ($\longleftrightarrow$ a parameter),

  randomly fluctuates from one state to another with a probability of visiting state x given by

  $$e^{-f(x)/(kT)}$$ (where k is the Boltzmann constant).

- To simulate an annihilation process,
  - from the current state x we generate a state y with probability f_{xy};
  - if $f(y) \leq f(x)$, y is **accepted**;
  - if $f(y) > f(x)$, y is **accepted with probability** $e^{(f(x)-f(y))/T}$
- Observations:
  - the probability of accepting a solution y worse than x decreases when $f(y) - f(x)$ grows;
  - the probability of accepting a solution y worse than x decreases when T decreases;
  - the temperature T decreases during execution;
  - when $T = 0$ uphill solutions are accepted with 0 probability ($\equiv$ local search).

Simulated Annealing algorithm for TSP

1. generate a starting solution C of value $z(C)$ and set $C^* := C$;

2. define an initial temperature T and a final temperature $T_{\min}$;

3. **while** $T > T_{\min}$ **do**

3.1 randomly select a move that transforms C to C';

3.2 $\Delta := z(C') - z(C)$;

3.3 **if** $\Delta \leq 0$ **then** (**comment:** downhill)

$C := C'$;

if $z(C) < z(C^*)$ **then** $z(C^*) := z(C)$

else (**comment:** possible uphill)

generate a random value $r \in [0, 1)$;

if $r < e^{-\Delta/T}$ **then** $C := C'$;

endif

3.4 decrease T

endwhile

- This version is called **non-homogeneous**: the temperature decreases at each iteration.
- In the **homogeneous** version the temperature is kept constant until an **equilibrium state** has been reached (usually a certain number of iterations) and then the temperature is decreased.

Decisions to be taken when implementing a Simulated Annealing algorithm

1. **Initial temperature**:
   - initially many moves have to be accepted $\Longrightarrow$ "high" initial temperature;
   - frequently found by trial-and-error (imposing $\approx 90\%$ acceptance).
2. **Cooling speed**:
   - it must be slow enough to ensure a good exploration. Two classical methods:
     - $T := \alpha T$ with $\alpha < 1$ but close to 1;
     - $T := \dfrac{T}{1 + \beta T}$, with $\beta > 0$ but close to 0.
3. **Final temperature**:
   - in principle $T_{\min} = 0$. In practice the search is halted when since P iterations the incumbent solution is not improved, or no move is accepted, or ...
4. **Equilibrium state** (homogeneous version): frequently given by a prefixed number of iterations.

Several Variants:

- **Reannealing**: 1st execution: we store the temperature T_0 at which the best solution was found.

  2nd execution: we perform a more accurate search with $T = T_0$.
- **Restricted neighborhood**: moves that are unlikely to produce good solutions are avoided

  (Example: for the TSP we only consider moves linking vertices that are "close" to each other).

Genetic algorithms

- Genetic algorithms are based on analogies with species evolution:
  - **solution** $\longleftrightarrow$ **individual**;
  - **set of solutions** $\longleftrightarrow$ **population**;
  - **solution value** $\longleftrightarrow$ individual **adaptation** to the environment;
  - **generation of new solutions** $\longleftrightarrow$ **reproduction**;
  - **elimination of bad solutions** $\longleftrightarrow$ **natural selection**.
- Let us consider a solution given by a binary vector x (e.g., KP): **Reproduction** occurs according to two main procedures:
  1. **Mutation**: randomly change the value of one or more x_j values (**chromosomes**);

     Example, KP: examine each x_j and, with **small** probability change its value, from 0 to 1 or from 1 to 0. (The new solution must be tested for feasibility.)

     Numerical example: (1 1 0 0 0 1 1 1 0) $\rightarrow$ (1 0 0 1 0 1 0 1 0).
  2. **Crossover**: from two solutions, randomly produce a new one which shares some characteristics of its parents;

     Example, KP: given $x^{(1)}$, $x^{(2)}$, generate a random value $a \in [1, n-1]$ and set $x_j^{(3)} = x_j^{(1)}$ for $j \leq a$, $x_j^{(3)} = x_j^{(2)}$ for $j > a$. (Test the new solution for feasibility.)

     Numerical example: $x^{(1)}$=(1 1 0 0 0 0 1 1 1), $x^{(2)}$=(1 0 1 0 0 1 1 1 0): $a = 6 \rightarrow$ $x^{(3)}$=(1 1 0 0 0 0 1 1 0).

Outline of a Genetic algorithm

1. generate a population of k solutions $\Sigma = \{S_1, \ldots, S_k\}$ (e.g., with a randomized Greedy);

2. **for each** $S \in \Sigma$ **do** improve S through a local search algorithm;

3. **while** a **convergence criterion** is not satisfied **do**

3.1 select k' disjoint subsets of Σ, of cardinality 1 or 2;

3.2 **for each** subset of cardinality 1 **do** produce a new feasible solution through mutation;

3.3 **for each** subset of cardinality 2 **do** produce a new feasible solution through crossover;

3.4 **for each** solution S produced in steps 3.2 e 3.3 **do**

improve S through local search;

let Σ' be the resulting set of new solutions;

enddo

3.5 use a **selection criterion** to select k **surviving individuals** from $\Sigma \cup \Sigma'$;

replace Σ with the selected surviving set

endwhile

- Steps **2.** and **3.4** are optional (or it can randomly be decided whether to execute them).
- The **selection criterion** is stochastic (various methods) and depending on the solution value.
- Other metaheuristics have been derived from Genetic Algorithms:

  The most important is **Scatter Search**.

Outline of a Scatter Search algorithm

1. generate an initial population P of solutions (**Pool**);
2. **for each** $s \in P$ **do**

   improve s through local search;

   associate two values to s: $q(s)$ (**quality**, depending on the solution value);

   $d(s)$ (**diversity** with respect to the solutions in P)

   end for
3. create a **reference set** $R = R_\alpha + R_\beta$ of distinct solutions, where:

   R_α contains the α solutions of P of higher quality;

   R_β contains the β solutions of P of higher diversity;
4. evolve the reference set R through:

   a. **subset generation**: generate a family F of subsets of R;

   b. **while** $F \neq \emptyset$ **do**

      combination: extract solutions from F and obtain a new solution s by combination;

      intensification: improve s through local search;

      update: on the basis of $q(s)$ and $d(s)$, possibly replace a solution of R_α or R_β with s

      endwhile;

   c. **if halting criteria** are not satisfied **then go to** a.

Ant Colony Optimization

Real ants

- In their search for food, **ants** initially move randomly;
- when an ant finds food, on its trip back to the nest it leaves a **pheromone trail**;
- when an ant finds a pheromone trail, it has a probability, proportional to the amount of pheromone, of following it;
- pheromone is volatile, and evaporates over time:

  the longer the travel to the nest, the more time the pheromone has to evaporate;
- as a result, after some time, the ant colony will follow the shortest path between nest and food.

Algorithmic ants

- **Ant Colony Optimization** algorithms are **multiagent systems** that imitate the ant behavior:
- **Ant** = simple computation agent which iteratively constructs a solution basing its decisions on
  - its **status** (the partial solution it has constructed so far), and
  - the pheromone trail (a value stored in a **global array** τ accessible to all ants) depending on the solutions constructed by other ants:
- the value of τ is decreased when proceeding from one iteration to the next one.

Outline of an Ant Colony Optimization algorithm

1. initialize the pheromone τ;
2. **while** a convergence criterion is not satisfied **do**

   for each ant **do** build a solution using the pheromone τ;

   decrease the value of τ (**evaporation**);

   increase the value of the τ_{ij}'s used in good solutions (**reinforcement**)

   endwhile

Swarm Intelligence (?) and other tools

In recent years, **undesirable proliferation** of methods based on metaphors of natural or manmade systems and processes;
mostly old ideas window dressed so as to be claimed as 'novel' on the basis of metaphors ☹:

- **Swarm Intelligence Algorithms**: ants, bees, wasps, termites ... almost every insect;
- other nature inspired methods: flies, bats, cuckoos, kangaroos, glow worms, bacteria;
- invasive weeds, musicians playing jazz, imperialist societies, intelligent water drops;
- even leapfrogs and mine blast!
  ☹☹☹

Practical issues related to metaheuristic algorithms

- **Implementation:**
  - high **quality over price** ratio:
  - metaheuristic algorithms are relatively easy to implement, even for very complex optimization problems, and generally give good practical results.
- **Experiments:**
  - the tuning of the (many) parameters can request heavy experimentations to produce good results.
- **Practical behavior:**
  - **Tabu Search** works well in most cases.
  - **Simulated Annealing** sometimes works well;

    it rarely works better than Tabu Search, but it is simple to implement it once tabu search has been implemented (using the same neighborhood).
  - Pure **Genetic algorithms** rarely work well;

    they can give good results in combination with other methods (e.g., with **Scatter search**).
  - **Ant colony optimization** can work well when the problem instance changes dynamically ($\Leftarrow$ the algorithm can adapt its behavior to changes).
- **What about exploiting the positive aspects of different paradigms?**

Variable Neighborhood Search (VNS)

Consider different paradigms (i.e., different neighborhood structures). **Basic facts:**

- a **local minimum wrt one neighborhood** is not necessary so with another;
- a **global minimum** is a local minimum wrt all possible neighborhoods.

Idea behind VNS: systematic change of neighborhood within the search.

Outline of a **Variable Neighborhood Search algorithm:**

1. Select a set of neighborhood structures $\mathcal{N}_k$ $(k = 1, \ldots, k_{\max})$;
2. find an initial solution x, and improve it through local search;
3. select a stopping condition;
4. **repeat**
   $k := 1$;
   repeat
   shaking: generate x' at random from $N_k(x)$;
   local search: apply a local search method to x' to find a local optimum x'';
   move or not: **if** x'' is better than the incumbent, **then** set $x := x''$ and $k := 1$
   else set $k := k + 1$ (or, if $k = k_{\max}$, set $k := 1$)
   until $k = k_{\max}$
   until stopping condition holds.