Heuristics for Combinatorial Optimization

Luigi De Giovanni

Dipartimento di Matematica, Università di Padova

Exact and heuristic methods

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

- "Sometimes" exact solution is mandatory
  - theoretical requirements (e.g. slave problem in "exact" column generation)
  - case study: covering public transportation services at Deutsche Bahn
  - case study: make **all!** surgical operations you can
- Always try to devise an exact approach, first!
  - search for scientific literature
  - search for an efficient algorithm (exploit special problem property, smart reformulations etc.)
  - "MIP it": MILP model + MILP solver
  - ...
- ... otherwise, heuristics!

When do we use heuristics?

- To formulate an exact model or method is unpractical or impossibile
  - case study: configuration of transportation networks with realistic congestion models (e.g. with simulation and no analytic model)
- Need for just "good" solution using a "reasonable" resources:
  - limited amount of time to provide a solution (running time)
    - ⋆ e.g., quick scenario evaluation in interactive Decision Support Systems
    - ⋆ e.g., *real time* system
  - limited amount of computational resources (memory, CPUs, hardware)
  - limite amount of time to develop an effective solution (e.g., off-the-shelf solvers cannot effectively solve an available formulation)
  - limited amount of economic resources to develop a solution algorithm (e.g.., costs for analysers and developers) or run it (e.g., costs for solver licenses, new hw etc.)
  - just estimates of the problem parameters are available (and we do not want to deal with uncertainty using robust or stochastic optimization...)
- Warning: NP-hard problem $\not\Rightarrow$ heuristics!

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 that come to our mind"

General heuristic approaches (algorithmic "templates")

- constructive heuristics
- simplified exact procedures
- meta-heuristics (algorithmic improvement schemes)
- approximation algorithms
- hyper-heuristics
- ...

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

K. Sörensen, "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* (and feasible) element

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

Example: greedy algorithm KP/0-1

[Knapsack Problem 0/1 (KP-0/1)]

Given: Item j with w_j and p_j; capacity W;

Determine: loadable subset of items that maximizes total 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

[SetCovering Problem (SCP)]

Given: set M; set $\mathcal{M} \subset 2^M$; c_J, $J \in \mathcal{M}$;

Determine: a min cost combination of subsets in $\mathcal{M}$ whose union is M

1. Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. compute the set $J \notin \mathcal{S}$ minimizing the ratio $\dfrac{c_J}{|\, J \setminus \bar{M} \,|}$;
4. set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.

- Dynamic dispatching rule

Greedy for SCP: sorting criterion through an exact method

$$\begin{aligned} \min \sum_{J \in \mathcal{M}} & c_J x_J \\ s.t. \sum_{J \in \mathcal{M}: i \in J} & x_J \quad \geq 1 \quad \forall\, i \in M \\ & x_J \quad \in \{0,1\} \quad \forall\, J \in \mathcal{M} \end{aligned}$$

1. Initialize: $\mathcal{S} := \emptyset$, $\bar{M} := \emptyset$, $z := 0$
2. if $\bar{M} = M$ ($\Leftrightarrow$ all elements are covered), STOP;
3. solve *linear programming relaxation* of SCP (with $x_J = 1$ $(J \in \mathcal{S})$, and let x^* be the corresponding optimal solution;
4. let $J = \arg\max_{J \notin \mathcal{S}} x^*_J$;
5. set $\mathcal{S} := \mathcal{S} \cup \{J\}$, $\bar{M} := \bar{M} \cup J$, $z := z + c_J$ and go to 2.

Algorithms embedding exact solution methods: in general

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

Remark: having a mathematical model is useful, even if the model does not directly solve the problem

Randomized constructive heuristics

- The expansion criterion can be randomized
  - random swap of consecutive elements in a static sorting
  - random choice (uniform or weighted) among the next k candidate elements
  - adding a random component to the score
  - etc.
- Can be iterated to obtain different solutions (e.g. up to a time limit): "easy" way to improve over the first solution

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, e.g. *Beam Search*)

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 on them
- Let: n levels, b branches per node, k beam size

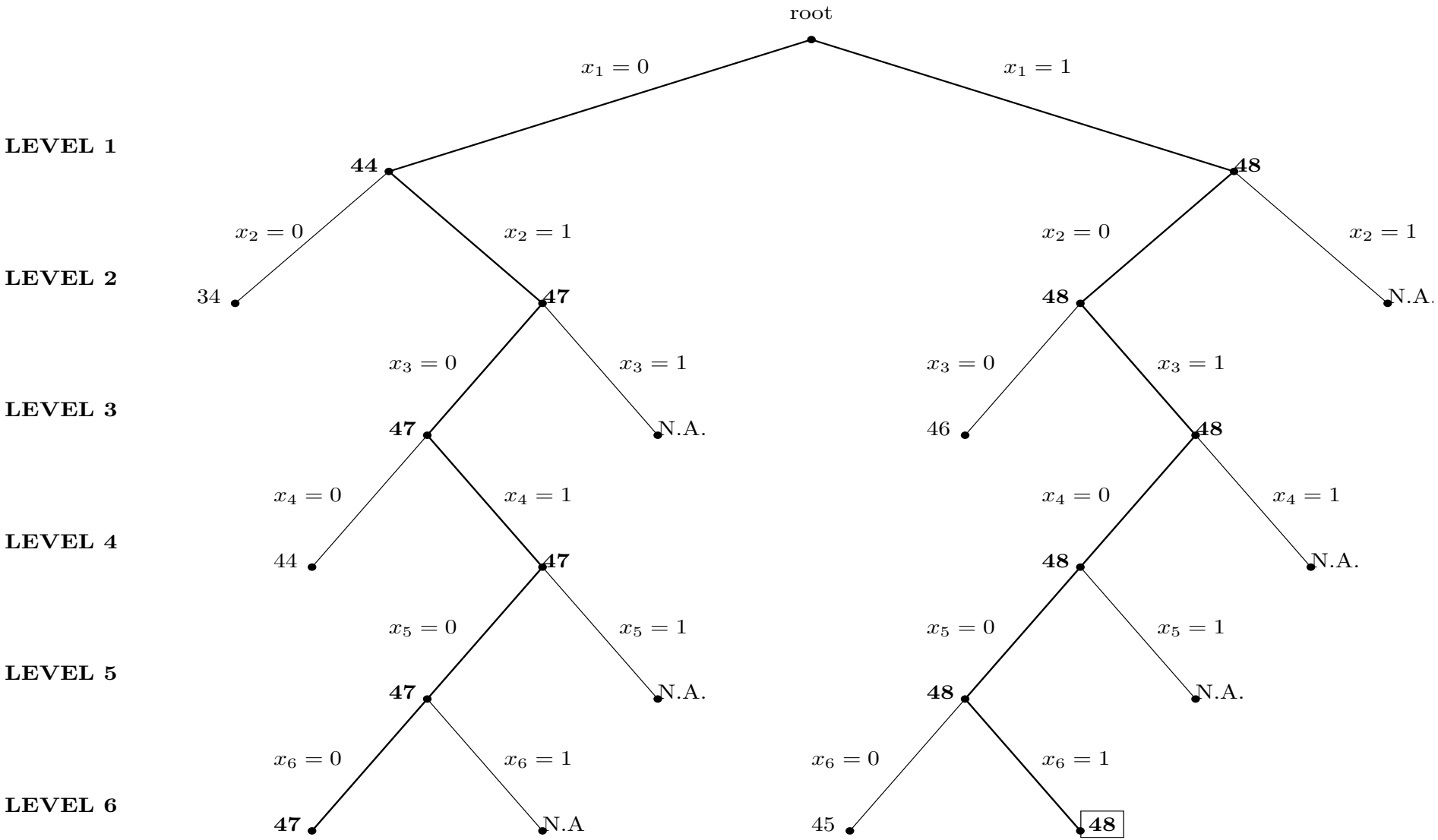
  $n \cdot k$ nodes in the final tree

  $n \cdot b \cdot k$ score evaluations
- Let: t time for single node evaluation, T overall time limite

  tune $k = \dfrac{T}{n \cdot b \cdot t}$
- Variant (with some backtrack): recovery beam search

From tree-search or B&B to Beam search for KP-0/1

$n = 6$ items; binary branching ($b = 2$); $k = 2$;
(any reasonable) scoring of nodes (e.g., LP relaxation or greedy on "residual" problem)



Neighbourhood Search and Local Search

How to improve over a solution?

- (continuous solution space and suitable objective functions: gradient!)
- combinatorial optimization: "look around", *neighborhood*!
- a neighbor identifies a search direction, try 'many' directions to find an improving one!

Let X be a (discrete) set of feasible solutions, and consider $\min_{x \in X} f(x)$:

a **neighbourhood** of a solution $s \in X$ is a function $N : s \to N(s)$ that identifies a subset $N(s) \subseteq X$

Remark:

- $N(s)$ obtained by *sistematically* applying *small* changes to s.
- A change is also called *move*: we move from s to a *neighbor solution*
- The move also identifies the applied rule, i.e., the neighborhood function

Neighbourhood: an example

KP0/1, items $i(p_i, w_i)$: a(3,4), b(4,5), c(5,4), d(3,3), e(8,9), f(4,7), W=20

$\sigma = \{a, b, d\}$ $\quad obj(\sigma) = 10$

$N(\sigma) = \{\tau \subseteq X \mid \tau = \sigma + i, i \in X \setminus \sigma$ or $\tau = \sigma - i, i \in \sigma$ (insert/remove)$\}$

$\tau_1 = \{b, d\}$	$obj(\tau_1) = 7$	$\tau_4 = \{a, b, c, d\}$	$obj(\tau_4) = 15$
$\tau_2 = \{a, d\}$	$obj(\tau_2) = 6$	$\tau_5 = \{a, b, d, e\}$	infeasible
$\tau_3 = \{a, b\}$	$obj(\tau_3) = 7$	$\tau_6 = \{a, b, d, f\}$	$obj(\tau_6) = 14$

Improving directions: from σ to τ_4 and from σ to τ_6

$N'(\sigma) = \{\tau \subseteq X \mid \tau = \sigma + i - j, i \in X, j \in \sigma$ (swap)$\}$

$\tau_1 = \{c, b, d\}$	$obj(\tau_1) = 12$	$\tau_7 = \{a, b, c\}$	$f(\tau_7) = 12$
$\tau_2 = \{e, b, d\}$	$obj(\tau_2) = 15$	$\tau_8 = \{a, b, e\}$	$f(\tau_7) = 15$
$\tau_3 = \{f, b, d\}$	$obj(\tau_3) = 11$	$\tau_9 = \{a, b, f\}$	$f(\tau_6) = 11$
$\tau_4 = \{a, c, d\}$	$obj(\tau_4) = 11$		
$\tau_5 = \{a, e, d\}$	$obj(\tau_5) = 14$	improving directions:	
$\tau_6 = \{a, f, d\}$	$obj(\tau_6) = 10$	all but τ_6	

Basic Local Search (LS) scheme

Determine an initial solution x

Define a neighborhood function N in the space X of solutions

while $(\exists\, x' \in N(x) : f(x') < f(x))$ **do** $\{$

$\quad x := x'$

$\}$

return(x) (x is a **local optimum***)

***Notice:** "combinatorial (local) convexity" depends on x, f **and** on $N(x)$

LS components

- a method to find an **initial solution**;
- a **solution representation**, i.e., a formalization of the solution space X: this 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 choice a solution
- 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 (≈*formulation* of the problem)
- 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)

A *neighborhood function* $N : x \rightarrow N(x)$ defines the *elements* of a solution x and a modifying action (or *move*) that perturbs x

Given a solution x (neighbourhood *centre*), we apply a **move** to each **element** of x and we obtain a set of *neighbour solutions* (neighborhood)

Example: add/remove neighborhood for KP-0/1:

- solution x: a subset S of items
- solution element: an item i
- add/remove move: if $i \in S$, remove it from S, f $i \notin S$, add it to from S,
- neighborhood: all the items subsets obtained from S by adding/removing one item

Example: 2-opt neighborhood for TSP:

- solution x: a sequence Σ of cities
- solution element: a sub-sequence
- 2-opt move: reverse the subsequence in Σ
- neighborhood: all the sequences obtained from Σ by reversing any subsequence

Neighbourhood design

- Normally, implicit definition by *moves* (how to perturb)

  Example KP-0/1: (i) insert; (ii) remove, (iii) swap one in/out; **(iv) ...**

- **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**: any solution $x \in X$ can be obtained from any $y \in X$ by a sequence of moves (desireble feature)

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. is not connected
- Swap neigh. is not connected
- Insertion+removing neigh. is connected (in theory, see solution evaluation...)

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., pairwise interchange:
  - from $1-2-3-4-5-6-7$
    to $1-6-3-4-5-2-7$ (pairwise interchange 2 and 6)
    or $5-2-3-4-1-6-7$ (pairwise interchange 1 and 5)
    or ...
  - size is $O(n^2)$, connected (with respect to maximal solutions)
  - neigh. evaluation in $O(n)$ (no fully-incremental evaluation)
  - overall complexity $O(n^3)$
  - (remark: only maximal solutions are visited)

Solution evaluation function

- Evaluation is used to compare neighbours to the centre solution (and between each others)
- Normally, the objective function

- May include some extra-feature (e.g. combined by means of a weighted sum) to identify "more promising" solutions
  - In KP-0/1, "prefer" solutions with larger residual capacity
    $\tilde{f}(X) = \sum_{i \in X} p_i + \epsilon \left(W - \sum_{i \in X} w_i \right)$

- May include penalty terms (e.g. infeasibility level to allow visiting infeasible solutions)
  - In KP-0/1, let X be the subset of loaded items
    $\tilde{f}(X) = \alpha \sum_{i \in X} p_i - \beta \max \left\{ 0, \sum_{i \in X} w_i - W \right\} \qquad (\alpha, \beta > 0)$
    it potentially activates "removing" move in a connected "insertion+removing" neighbourhood

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)
- **Granularization**: apply a filter (deterministic rules, a pre-trained classifier) to exclude part of the neighbours

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

[Traveling Salesman Problem (TSP)]

Given: a (complete) graph $G(V, A)$; cost c_{ij}, $(i, j) \in A$;

Determine: a min cost hamiltonian cycle.

Prototype application: optimal tour of a sales(wo)man to visit all (her)his clients

- 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:

  $G = (V, A)$ $\quad$ G is complete $\quad$ $|V| = n$

  cost $c_{ij} \neq c_{ji}$ (**asymmetric** case) $\quad$ or $\quad$ $c_{ij} = c_{ji}$ (**symmetric** case)

- Define all the elements of LS

LS for TSP: initial solution by Nearest Node[1] 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 $Cycle = Cycle \oplus \{j\}$; $cost = cost + c_{ij}$
4. set $i = j$
5. if still nodes to be visited, go to 2
6. $Cycle = Cycle \oplus \{i_0\}$; $cost = cost + c_{ii_0}$

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

[1] aka "Nearest Neighbour", but no local search involved yet

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.

- $O(n^2)$: rather effective
- may randomize initial pair and/or r selection

LS for TSP: Nearest/Farthest Insertion

1. Choose the two nearest/farthest nodes i and j an build the initial cycle $C = i - j - i$
2. select the node $r = \arg\min_{v \in V \setminus C} / \max_{v \in V \setminus C} \{c_{vj} : j \in C\}$
3. modify C by inserting r between pairs of consecutive nodes i and j in the current cycle such that $c_{ir} + c_{rj} - c_{ij}$ is *minimized*
4. if still nodes to be visited, go to 2.

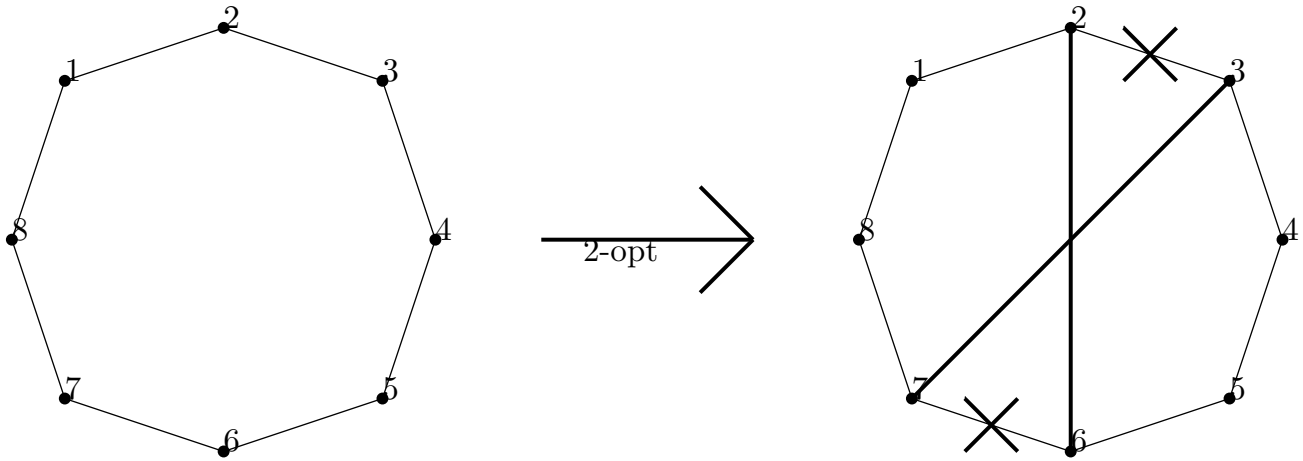
- $O(n^2)$: rather effective (farthest version better, more balanced cycles)
- may randomize initial pair and/or r selection

LS for TSP: Solution Representation

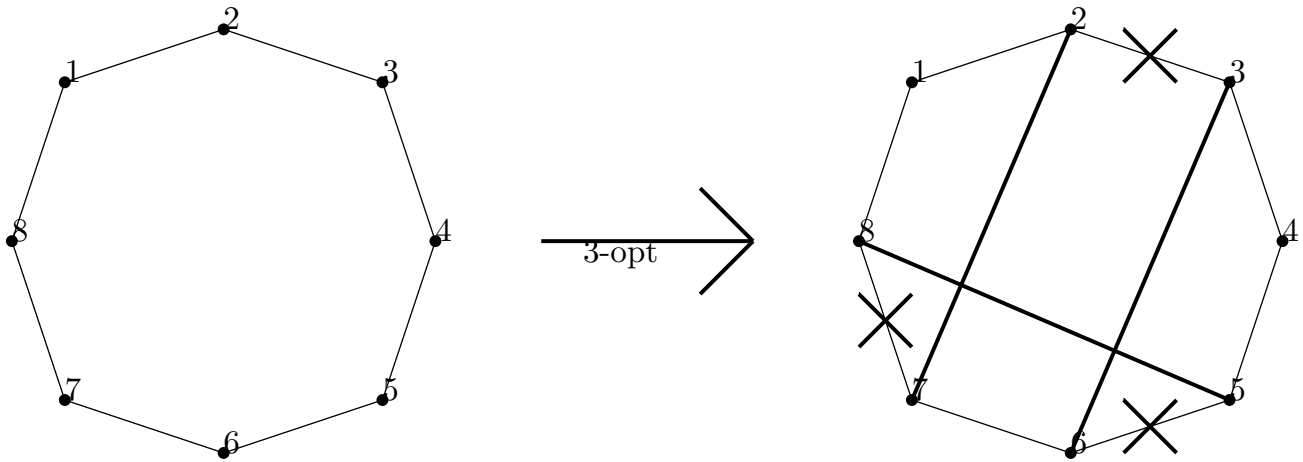
- **arc representation**: arcs in the solution, e.g. as a binary adjacency matrix
- **adjacency representation**: a vector of n elements between 1 and n (representing nodes), $v[i]$ reports the node to be visited after node i
- **path representation**: ordered sequence of the n nodes (a solution is a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

LS for TSP: k-opt neighbourhoods

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



<1,2,3,4,5,6,7,8,1> <1,2,**6,5,4,3**,7,8,1>



<1,2,3,4,5,6,7,8,1> <1,2,**7,6,*3,4,5*,8,1>

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 (symmetric case): reversing sequence between i and j in the sequence $< 1 \ldots h, i, \ldots, j, l, \ldots, 1 >$

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

- 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: start from different (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)
  - ...
  - Accept (move to) non-improving neighbor solutions
- *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

Simulated Annealing [Kirkpatrick, 1983]

Metaphor: annealing process of, e.g., metals.
Alternate warming/cooling to obtain "optimal" molecular structure

search scheme (one possible):

Determine an initial solution x
intialize: best solution $x^* \leftarrow x$ and iteration $k = 0$
repeat
  $k \leftarrow k + 1$
  generate a (random) neighbour y
  if y is better than x^*, **then** $x^* \leftarrow y$
  $Loss = \max\{0, f(y) - f(x)\}$ (minimization problems)[2]
  accept y *with* **probability** $p = exp\left(-\frac{Loss}{T(k)}\right)$
  if accepted, $x \leftarrow y$
until (no further neighbours of x, or max trials)
return x^*

[2] $Loss = \max\{0, f(x) - f(y)\}$ (maximization problems)

SA: cooling schedule

- Parameter $T(k)$: temperature, *cooling schedule*
- $T(first) > T(last)$: the probability of accepting not improving solutions is decreasing over time
- Example: "stepwise" cooling schedule, defined by parameters
  - initial T (maximum)
  - number of iterations at constant T
  - T decrement
  - minimum T

\+ (one of) the first NS scheme inspired by a natural metaphore

\+ provably converges to the global optimum (in theory, under strong assumptions)

\+ simple to implement

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

Tabu Search [Fred Glover, 1989]

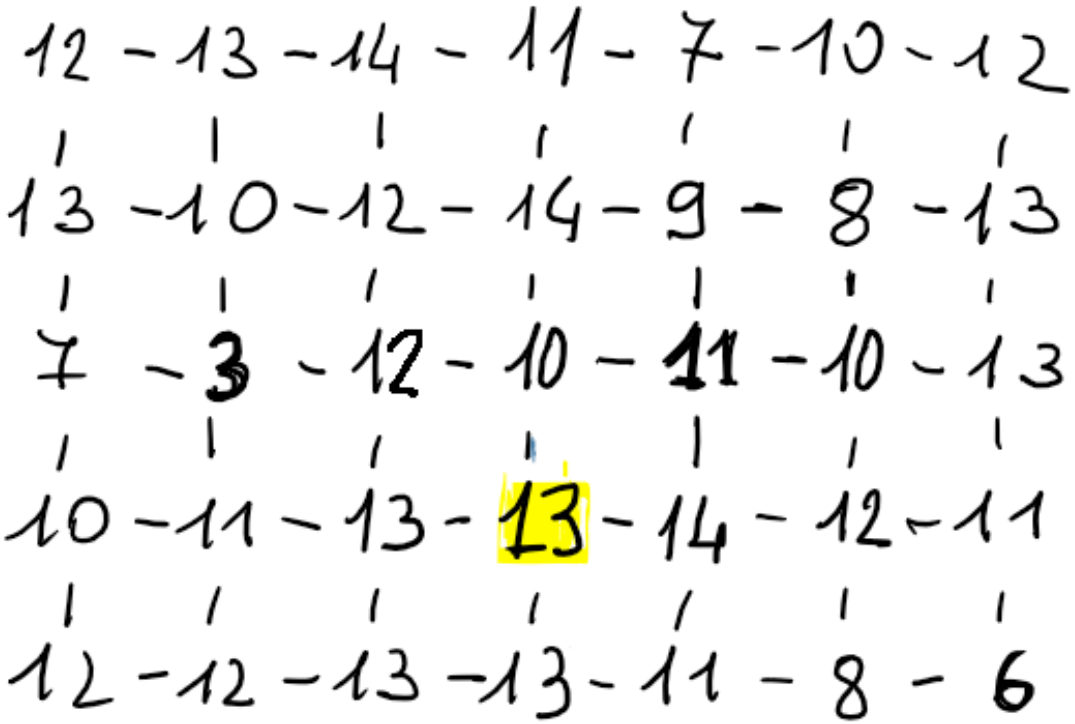
- **Memory** is used to avoid cycling: store *information on visited solutions* (allows exploiting structure of the problem)
- Basic idea: store visited solutions and **exclude them (= make tabu)** from neighbourhoods
- Implementation by storing **Tabu List** of the **last** t **solutions**

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

at iteration k, avoid cycles of length $\leq t$

- t is a parameter to be calibrated (see example*)
- From $N(x)$ to $N(x, k)$: the neighborhood of a same solution may be different at different times, it depends on iteration k
- Notice. Visiting a same solution is allowed: we just need to avoid choosing the same neighbour (recall $N(x, k) \neq N(x, l)$, see example**)

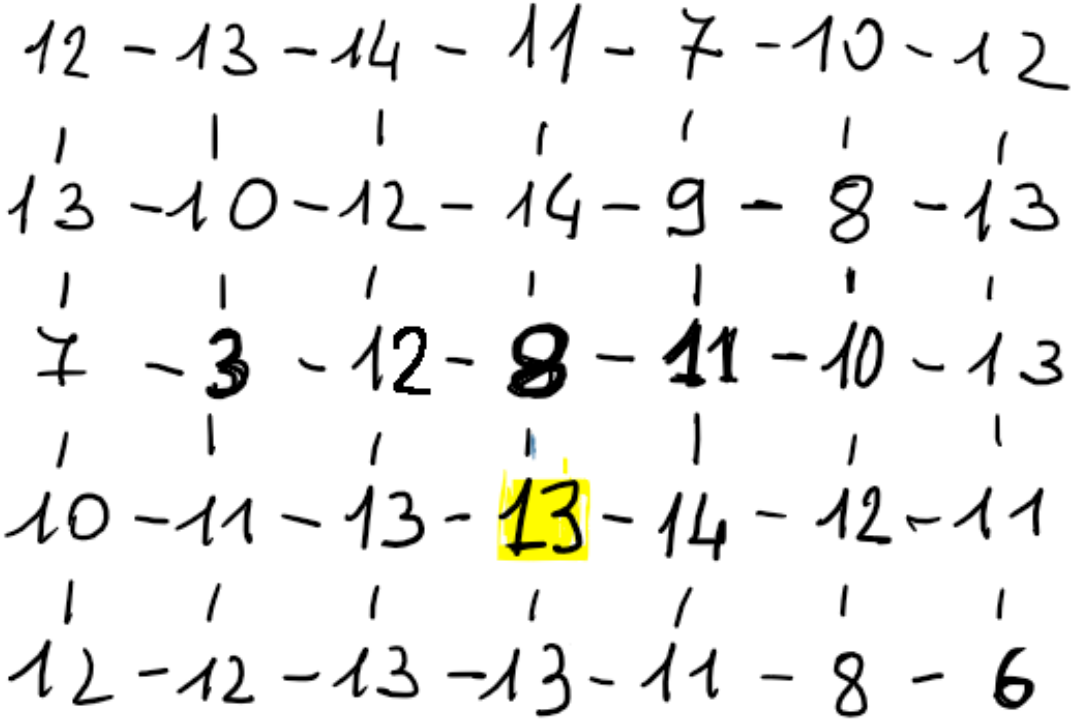
*Example: tabu list length



$t = 6 \Rightarrow$ loc opt 6

$t = 5 \Rightarrow$ loop

**Example: re-visiting solutions



$t = 6 \Rightarrow$ loc opt 6

$t = 5 \Rightarrow$ loc opt 3

Tabu list features

- Tabu List (may) store the *last* t solutions: *short term memory*
- Often it stores *information* about solutions, rather than solutions
  - ▸ E.g., *(reverse) moves* are stored instead of solutions
  - \+ *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)
  - – may declare more tabu directions than needed (see aspiration)
- Example: TSP, 2-opt. TL stores the last t pairs of arcs added (to avoid arcs or involved nodes)
- In any case, t (tabu tenior) has to be calibrated:

- too small: TS may cycle
- too large: too many tabu neighbours

Aspiration criteria

- By storing "information", even 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

- Maximum number of iterations, or time limit *
- Maximum number of NOT (locally or globally) IMPROVING iterations *
- A solution is found satisfying an optimality or "acceptability" certificate, if available...
- 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

* parameter to be calibrated

TS basic scheme

Input: neigh. function N, evaluation function $\tilde{f}$, objective function f, tabu tenure t

Notation: incumbent-current-neighbor solution x^*-x-y, iteration counter k, Tabu List T

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 $f(y)$ better than $f(x^*)$ **then** let $x^* := y$

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

until (stopping criteria)

return (x^*).

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

Intensification 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 search scheme may be improved by **alternating** intensification and diversification, to find and exploit new promising regions and, hence, new (and possibly better) local optima
- in TS, **memory** may play a role (store information on visited solutions, e.g. to allow avoiding the same features during diversification)
  - e.g., in TSP, maintain statistics on how many times each arc has been included in a visited solution and either (i) fix more frequent arcs for intensification or (ii) exclude more frequent arcs for diversification

Intensification and diversification are general principle
that can be applied to **any** metaheuristics (not only to TS)

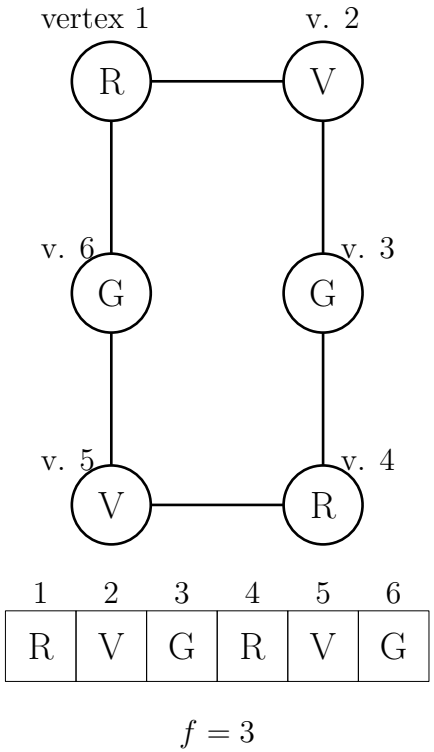
Intensification

- 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)
- temporarily use a more detailed neighbourhood (e.g. allowing many possible moves)
- relax aspiration criteria
- modify evaluation function to penalize far away solutions

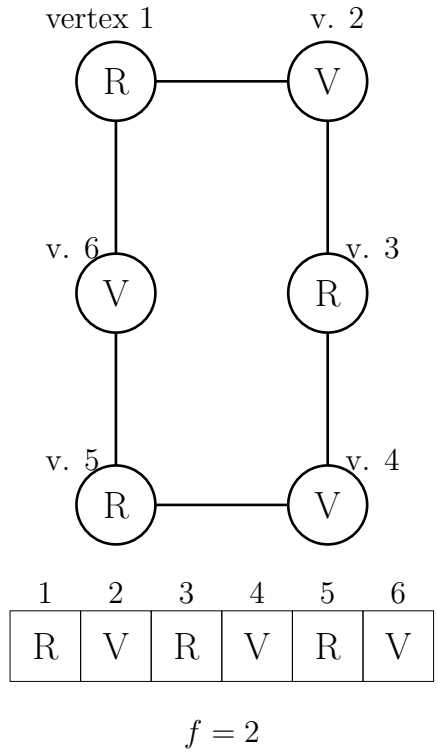
Diversification

- use "larger" neighbourhoods (e.g. k-opt $\rightarrow$ $(k+1)$-opt in TSP, until a better solution is found [*V*ariable *N*eighborhood *D*escent])
  - in the case of tabu search, 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 exclude them (or 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



$f = 3$



$f = 2$

- 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*, to be minimized (minimize non-feasibilities[3])
- 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) (vertex v kept color r)

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

[3] Searching for solutions that minimize constraint infeasibilities is also said "dual search", as opposed to the usual search for better solutions in terms of objective function, also said "primal search"

Population based heuristics

At each iteration

- a *set* [4] of solutions (**population**) is maintained
- some solutions are *recombined* [5] to obtain *new set* of solutions (among which a better one, hopefully)

Several paradigms (often just the metaphor changes![6])

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

General purpose (soft computing), easy to implement (rather than effective!)

[4] In trajectory/based metaheuristics, a single
[5] In trajectory/based metaheuristics, perturbation, move
[6] K. Sörensen, "Metaheuristics – the metaphor exposed"

Genetic Algorithms [Hollande, 1975]

Metaphore: biological evolution as an optimization process:

Survival of the fittest	$\leftrightsquigarrow$	*Optimization*
Individual	$\leftrightsquigarrow$	Solution
Chromosome	$\leftrightsquigarrow$	Encoding
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 affects 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*

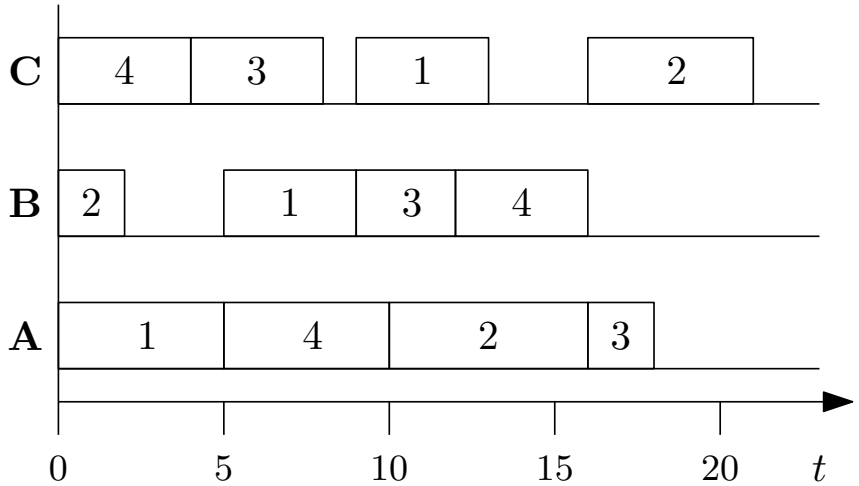
- Job shop scheduling problem instance:

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

Solution Encoding: gene = job ; chromosome = $n * m$ genes

4	2	1	1	3	4	2	3	1	2	3	4

Solution Decoding: sequence of genes gives the priority of jobs on machines, a greedy procedure determines the task starting times



Genetic operators

- **Initial population**: random + *some* heuristic/local search
  - random $\rightarrow$ initial diversification (very important!!!)
  - heuristic (randomized) $\rightarrow$ faster convergence (not too many heuristic solutions, otherwise fast convergence to "local" optimum)
  - Remark: in population based heuristics, a **local optimum** does not correspond to a single solution, but to a population made of identical (or very similar) individuals, i.e., **loss of population diversity** such that recombination is not able to generate improving individuals
- **Fitness**: (variants of the) objective function (see Neighbourhood Search)

Genetic operators: Selection

- **Selection**: larger fitness $\rightsquigarrow$ larger *probability* to be selected
- 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 individuals in the mating pool.
- Selection aims at identifying fittest individual *however*, to **avoid premature convergence**, even worse individuals should be selected (with small probability): they may contain good features (genes), even if their overall fitness is poor

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
- **Linear ranking**: sort individual by increasing fitness and σ_i is the position of i, set $p_i = \frac{2\sigma_i}{N(N+1)}$ $\left[= \text{constant} \cdot \sigma_i \text{ (linear in } \sigma_i) \right]$
- n-**tournament**: in order to select one individual, first select a small subset of n individuals uniformly in the population, then select the best individual in the subset

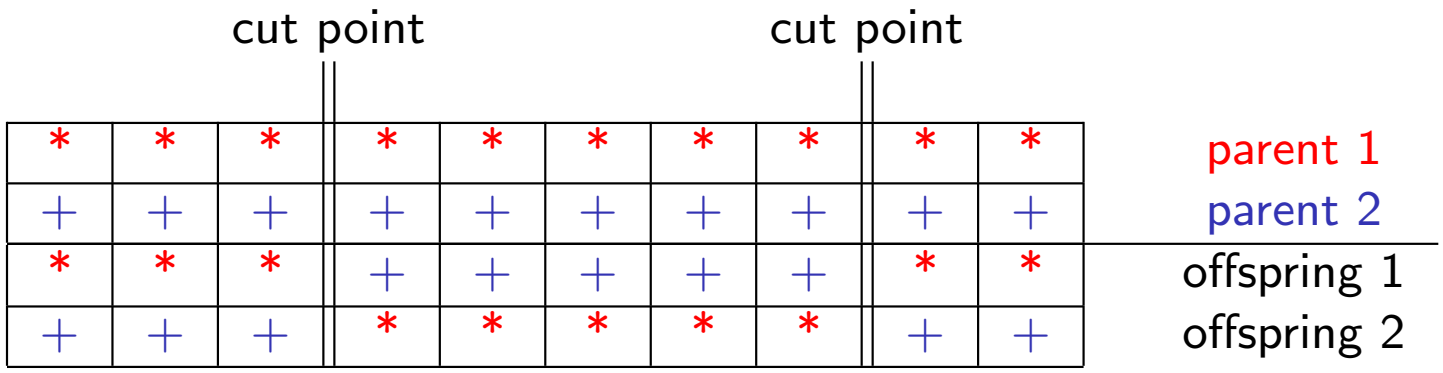
Genetic operators: recombination [crossover]

- 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"



Mutation

After or during crossover, some genes are randomly changed

- **Mutation is primarily intended to avoid genetic drift**: statistical arguments show that **a single** gene tends to take the same value in all the individuals of the population, which implies loss of genetic diversity (loss of diversity in a single gene)
  - Even if individuals may be different from each other according to other genes (population diversity is preserved, e.g., thanks to the other genetic operators), one gene is the same for every individual (genetic drift): e.g, in KP/0-1, we have different solutions in the population, but all of them include item '4'
- Mutation's effects and side effects (sometimes we want them!):
  - (re)introduce *genetic* diversity (= diversity in each gene)
  - slow population convergence (for a good trade-off, normally we change very few genes with very small probability)
  - can be used to obtain, as a side effect, *chromosome* diversity (i.e. diversification, diversity among individuals), if more genes with larger probability undergo mutation (simple way to diversify the population, often 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! Suggestions:
  - use simple and fast LS
  - apply to a selected subset of individuals
  - more sophisticated NS only at the end, as post-optimization

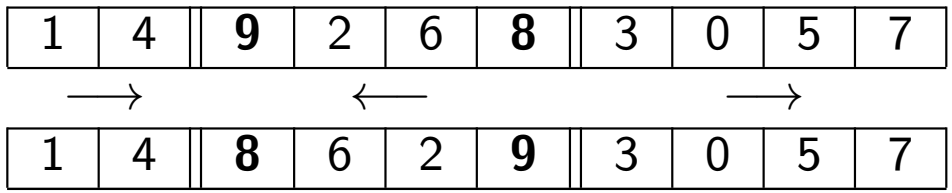
Crossover, mutation and non-feasible offspring

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** (provides *similar* offspring because reciprocal order is maintained for most pairs)

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 move)



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 aims at keeping the **population diversified** (diversity between chromosomes), whilst obtaining (at least one) better and better solution. E.g.:

- 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 designer/developer (or the user...)*, that should find the optimal combination of the parameters.

General remark: normally, the designer/developer should provide the user with a method able to directly find the optimal combination of decision variables. In fact, the algorithm designer/developer 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 provides a good solution)

Validation / evaluation 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)

**see lab example (tabu tenure calibration, tests with multiple runs, statistics and results)*

Parameter calibration (or estimation)

- **Pre-deployment** activity (designer should do, not the user!)
- Estimation to be used for *every* instance (for evaluation purposes)
- Standard technique, given a sufficiently large and representatice set of instances:
  - select an instance **sub**set for training (= training set)
  - extensive runs on the training set
  - select an instance **sub**set for validation (= validation set)
  - performance analysis to select better parameters
  - take **interaction** among parameters into account
  - stochastic components make the calibration harder
  - select an instance **sub**set for test (= test set)
  - runs with the estimated parameter to evaluate the "final" performance

Parameter calibration (or estimation): advanced and automatic techniques

- Black box optimization
- Automatic estimation (e.g. *i-race* package, off-line application of machine learning techniques)
- Adaptivity: parameters are changed at running time, following the search history (e.g., adjusting tabu tenure by cycle-checking, "on-line" application of machine learning techniques)

Warning: metaheuristc principles .vs. metaphores

Many (good) metaheuristics are inspired by (good) metaphores

Recent literature proposed *a true tsunami of "novel" metaheuristic methods, most of them based on a metaphor of some natural or man-made process*: the behavior of any species (bees, wasps, monkeys, apes, birds etc.), the flow of water, musicians playing together etc.

Actually, the basic principles are often not novel, but the same as for trajectory or population based methods

Good or new metaphores do not necessarily lead to good or new metaheuristics!

Golden Rule

An algorithm is good if it provides good results (validation!), and not if it is described by a suggestive metaphor. *See Sörensen, 2015*

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, fixing heuristics
  - Algorithmic schemas: Local branching, Kernel Search ...
  - ...
- **Data driven optimization** (hot research topic, thesis avaialble!)
  - Machine Learning to set optimization model parameters
  - Artificial Intelligence to detect or learn promising regions or search directions (e.g., ML-driven granular search, ML-driven kernel search)
  - ...
- ...

Ideas from sample applications [see proposed readings] (i)

Two-level local search heuristic for pickup/delivery problems in express freight trucking

- Neighbor solution evaluation is difficult: second level local search heuristic
- Tabu Search (intensification) + Variable Neighborhood Search (diversification)
- Randomization (further diversification)
- efficiency: granular exploration + parallel implementation + fast route sequencing

Evolving Neural Networks Through Augmenting Topologies

- Evolutionary algorithm, crossover as "union" of nodes and connections
- Mutation operators obtained by neighborhood moves (+/- node; +/- connection; change weight)
- Independent evolution and management of "species" (subsets of similar solutions)

Models and heuristics for resourse allocation in cloud-oriented distributed engines

- optimal allocation of engine modules to compute nodes in the cloud
- QoS and QoE requirements and indicators
- mathematical models for the general case, heuristics for special cases
- Application to cloud gaming engines

Ideas from sample applications [see proposed readings] (ii)

Data-driven matheuristic for the Air Traffic Flow Management Problem

- Takes advantage from data repositories on historically flown trajectories
- Specific Integer Programming formulation integrating airspace user's preferences and predefined routes
- Model components (routes and preference parameters) determined through Data Analytics (cluster analysis, tree classifiers)
- Granular exploration of very large neighborhoods through Integer Programming and Data Analytics (good move classifiers)

Generalized TSP for improved equity in eletric vehicle sharing systems

- Mathematical model to define relocation tours that optimize equity indicators and operational costs
- Model parameters learned from hystorical data (impact of location on equity indicators)
- Model solved by matheuristics (partition heuristic, MILP for TSP, neighborhood search)