# Heuristics for Combinatorial Optimization

Luigi De Giovanni

Dipartimento di Matematica, Università di Padova

Notice: lecture notes cover up to slide 20

# Exact and heuristic methods

- **Exact methods**: devised to provide a provably optimal solution
- **Heuristic methods**: provides "good" solution with *no optimality guarantee*

- Try to devise an exact approach, first!
  - search for an efficient algorithm (e.g. shortest path-like problem)
  - MILP model + MILP solver
  - exploit some special property
  - suitable (re)formulation of the problem
  - search for (scientific) literature
  - ...
- ... otherwise, heuristics! (*eurìskein* = *to find*)
  - example: optimal transportation-network configuration ("hard" congestion models)
  - limited available time

# When do we use heuristics?

- Sometime cannot be used, since an optimal solution is mandatory!
- NP-hard problem $\not\Rightarrow$ heuristics! (e.g., MILP solver are now able to solve some of them!)
- Use of heuristic to provide a "good" solution in a "reasonable" amount of time. Some appropriate cases:
  - limited amount of time to provide a solution (running time)
  - limited amount of time to develop a solution algorithm
  - just estimates of the problem parameters are available
  - quick scenario evaluation in interactive Decision Support Systems
  - *real time* system

# One (among many) possible classification

Specific heuristics

- exploits special features of the problem at hand
- may encode the current “manual” solution, good practice
- may be “the first reasonable algorithm come to our mind”

General heuristic approaches

- constructive heuristics
- meta-heuristics (algorithmic schemes)
- approximation algorithms
- iper-heuristics
- ...

*C. Blum and A. Roli, “Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison”, ACM Computer Surveys 35:3, 2003 (p. 268-308)*

*K. S orensen, “Metaheuristics – the metaphor exposed”, International Transactions in Operational Research (22), 2015 (p. 3-18)*

# Constructive heuristics

- Build a solution incrementally selecting a subset of alternatives
- Expansion criterion (no backtracking)

**Greedy algorithms** (strictly local optimality in the expansion criterion)

Initialize solution $S$;
While (there are choice to make)
    add to $S$ the *most convenient* element [1]

- Widespread use: simulate practice; simple implementation; small running times ($\sim$ linear); embedded as sub-procedure.
- Sorting elements by **Dispatching rules**: static or dynamic scores
- Randomization (randomized scores, random among the best $n$ etc.)
- Primal / dual heuristics

[1] Taking feasibility constraints into account, e.g., by excluding elements that make the solution unfeasible

# Example: greedy algorithm KP/0-1

*Item $j$ with $w_j$ and $p_j$; capacity $W$; select items maximizing profit!*

1. Sort object according to ascending $\frac{p_j}{w_j}$.
2. Initialize: $S := \emptyset$, $\bar{W} := W$, $z := 0$
3. **for** $j = 1, \ldots, n$ **do**
4. **if** $(w_j \leq \bar{W})$ **then**
5. $S := S \cup \{j\}$, $\bar{W} := \bar{W} - w_j$, $z := z + p_j$.
6. **endif**
7. **endfor**

- Static dispatching rule

# Example: Greedy algorithm for the Set Covering Problem

*SCP: given set $M$ and $\mathcal{M} \subset 2^M$, $c_j$, $j \in \mathcal{M}$;*
*select a min cost combination of subsets in $\mathcal{M}$ whose union is $M$*

1. Initialize: $S := \emptyset$, $\bar{M} := \emptyset$ , $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. compute the set $j \notin S$ minimizing the ratio $\dfrac{c_j}{\displaystyle\sum_{i \in M \setminus \bar{M}} a_{ij}}$;
4. set $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i : a_{ij} = 1\}$, $z := z + c_j$ and go to 2.

- Dynamic dispatching rule

# Algorithms embedding exact solution methods

- Expansion criterion based on solving a sub-problem to optimality (once or at each expansion)
- Example: best (optimal!) element to add by MILP
- normally longer running times but better final solution
- "Less greedy": solving the sub-problem involves all (remaining) decisions variables (global optimality)

# Algorithm for SCP

$$\min \sum_{j \in \mathcal{M}} c_j x_j$$
$$s.t. \sum_{j \in \mathcal{M}} a_{ij} x_j \geq 1 \quad \forall\, i \in M$$
$$x_j \in \{0, 1\} \quad \forall\, j \in \mathcal{M}$$

1. Initialize: $S := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. se $\bar{M} = M$ ($\Leftrightarrow$ tall elements are covered), STOP;
3. solve *linear programming relaxation* of SCP (with $x_j = 1$ ($j \in S$), and let $x^*$ be the corresponding optimal solution;
4. let $j = \arg\max_{j \notin S} x_j^*$;
5. set $S := S \cup \{j\}$, $\bar{M} := \bar{M} \cup \{i : a_{ij} = 1\}$, $z := z + c_j$ and go to 2.

# Simplifying exact procedures: some examples

- Run Cplex on a MILP model for a limited amount of time
- simplify an enumeration scheme (select only a limited subset of alternatives)

  **Beam search**
- partial breath-first visit ot the enumeration tree

  compute a score for each node (likelihood it leads to an optimal leave)

  at each level select the $k$ best-score nodes and branch them
- let: $n$ levels, $b$ branches per node, $k$ beam size

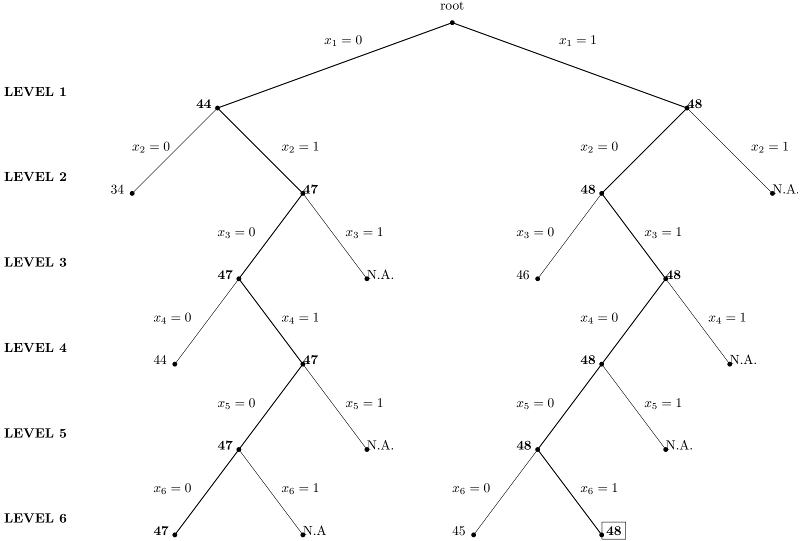
  $n \cdot k$ nodes in the final tree

  $n \cdot b \cdot k$ score evaluations

  calibrate $k$ so that specific time limits are met
- variant (with some backtrack): recovery beam search

# Beam search for KP-0/1

$n = 6$ items; binary branchining ($b = 2$); $k = 2$; greedy evaluation of nodes

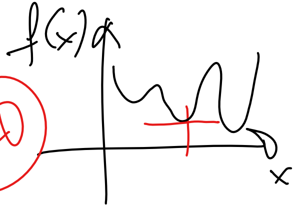


# Neighbourhood Search and Local Search

Neghbourhood of a solution $s \in X$ is $N : s \to N(s)$, $N(s) \subseteq X$

Basic LS scheme:

1. Determine an initial solution $x$;
2. **while** $(\exists\, x' \in N(x) : f(x') < f(x))$ **do** {
3. $\qquad x := x'$
4. }
5. **return**$(x)$ ($x$ is a **local optimum***)




* WITH RESPECT TO f and N

# LS components

- a method to find an **initial solution**;
- a **solution representation**, which is the base for the following elements;
- the application that, starting from a solution, generates the **neighbourhood** (moves);
- the function that **evaluates** solutions;
- a neighbourhood **exploration strategy**.

# Initial solution

- random
- from current practice
- (fast) heuristics
- randomized heuristics
- ...
- no theoretical preference: better initial solutions may lead to worst local optima
- random or randomized + multistart

# Initial solution

- random
- from current practice
- (fast) heuristics
- randomized heuristics
- ...
- no theoretical preference: better initial solutions may lead to worst local optima
- random or randomized + multistart

# Solution representation

- **Encodes** the features of the solutions
- Very important: impact on the following design steps (related to how we imagine the solutions and the solution space to be explored!)
- Example: KP-0/1

  list of loaded items

  characteristic (binary) vector

  ordered item sequence
- **Decoding** may be needed
- Example: KP-0/1

  list and vector representation: immediate decoding

  ordered sequence: a solution is derived by loading items in the given order up to saturating the knapsack

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood (moves)

*Neighbour solutions* by *moves* that perturb $x$ (neighbourhood *centre*)

Example KP/0-1: (i) insertion; (ii) swap one in/out; (iii) ...

- **Neighbourhood size**: number of neighbour solutions
- **Evaluation complexity**: should be quick! possibly incremental evaluation
- **Neighbourhood complexity**: time to explore (evaluate) all the neighbour solutions of a the current one (efficiency!)
- **Neighbourhood strength**: ability to produce good local optima (notice: local optima depend also on the neighbourhood definition)

  little perturbations, small size, fast evaluation, less strong .vs. large perturbation, large size, slow evaluation, larger improving power
- **Connection** feature is desirable

# Neighbourhood: KP/0-1 example

- Insertion neighbourhood has $O(n)$ size; Swap neigh. has $O(n^2)$ size
- A stronger neigh. by allowing also double-swap moves, size $O(n^4)$
- An insertion or a swap move can be incrementally evaluated in $O(1)$
- Overall neigh. complexity: insertion $O(n)$, swap $O(n^2)$
- Insertion neigh. or Swap neigh. are not connected.
  Insertion+removing neigh. is connected

# Neighbourhood definition: solution representation is important!

- Insertion, swapping, removing moves are based on list or vector representation!
- Difficult to implement (and imagine) them on the ordered-sequence representation
- For the ordered-sequence representation, moves that perturb the order are more natural. e.g.swapping position:
  - from $1-2-3-4-5-6-7$ to $1-6-3-4-5-2-7$ (swap 2 and 6) or $5-2-3-4-1-6-7$ (swap 1 and 5)
  - size is $O(n^2)$, connected (with respect to maximal solutions)
  - neigh. evaluation in $O(n)$ (no fully-incremental evaluation)
  - overall complexity $O(n^3)$

# Solution evaluation function

$\hat{f} = \alpha \sum_{i \in X} p_i + \gamma \left(W - \sum_{i \in X} w_i\right)$ $\alpha, \gamma > 0$

- Evaluation is used to compare neighbours to each other and select the best one
- Normally, the objective function
- May include some extra-feature (e.g. weighted sum) (e.g. $\hat{f}$ for KP)
- May include penalty terms (e.g. infeasibility level)
  - In KP/0-1, let $X$ be the subset of loaded items
  - $\tilde{f}(X) = \alpha \sum_{i \in X} p_i - \beta \max\left\{0, \sum_{i \in X} w_i - W\right\}$ $\quad (\alpha, \beta > 0)$
  - activate "removing" move in a connected "insertion+removing" neighbourhood (BOTH $\hat{f}$ and $\tilde{f}$)

# Exploration strategies

Which improving neighbour solution to select?

- **Stepest descent** strategy: the best neighbour (all evaluated!)
- **First improvement** strategy: the first improving neighbour. Sorting matters! (heuristic, random)

Possible variants:

- **random** choice among the best $k$ neighbours
- **store** interesting second-best neighbours and use them as recovery starting points for LS

# Sample application to TSP

- First question: is LS justified? Exact approaches exists, not suitable for large instances and small running times. Notice that TSP is NP-Hard
- Notation and assumptions:

  $G = (V, A)$ (undirected)

  $G$ is complete

  $|V| = n$

  cost $c_{ij}$ (may be $= c_{ji}$ in the symmetric case)

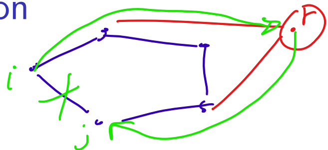
- Define all the elements of LS

# LS for TSP: initial solution by Nearest Neighbour heuristic

1. select node $i_0 \in V$; $cost = 0$, $Cycle = \{i_0\}$, $i = i_0$.
2. select $j = \arg \min_{j \in V \setminus Cycle} \{c_{ij}\}$
3. set $Cycle = Cycle \cup \{j\}$; $cost = cost + c_{ij}$
4. set $i = j$
5. if still nodes to be visited, go to 2
6. $Cycle = Cycle \cup \{i_0\}$; $cost = cost + c_{ii_0}$

- $O(n^2)$ (or better): simple but not effective (too greedy, last choices are critical)
- repeat with different $i_0$
- randomize Step 2

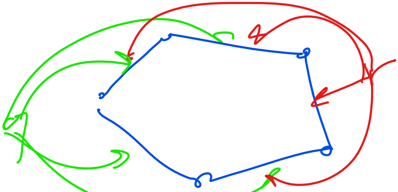
# LS for TSP: Nearest/Farthest Insertion




1. Choose the nearest/farthest nodes $i$ and $j$: $C = i - j - i$, $cost = c_{ij} + c_{ji}$
2. select the node $r = \arg\min_{i \in V \setminus C} / \max_{i \in V \setminus C} \{c_{ij} : j \in C\}$
3. modify $C$ by inserting $r$ between nodes $i$ and $j$ minimizing $c_{ir} + c_{rj} - c_{ij}$
4. if still nodes to be visited, go to 2.

- $O(n^3)$: rather effective (farthest version better, more balanced cycles)
- may randomize initial pair and/or $r$ selection

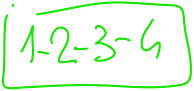
# LS for TSP: Best Insertion

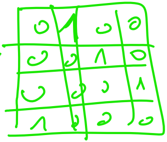


1. Choose the nearest nodes $i$ and $j$: $C = i - j - i$, $cost = c_{ij} + c_{ji}$
2. select the node $r = \arg\min_{i \in V \setminus C} \{c_{ir} + c_{rj} - c_{ij} : i, j \text{ consecutive in } C\}$
3. modify $C$ by inserting $r$ between nodes $i$ and $j$ minimizing $c_{ir} + c_{rj} - c_{ij}$
4. if still nodes to be visited, go to 2.

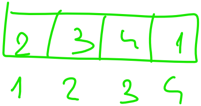
- $O(n^3)$: rather effective (lest than farthest/nearest insertion)
- may randomize initial pair and/or $r$ selection

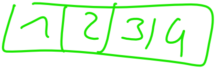
# LS for TSP: Solution Representation



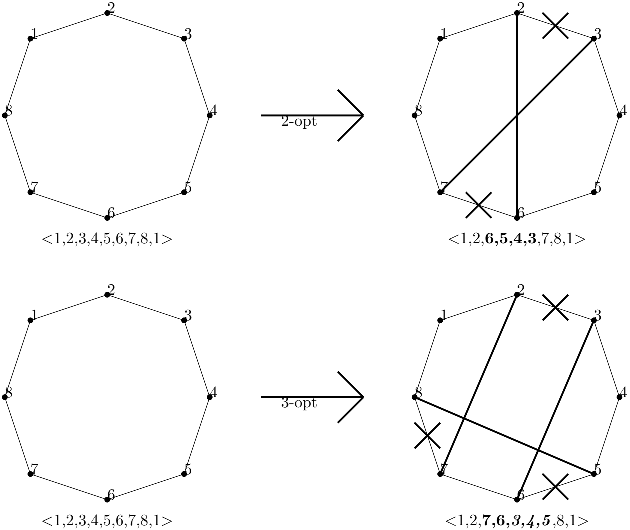

- **arc representation**: arcs in the solution, e.g. as a binary adjacency matrix
- **adjacency representation**: a vector of $n$ elements between 1 and $n$ (representing nodes), $v[i]$ reports the node to be visited after node $i$
- **path representation**: ordered sequence of the $n$ nodes (a solution is a node permutation!)



# LS for TSP: *k*-opt neighbourhoods

Concept: replace *k* arcs in with *k* arcs out [Lin and Kernighan, 1973]



# LS for TSP: $k$-opt neighbourhoods

- In terms of path representation, 2-opt is a substring reversal
- Example: $< 1, 2, 3, 4, 5, 6, 7, 8, 1 > \longrightarrow < 1, 2, 6, 5, 4, 3, 7, 8, 1 >$
- 2-opt size: $\frac{(n-1)(n-2)}{2} = O(n^2)$
- $k$-opt size: $O(n^k)$
- Neighbour evaluation: incremental for the symmetric case, $O(1)$
- 2-opt move evaluation: reversing sequence between $i$ and $j$ in the sequence $< 1 \dots h, i, \dots, j, l, \dots, 1 >$

$$C_{new} = C_{old} - c_{hi} - c_{jl} + c_{hj} + c_{il}$$

- which $k$? $k = 2$ good, $k = 3$ fair improvement, $k = 4$ little improvement

# LS for TSP: evaluation function and exploration strategy

No specific reason to adopt special choices:

- Neighbours evaluated by the objective function (cost of the related cycle)
- Steepest descent (or first improvement)

# Neighbourhood search and Trajectory methods

- LS trades-off simplicity/efficiency and effectiveness, but it gets stuck in local optima
- Need to escape from local optima (only convexity implies global optimality)

- Random multistart (random initial solutions)
- Variable neighbourhood (change neighbourhood if local optimum)
- Randomized exploration strategy (e.g. random among best $k$ neigh)
- Backtrack (memory and recovery of unexplored promising neighbours)
- ...

- *Neighbourhood search* or *Trajectory methods*: a walk trough the solution space, recording the best visited solution

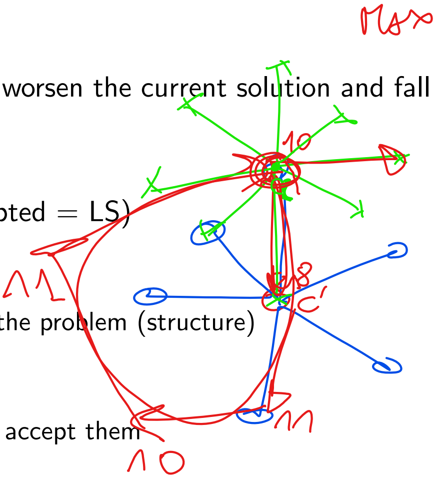
# Neighbourhood search and Trajectory methods

- LS trades-off simplicity/efficiency and effectiveness, but it gets stuck in local optima
- Need to escape from local optima (only convexity implies global optimality)
  - Random multistart (random initial solutions)
  - Variable neighbourhood (change neighbourhood if local optimum)
  - Randomized exploration strategy (e.g. random among best $k$ neigh)
  - Backtrack (memory and recovery of unexplored promising neighbours)
  - ...
- *Neighbourhood search* or *Trajectory methods*: a walk trough the solution space, recording the best visited solution

# Avoiding loops

- A walk escaping local optima may worsen the current solution and fall into loops
- In order to avoid loops:

- (only improving solutions are accepted = LS)
- randomized exploration
  - alternative random ways
  - does not exploit information on the problem (structure)
  - e.g. Simulated Annealing
- memory of visited solutions
  - store visited solution and do not accept them
  - structure can be exploited
  - e.g. Tabu Search



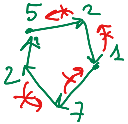
- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour

# Avoiding loops

- A walk escaping local optima may worsen the current solution and fall into loops*
- In order to avoid loops:

* ① You may come back to a local optimum (loop of length 2)

② if we avoid ①, we may have (M|H|)




- (only improving solutions are accepted = LS)
- randomized exploration
  - alternative random ways
  - does not exploit information on the problem (structure)
  - e.g. Simulated Annealing
- memory of visited solutions
  - store visited solution and do not accept them
  - structure can be exploited
  - e.g. Tabu Search

- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour

# Simulated Annealing [Kirkpatrick, 1983]

- Metaphore: annealing process of glass, metal. Alternate warming/cooling to obtain "optimal" molecular structure
- (One possible) search scheme (min problem):

Determine an initial solution $x$; $x^* \leftarrow x$ $k = 0$

**repeat**

  $k \leftarrow k + 1$

  generate a (random) neighbour $y$

  **if** $y$ is better **than** $x^*$, then $x^* \leftarrow y$

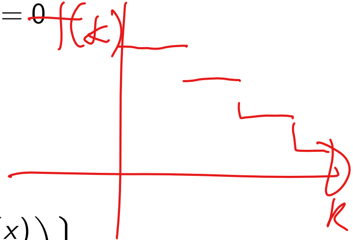
  compute $p = \min \left\{ 1, exp\left( -\frac{f(y) - f(x)}{T(k)} \right) \right\}$

  **accept** *y with* **probability** *p*

  **if** accepted, $x \leftarrow y$

**until** (no further neighbours of $x$, or max trials)

**return** $x^*$



# SA: cooling schedule

- Parameter $T(k)$: temperature, *cooling schedule*
- $T(\mathit{first}) > T(\mathit{last})$
- Example of cooling schedule:

- initial $T$ (maximum)
- number of iterations at constant $T$
- $T$ decrement
- minimum $T$

+ (one of) the first NS metaphors
+ provably converges to the global optimum (under strong assumptions)
+ simple to implement

— there are better (on-the-field) NS metaheuristics!

# Tabu Search [Fred Glover, 1989]

- **Memory** is used to avoid cycling: store *information on visited solutions* (allows exploiting structure of the problem)
- Basic idea: store visited solutions and **exclude them (= make tabu)** from neighbourhoods
- Implementation by storing **Tabu List** of the **last** $t$ **solutions**

$$T(k) := \{x^{k-1}, x^{k-2}, \ldots, x^{k-t}\}$$

  at iteration $k$, avoid cycles of length $\leq t$
- $t$ is a parameter to be calibrated
- From $N(x)$ to $N(x, k)$

# Storing “information” instead of solutions

- Tabu List (may) store *information* on the last $t$ solutions
- E.g., often *moves* are stored instead of solutions because of
  - *efficiency* (checking equality between full solutions may take long time and slow down the search)
  - *storage* capacity (storing full solution information may take large memory)
- Example: TSP, 2-opt. TL stores the last $t$ pairs of arcs added (to avoid arcs or involved nodes)
- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour (recall $N(x,k) \neq N(x,l)$)
- $t$ (tabu tenior) has to be calibrated:
  - too small: TS may cycle
  - too large: too many tabu neighbours

# Storing "information" instead of solutions

- Tabu List (may) store *information* on the last $t$ solutions
- E.g., often *moves* are stored instead of solutions because of
  - *efficiency* (checking equality between full solutions may take long time and slow down the search)
  - *storage* capacity (storing full solution information may take large memory)
- Example: TSP, 2-opt. TL stores the last $t$ pairs of arcs added (to avoid arcs or involved nodes)
- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour (recall $N(x, k) \neq N(x, l)$)
- $t$ (tabu tenior) has to be calibrated:
  - too small: TS may cycle
  - too large: too many tabu neighbours

# Storing "information" instead of solutions

- Tabu List (may) store *information* on the last $t$ solutions
- E.g., often *moves* are stored instead of solutions because of
  - *efficiency* (checking equality between full solutions may take long time and slow down the search)
  - *storage* capacity (storing full solution information may take large memory)
- Example: TSP, 2-opt. TL stores the last $t$ pairs of arcs added (to avoid arcs or involved nodes)
- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour (recall $N(x, k) \neq N(x, l)$)
- $t$ (tabu tenior) has to be calibrated:
  - too small: TS may cycle
  - too large: too many tabu neighbours

# Aspiration criteria

- By storing “information”, unvisited solutions may be declared as tabu
- If a tabu neighbour solution satisfies one or more **aspiration criteria**, tabu list is *overruled*
- Aspiration criterion: a solution is “interesting”, e.g. the solution is the best found so far (not visited before!)

# Stopping criteria

- (A solution is found satisfying an optimality certificate, if available...)
- Maximum number of iterations, or time limit
- Maximum number of NOT IMPROVING iterations
- Empty neighbourhood and no overruling
  - perhaps $t$ is too long
  - perhaps visit non-feasible solutions (e.g. COP with many constraints): modifying the evaluation function, alternate dual and primal search

# TS basic scheme

Determine an **initial** solution $x$; $k := 0$, $T(k) = \emptyset$, $x^* = x$;

**repeat**

let $y = \arg\text{best}\big(\{\tilde{f}(y), y \in N(x,k)\} \cup$
$\{y \in N(x) \setminus N(x,k) \mid y$ satisfies aspiration$\}\big)$

compute $T(k+1)$ from $T(k)$ by inserting $y$ (or move $x \mapsto y$, or information) and, if $|T(k)| \geq t$, removing the elder solution (or move or information)

if $f(y)$ improves $f(x^*)$, let $x^* := y$;

$x = y$, $k{+}{+}$

**until** (stopping criteria)

**return** $(x^*)$.

Same basic elements as LS (+ tabu list, aspiration, stop)

# TS basic scheme

Determine an **initial** solution $x$; $k := 0$, $T(k) = \emptyset$, $x^* = x$;

**repeat**

let $y = \arg\text{best}\big(\{\tilde{f}(y), y \in N(x,k)\} \cup$
$\{y \in N(x) \setminus N(x,k) \mid y \text{ satisfies aspiration}\}\big)$

compute $T(k+1)$ from $T(k)$ by inserting $y$ (or move $x \mapsto y$, or information) and, if $|T(k)| \geq t$, removing the elder solution (or move or information)

if $f(y)$ improves $f(x^*)$, let $x^* := y$;

$x = y$, $k{+}{+}$

**until** (stopping criteria)

**return** $(x^*)$.

Same basic elements as LS (+ tabu list, aspiration, stop)

# Intesification and diversification phases

- **Intensification** explores more solutions in a small portion of the solution space: solutions with similar features
- **Diversification** moves the search towards unexplored regions of the search space: solutions with different features
- the basic TS scheme may be improved by **alternating** intensification and diversification, to find and exploit new promising regions and, hence, new (and possibly better) local optima
- **memory** may play a role (store information on visited solutions, e.g. to allow avoiding the same features during diversification)

Intesification and diversification can be applied to **any** metaheuristics (not only to TS)

# Intesification and diversification phases

- **Intensification** explores more solutions in a small portion of the solution space: solutions with similar features
- **Diversification** moves the search towards unexplored regions of the search space: solutions with different features
- the basic TS scheme may be improved by **alternating** intensification and diversification, to find and exploit new promising regions and, hence, new (and possibly better) local optima
- **memory** may play a role (store information on visited solutions, e.g. to allow avoiding the same features during diversification)

Intesification and diversification can be applied to **any** metaheuristics (not only to TS)

# Intensification

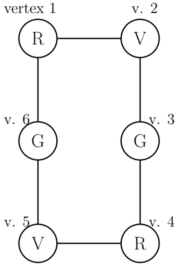
- enumerate (implicitly) all the solutions in a (small) region where good solutions have been found (e.g. fix some variables in a MILP model and run a solver)
- use a more detailed neighbourhood (e.g. allowing many possible moves)
- relax aspiration criteria
- modify evaluation function to penalize far away solutions

# Diversification

( up to VARIABLE NEIGHBOURHOOD )

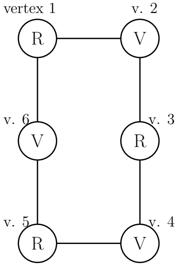
- use "larger" neighbourhoods (e.g. $k$-opt $\rightarrow$ $(k+1)$-opt in TSP, until a better solution is found)
  - if more neighbourhoods are used, they rely on independent tabu lists
- modify the evaluation function to promote far away solutions
- use the last local minimum to build a far-away ("complementary") solution to start a new intensification
- use a long term memory to store the "more visited" features and penalize them in the evaluation function
  - as a quick-and-dirty approximation, use a dynamic tabu list length $t$: $t$ is short during intensification and long during diversification (we may start with small $t = t_0$ and increment it as long as we do not find improving solutions, until a maximum $t$ is reached or an improvement resets $t = t_0$ for a new intensification)

# Example: Tabu Search for Graph Coloring




| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| R | V | G | R | V | G |

$f = 3$




| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| R | V | R | V | R | V |

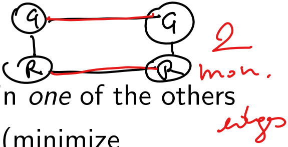
$f = 2$

- move: change the color of one node at a time (no new color). 12 neighbours: VVGRVG, GVGRVG, RRGRVG, RGGRVG, RVRRVG etc. **none feasible!**
- objective function to evaluate: little variations (**plateau!**)

$\tilde{f}$ that penalizes non-feasibilities, includes (weighted sum) other features, **but ...**

# Too many constraints: change perspective!

Given a $k$-coloring, search for a $k-1$-coloring




- Initial solution: delete *one* color by changing it in *one* of the others
- Evaluation $\tilde{f}$: number of *monochromatic edges* (minimize non-feasibilities)
- Move: as before, change the color of one vertex
- **Granular TS**: consider only nodes belonging to monochromatic edges
- Tabu list: last $t$ pairs $(v, r)$ (vertx $v$ kept color $r$)

- if $\tilde{f} = 0$, new feasible solution with $k-1$ colors: set $k = k - 1$ and start again!

# Too many constraints: change perspective!

Given a $k$-coloring, search for a $k-1$-coloring

- Initial solution: delete *one* color by changing it in *one* of the others
- Evaluation $\tilde{f}$: number of *monochromatic edges* (minimize non-feasibilities)
- Move: as before, change the color of one vertex
- **Granular TS**: consider only nodes belonging to monochromatic edges
- Tabu list: last $t$ pairs $(v, r)$ (vertx $v$ kept color $r$)

- if $\tilde{f} = 0$, new feasible solution with $k-1$ colors: set $k = k-1$ and start again!

# Population based heuristics

At each iteration

- a set[2] of solutions (**population**) is maintained
- some solutions are recombined[3] to obtain new solutions (among which a better one, hopefully)

Several paradigms (often just the metaphor changes!)

- Evolutionary Computation (Genetic algorithms)
- Scatter Search and path relinking
- Ant Colony Optmization
- Swarm Optmization
- etc.

General purpose (soft computing) and easy to implement (more than effective!)

[2] In trajectory/based metaheuristics, a single

[3] In trajectory/based metaheuristics, perturbation, move

# Population based heuristics

At each iteration

- a set[2] of solutions (**population**) is maintained
- some solutions are recombined[3] to obtain new solutions (among which a better one, hopefully)

Several paradigms (often just the metaphor changes!)

- Evolutionary Computation (Genetic algorithms)
- Scatter Search and path relinking
- Ant Colony Optmization
- Swarm Optmization
- etc.

General purpose (soft computing) and easy to implement (more than effective!)

[2] In trajectory/based metaheuristics, a single

[3] In trajectory/based metaheuristics, perturbation, move

# Population based heuristics

At each iteration

- a set[2] of solutions (**population**) is maintained
- some solutions are recombined[3] to obtain new solutions (among which a better one, hopefully)

Several paradigms (often just the metaphor changes!)

- Evolutionary Computation (Genetic algorithms)
- Scatter Search and path relinking
- Ant Colony Optmization
- Swarm Optmization
- etc.

General purpose (soft computing) and easy to implement (more than effective!)

---

[2] In trajectory/based metaheuristics, a single

[3] In trajectory/based metaheuristics, perturbation, move

# Genetic Algorithms [Hollande, 1975]

| | | |
|---|---|---|
| *Survival of the fittest* (evolution) | $\leftrightsquigarrow$ | Optimization |
| Individual | $\leftrightsquigarrow$ | Solution |
| Fitness | $\leftrightsquigarrow$ | Objective function |

*Encode* solutions *of the specific problem*.

Create an initial set of solutions (*initial population**).

**Repeat**

*Select** pairs (or groups) of solutions (parent).

*Recombine** parents to generate new solutions (offspring).

Evaluate the *fitness** of the new solutions

*Replace** the population, using the new solutions.

**Until** (*stopping criterion*)

**Return** the best generated solution.

*** Genetic Operators**

# Encoding: *chromosome*, sequence of *genes*

- KP 0/1: binary vector, $n$ genes = 0 / 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

- TSP: path representation: $n$ genes = cities

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 6 | 1 | 8 | 0 | 4 | 7 | 1 | 5 |

- Normally, each gene is related to one of the decision variables of the Combinatorial Optimization Problem (COP)
- Encoding is important and affect following design steps (like solution representation in neighbourhood search)
- **Decoding** to transform a chromosome (or individual) into a solution of the COP (in the cases above it is straightforward)

# Encoding: *chromosome*, sequence of *genes*

- KP 0/1: binary vector, $n$ genes = 0 / 1

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 |

- TSP: path representation: $n$ genes = cities

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 2 | 6 | 1 | 8 | 0 | 4 | 7 | 1 | 5 |

- Normally, each gene is related to one of the decision variables of the Combinatorial Optimization Problem (COP)

- Encoding is important and affect following design steps (like solution representation in neighbourhood search)

- **Decoding** to transform a chromosome (or individual) into a solution of the COP (in the cases above it is straightforward)

# Encoding: *chromosome*, sequence of *genes*

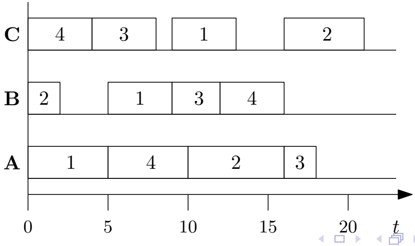
- Job shop scheduling: $n * m$ genes = jobs (decoding!!!)

| Job | machine , $t_{ij}$ | | |
|---|---|---|---|
| 1 | A , 5 | B , 4 | C , 4 |
| 2 | B , 2 | A , 6 | C , 5 |
| 3 | C , 4 | B , 2 | A , 2 |
| 4 | C , 4 | A , 5 | B , 4 |

Encoding:

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 2 | 1 | 1 | 3 | 4 | 2 | 3 | 1 | 2 | 3 | 4 |

Decoding:



# Genetic operators

- **Initial population**: random + *some* heuristic/local search
  - random $\rightarrow$ diversification (very important!!!)
  - heuristic (randomized) $\rightarrow$ faster convergence (not too many heuristic solutions, otherwise fast convergence to local optimum)
- **Fitness**: (variants of the) objective function (see Neighbourhood Search)

# Genetic operators: Selection

- **Selection**: larger fitness $\rightsquigarrow$ larger *probability* to be selected
- Notice: even worse individual should be selected with small probability to (*avoid premature convergence!*): they may contain good features (genes), even if their overall fitness is poor
- Mode 1: select one $t$-uple of individuals to be combined at a time
- Mode 2: select a subset of individuals to form a *mating pool*, and combine all the individual in the mating pool.

# Genetic operators: Selection schemes

- $p_i$: probability of selecting individual $i$; $f_i$: fitness of $i$

In general, compute $p_i$ such that the higher $f_i$, the higher $p_i$

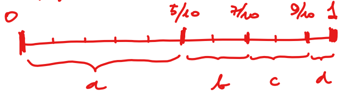
- **Montecarlo**: $p_i$ is proportional to $f_i$

$$p_i = f_i \,/\, \sum_{k=1}^{N} f_k$$

$f_i$: fitness of $i$

Super-individuals may be selected too often

IMPLEMENTATION: roulette wheel
$f_a = 5$, $f_b = 2$, $f_c = 2$, $f_d = 1$




- **Linear ranking**: sort individual by increasing fitness and $\sigma_i$ is the position of $i$, set $p_i = \dfrac{2\sigma_i}{N(N+1)}$

IMPLEMENTATION: roulette wheel that uses the rank (position) instead of the fitness.

- $n$-**tournament**: select a small subset of individuals uniformly in the population, then select the best individual in the subset

# Genetic operators: recombination [crossover]

- From $n \geq 1$ *parents*, obtain *m offspring* **different but similar**
- offspring inherits genes (features) from one of the parents at random

- Uniform (probability normally depends on the parent fitness)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | parent 1 (fitness 8) |
| 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | parent 2 (fitness 5) |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | offspring |

- *k*-cut-point: "adjacent genes represents correlated features"

| | | | cut point | | | | | cut point | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| * | * | * | | * | * | * | * | * | | * | * | parent 1 |
| + | + | + | | + | + | + | + | + | | + | + | parent 2 |
| * | * | * | | + | + | + | + | + | | * | * | offspring 1 |
| + | + | + | | * | * | * | * | * | | + | + | offspring 2 |

# Mutation

After or during crossover, some genes are randomly changed

- Against *genetic drift*: **one** gene takes the same value in all the individuals of the population (loss of genetic diversity)
- Effects and side effects (sometimes we want them!):
  - (re)introduce genetic diversity
  - slow population convergence (normally we change very few genes with very small probability)
  - can be used to obtain diversification (more genes with more probability: simple way to diversify, not the best one)

# Integrating Local Search

**Local search** may be used to improve offspring (simulate children education)

- Replace an individual with the related local minimum
- May lead to premature convergence
- Efficiency may degrade!
  - simple, fast LS
  - apply to a selected subset of individuals
  - more sophisticated NS only at the end, as post-optimization

# Crossover, mutation and non-fesible offspring

Crossover/mutation operators may generate unfeasible offspring. We can:

- Reject unfeasible offspring
- Penalize (modified fitness)
- Repair (during the decoding)
- Design specific operators guaranteeing feasibility. E.g. for *TSP*:
    - **Order crossover** (similar, since reciprocal order is maintained)

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 9 | 2 | 6 | 8 | 3 | 0 | 5 | 7 | parent 1 |
| 0 | 2 | 1 | 5 | 3 | 9 | 4 | 7 | 6 | 8 | parent 2 |
| 1 | 4 | 9 | 2 | 3 | 6 | 8 | 0 | 5 | 7 | offspring 1 |
| 0 | 2 | 1 | 4 | 9 | 3 | 5 | 7 | 6 | 8 | offspring 2 |

    - Mutation by substring reversal (= 2-opt)

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 4 | **9** | 2 | 6 | **8** | 3 | 0 | 5 | 7 |
| $\longrightarrow$ | | | $\longleftarrow$ | | | | | $\longrightarrow$ | |
| 1 | 4 | **8** | 6 | 2 | **9** | 3 | 0 | 5 | 7 |

# Generational Replacement

**Generational replacement**: old individuals are replaced by offspring

- **Steady state**: a few individuals (likely the *worst* ones) are replaced
- **Elitism**: a few individuals (likely the *best* ones) are kept
- **Best individuals**: generate $R$ new individuals from $N$ old ones; keep the best $N$ among the $N + R$

**Population management**: keep the population diversified, whilst obtaining (at least one) better and better solution

- Acceptance criteria for new individuals (e.g. fitness)
- Diversity threshold (e.g. Hamming distance)
- Variable threshold to alternate *intensification* and *diversification*

# Stopping criteria

- Time limit
- Number of (not improving) iterations (=generations)
- Population convergence: all individuals are similar to each other (pathology: not well designed or calibrated)

# Observations

- Advantages: general, robust, adaptability (just an encoding and a fitness function!)
- Disadvantages: many parameters! (you may save time in developing the code but spend it in calibration)
- Overstatement: *complexity comes back to the user*, that should find the optimal combination of the parameters.
  Normally, the designer should provide the user with a method able to directly find the optimal combination of decision variables. In fact, the algorithm designer should also provide the user with the **parameter calibration**!

- Genetic algorithms are in the class of *weak methods* or *soft computing* (exploit little or no knowledge of the specific problem)

# Validating optimization algorithms

Some criteria:

- (Design and implementation time / cost)
- Efficiency (running times)
- Effectiveness (quality of the provided solutions)
- *Reliability*, if stochastic (every run provide a good solution)

Evaluation/validation techniques:

- **Computational experiments**. Steps
  - desing and implementation of the optimization algorithm
  - benchmark selection (real, literature, ad-hoc): "many" instances
  - parameter calibration (before -not during- test)
  - test (notice: multiple [e.g. 10] running if stochastic)
  - statistics (including reliability) and comparison with alternative
- Probabilistic analysis (more theoretical, e.g. probability of optimum)
- Worst case analysis (performance guarantee, often too pessimistic)

# Parameter calibration (or estimation)

- **Pre-deployment** activity (designer should do, not the user!)
- Estimation valid for *every* instance (for evaluation purposes)
- Standard technique:
  - select an instance **sub**set (= training set)
  - extensive test on the training set
  - take **interaction** among parameters into account
  - stochastic components make the calibration harder
- Advanced techniques:
  - Black box optimization
  - Automatic estimation (e.g. *i-race* package)
  - Adaptivity

# Hybrid metaheuristics: very brief introduction!

Integration between different techniques, at different levels (components, concepts, etc.). Examples:

- population based + trajectory methods (find good regions + intensification)
- tabu search + simulated annealing
- **Matheuristics** (hot research topic, thesis avaialble!)
  - mathematical programming driven constructive heuristics
  - exact methods to find the best move in large neighbourhoods
  - heuristics to help exact methods (e.g. primal and dual bounds)
  - Rounding heuristics
  - Local branching
  - ...

Warning: an algorithm is good if it provides good results (validation), and not if it is described by a suggestive metaphor. See Sörensen, 2015