

# Heuristic Algorithms for Combinatorial Optimization problems

Ph.D. course in Computer Science

Roberto Cordone  
DI - Università degli Studi di Milano



E-mail: [roberto.cordone@unimi.it](mailto:roberto.cordone@unimi.it)

Web page: <https://homes.di.unimi.it/cordone/courses/2023-haco/2023-haco.html>

# Recombination heuristics

Constructive and exchange heuristics manage one solution at a time  
(except for the *Ant System*)

Recombination heuristics manage several solutions in parallel

- start from a set (**population**) of solutions (**individuals**) obtained somehow
- recombine the individuals generating a new population

Their original aspect is the use of operations working on several solutions, but they often include features of other approaches (sometimes renamed)

Some are nearly or fully deterministic

- Scatter Search
- Path Relinking

others are strongly randomised (often based on biological metaphors)

- genetic algorithms
- memetic algorithms
- evolution strategies

*Of course the effectiveness of a method does not depend on the metaphor*

# General scheme

The basic idea is that

- good solutions share components with the global optimum
- different solutions can share different components
- combining different solutions, it is possible to merge optimal components more easily than building them step by step

The typical scheme of recombination heuristics is

- build a **starting population** of solutions
- as long as a suitable **termination condition** does not hold
- at each iteration (**generation**) update the population
  - **extract single individuals** and **apply exchange operations to them**
  - **extract subsets of individuals** (usually, pairs) and **apply recombination operations to them**
  - collect the individuals thus generated and **choose whether to accept or not each of them** and **how many copies** into the new population

# Scatter Search

*Scatter Search (SS)*, proposed by Glover (1977), proceeds as follows

- 1 generate a starting population of solutions
- 2 improve all of them with an exchange procedure
- 3 build a *reference set*  $R = B \cup D$  where
  - subset  $B$  includes the best known solutions
  - subset  $D$  includes the “farthest” solutions (from  $B$  and each other) (this requires a distance definition, e.g. the Hamming distance)
- 4 for each pair of solutions  $(x, y) \in B \times (B \cup D)$ 
  - “recombine”  $x$  and  $y$ , generating  $z$
  - improve  $z$  obtaining  $z'$  with an exchange procedure
  - if  $z' \notin B$  and  $B$  contains a worse solution, replace it with  $z'$  (we want no duplicates in the reference set)
  - if  $z' \notin D$  and  $D$  includes a closer solution, replace it with  $z'$  (we want no duplicates in the reference set)
- 5 terminate when  $R$  is unchanged

The rationale is that

- the recombinations in  $B \times B$  intensify the search
- the recombinations in  $B \times D$  diversify the search

# General scheme of the *Scatter Search* approach

Algorithm ScatterSearch( $I, P, n_B, n_D$ )

$B := \emptyset; D := \emptyset;$

Repeat

  Stop = true;

  For each  $x \in P$  do

$z := \text{SteepestDescent}(I, x);$  If  $f(z) < f(x^*)$  then  $x^* := z;$

$y_B := \arg \max_{y \in B} f(y); y_D := \arg \min_{y \in D} d(y, B \cup D \setminus \{y\});$

    If  $z \notin B$  and ( $|B| < n_B$  or  $f(z) < f(y_B)$ ) then

      {  $B$  keeps the  $n_B$  best unique solutions }

$B := B \cup \{z\};$  Stop := false; If  $|B| > n_B$  then  $B := B \setminus \{y_B\};$

    Elseif  $z \notin D$  and ( $|D| < n_D$  or  $d(z, B \cup D \setminus \{y_D\}) > d(y_D, B \cup D \setminus \{y_D\})$ ) then

      {  $D$  keeps the  $n_D$  most diverse unique solutions }

$D := D \cup \{z\};$  Stop := false; If  $|D| > n_D$  then  $D := D \setminus \{y_D\};$

    EndIf

  EndFor

$P := \emptyset;$

  For each  $(x, y) \in B \times (B \cup D)$  do      { Recombine to build the new population }

$P := P \cup \text{Recombine}(x, y, I);$

  EndFor

until Stop = true;

Return  $(x^*, f(x^*));$

# Recombination procedure

The recombination procedure depends on the problem

Usually, solutions  $x$  and  $y$  are manipulated as subsets

- 1 include in  $z$  all the elements shared by  $x$  and  $y$ :

$$z := x \cap y$$

*(both solutions concur in suggesting those elements)*

- 2 augment solution  $z$  adding elements from  $x \setminus z$  or  $y \setminus z$ 
  - chosen at random or with a greedy selection criterium
  - alternatively from each source or freely from the two sources

*(this is similar to a restricted constructive heuristic)*

- 3 if necessary, add external elements from  $B \setminus (x \cup y)$
- 4 if subset  $z$  is unfeasible, apply an auxiliary exchange heuristic to make it feasible (repair procedure)

## *MDP*

- start with  $z := x \cap y$
- augment  $z$  with  $k - |z|$  random or greedy points from  $x \setminus z$  or  $y \setminus z$
- no repair procedure is required

## *Max-SAT*

- start with  $z := x \cap y$
- augment  $z$  with  $n - |z|$  random or greedy truth assignments from  $x \setminus z$  or  $y \setminus z$
- no repair procedure is required

## *KP*

- start with  $z := x \cap y$
- augment  $z$  with random or greedy elements from  $x \setminus z$  or  $y \setminus z$  respecting the capacity
- no repair procedure is required
- the solution could be augmented with elements from  $B \setminus (x \cup y)$

## *SCP*

- start with  $z := x \cap y$
- augment  $z$  with random or greedy columns from  $x \setminus z$  or  $y \setminus z$  (avoiding the redundant ones)
- remove the redundant columns with a destructive phase



# Path Relinking

*Path Relinking (PR)*, proposed by Glover (1989), is generally used as a final intensification procedure more than as a stand-alone method

Given a neighbourhood  $N$  and an exchange heuristic based on it

- collect in a reference set  $R$  the best solutions generated by the auxiliary heuristic (**elite solutions**)
- **for each pair of solutions  $x$  and  $y$  in  $R$** 
  - **build a path  $\gamma_{xy}$  from  $x$  to  $y$  in the search space of neighbourhood  $N$  applying to  $z^{(0)} = x$  the auxiliary exchange heuristic, but **choosing at each step the solution closest to the destination  $y$****

$$z^{(k+1)} := \arg \min_{z \in N(z^{(k)})} d(z, y)$$

where  $d$  is a suitable metric function on the solutions

**In case of equal distance, optimise the objective function  $f$**

- find the best solution  $z_{xy}^*$  along the path (and improve it)

$$z_{xy}^* := \arg \min_{k \in \{1, \dots, |\gamma_{xy}| - 1\}} f(z^{(k)})$$

- if  $z_{xy}^* \notin R$  and is better than the worst in  $R$ , add it to  $R$

# General scheme of the *Path Relinking* approach

Algorithm PathRelinking( $I, P, n_R$ )

Repeat

$R := \emptyset$ ;

For each  $x \in P$  do

$z := \text{SteepestDescent}(I, x)$ ; If  $f(z) < f(x^*)$  then  $x^* := z$ ;

$y_R := \arg \max_{y \in R} f(y)$ ;

If  $z \notin R$  and ( $|R| < n_R$  or  $f(z) < f(y_R)$ ) then

{  $R$  keeps the  $n_R$  best unique solutions }

$R := R \cup \{z\}$ ; Stop := false; If  $|R| > n_R$  then  $R := R \setminus \{y_R\}$ ;

EndIf

EndFor

$P := \emptyset$ ;

For each  $x \in R$  and  $y \in R \setminus \{x\}$  do { Recombine to build the new population }

$z := x$ ;  $z^* := x$ ;

While  $z \neq y$  do { Build a path from  $x$  to  $y$  }

$Z := \arg \min_{z' \in N(z)} d(z', y)$ ;  $z := \arg \min_{z' \in Z} f(z')$ ;

If  $f(z) < f(z^*)$  then  $z^* := z$

EndWhile;

If  $z^* \notin P$  then  $P := P \cup \{z^*\}$ ;

EndFor

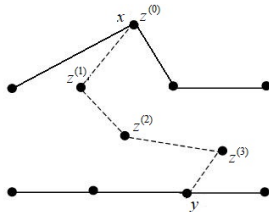
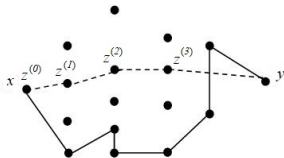
until Stop = true;

Return ( $x^*$ ,  $f(x^*)$ );

# Relinking paths

The paths explored in this way

- **intensify the search**, because they connect good solutions
- **diversify the search**, because they follow different paths with respect to the exchange heuristic (especially if the extremes are far away)



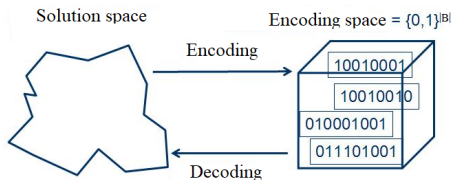
- since the distance of  $z^{(k)}$  from  $y$  is decreasing, one can explore
  - worsening solutions without the risk of cyclic behaviours
  - unfeasible subsets without the risk of not getting back to feasibility  
(*they do not improve directly, but open the way to improvements*)

Given two solutions  $x$  and  $y$ , Path Relinking has several variants:

- *forward path relinking*: build a path from the worse to the better one
- *backward path relinking*: build a path from the better to the worse one
- *back-and-forward path relinking*: build both paths
- *mixed path relinking*: build a path with alternative steps from each extreme (updating the destination)
- *truncated path relinking*: build only the first steps of the path (if the good solutions are experimentally close to each other)
- *external path relinking*: build a path from one moving away from the other (if the good solutions are experimentally far from each other)

# Encoding-based algorithms

Many recombination heuristics define and manipulate **encodings** of the solutions (i.e., **compact representations**), rather than the solutions



The aims of this approach are

- **abstraction**: conceptually distinguishing the method from the problem to which it is applied
- **generality**: build operators effective on every problem represented with a given family of encodings

In a strict sense, every representation of a solution in memory is an encoding: the term “encoding” tends to be used for the more involved and compact ones

*The difference is blurred*

# Genetic algorithm

The genetic algorithm, proposed by Holland (1975), is the most famous. It builds and **encodes** a population  $X^{(0)}$ , and repeatedly applies:

- 1 **selection**: generate a new population starting from the current one
- 2 **crossover**: recombine subsets of two or more individuals
- 3 **mutation**: modify the individuals

*Algorithm* GeneticAlgorithm( $I, X^{(0)}$ )

$\Xi^{(0)} := \text{Encode}(X^{(0)}); x^* := \arg \min_{x \in X^{(0)}} f(x); \quad \{ \text{Best solution found so far} \}$

*For*  $g = 1$  to  $n_g$  *do*

$\Xi := \text{Selection}(\Xi);$

$\Xi := \text{Crossover}(\Xi);$

$x_c := \arg \min_{\xi \in \Xi} f(x(\xi));$

*If*  $f(x_c) < f(x^*)$  *then*  $x^* := x_c;$

$\Xi := \text{Mutation}(\Xi);$

$x_m := \arg \min_{\xi \in \Xi} f(x(\xi));$

*If*  $f(x_m) < f(x^*)$  *then*  $x^* := x_m;$

*EndFor*;

*Return*  $(x^*, f(x^*));$

# Features of a good encoding

The performance of a genetic algorithm depends on the encoding

The following properties should be satisfied (with decreasing importance)

- 1 each solution should have an encoding, except for dominated ones; otherwise, there would be unreachable solutions
- 2 different solutions should have different encodings (or the best solution with a given encoding should be easy to find); otherwise, there would be unreachable solutions
- 3 each encoding should correspond to a feasible solution; otherwise, the population would include useless individuals
- 4 each solution should correspond to the same number of encodings; otherwise, some solutions would be unduly favoured
- 5 the encoding and decoding operations should be efficient, otherwise, the algorithm would be inefficient
- 6 locality: small changes to the encoding should induce small changes to the solution, otherwise intensification and diversification would be impossible

These conditions depend very much on the constraints of the problem

*(so much for abstraction...)*

# Feasible and unfeasible encodings

Mutation and crossover operators easily produce unfeasible subsets

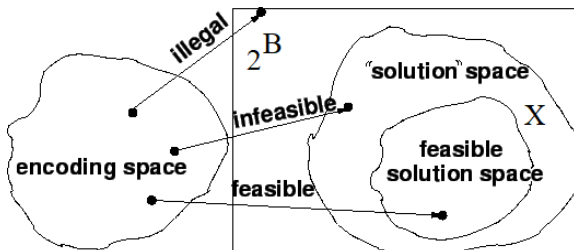
if property 3 is not satisfied; this may imply the violation of

- ① quantitative constraints (e.g., a capacity is exceeded)
- ② structural constraints (e.g., the solution is not made of circuits)

*The distinction is conventional*

The second kind of infeasibility is harder to repair, because it concerns constraints that interact more strongly with each other

Some encodings guarantee structural (though not quantitative) feasibility





# Encodings: the incidence vector

The most direct encoding for Combinatorial Optimization problems is the **binary incidence vector**  $\xi \in \mathbb{B}^{|B|}$

$$\begin{cases} \xi_i = 1 \text{ indicates that } i \in x \\ \xi_i = 0 \text{ indicates that } i \notin x \end{cases}$$

A generic binary vector corresponds

- in the *KP* to a set of objects: its weight could be excessive
- in the *SCP* to a set of columns: it could leave uncovered rows
- in the *PMSP* and in the *BPP* to a set of assignments of tasks (objects) to machines (containers): it could make zero or more assignments for an element; in the *BPP*, it could violate the capacity of some container;
- in the *TSP* to a set of arcs: it could not form a hamiltonian circuit
- in the *CMSTP* (*VRP*) to a set of edges (arcs): it could not form a tree (set of cycles), or exceed the capacity of the subtrees (circuits)

# Encodings: symbolic strings

If the ground set is partitioned into components

(objects, tasks, Boolean variables, vertices, nodes...)

$$B = \bigcup_{c \in C} B_c \quad \text{with } B_c \cap B_{c'} = \emptyset \text{ for each } c \neq c'$$

and the feasible solutions contain one element of each component

$$|x \cap B_c| = 1 \text{ for each } c$$

one can

- define for each  $c \in C$  an alphabet of symbols describing component  $B_c$
- encode the solution into a string of symbols  $\xi \in B_1 \times \dots \times B_{|C|}$

$$\xi_c = \alpha \text{ indicates that } x \cap B_c = \{(c, \alpha)\}$$

Examples of encodings:

- *Max-SAT*: a string of  $n$  Boolean values, one for each logical variable
- *PMSP*: a string of machine labels, one for each task
- *BPP/CMSTP*: a string of container/subtree labels, one for each object/vertex:
  - the structural constraint on object assignment is enforced
  - the quantitative constraint on capacity is neglected
- for the *VRP*, a string of vehicle labels, one for each node (but capacity is neglected and decoding the circuit for each vehicle is an  $\mathcal{NP}$ -complete problem)
- the solutions of the *TSP*, the *KP*, the *SCP* are not partitions

# Encodings: permutations of a set

A common encoding is given by the **permutations of a set**

- if the solutions are permutations, this is the natural encoding  
(*TSP* solutions are subsets of arcs, but also permutations of nodes)
- if the solutions are partitions and the objective is additive on the subsets, the *order-first split-second* method transforms permutations into partitions  
(*but solutions and encodings do not correspond one-to-one!*)
- if the problem admits a constructive algorithm that at each step
  - 1 chooses an element
  - 2 chooses how to add it to the solution (if many ways exist)we can feed elements to the algorithm following the permutation  
(*if step 2 is not unique, some solutions could have no encoding*)

# Selection

At each generation  $g$  a new population  $\Xi^{(g)}$  is built extracting  $n_p = |\Xi^{(g)}|$  individuals from the current population  $\Xi^{(g-1)}$

$$\Xi^{(g)} := \text{Selection}(\Xi^{(g-1)});$$

The extraction follows two fundamental criteria

- 1 an individual can be extracted more than once
- 2 better individuals are extracted with higher probability

$$\varphi(\xi) > \varphi(\xi') \Rightarrow \pi_\xi \geq \pi_{\xi'}$$

where the fitness  $\varphi(\xi)$  is a measure of the quality of individual  $\xi$

- for a maximization problem, commonly

$$\varphi(\xi) = f(x(\xi))$$

- for a minimization problem, commonly

$$\varphi(\xi) = UB - f(x(\xi))$$

where  $UB \geq f^*$  is a suitable upper bound on the optimum

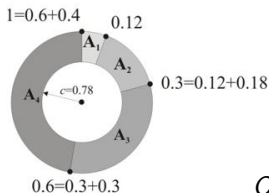
# Proportional selection

The original scheme proposed by Holland (1975) assumed a **probability proportional to fitness**

$$\pi_{\xi} = \frac{\varphi(\xi)}{\sum_{\xi \in \Xi} \varphi(\xi)}$$

This is named **roulette-wheel selection** or **spinning wheel selection**:

- given the fitness for  $\xi_i \in \Xi$  ( $i = 1, \dots, n_p$ )
- build the intervals  $\Gamma_i = \left( \sum_{k=1}^{i-1} \pi_{\xi_k}; \sum_{k=1}^i \pi_{\xi_k} \right]$  in  $O(n_p)$  time
- extract a random number  $r \in U(0; 1]$
- choose individual  $i^*$  such that  $r \in \Gamma_{i^*}$  in  $O(\log n_p)$  time each



Overall  $O(n_p \log n_p)$  time

# Rank selection

The proportional selection suffers from

- **stagnation**: in the long term, all individuals tend to have a good fitness, and therefore similar selection probabilities
- **premature convergence**: if the best individuals are bad and the other ones very bad, the selection quickly generates a bad population

To overcome these limitations, one should **at the same time**

- **assign different probabilities to the individuals**
- **limit the difference of probability among the individuals**

The rank selection method

- **sorts the individuals by nondecreasing fitness**

$$\Xi^{(g)} = \{\xi_1, \dots, \xi_{n_p}\} \text{ with } \varphi_{\xi_1} \leq \dots \leq \varphi_{\xi_{n_p}}$$

- assigns to the  $k$ -th individual a probability equal to

$$\pi_{\xi_k} = \frac{k}{\sum_{k=1}^{n_p} k} = \frac{2k}{n_p(n_p - 1)}$$

It can be done in  $O(n_p \log n_p)$  time as in the previous case

# Tournament selection

An efficient compromise consists in

- extracting  $n_p$  random subsets  $\Xi_1, \dots, \Xi_{n_p}$  of size  $\alpha$
- selecting the best individual from each subset

$$\xi_k := \arg \max_{\xi \in \Xi_k} \varphi(\xi) \quad k = 1, \dots, n_p$$

in time  $O(n_p \alpha)$

Parameter  $\alpha$  tunes the strength of the selection:

- $\alpha \approx n_p$  favours the best individuals
- $\alpha \approx 2$  leaves chances to the bad individuals

All selection procedures admit an **elitist variant**, which includes in the new population the best individual of the current one

*(always keep the best individual found so far)*

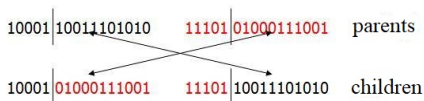
# Crossover

The **crossover** operator combines  $k \geq 2$  individuals to generate other  $k$

The most common ones set  $k = 2$  and are

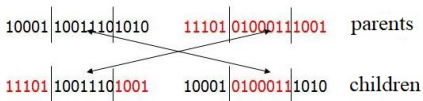
- **simple crossover:**

- extract a random position with uniform probability
- split the encoding in two parts at the extracted position
- exchange the final parts of the encodings of the two individuals



- **double crossover:**

- extract two positions at random with uniform probability
- split the encoding in three parts at the extracted positions
- exchange the extreme parts of the encodings of the two individuals





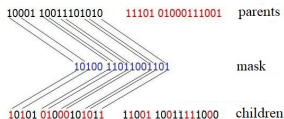
Generalizing, one obtains the

- $\alpha$  points crossover:
  - extract  $\alpha$  positions at random with uniform probability
  - split the encoding in  $\alpha + 1$  parts at the extracted positions
  - exchange the odd parts of the encodings of the two individuals (first, third, etc. . .)

For small values of  $\alpha$ , this implies a **positional bias**:  
symbols close in the encoding tend to remain close

To cancel this bias, one can adopt the

- **uniform crossover**:
  - build a random binary vector  $m \in U(\mathbb{B}^n)$  (“mask”)
  - if  $m_i = 1$  exchange the symbols in position  $i$  of the two individuals, if  $m_i = 0$  keep them unmodified



# Crossover versus Scatter Search and Path Relinking

The crossover operator resembles the recombination phase of *SS* and *PR*

The main differences are that

- 1 it recombines the symbols of the encodings, instead of
  - recombining the solutions (*SS*)
  - performing a chain of exchanges on the solutions (*PR*)
- 2 it operates on the whole population, instead of only a reference set *R*
- 3 it operates on random pairs of individuals, instead of methodically scanning all pairs of solutions of *R*
- 4 it generates a pair of new individuals, instead of
  - generating a single intermediate solution (*SS*)
  - visiting the intermediate solutions and choosing the best one (*PR*)
- 5 the new individuals enter the population directly, instead of becoming candidates for the reference set

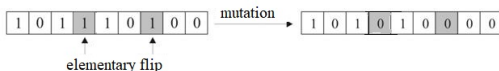
*However, classifying an operator can be a matter of taste*

The **mutation** operator **modifies an individual to generate a similar one**

- scan encoding  $\xi$  one symbol at a time
- decide with probability  $\pi_m$  to modify the current symbol

The kind of modification usually depends on the encoding

- **binary encodings**: flip  $\xi_i$  into  $\xi'_i := 1 - \xi_i$



- **symbol strings**: replace  $\xi_c$  with a random symbol  $\xi'_c \in B_c \setminus \{\xi_c\}$  selected with a uniform probability
- **permutations**: there are many proposals
  - **exchange two random elements** in the permutation (*swap*)
  - **reverse the stretch between two random positions** of the permutation
  - ...

# Mutation versus exchange heuristics

The mutation operator has strong relations with exchange operations

The main differences are that

- ① it modifies the symbols of an encoding, instead of exchanging elements of a solution
- ② it operates on random symbols, instead of exploring a neighbourhood systematically
- ③ it operates on a random subset of symbols of size unknown *a priori*, somewhat like sampling a very large scale neighbourhood, instead of exchanging a fixed number of elements

*However, classifying an operator can be a matter of taste*

# The feasibility problem

If the encoding is not fully invertible, **crossover and mutations sometimes generate encodings that do not correspond to feasible solutions**

We distinguish between

- **feasible encodings** that correspond to feasible solutions
- **unfeasible encodings** that correspond to legal, but unfeasible subsets

The existence of unfeasible encodings implies several disadvantages:

- **inefficiency**: computational time is lost handling meaningless objects
- **ineffectiveness**: the heuristic explores less solutions (possibly, none)
- **design problems**: fitness must be defined also on unfeasible subsets

There are three main approaches to face this problem

- 1 **special encodings and operators** (*avoid or limit infeasibility*)
- 2 **repair procedures** (*turn infeasibility into feasibility*)
- 3 **penalty functions** (*accept infeasibility, but discourage it*)

# Special encodings and operators

The idea is to investigate

- **encodings that** (nearly) **always yield feasible solutions**, such as
  - permutation encodings and order-first split-second decodings for partition problems (*CMSTP*, *VRP*, etc. . .)
  - permutation encodings and constructive heuristic decodings for scheduling problems (*PMSP*, . . .)
- **crossover and mutation operators that maintain feasibility**, such as
  - operators that simulate moves on solutions (*k*-exchanges)
  - specialised operators (*PMX* operator for the *TSP*)

These methods

- tend to closely approximate exchange and recombination heuristics based on the concept of neighbourhood
- give up the idea of abstraction and focus on the specific problem, contrary to classical genetic algorithms

# Repair procedures

A **repair procedure** is a **function**  $x_R : \Xi \rightarrow X$  that

- given an encoding  $\xi$ , yielding a possibly unfeasible subset  $x(\xi) \notin X$
- transforms  $x(\xi)$  into a feasible solution  $x_R(\xi) \in X$
- returns  $x_R$

The procedure is applied to each unfeasible encoding  $\xi \in \Xi^{(g)}$

- in some methods, **the encoding  $\xi(x_R(\xi))$  replaces  $\xi$  in  $X^{(g)}$**
- in other ones,  **$\xi$  remains in  $\Xi^{(g)}$  and  $x_R(\xi)$  is used only to update  $x^*$**

The methods of the first family

- **maintain a population of feasible solutions**

but they introduce

- a strong **bias in favour of feasible encodings**
- a **bias in favour of the feasible solutions most easily obtained** with the repair procedure

# Penalty functions: measuring the infeasibility

If the objective function is extended to unfeasible subsets  $x \in 2^B \setminus X$ , the fitness function  $\phi(\xi)$  can be extended to any encoding, but **many unfeasible subsets have a fitness larger than the optimal solution**

The selection operator tends to favour such unfeasible subsets

To avoid that, **the fitness function must combine**

- the **objective value**  $f(x(\xi))$
- a **measure of infeasibility**  $\psi(x(\xi))$

$$\begin{cases} \psi(x(\xi)) = 0 & \text{if } x(\xi) \in X \\ \psi(x(\xi)) > 0 & \text{if } x(\xi) \notin X \end{cases}$$

If the constraints of the problem are expressed by equalities or inequalities,  $\psi(x)$  can be defined as a weighted sum of their violations

*How to define the weights?  
Are they fixed, variable or adaptive?*



# Penalty functions: definition of the fitness

The most typical combinations are

- **absolute penalty**: given two encodings  $\xi$  and  $\xi'$ 
  - if both are feasible, the one with smaller  $f$  is better
  - if exactly one is feasible, the feasible one is better than the other
  - if both are unfeasible, the one with smaller  $\psi$  is better

*(suitable for rank and tournament selection)*

- **proportional penalty**: the unfeasible encodings have a penalty increasing with the violation

$$\varphi(\xi) = f(x(\xi)) - \alpha\psi(x(\xi)) + M \quad \text{for maximisation problems}$$

$$\varphi(\xi) = -f(x(\xi)) - \alpha\psi(x(\xi)) + M \quad \text{for minimisation problems}$$

where offset  $M$  guarantees that  $\varphi(\xi) \geq 0$  for all encodings

- **penalty obtained by repair**, that is keep the unfeasible encoding, but derive its fitness from the objective value of the repaired solution

$$\varphi(\xi) = f(x_R(\xi)) \text{ or } \varphi(\xi) = UB - f(x_R(\xi))$$

since usually  $f(x_R(\xi))$  is worse than  $f(x(\xi))$

# Proportional penalty functions: weight tuning

Experimentally, **it is better to use the smallest effective penalty**

- if the penalty is too small, too few feasible solutions are found
- if the penalty is too large, the search is confined within a part of the feasible region (*“hidden” feasible solutions are hard to find*)

A good value of the parameter  $\alpha$  tuning the penalty can be found with

- **dynamic methods**: increase  $\alpha$  over time according to a fixed scheme (*first reach good subsets, then enforce feasibility*)
- **adaptive methods**: update  $\alpha$  depending on the situation
  - increase  $\alpha$  when unfeasible encodings dominate the population
  - decrease  $\alpha$  when feasible encodings dominate
- **evolutionary methods**: encode  $\alpha$  in each individual, in order to select and refine both the solution and the algorithm parameter

# Memetic algorithms

Memetic algorithms (Moscato, 1989) are inspired by the concept of **meme** (Dawkins, 1976) that is a **basic unit of reproducible cultural information**

- genes are selected only at the phenotypic expression level
- memes also adapt directly, as in Lamarckian evolution

Out of the metaphor, **memetic algorithms combine**

- **“genotypic” operators that manipulate the encodings** (crossover and mutation)
- **“phenotypic” operators that manipulate the solutions** (local search)

In short, **the solutions are improved with exchanges before reencoding**

Several parameters determine how to apply local search

- how often (at every generation, or after a sufficient diversification)
- to which individuals (all, the best ones, the most diversified ones)
- for how long (until a local optimum, beyond, or stopping before)
- with what method (steepest descent, VNS, ILS, etc. . . )

# Evolution strategies

They have been proposed by Rechenberg and Schwefel (1971)

The main differences with respect to classical genetic algorithms are:

- the solutions are encoded into **real vectors**
- **a small population of  $\mu$  individuals generate  $\lambda$  candidate descendants**  
(originally,  $\mu = 1$ )
- the new individuals compete to build the new population
  - in the  **$(\mu, \lambda)$  strategy** the best  $\mu$  descendants replace the original population, even if some are dominated
  - in the  **$(\mu + \lambda)$  strategy** the best  $\mu$  individuals overall (predecessors or descendants) survive in the new population
- the mutation operator **sums to the encoding a random noise with a normal distribution of zero average**

$$\xi' := \xi + \delta \text{ with } \delta \in N(0, \sigma)$$

- originally, the crossover operator was not used (now it is)