Approximate solution of combinatorial optimization problems

Silvano Martello
DEIS, Università di Bologna, Italy

May 16, 2011
1 Combinatorial Optimization problems

2 Approximation algorithms

3 Heuristic algorithms

4 Meta-heuristic algorithms
Combinatorial Optimization problems
Combinatorial Optimization problems

• given \( n \) decision variables \( x_1, \ldots, x_n \), find \( x \) so as to

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to:} & \quad g_i(x) \geq b_i \quad (i = 1, \ldots, m)
\end{align*}
\]

• \( f(x) = \text{objective function} \)

(alternatively: maximize \( f(x) \))

• \( g_i(x) \geq b_i = \text{constraints} \)

(alternatively: \( g_i(x) \leq b_i \), or \( g_i(x) = b_i \))

• Many specific problem classes are obtained by restricting the kind of functions \( f \) and \( g_i \) and/or the values variables \( x \) can take.

• Two main categories:
  – continuous variables;
  – discrete variables (Combinatorial Optimization).

• Most famous problem: \textbf{Linear Programming (LP)}:
  – \( f(x) \) and \( g_i(x) \) \((i = 1, \ldots, m)\) linear, \( x \in R^n \)
  – LP is continuous, but the number of potentially optimal solutions is finite, so it can also be seen as a combinatorial one.
Examples of Combinatorial Optimization problems

- **Assignment Problem:**

  given a cost matrix \([c_{ij}] (n \times n)\),

  select \(n\) elements, one per row and one per column,

  so that the sum of the selected costs is minimized.

\[
x_{ij} = \begin{cases} 
1 & \text{if row } i \text{ is assigned to column } j \\
0 & \text{otherwise} 
\end{cases} \quad (i, j = 1, \ldots, n)
\]

\[
\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij}
\]

\[
\sum_{j=1}^{n} x_{ij} = 1 \quad \text{for } i = 1, \ldots, n
\]

\[
\sum_{i=1}^{n} x_{ij} = 1 \quad \text{for } j = 1, \ldots, n
\]

\[
x_{ij} \in \{0, 1\} \quad \text{for } i, j = 1, \ldots, n
\]
Examples of Combinatorial Optimization problems

- **0–1 Knapsack Problem (KP):**
  
  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 $c$:

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

  $$\max \sum_{j=1}^{n} p_j x_j$$

  $$\sum_{j=1}^{n} w_j x_j \leq c$$

  $$x_j \in \{0, 1\} \quad j = 1, \ldots, n$$
Examples of Combinatorial Optimization problems

- 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** ($\text{edge } (i, j) \equiv \text{edge } (j, i)$), ⇒ **Symmetric TSP**

If the graph is **oriented** ($\text{arc } (i, j) \not\equiv \text{arc } (j, i)$), ⇒ **Asymmetric TSP**
Mathematical model of the Asymmetric TSP

\[ x_{ij} = \begin{cases} 
1 & \text{if arc } (i, j) \text{ belongs to the solution} \\
0 & \text{otherwise} 
\end{cases} \quad (i, j = 1, \ldots, n) \]

\[
\min \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} 
\]

\[
\sum_{i=1}^{n} x_{ij} = 1, \quad j = 1, \ldots, n \quad (1)
\]

\[
\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n \quad (2)
\]

Constraints (1): exactly one arc entering each vertex;

Constraints (2): exactly one arc emanating from each vertex;

Same model as the Assignment Problem. Question: is it correct for the Asymmetric TSP?

Answer: NO! Without additional conditions, partial circuits are possible.
Additional constraints: every \textbf{cut} $(S, V \setminus S)$ \textbf{must be crossed} by at least one arc:

\[
\begin{align*}
\min & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} \\
\sum_{i=1}^{n} x_{ij} & = 1, \quad j = 1, \ldots, n \\
\sum_{j=1}^{n} x_{ij} & = 1, \quad i = 1, \ldots, n \\
\sum_{i \in S, j \notin S} x_{ij} & \geq 1, \quad \forall S \subset V, S \neq \emptyset (*)
\end{align*}
\]

\( (*) \) \textbf{Alternatively:} \( \sum_{i \in S, j \in S} x_{ij} \leq |S| - 1, \quad \forall S \subset V. \)

\( (*) \) \textbf{In both cases, Exponential number of constraints!}

The assignment problem is used as a relaxation of TSP in branch-and-bound algorithms.
Mathematical model of the Symmetric TSP

\[ x_e = \begin{cases} 1 & \text{if edge } e (= (i, j) = (j, i)) \text{ belongs to the solution} \\ 0 & \text{otherwise} \end{cases} \]

\[ (e = 1, \ldots, m) \]

\[ \delta(i) = \{(i, j) : j \in V\} \text{ for } i \in V \]

\[ E(S) = \{(i, j) : i \in S \text{ and } j \in S\} \text{ for } S \subset V, S \neq \emptyset \]

\[
\begin{align*}
\min \frac{1}{2} \sum_{i=1}^{n} \sum_{e \in \delta(i)} c_e x_e \\
\sum_{e \in \delta(i)} x_e &= 2, \ i = 1, \ldots, n \quad (4) \\
\sum_{e \in E(S)} x_e &\leq |S| - 1, \ \forall S \subset V, S \neq \emptyset \quad (5) \\
x_e &\in \{0, 1\}, \ \forall \ e \in E
\end{align*}
\]

Constraints (4): **exactly two edges** incident with each vertex;

Constraints (5): no vertex subset can have a **partial circuit** (Exponential number!);

Objective function: the cost of each edge \((j, l)\) of the solution is summed **twice** in

\[
\sum_{i=1}^{n} \sum_{e \in \delta(i)} c_e x_e \text{ (for } i = j \text{ and for } i = l) \implies \frac{1}{2}.
\]

---

*S. Martello, Approximate solution*
Exact and approximate solution of Combinatorial Optimization problems

• For some (few) Combinatorial Optimization problems we know algorithms that find the optimal solution requiring a computing time that, in the worst case, is a polynomial function of the dimension of the problem instance.

• Example: the Assignment Problem can be solved in $O(n^3)$ time.

• For most Combinatorial Optimization problems we only know algorithms that find the optimal solution requiring a computing time that, in the worst case, is an exponential function of the dimension of the problem. (NP-hard problems).

• Example: 0-1 knapsack problem: $O(2^n)$, TSP: $O((n - 1)!)$

• An exponentially growing computing time is unacceptable for real world problems.

• In the last 40 years we have been unable to answer the question: can an NP-hard problem be solved in polynomial time?

• For no NP-hard problem we know a polynomial-time algorithm.

• A polynomial-time algorithm for a single (any) NP-hard problem, would automatically produce a polynomial-time algorithm for all NP-hard problems.

• Hence it is “improbable” that such algorithm can exist.

• And hence the need for approximation algorithms.
Approximation algorithms and heuristic algorithms

- An approximation (or heuristic) algorithm is a method that looks for a good solution within an acceptable computing time, without being able of ensuring the optimality of the solution found and, in some cases, without being able of finding a feasible solution.

- Normally the term “approximation” is preferred for algorithms for which theoretical results too can be obtained, such as, e.g.,
  - maximum relative error in the worst case;
  - expected error;
  - maximum asymptotic relative error;
  - etc.

- We prefer instead the term “heuristic” for algorithms for which the main interest is only a satisfactory practical behavior.

- Recently, many studies have been conducted on meta-heuristic algorithms, which frequently have an excellent experimental behavior.
Approximation algorithms
Worst-case behavior

- $I$ = problem instance (= specific numerical case);
- $A(I)$ = value of the solution found by an approximation algorithm $A$ for instance $I$;
- $OPT(I)$ = optimal solution value of $I$.

**Absolute worst-case performance ratio** of $A$:
- for a maximization problem:
  
  $$R_A = \inf_I \left\{ \frac{A(I)}{OPT(I)} \right\} \quad (R_A \leq 1)$$

- for a minimization problem:
  
  $$R_A = \sup_I \left\{ \frac{A(I)}{OPT(I)} \right\} \quad (R_A \geq 1)$$
0-1 Knapsack Problem

- Greedy algorithm:
  \[ z^g = \text{current profit}; \]
  \[ \overline{c} = \text{current residual capacity}; \]

- **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:
  \[
  \begin{align*}
  (p_j) &= 100 \quad 60 \quad 70 \quad 45 \quad 45 \quad 4 \quad 4 \quad 4 \quad 15 \\
  (w_j) &= 10 \quad 10 \quad 12 \quad 8 \quad 8 \quad 1 \quad 1 \quad 1 \quad 4 \\
  \end{align*}
  \]
  \[
  c = 26
  \]
  \[
  \text{GREEDY}: \\
  (x_j) = 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0 \\
  z = 172 \quad \overline{c} = 3
  \]
Worst-case behavior of GREEDY

- GREEDY can be arbitrarily bad, i.e., $R_{\text{GREEDY}}$ can be arbitrarily close to 0.

  - Example: $(p_j) = (2, M)$,
    $(w_j) = (1, M)$,
    $c = M > 2$

  - $\text{OPT}(I) = M, \text{GREEDY}(I) = 2 \Rightarrow \frac{\text{GREEDY}(I)}{\text{OPT}(I)} \xrightarrow{M \to \infty} 0$.

- Improved algorithm: $\overline{G}(I) = \max\left(G_{\text{GREEDY}}(I), \max_j \{p_j\}\right)$.

  - It can be proved that $R_{\overline{G}} = \frac{1}{2}$.

  - $R_{\overline{G}}$ is tight: there exist instances for which $R_{\overline{G}}$ is arbitrarily close to $\frac{1}{2}$.

  - Example: $(p_j) = (2, M/2, M/2)$,
    $(w_j) = (1, M/2, M/2)$,
    $c = M > 2$.

  - $\text{OPT}(I) = M, \overline{G}(I) = M/2 + 2 \Rightarrow \frac{\text{GREEDY}(I)}{\text{OPT}(I)} \xrightarrow{M \to \infty} \frac{1}{2}$.

- 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.

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

---

S. Martello, Approximate solution
PTAS for the 0-1 Knapsack Problem

• procedure $S(k)$ (comment: $k$ 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 $\varepsilon$:
  
  $\varepsilon = 1 - \frac{S(I)}{OPT(I)} \leq 1 - \frac{k}{k+1} = \frac{1}{k+1}$

• ∃ Fully polynomial-time approximation schemes (FPTAS) for which the running time grows polynomially with the inverse of the relative error (based on dynamic programming).
Approximability of the TSP

- Can any NP-hard problem be approximated in some way?
- **Bad news:** 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 an 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 $n$ (only unit cost edges are used).

It follows that algorithm $A$ must find a solution of value $\leq R \cdot n$.

Hence $A$ cannot use any edge having cost $R \cdot n$ as otherwise the cost of the circuit would be at least $R \cdot n + (n - 1)$.

The HC problem would then be solved in polynomial time by algorithm $A$: $G$ has an HC if and only if the solution produced by $A$ has value $n$ (and such solution would be the required HC).
**Things are better for a special case of the Symmetric TSP**

- 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 \((i, j) \in E \quad \forall \ i, j\).

- **Spanning Tree** of a graph \( G \ (n \ \text{vertices}) = \) connected graph containing \( n - 1 \) edges of \( G \).

- **Shortest Spanning Tree** of a graph \( G = \) Spanning Tree having minimum total cost.

  \[
  \begin{array}{c}
  1 \\
  2 \\
  3 \\
  \end{array}
  \begin{array}{c}
  4 \\
  5 \\
  6 \\
  7 \\
  \end{array}
  \]

  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 Symmetric TSP

- 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 $< 2 \cdot (\text{optimal TSP cost})$;
The worst-case bound is tight

(a) Euclidean graph (all edges)

(b) $q$ "squares"

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

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

(d) Optimal solution
$\sim 2q + 2$
Approximability status for the TSP

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

2. The **symmetric TSP with triangularity condition**
   - 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**);
   - **no better algorithm** is known.

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.
Example: 0-1 Knapsack Problem, local search algorithms

- **Procedure LOCAL_SEARCH_1:**
  - 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_1**
  begin
  call GREEDY ($z^g = \text{solution value}$, $c = \text{residual capacity}$);\[1.5ex]
  $z := z^g$;\[1.5ex]
  for $i := 1$ to $n$ do\[1.5ex]
    if $x_i = 1$ then\[1.5ex]
      for $j := i + 1$ to $n$ do (comment: if $j < i$ the exchange is infeasible)\[1.5ex]
        if $x_j = 0$ and $c + w_i \geq w_j$ and $p_j > p_i$ then\[1.5ex]
          $x_i := 0$, $x_j := 1$;\[1.5ex]
          $z := z - p_i + p_j$;\[1.5ex]
          $c := c + w_i - w_j$
        endif
    endif
  end.

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

- The operation of changing the current solution is called a **move**.
Example:

\[(p_j) = 100 \quad 60 \quad 70 \quad 45 \quad 45 \quad 4 \quad 4 \quad 4 \quad 15\]
\[(w_j) = 10 \quad 10 \quad 12 \quad 8 \quad 8 \quad 1 \quad 1 \quad 1 \quad 4\]
\[c = 26\]

**GREEDY:**

\[(x_j) = 1 \quad 1 \quad 0 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0\]
\[z = z^g = 172 \quad \bar{c} = 3\]

**LOCAL_SEARCH_1:**

\[i = 2, j = 3: \]
\[(x_j) = 1 \quad 0 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 0\]
\[z = 182 \quad \bar{c} = 1\]

No further improvement is possible
• **Procedure LOCAL\_SEARCH\_2:**
  - iteratively find **the best** one-to-one exchange and perform it;
  terminate when no improving exchange exists.

• **procedure LOCAL\_SEARCH\_2**
  begin
  call GREEDY \((z^g = \text{solution value, } \bar{c} = \text{residual capacity})\);
  \(z := z^g;\)
  \(\text{while } \emptyset \neq J = \{(i, j) : x_i = 1, x_j = 0 \text{ and } \bar{c} + w_i \geq w_j \text{ and } p_j > p_i\} \text{ do}\)
  \(\quad \text{let } (i^*, j^*) : p_{j^*} - p_{i^*} = \max_{(i, j) \in J} \{p_j - p_i\};\)
  \(\quad x_{i^*} := 0, x_{j^*} := 1;\)
  \(\quad z := z - p_{i^*} + p_{j^*};\)
  \(\quad \bar{c} := \bar{c} + w_{i^*} - w_{j^*};\)
  endwhile
  end.

• The time complexity depends on the implementation.
Example:

\[(p_j) = \begin{pmatrix} 100 & 60 & 70 & 45 & 45 & 4 & 4 & 4 & 15 \end{pmatrix} \]
\[(w_j) = \begin{pmatrix} 10 & 10 & 12 & 8 & 8 & 1 & 1 & 1 & 4 \end{pmatrix}, \quad c = 26 \]

**GREEDY:**

\[(x_j) = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{pmatrix} \]
\[z = z^g = 172, \quad \bar{c} = 3 \]

**LOCAL_SEARCH_2:**

\[i^* = 6, \quad j^* = 9: \]
\[(x_j) = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix} \]
\[z = 183, \quad \bar{c} = 0 \]

**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 (computationally too heavy).
Example: Symmetric TSP, greedy algorithms

- General idea: iterative extension of a path through the shortest emanating edge.
- $\pi(1), \pi(2), \ldots = \text{sequence of already selected vertices.}$
- **procedure GREEDY_1** (Nearest Neighbor)
  
  ```
  begin
  $\pi(1) := 1$ (*comment: starting vertex*);
  for $i := 1$ to $n - 1$ do
  find the vertex $k$ that minimizes $\{c_{\pi(i),k} : k \neq \pi(j) \text{ for } 1 \leq j \leq i\}$
  $\pi(i + 1) := k$
  end.
  ```

- Time complexity $O(n^2)$.
• General idea: iterative construction of the circuit through the shortest non-forbidden edge.
• \( \Sigma = \{ \sigma(1), \sigma(2), \ldots \} \) = set of the already selected edges.
• procedure GREEDY\textsubscript{2} (Multifragment)
  \begin{align*}
  & \text{begin} \\
  & \quad \text{find the edge } (k, j) \text{ with minimum } c_{k,j}; \\
  & \quad \sigma(1) := (k, j), \quad \Sigma := \{ \sigma(1) \}; \\
  & \quad \text{for } i := 2 \text{ to } n - 1 \text{ do} \\
  & \qquad \text{find the edge } (k, j) \text{ with minimum } c_{k,j} \text{ among those of } \\
  & \qquad \quad F(\Sigma) = \left\{ (k, j) \notin \Sigma : \Sigma \cup \{(k, j)\} \right\} \\
  & \qquad \quad 1. \text{ does not have vertices of degree } 3, \\
  & \qquad \quad 2. \text{ does not contain circuits} \\
  & \qquad \quad \sigma(i) := (k, j), \quad \Sigma := \Sigma \cup \{ \sigma(i) \}\}; \\
  & \quad \text{endfor}; \\
  & \quad \text{add the (unique) edge that closes the circuit} \\
  & \text{end}.
\end{align*}
• Time complexity \( O(n^2 \log n) \).
Example: Symmetric TSP, local search algorithms

- \( C \) = set of the edges of the current circuit.
- procedure LOCAL_SEARCH_1 (two-opt)
  
  begin
  
  while \( \exists (i, j), (k, \ell) \in C : c_{(i,j)} + c_{(k,\ell)} > c_{(i,\ell)} + c_{(k,j)} \) do
  
  \( C := C \setminus \{(i, j), (k, \ell)\} \cup \{(i, \ell), (k, j)\} \)
  
  endwhile
  
  end.

- The number of moves can be exponential.
• **procedure** LOCAL_SEARCH_2 (**three-opt**)

  begin
  
  while ∃ (i, j), (k, ℓ), (r, s) ∈ C : \( c_{(i,j)} + c_{(k,\ell)} + c_{(r,s)} > \ldots \) do

  \( C := C \setminus \{(i, j), (k, \ell), (r, s)\} \cup \ldots \)

  endwhile

  end.\[1\]

\[1\] S. Martello, Approximate solution
Metaheuristic Algorithms
**The metaheuristic approach**

- 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):

  \[ z = 50 \quad z = 63 \quad z = 81 \quad z = 49 \quad z = 64 \quad z = 65 \]

  \[ z = 47 \quad z = 45 \]

  \[ z = 48 \quad z = 74 \quad z = 90 \quad z = 60 \]

  \[ z = 39 \quad z = 57 \]

- Various algorithms \(\leftrightarrow\) 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

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, ...
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.
- **Example: TSP:**

```plaintext
procedure GREEDY₂^R (Randomized Multifragment)
begin
  let Q be a prefixed (input) value;
  randomly select a set of Q edges σ(1), ..., σ(Q) with no vertex in common;
  Σ := {σ(1), ..., σ(Q)};
  for i := Q + 1 to n − 1 do
  ...
end.
```

- The improvement is generally limited:
  - it does not exploit the possibility that different local optima are “close” to each other;
  - it is preferable to start a new local search “close” to the last local optimum found;
  - best method: **GRASP (Greedy Randomized Adaptive Search Procedure).**
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.
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;
   - ⇒ each iteration must require a very short CPU time;
   - ⇒ 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:**
     - **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.
Main components of a Tabu Search algorithm

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 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 \( \leftrightarrow \) a feasible solution;
  - \( f(x) \) is the energy of state \( x \) \( \leftrightarrow \) the solution value;
  - \( T \) is the system temperature \( \leftrightarrow \) a parameter,

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

\[ e^{-f(x)/(kT)} \quad (\text{where } k \text{ is the Boltzmann constant}). \]

- 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 $T_{\text{min}}$;

3. while $T > T_{\text{min}}$ 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 $\implies$ “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;
     - $T := \frac{T}{1 + \beta T}$, with $\beta > 0$ but close to 0.

3. **Final temperature:**
   - In principle $T_{\text{min}} = 0$. 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 ⇐⇒ individual;
  • set of solutions ⇐⇒ population;
  • solution value ⇐⇒ individual adaptation to the environment;
  • generation of new solutions ⇐⇒ reproduction;
  • elimination of bad solutions ⇐⇒ 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: $(1\ 1\ 0\ 0\ 0\ 1\ 1\ 1\ 0) \to (1\ 0\ 0\ 1\ 0\ 1\ 0\ 1\ 0)$.

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

  S. Martello, Approximate solution 43
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 $\Sigma$, 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 $\Sigma'$ be the resulting set of new solutions;

   enddo

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

   replace $\Sigma$ 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 probably 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:
   - $R_\alpha$ contains the $\alpha$ solutions of $P$ of higher quality;
   - $R_\beta$ contains the $\beta$ 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 $R_\alpha$ or $R_\beta$ 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 $\tau$ accessible to all ants) depending on the solutions constructed by other ants:
- the value of $\tau$ is decreased when proceeding from one iteration to the next one.
Outline of an Ant Colony Optimization algorithm

1. initialize the pheromone $\tau$;

2. while a convergence criterion is not satisfied do
   for each ant do build a solution using the pheromone $\tau$;
   decrease the value of $\tau$ (evaporation);
   increase the value of the $\tau_{ij}$’s used in good solutions (reinforcement);
endwhile

Swarm Intelligence Algorithms

- Algorithms based on the collective behavior of species
  - ants
  - bees
  - wasps
  - termites
  - ...

S. Martello, Approximate solution
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 change dynamically (the algorithm can adapt its behavior to changes).
To study these issues in more depth

**Complexity**

- M.R. Garey, D.S. Johnson

**Knapsack Problem**

- S. Martello, P. Toth.

**Traveling Salesman Problem**

- E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan, D. B. Shmoys (eds.).
- G. Gutin, A. Punnen (eds.).
Assignment Problem

- R. Burkard, M. Dell’Amico, S. Martello.

Heuristics and Metaheuristics

- I.H. Osman, J.P. Kelly (eds.).
- F. Glover, M. Laguna (eds.).
- E. Aarts, J.K. Lenstra (eds.).
- S. Voss, S. Martello, I. Osman, C. Roucairol (eds.).
- C.C. Ribeiro, P. Hansen (eds.).
- E.-G. Talbi.
Freeware

Usually copyrighted for professional use.

Knapsack Problem

- [http://www.or.deis.unibo.it/kp/Codes.zip](http://www.or.deis.unibo.it/kp/Codes.zip):
- Fortran codes for the 0-1 Knapsack Problem and many variants.

Traveling Salesman Problem

- [http://www.or.deis.unibo.it/research_pages/tspsoft.html](http://www.or.deis.unibo.it/research_pages/tspsoft.html):
- list of links to codes for exact and approximation algorithms for TSP, ATSP, Hamiltonian circuit and other variants (Few proprietary codes);
- various languages: Pascal, Turbo Pascal, Fortran, C, C++, AMPL, Java, Mathematica.

Assignment Problem

  direct downloads and links to Pascal, Fortran and C codes.
  Java applets for the Hungarian algorithm.