EAIA'96

ADVANCED SCHOOL ON
ARTIFICIAL INTELLIGENCE

Estoril, Portugal

October 7-11, 1996

METAHEURISTICS AND APPLICATIONS

Celso C. Ribeiro

Department of Computer Science

Catholic University of Rio de Janeiro, Brazil

E-mail: celso@inf.puc-rio.br

http://www.inf.puc-rio.br/~celso

Last update: May 1997

Contents:

1. INTRODUCTION TO COMPLEXITY THEORY

- ***Decision Problems***

"Is there a particular structure satisfying some specific property?"

Result: "YES" ou "NO"

- ***Optimization Problems***

"Among all structures satisfying some specific property, find one optimizing some specific cost function."

Result: an optimal feasible solution

- ***Example: Traveling Salesman Problem***

Input: $\boldsymbol{n}$ cities and a matrix $\boldsymbol{C}=\{\boldsymbol{c_{ij}}\}$ of distances between them

Decision problem: "Given $\boldsymbol{n}$, a distance matrix $\boldsymbol{C}=\{\boldsymbol{c_{ij}}\}$ and an integer $\boldsymbol{L}$, is there a Hamiltonian cycle with length less than or equal to $\boldsymbol{L}$?"

Optimization problem: "Given $\boldsymbol{n}$ and a distance matrix $\boldsymbol{C}=\{\boldsymbol{c_{ij}}\}$, find a minimum length Hamiltonian cycle."

- ***Example: Knapsack Problem***

Input: $\boldsymbol{n}$ items, profits $\boldsymbol{c_j}$ and weigths $\boldsymbol{a_j}$ for any item $\boldsymbol{j} \in \{1,2,...,\boldsymbol{n}\}$, maximum weigth $\boldsymbol{b}$

Decision problem: "Given $\boldsymbol{n}$, $\boldsymbol{c}$, $\boldsymbol{a}$, $\boldsymbol{b}$, and an integer $\boldsymbol{L}$, is there a subset $\boldsymbol{S} \subseteq \{1,2,...,\boldsymbol{n}\}$ such that $\Sigma_{j\in S}\, \boldsymbol{a_j} \leq \boldsymbol{b}$ and $\Sigma_{j\in S}\, \boldsymbol{c_j} \geq \boldsymbol{L}$?"

Optimization problem: "Given $\boldsymbol{n}$, $\boldsymbol{c}$, $\boldsymbol{a}$ and $\boldsymbol{b}$, find a subset $\boldsymbol{S^*} \subseteq \{1,2,...,\boldsymbol{n}\}$ maximizing $\Sigma_{j\in S}\, \boldsymbol{c_j}$ among all $\boldsymbol{S} \subseteq \{1,2,...,\boldsymbol{n}\}$ such that $\Sigma_{j\in S}\, \boldsymbol{a_j} \leq \boldsymbol{b}$."

- ***Complexity Theory: based on decision problems***

It can be shown that the decision version associated to some specific problem cannot be more difficult than its optimization version.

- ***Class* P*: decision problems for which polynomial time (deterministic) algorithms are known***

A (deterministic) algorithm for a decision problem A is said to be polynomial if it runs in time limited by a polynomial in the size $L(I_A)$ of its input I_A.

Examples:
Shortest paths
Minimum spanning trees
Maximum flow
Linear programming

- ***Non-deterministic algorithm***

Primitives:
`Choice` - guesses a solution (certificate)
`Check` - checks (in polynomial time) if a "guessed" solution leads or not to a "yes" answer
`Success` - the algorithm gets a "yes" answer after applying **`Check`**
`Fail` - the algorithm is not able to finish with a "yes" answer

Result: if **`Choice`** guesses a solution leading to a "yes" answer (and it has this unusual capability), then the algorithm runs in polynomial time.

Alternatively, the following ***interpretation*** is possible:

Non-deterministic algorithms make use of a special instruction

`GO TO Label_A, Label_B`

which creates copies of the execution, as if within an environment of unlimited parallelism, each copy correponding to a different possible guess.

A non-deterministic algorithm runs in polynomial time if the first branch (or "copy") leading to a "yes" answer stops in polynomial time.

- ***Class* NP*: decision problems for which polynomial time non-deterministic algorithms are known***
  (or, equivalently, decision problems for which any certificate - solution proposal - can be checked in polynomial time for a "yes" answer)

Examples:
Traveling salesman problem
Graph coloring
0-1 optimization
Integer programming

Important result: $P \subseteq NP$
Open question: $P = NP$ or $P \subset NP$?

- **NP*-complete problems: problems in* NP *such that if there ever exists a polynomial time deterministic algorithm for its solution, then all problems in* NP *are also solvable by polynomial time deterministic algorithms***

We say that a decision problem A *polynomially transforms* to another decision problem B if, given any input I_A for problem A, we can construct an input I_B for problem B in polynomial (in $L(I_A)$) time, such that I_A is a "yes" answer for A if and only if I_B is a "yes"answer for B.

A decision problem $A \in NP$ is said to be NP-complete if all other problems in NP polynomially transform to A.

Examples:
Traveling salesman problem
Knapsack problem
Graph coloring
Integer programming

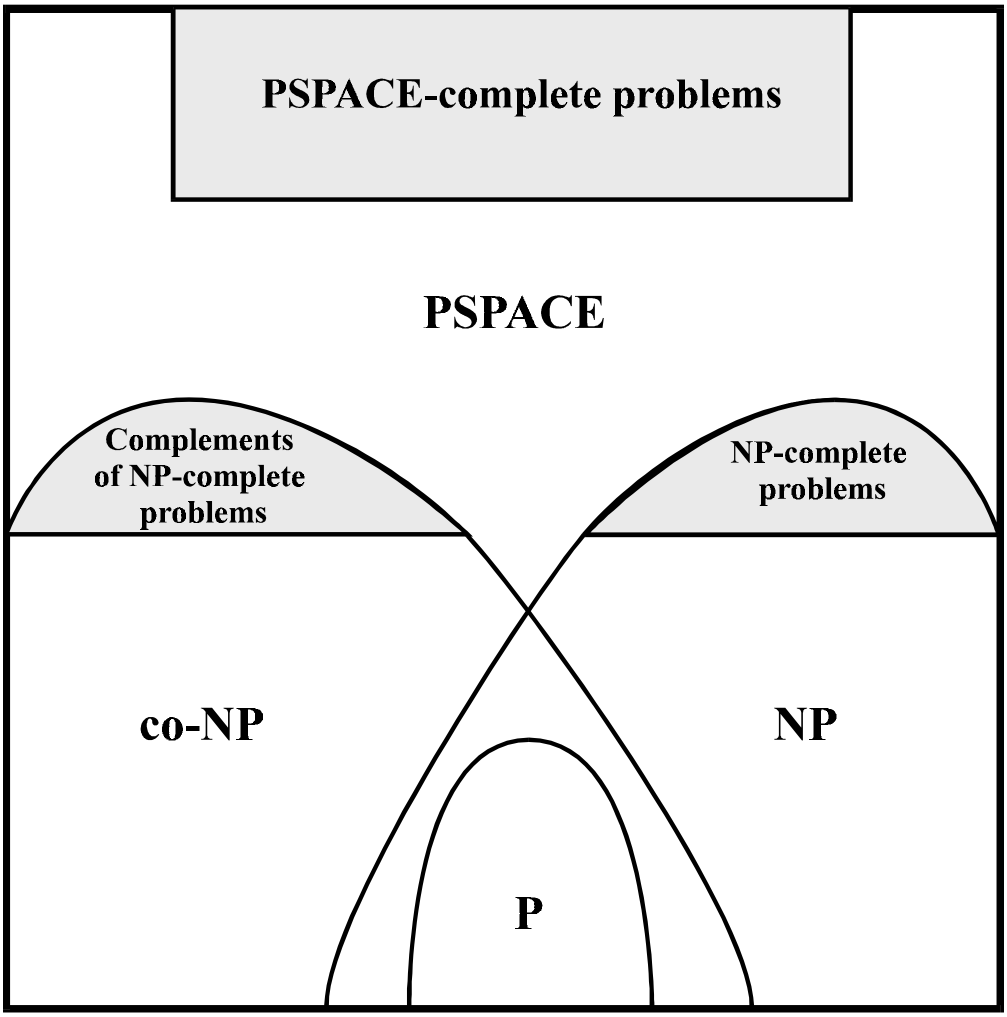
Polynomial time deterministic algorithms for **NP**-complete problems either ***do not exist*** or ***have not been found***.

- **NP**-*hard problem: optimization version of any* **NP**-*complete problem*

- *Class* **P**-*space: decision problems solvable by a deterministic algorithm using only a polynomial amount of memory space*

- *Simplified view of the world of decision problems*

Figure 1: Complexity classes



- ***Approaches to* NP*-hard problems***

1. Exact non-polynomial algorithms:
   - Dynamic programming
   - Branch-and-bound
   - Backtracking

2. Pseudo-polynomial algorithms: polynomial time deterministic algorithms on both the instance size and the largest problem data
   - Knapsack problem

3. Parallel processing: speedups in practice, without reduction in complexity

4. Special cases polynomially solvable:
   - Graph coloring for interval graphs

5. Approximation algorithms: feasible solution within some guaranteed deviation from the actual optimal value
   - Bin packing

6. Probabilistic algorithms:
   - Convergence in expected value: the expected value of the solutions found goes to the optimum when problem size increases asyntotically.
   - Convergence in probability: for any $\varepsilon > 0$, the probability that the difference between the optimum value and the expected value of the solutions found be greater than ε goes to zero when problem size increases asyntotically.

7. Heuristics: any approximate method built up on the basis of the structural properties or on the characteristics of problem solutions, with reduced complexity with respect to exact algorithms and providing, in general, good quality feasible solutions (without a formal guarantee of solution quality).
   - constructive tailored methods
   - local search
   - metaheuristics

2. CONSTRUCTIVE METHODS

- ***Combinatorial optimization problem (minimization version):***

Given a finite set $\boldsymbol{E} = \{1,2,...,\boldsymbol{n}\}$ and a cost function $\boldsymbol{c}: 2^{\boldsymbol{E}} \to \boldsymbol{R}$,
find $\boldsymbol{S^*} \in \boldsymbol{F}$ such that $\boldsymbol{c}(\boldsymbol{S}^*) \leq \boldsymbol{c}(\boldsymbol{S})$ (minimization problem), $\forall \boldsymbol{S} \in \boldsymbol{F}$,
where $\boldsymbol{F} \subseteq 2^{\boldsymbol{E}}$ is a collection of subsets (feasible solutions).

Linear combinarial optimization problem:
$\boldsymbol{c}: \boldsymbol{E} \to \boldsymbol{R}$ and $\boldsymbol{c}(\boldsymbol{S}) = \sum_{\boldsymbol{e} \in \boldsymbol{S}} \boldsymbol{c}(\boldsymbol{e})$, $\forall \boldsymbol{S} \in \boldsymbol{F}$

- ***Examples:***

Traveling salesman problem:
$\boldsymbol{E}$: set of edges, $\boldsymbol{c}(\boldsymbol{e}) = \boldsymbol{c_e}$ $\forall \boldsymbol{e} \in \boldsymbol{E}$
$\boldsymbol{F}$: any subset of edges forming a Hamiltonian cycle

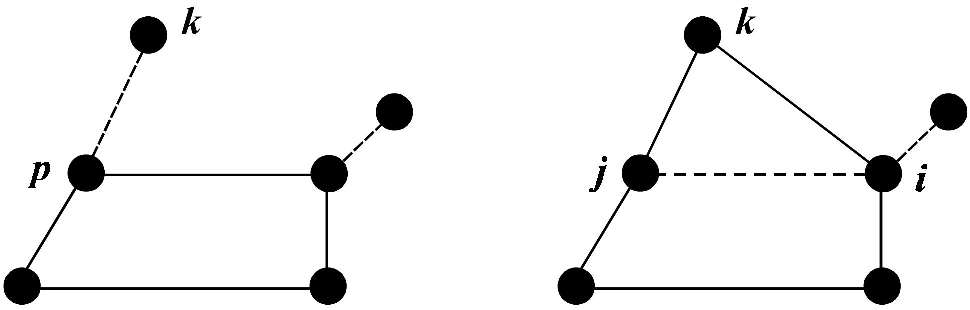
Knapsack problem:
$\boldsymbol{E}$: set of items, $\boldsymbol{c}(\boldsymbol{e}) = \boldsymbol{c_e}$ $\forall \boldsymbol{e} \in \boldsymbol{E}$
$\boldsymbol{F}$: any subset of items satisfying a weigth constraint $\sum_{\boldsymbol{e} \in \boldsymbol{F}} \boldsymbol{a_e} \leq \boldsymbol{b}$

- ***Build up a feasible solution from scratch***

Traveling salesman problem: Farthest insertion algorithm

Step 1. Start with a partial tour formed by a single, arbitrarily chosen city.
Step 2. If the current partial tour does not include all cities, then find those cities $\boldsymbol{k}$ and $\boldsymbol{p}$, $\boldsymbol{p}$ on the tour and $\boldsymbol{k}$ not, for which $\boldsymbol{c_{kp}}$ is maximized.
Step 3. Let $\boldsymbol{i^*}$ and $\boldsymbol{j^*}$ be two cities minimizing $\{\boldsymbol{c_{ik}} + \boldsymbol{c_{kj}} - \boldsymbol{c_{ij}}\}$, among all pairs of cities $\boldsymbol{i}$ and $\boldsymbol{j}$ on the tour.
Step 4. Replace edge $(\boldsymbol{i^*},\boldsymbol{j^*})$ in the tour by edges $(\boldsymbol{i^*},\boldsymbol{k})$ and $(\boldsymbol{k},\boldsymbol{j^*})$ and go back to step 2.

Figura 2: Choosing a city to be inserted and an edge to be removed



Algorithm stops either:
(a) after constructing the first (complete) feasible solution, or
(b) failing to find one.

Notice that some edges may be inserted in the current solution, but afterwards removed.

- ***Template: Greedy algorithm (minimization)***

Step 1. Sort the elements of $\boldsymbol{E}$ in increasing order: $\boldsymbol{c}(\boldsymbol{e}_1) \le \boldsymbol{c}(\boldsymbol{e}_2) \le \ldots \le \boldsymbol{c}(\boldsymbol{e}_n)$
Step 2. $\boldsymbol{S} \leftarrow \emptyset$
Step 3. For all $\boldsymbol{i} = 1,2,\ldots,\boldsymbol{n}$ do:
(a) If $\boldsymbol{S} \cup \{\boldsymbol{e}_i\} \in \boldsymbol{F}$ then $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{\boldsymbol{e}_i\}$
Step 4. Best solution found: $\boldsymbol{S}^* \leftarrow \boldsymbol{S}$ and $\boldsymbol{c}(\boldsymbol{S}^*) = \Sigma_{i \in S^*} \boldsymbol{c}(\boldsymbol{e}_i)$

Minimum spanning tree (Kruskal's greedy algorithm is optimal)
Knapsack problem (greedy algorithm is not optimal)

The greedy algorithm gives the optimal solutions for matroids:
(a) $\boldsymbol{S}_1 \subseteq \boldsymbol{S}_2 \in \boldsymbol{F} \Rightarrow \boldsymbol{S}_1 \in \boldsymbol{F}$
(b) $\emptyset \in \boldsymbol{F}$
(c) $\boldsymbol{S}_1, \boldsymbol{S}_2 \in \boldsymbol{F}$ and $|\boldsymbol{S}_1| = |\boldsymbol{S}_2| + 1 \Rightarrow$
$\exists\, \boldsymbol{s} \in \boldsymbol{S}_1 \setminus \boldsymbol{S}_2$ such that $\boldsymbol{S}_2 \cup \{\boldsymbol{s}\} \in \boldsymbol{F}$

- ***Template: Greedy randomized algorithm (minimization)***

Step 1. Sort the elements of $\boldsymbol{E}$ in increasing order:
$\boldsymbol{c}(\boldsymbol{e}_1) \leq \boldsymbol{c}(\boldsymbol{e}_2) \leq \ldots \leq \boldsymbol{c}(\boldsymbol{e}_{\boldsymbol{n}})$
Step 2. $\boldsymbol{S} \leftarrow \varnothing$
Step 3. For all $\boldsymbol{i} = 1,2,\ldots,\boldsymbol{n}$ do:
(a) Create a list $\boldsymbol{L} \subseteq \{1,2,\ldots,\boldsymbol{n}\} \setminus \boldsymbol{S}$ such that $\boldsymbol{S} \cup \{\boldsymbol{e}\} \in \boldsymbol{F}$ for any $\boldsymbol{e} \in \boldsymbol{L}$
(b) Randomly select an element $\boldsymbol{e} \in \boldsymbol{L}$
(c) Set $\boldsymbol{S} \leftarrow \boldsymbol{S} \cup \{\boldsymbol{e}\}$
Step 4. Best solution found: $\boldsymbol{S}^* \leftarrow \boldsymbol{S}$ and $\boldsymbol{c}(\boldsymbol{S}^*) = \Sigma_{\boldsymbol{i} \in \boldsymbol{S}^*} \boldsymbol{c}(\boldsymbol{e}_{\boldsymbol{i}})$

Greedy randomized algorithm with candidate list:

(a) Solution quality depends on the ***quality*** of the elements in the candidate list ***L***.

(b) Solution diversity depends on the ***cardinality*** of the candidate list ***L***.

(c) Randomization helps!

3. LOCAL SEARCH

- ***Optimization problems and algorithms***

Optimization problem	Polynomial algorithm	Exponential algorithm
Optimum (exact)	Fast optimization • Linear programming	Enumerative optimization • Branch-and-bound
Approximation	Fast approximation • Performance analysis	LOCAL SEARCH

Optimization problem: Given a symmetric TSP instance,
find a shortest tour (whose length will be L^*).

Approximation problem: Given a symmetric TSP instance and
any $\alpha > 1$, find a tour of length $L_{\text{approx}} < \alpha . L^*$.

Easy approximation problem (fast algorithm): Given a symmetric TSP instance that satisfy the triangle inequality, find a tour of length less than $(3/2).L^*$.

- ***Built around a neighborhood search procedure***

$F \subseteq 2^E$ is the set of feasible solutions.

Start at a feasible solution S_0 and improve it by local changes at each iteration: S is the current feasible solution.

Neighborhood N: $F \to 2^F$ may be seen as a (symmetric) adjacency relation on the set F of feasible solutions.

A subordinate heuristic is used to obtain a new solution S' in the neighborhood $N(S)$ of the current feasible solution S.

May take exponential time!

- ***Main issues***

(a) Neighborhood definition
(b) Search strategy

- ***Typical neighborhood definitions***

Knapsack problem:
The neighborhood $\boldsymbol{N}(\boldsymbol{S})$ is formed by either inserting or removing an item to/from the current set $\boldsymbol{S}$ of selected items.

0-1 integer programming:
$\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_{\boldsymbol{n}})$
$\boldsymbol{N}(\boldsymbol{x}) = \{\boldsymbol{x}' \in \{0,1\}^{\boldsymbol{n}} : \exists$ exactly one index $\boldsymbol{k} \in \{1,2,...,\boldsymbol{n}\}$ such that
$\boldsymbol{x}'_{\boldsymbol{k}} = 1 - \boldsymbol{x}_{\boldsymbol{k}}$ and $\boldsymbol{x}'_{\boldsymbol{j}} = \boldsymbol{x}_{\boldsymbol{j}}$ for any $\boldsymbol{j} \neq \boldsymbol{k}\}$

Scheduling problem:
A solution is a permutation of the $\boldsymbol{n}$ tasks to be executed.
$\boldsymbol{S}$: $1 - 2 - ... - \boldsymbol{i} - \boldsymbol{a} - \boldsymbol{b} - ... - \boldsymbol{c} - \boldsymbol{j} - \boldsymbol{d} - ... - \boldsymbol{n}$
$\boldsymbol{S}' \in N_1(\boldsymbol{S})$: $1 - 2 - ... - \boldsymbol{j} - \boldsymbol{a} - \boldsymbol{b} - ... - \boldsymbol{c} - \boldsymbol{i} - \boldsymbol{d} - ... - \boldsymbol{n}$ (exchange)
$\boldsymbol{S}' \in N_2(\boldsymbol{S})$: $1 - 2 - ... - \boldsymbol{j} - \boldsymbol{i} - \boldsymbol{a} - \boldsymbol{b} - ... - \boldsymbol{c} - \boldsymbol{d} - ... - \boldsymbol{n}$ (shift)

*Traveling salesman problem: **k**-exchange procedure (2-opt, 3-opt)*
The neighborhood $\boldsymbol{N}(\boldsymbol{S})$ is formed by tours obtained by replacing $\boldsymbol{k}$ edges of the current tour $\boldsymbol{S}$ by $\boldsymbol{k}$ edges not in the tour.

Figure 3: 2-opt neighborhood for TSP

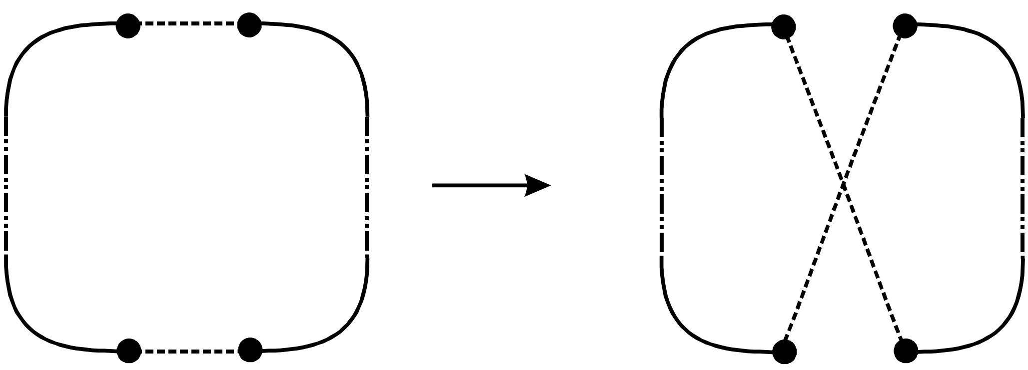
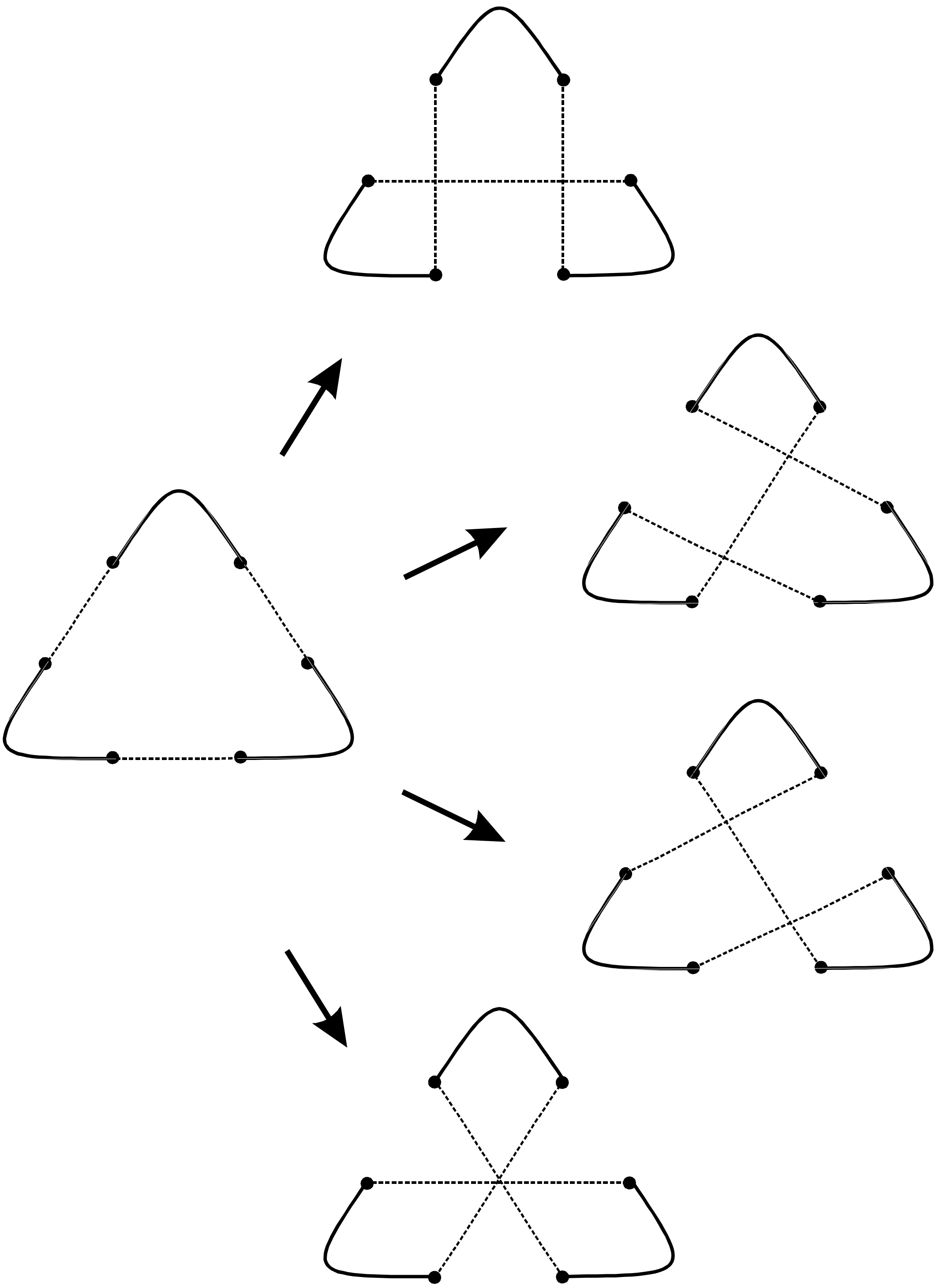


Figure 4: 3-opt neighborhood for TSP



- ***Template: Iterative improvement or hill-descending algorithm (minimization)***

> ***Step 1.*** Let $\boldsymbol{S}_0$ be an initial feasible solution and set $\boldsymbol{S} \leftarrow \boldsymbol{S}_0$
> ***Step 2.*** While there exists $\boldsymbol{S'} \in \boldsymbol{N}(\boldsymbol{S})$ such that $\boldsymbol{c}(\boldsymbol{S'}) < \boldsymbol{c}(\boldsymbol{S})$ do:
> (a) Set $\boldsymbol{S} \leftarrow \boldsymbol{S'}$
> ***Step 3.*** Set $\boldsymbol{S^*} \leftarrow \boldsymbol{S}$ and output the best solution $\boldsymbol{S^*}$

Monotonic local search approach which stops at the first local optimum: $\boldsymbol{S}$ is locally optimal with respect to the neighnorhood $\boldsymbol{N}$.

First choice: construction of the initial feasible solution
Second choice: identification of a "good", suitable neighborhood
Third choice: finding an improving (if any) solution within the neighborhood

- ***More advanced search strategies: Steepest-descent (minimization)***

> ***Step 1.*** Let $\boldsymbol{S}_0$ be an initial feasible solution and set $\boldsymbol{S} \leftarrow \boldsymbol{S}_0$
> ***Step 2.*** While there exists $\boldsymbol{S'} \in \boldsymbol{N}(\boldsymbol{S})$ such that $\boldsymbol{c}(\boldsymbol{S'}) \leq \boldsymbol{c}(\boldsymbol{S''})$ for all $\boldsymbol{S''} \in \boldsymbol{N}(\boldsymbol{S})$ and $\boldsymbol{c}(\boldsymbol{S'}) < \boldsymbol{c}(\boldsymbol{S})$, do:
> (a) Set $\boldsymbol{S} \leftarrow \boldsymbol{S'}$
> ***Step 3.*** Set $\boldsymbol{S^*} \leftarrow \boldsymbol{S}$ and output the best solution $\boldsymbol{S^*}$

Other advanced strategies: Variable neighborhood search (section 4.5)
Large step optimization (section 4.6)

- ***Example: TSP computations***

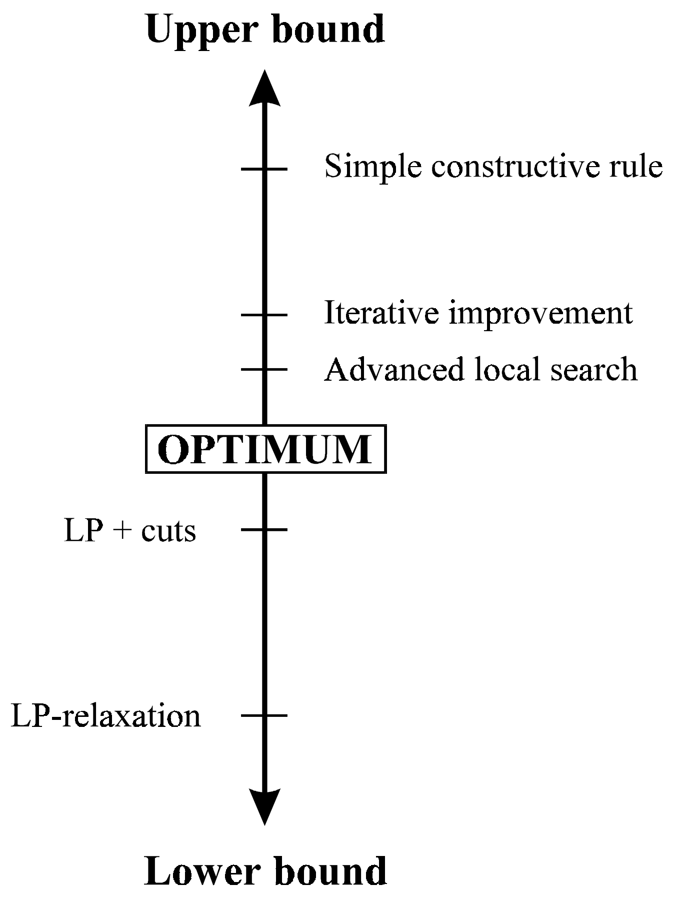
	Optimization	**Approximation**
1970's	**toy**	**hundreds**
1980's	**hundreds**	**thousands**
1990's	**thousands**	**millions**

Approximate algorithms:
- Upper bounds (feasible solutions) on optimum (minimization)

Relaxations:
- Lower bounds on optimum (minimization), for
  → measuring the quality of upper bounds (feasible solutions)
  → eliminating subsets of solutions in enumerative exact algorithms

Largest problem exactly solved (1994): 7397 cities

Figure 4: Approximation and relaxation



- ***Embedding local search***

(a) constructive rules
(b) other local search methods
(c) enumeration schemes

4. METAHEURISTICS

- ***Success of a heuristic depends on its capability of:***

(a) adapting to special instances,

(b) escaping from local optima,

(c) making use of problem structure (e.g., natural order of solution elements),

and, moreover, on the use of:

(d) efficient data structures,

(e) preprocessing,

(f) good techniques for constructing initial solutions,

(g) restarting (reinitialization) procedures,

(h) solution improvement by local search,

(i) controlled randomization,

(j) search diversification when no further improvements seem possible, and

(k) search intensification into promising regions.

- ***New classes of heuristics***

These principles and ideas lead to the development of new classes of heuristics, which considerably improved the efficiency of approximate (non-exact) algorithms and the size of solvable instances of difficult problems found in practice.

Very often based on models from nature (e.g., physics, biology, evolution).

- Multi-start
- Simulated annealing
- Tabu search
- Greedy randomized adaptive procedures (GRASP)
- Variable (dynamic) neighborhood search
- Large step optimization
- Genetic algorithms
- Asynchronous teams (A-Teams)
- Ant colonies

- ***Metaheuristics***

The process of finding a good solution (eventually the optimum) consists in applying at each step a ***subordinate heuristic*** which has to be designed ***for each particular problem***.

As an attempt to escape from local optima, some of them allow controlled deterioration of the solutions in order to diversify the search.

4.1. MULTI-START

- ***Principle***

Apply iterative improvement several times from different starting points, with randomly generated initial solutions.

- ***Template: Multi-start algorithm (minimization)***

> ***Step 1.*** Set $\boldsymbol{c}(\boldsymbol{S^*}) \leftarrow \infty$
> ***Step 2.*** Randomly generate a feasible solution $\boldsymbol{S}_0$ and set $\boldsymbol{S} \leftarrow \boldsymbol{S}_0$
> ***Step 3.*** While there exists $\boldsymbol{S'} \in \boldsymbol{N}(\boldsymbol{S})$ such that $\boldsymbol{c}(\boldsymbol{S'}) < \boldsymbol{c}(\boldsymbol{S})$ do:
> (a) Set $\boldsymbol{S} \leftarrow \boldsymbol{S'}$
> ***Step 4.*** If $\boldsymbol{c}(\boldsymbol{S}) < \boldsymbol{c}(\boldsymbol{S^*})$, then set $\boldsymbol{S^*} \leftarrow \boldsymbol{S}$
> ***Step 5.*** If another initial solution has to be investigated, go back to step 2
> ***Step 6.*** Output the best solution $\boldsymbol{S^*}$

- ***Comments***

(a) Basic step consists in the successive repetition of iterative improvement
(b) Easy to implement
(c) Randomly generated solutions may be very bad
(d) Computation times may be very large if the initial solutions are far from local optima

4.2. SIMULATED ANNEALING

- ***Analogy between statistical mechanics and combinatorial optimization***

The term ***annealing*** refers to the process of a thermal system by first melting at high temperatures and then lowering the temperature slowly, based on an annealing schedule, until the vicinity of a solidification temperature is reached, where the system is allowed to reach the ***ground state*** (the lowest energy state of the system).

Simulated annealing is a simple ***Monte Carlo approach to simulate*** the behavior of this system to achieve thermal equilibrium at a given temperature in a given annealing schedule.

Combinatorial optimization problem:

objective function $\boldsymbol{c}(\boldsymbol{S})$	↔ energy level
(feasible) solution	↔ system state
neighbor solution (move)	↔ change of state
Control parameter	↔ temperature
Best solution found	↔ frozen state

- ***Template: Simulated annealing algorithm (minimization)***

Step 1. Generate an initial solution S_0 and set S, $S^* \leftarrow S_0$
Step 2. Set the initial temperature T
Step 3. While $T > 0$ do:
Step 3.1. While (thermal equilibrium not reached) do:
(a) Randomly generate a neighbor solution $S' \in N(S)$
(b) Evaluate the change ΔE in the energy level:
$\Delta E = c(S') - c(S)$
(c) If S' improves the current solution S, i.e. $\Delta E < 0$, then set $S \leftarrow S'$
(d) Otherwise, generate random x in interval [0,1] and set $S \leftarrow S'$ with (decreasing) probability $exp(-\Delta E/K_B T)$, where K_B is a constant (i.e., if $x < exp(-\Delta E/K_B T)$ then set $S \leftarrow S'$).
(e) If $c(S') < c(S^*)$, then update the best solution setting $S^* \leftarrow S'$
Step 3.2. Decrease temperature T according to the annealing schedule
Step 4. Output the best solution S^* (lowest energy level)

- ***Typical annealing schedule*** **(steps 3, 3.1, 3.2)**

While temperature $T_k > \varepsilon$, with ε sufficiently close to zero:
(a) Perform a fixed number $h(T_k)$ of iterations (thermal equilibrium) at temperature level T_k.
(b) Update the current temperature by setting $T_{k+1} \leftarrow \beta . T_k$, with $\beta < 1$.
(c) Set $k \leftarrow k + 1$.

Notice that:
(a) For high temperatures, the probability of acceptance $exp(-\Delta E/K_B T)$ is very close to one and the algorithm behaves as a multi-start method.

(b) When the temperature goes to zero, the probability of acceptance $\boldsymbol{exp}(-\Delta \boldsymbol{E}/\boldsymbol{K_B T})$ vanishes and the algorithm behaves as iterative improvement.

(c) Controlled randomization with deterioration: when the algorithm reaches a "local optimum" (generated neighbor is worse than the current solution), it moves with high probability to a worse solution in the beginning of the search, and with very small probability at the end of the search (diversification/stabilization).

- ***Comments***

Contrarily to iterative improvement, may also take worse neighbor, with decreasing probability going to zero when the temperature vanishes (i.e., when the number of iterations increase).

(a) Convergence proof (depending on the annealing schedule)
(b) Robust approach
(c) Easy to implement
(d) Large computation times

- ***Parallelization***

(a) Different processors proceed with annealing using different streams of random numbers, until the temperature is about to be reduced. The best current solution among all processors is chosen and all processors restart at the new temperature from this same solution.
→ Significantly different chains of solutions at high temperatures.
→ When the temperature is low and there is less flexibility, it is likely that the processors will end up with solutions which are close in terms of their cost and neighborhood structure.

(b) Different processors generate different random neighbors and test them for acceptance independently, at each iteration. Once a processor accepts a neighbor, it is broadcasted to all other processors and the search moves to the neighborhood of this new current solution.

→ Wasteful at high temperatures, when almost all neighbors are accepted.
→ Conversely, should speed up the search at low temperatures, when the majority of solutions are rejected.

(c) Start with strategy (a) until the ratio of rejections exceeds a certain level and then switch to strategy (b).

(d) Hold the current best solution in common memory and allow all processors to act on it independently, each of them generating a neighbor and updating the solution if appropriate.