A4M33EOA
Optimization. Local Search. Evolutionary methods.

Petr Pošík

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

What is this course about?

Problem solving by means of **evolutionary algorithms**, especially for hard problems where

- no low-cost, analytic and complete solution is known.

What makes 'hard problems' hard?

1. *Barriers inside the people* solving the problem.
   - Insufficient equipment (money, knowledge, . . .)
   - Psychological barriers (insufficient abstraction or intuition ability, 'fossilization', influence of ideology or religion, . . .)
2. *Number of possible solutions* grows very quickly with the problem size.
   - Complete enumeration intractable
3. The goal must fulfill some *constraints*.
   - Constraints make the problem much more complex, sometimes it is very hard to find *any feasible solution*.
4. Two or more *antagonistic goals*.
   - It is not possible to improve one without compromising the other.
5. The goal is *noisy* or *time dependent*.
   - The solution process must be repeated over and over.
   - Averaging to deal with noise.

Contents

Question you should be able to answer right now

- What is *optimization*? Give some examples of optimization tasks.
- In what courses did you meet optimization?
- What *sorts of optimization tasks* do you know? What are their characteristics?
- What is the difference between *exact methods* and *heuristics*?
- What is the difference between *constructive* and *improving (generative, perturbative)* methods?
- What is the *black-box optimization*? What can you do to solve such problems?
- What is the difference between *local* and *global* search?

(Skip the rest of this section
if you know the answers to the above questions.)

Optimization problems: definition

Among all possible objects $x \in \mathcal{F} \subset \mathcal{S}$, we want to determine such an object x_{OPT} that optimizes (minimizes) the function f:

$$x_{\text{OPT}} = \arg \min_{x \in \mathcal{F} \subset \mathcal{S}} f(x), \quad (1)$$

where

- $\mathcal{S}$ is the search space (of all possible candidate solutions),
- $\mathcal{F}$ is the space of all feasible solutions (which satisfy all constraints), and
- f is the objective function which measures the quality of a candidate solution x.

The task can be written in a different format, e.g.:

minimize $f(x)$
subject to $x \in \mathcal{F}$

The representation of a solution is

- a data structure that holds the variables manipulated during optimization, and
- induces the search space $\mathcal{S}$.

The constraints then define the feasible part $\mathcal{F}$ of the search space $\mathcal{S}$.

The optimization criterion (aka objective or evaluation function) f

- must "understand" the representation, and adds the meaning (semantics) to it.
- It is a measure of the solution quality.
- It is not always defined analytically, it may be a result of a simulation or experiment, it may be a subjective human judgement, . . .

Representation

Representation is a data structure holding the characteristics of a candidate solution, i.e. its tunable variables. Very often this is

- a vector of real numbers,
- a binary string,
- a permutation of integers,
- a matrix,

but it can also be (or be interpreted as)

- a graph, a tree,
- a schedule,
- an image,
- a finite automaton,
- a set of rules,
- a blueprint of certain device,
- ...

Features of optimization problems

- Discrete (combinatorial) vs. continuous vs. mixed optimization.
- Constrained vs. unconstrained optimization.
- None (feasibility problems) vs. single vs. many objectives.
- Deterministic vs. stochastic optimization.
- Static vs. time-dependent optimization.

E.g., continuous constrained subclass may have other features:

- Convex vs. non-convex optimization.
- Smooth vs. non-smooth optimization.
- ...

Taxonomy of single-objective deterministic optimization

Part of one possible taxonomy:

- Discrete
  - Integer Programming, Combinatorial Optimization, . . .
- Continuous
  - Unconstrained
    - Nonlinear least squares, Nonlinear equations, Nondifferentiable optimization, Global optimization, . . .
  - Constrained
    - Bound constrained, Nondifferentiable optimization, Global optimization, . . .
    - Linearly constrained
      - Linear programming, Quadratic programming
    - Nonlinear programming
      - Semidefinite programming, Second-order cone programming, Quadratically-constrained quadratic programming, Mixed integer nonlinear programming, . . .

Black-box optimization

The more we know about the problem, the narrower class of tasks we want to solve, and the better algorithm we can make for them. If we know nothing about the problem. . .

Black-box optimization (BBO)

- The inner structure of the objective function f is unknown.
- Virtually *no assumptions can be taken as granted* when designing a BBO algorithm.
- BB algorithms are thus *widely applicable*
  - continuous, discrete, mixed
  - constrained, unconstrained
  - . . .
- But generally they have *lower performance* than algorithms using the right assumptions.
- *Swiss army knives:* you can do virtually everyting with them, but sometimes a hammer, or a needle would be better.

What can a BBO algorithm do?

- Sample (create) a candidate solution,
- check whether it is feasible, and
- evaluate it using the objective function.

Anything else (gradients? noise? . . .) must be estimated from the samples!

Features of optimization methods

Do they provably provide the optimal solution?

- **Exact methods**
  - ensure optimal solutions, but
  - are often tractable only for small problem instances.
- **Heuristics**
  - provide only approximations, but
  - use techniques that "usually" work quite well, even for larger instances.

How do they create the solution?

- **Constructive algorithms**
  - require discrete search space,
  - construct full solutions *incrementally*, and
  - must be able to *evaluate partial solutions*.
  - They are thus *not suitable for black-box optimization*.
- **Generative algorithms**
  - generate complete candidate *solutions as a whole*.
  - They are *suitable for black-box optimization*, since only complete solutions need to be evaluated.

Optimization algorithms you may have heard of

Methods for discrete spaces:

- Complete (enumerative) search
- Graph-based: depth-, breadth-, best-first search, greedy search, A^*
- Decomposition-based: divide and conquer, dynamic programming, branch and bound

Methods for continuous spaces:

- Random search
- Gradient methods, simplex method for linear programming, trust-region methods
- Local search, Nelder-Mead downhill simplex search

Neighborhood, local optimum

The **neighborhood** of a point $x \in \mathcal{S}$:

$$N(x, d) = \{y \in \mathcal{S} | dist(x, y) \leq d\} \tag{2}$$

Measure of the **distance between points x and y**: $\mathcal{S} \times \mathcal{S} \rightarrow \mathcal{R}$:

- Binary space: Hamming distance, ...
- Real space: Euclidean, Manhattan (City-block), Mahalanobis, ...
- Matrices: Amari, ...
- In general: number of applications of some operator that would transform x into y in $dist(x, y)$ steps.

Local optimum:

- Point x is a *local optimum*, if $f(x) \leq f(y)$ for all points $y \in N(x, d)$ for some positive d.
- Small finite neighborhood (or the knowledge of derivatives) allows for validation of local optimality of x.

Global optimum:

- Point x is a *global optimum*, if $f(x) \leq f(y)$ for all points $y \in \mathcal{F}$.

Local Search, Hill-Climbing

Algorithm 1: LS with First-improving Strategy

```
begin
    x ← Initialize()
    while not TerminationCondition() do
        y ← Perturb(x)
        if BetterThan(y, x) then
            x ← y
```

Features:

- usually stochastic, possibly deterministic, applicable in discrete and continuous spaces

Algorithm 2: LS with Best-improving Strategy

```
begin
    x ← Initialize()
    while not TerminationCondition() do
        y ← BestOfNeighborhood(N(x, d))
        if BetterThan(y, x) then
            x ← y
```

Features:

- deterministic, applicable only in discrete spaces, or in descretized real-valued spaces, where $N(x, d)$ is finite and small

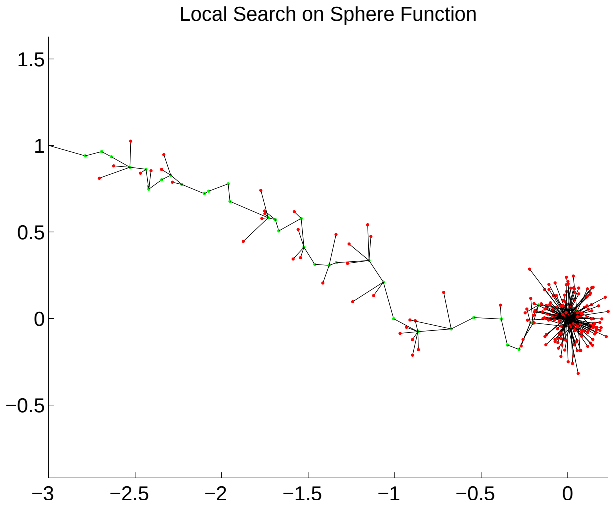
The influence of the neighborhood size:

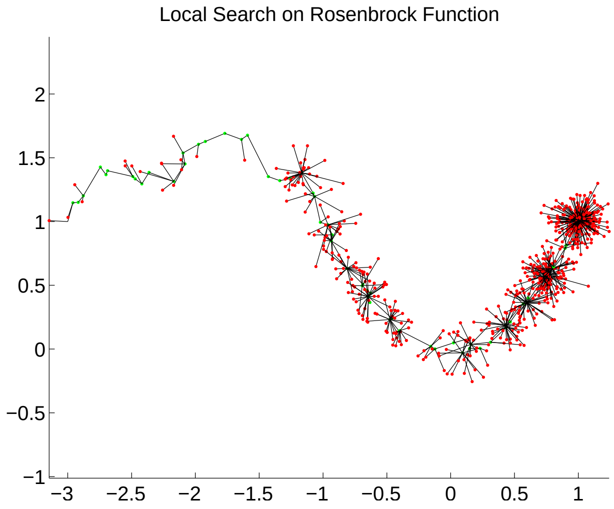
- Small neighborhood: fast search, huge risk of getting stuck in local optimum (in zero neghborhood, the same point is generated over and over)
- Large neighborhood: lower risk of getting stuck in LO, but the efficiency drops. If $N(x, d) = \mathcal{S}$, the search degrades to
  - random search in case of first-improving strategy, or to
  - exhaustive search in case of best-improving strategy.

Local Search Demo

LS with first-improving strategy:

- Neighborhood given by Gaussian distribution.
- Neighborhood is static during the whole algorithm run.





Rosenbrock's Optimization Algorithm

Described in [Ros60]:

Algorithm 3: Rosenbrock's Algorithm

Input: $\alpha > 1$, $\beta \in (0, 1)$

begin
  $x \leftarrow$ `Initialize()`; $x_o \leftarrow x$
  $\{e_1, \dots, e_D\} \leftarrow$ `InitOrtBasis()`
  $\{d_1, \dots, d_D\} \leftarrow$ `InitMultipliers()`
  while not `TerminationCondition()` **do**
    for i=1...D **do**
      $y \leftarrow x + d_i e_i$
      if `BetterThan(y,x)` **then**
        $x \leftarrow y$
        $d_i \leftarrow \alpha \cdot d_i$
      else
        $d_i \leftarrow -\beta \cdot d_i$
    if `AtLeastOneSuccInAllDirs()` **and** `AtLeastOneFailInAllDirs()` **then**
      $\{e_1, \dots, e_D\} \leftarrow$ `UpdOrtBasis`$(x - x_o)$
      $x_o \leftarrow x$

Features:

- D candidates generated each iteration
- neighborhood in the form of a pattern
- adaptive neighborhood parameters
  - distances
  - directions

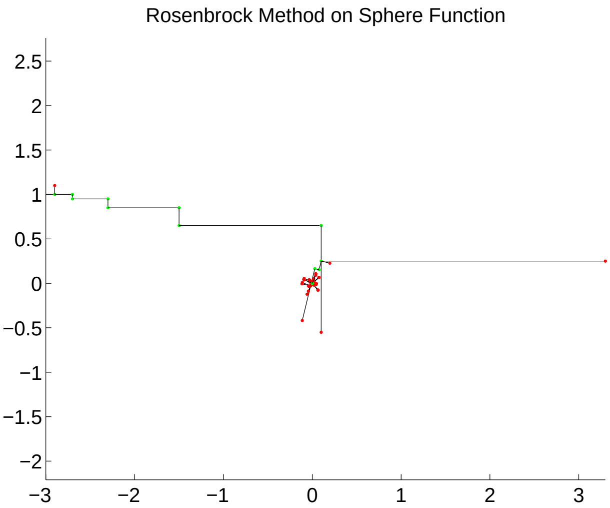
DEMO

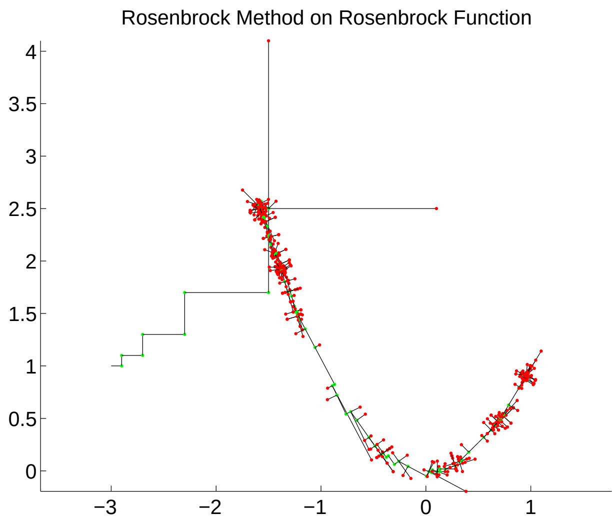
[Ros60] H. H. Rosenbrock. An automatic method for finding the greatest or least value of a function. *The Computer Journal*, 3(3):175–184, March 1960.

Rosenbrock's Algorithm Demo

Rosenbrock's algorithm:

- Neighborhood given by a pattern.
- Neighborhood is adaptive (directions and lengths of the pattern).





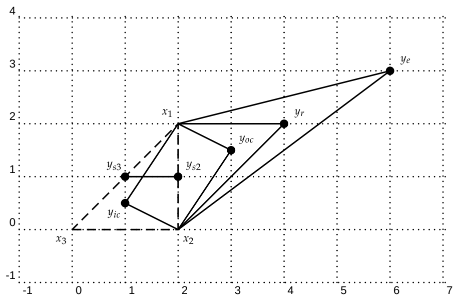
Nelder-Mead Simplex Search

Simplex downhill search (amoeba) [NM65]:

Algorithm 4: Nelder-Mead Simplex Algorithm

begin
  $(x_1, \dots, x_{D+1}) \leftarrow$ `InitSimplex()`
  so that $f(x_1) \le f(x_2) \le \dots \le f(x_{D+1})$

  while not `TerminationCondition()` **do**
    $\bar{x} \leftarrow \frac{1}{D} \sum_{d=1}^{D} x_d$
    $y_r \leftarrow \bar{x} + \rho(\bar{x} - x_{D+1})$
    if `BetterThan(`y_r, x_D`)` **then** $x_{D+1} \leftarrow y_r$
    if `BetterThan(`y_r, x_1`)` **then**
      $y_e \leftarrow \bar{x} + \chi(x_r - \bar{x})$
      if `BetterThan(`y_e, y_r`)` **then** $x_{D+1} \leftarrow y_e$; `Continue`
    if not `BetterThan(`y_r, x_D`)` **then**
      if `BetterThan(`y_r, x_{D+1}`)` **then**
        $y_{oc} \leftarrow \bar{x} + \gamma(x_r - \bar{x})$
        if `BetterThan(`y_{oc}, y_r`)` **then** $x_{D+1} \leftarrow y_{oc}$;
        `Continue`
      else
        $y_{ic} \leftarrow \bar{x} - \gamma(\bar{x} - x_{D+1})$
        if `BetterThan(`y_{ic}, x_{D+1}`)` **then** $x_{D+1} \leftarrow y_{ic}$;
        `Continue`
    $y_{si} \leftarrow x_1 + \sigma(x_i - x_1), \quad i \in 2, \dots, D+1$
    `MakeSimplex(`$x_1, y_{s2}, \dots, y_{s(D+1)}$`)`



Features:

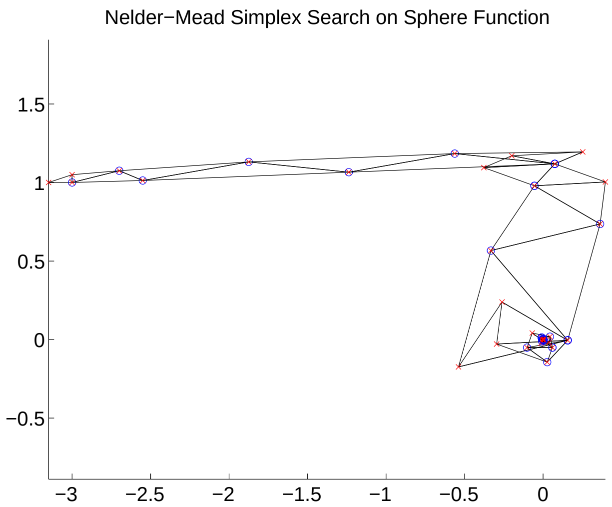
- universal algorithm for BBO in real space
- in $\mathcal{R}^D$ maintains a *simplex* of $D+1$ points
- neighborhood in the form of a pattern (reflection, extension, contraction, reduction)
- static neighborhood parameters!
- adaptivity caused by *changing relationships* among solution vectors!
- slow convergence, for low D only

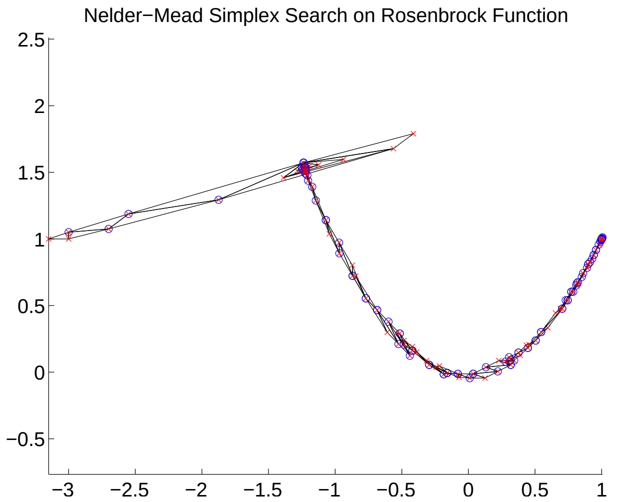
[NM65] J.A. Nelder and R. Mead. A simplex method for function minimization. *The Computer Journal*, 7(4):308–313, 1965.

Nelder-Mead Simplex Demo

Nelder-Mead downhill simplex algorithm:

- Neighborhood is given by a set of operations applied to a set of points.
- Neighborhood is adaptive due to changes in the set of points.





Lessons Learned

- To *search for the optimum*, the algorithm must *maintain at least one base solution* (fullfiled by all algorithms).
- To *adapt to the changing position in the environment* during the search, the algorithm must either
  - *adapt the neighborhood (model)* structure or parameters (as done in Rosenbrock method), or
  - *adapt more than 1 base solutions* (as done in Nelder-Mead method), or
  - both of them.
- The neighborhood
  - can be *finite* or *infinite*
  - can have a form of a *pattern* or a *probabilistic distribution*.
- Candidate solutions can be generated from the neighborhood of
  - one base vector (LS, Rosenbrock), or
  - all base vectors (Nelder-Mead), or
  - some of the base vectors (requires *selection* operator).

The Problem of Local Optimum

All the above LS algorithms often **get stuck in the neighborhood of a local optimum!**

How to escape from local optimum?

1. Run the optimization algorithm from a different initial point.
   - restarting, iterated local search, ...
2. Introduce memory and do not search in already visited places.
   - taboo search
3. Make the algorithm stochastic.
   - stochastic hill-climber, simulated annealing, evolutionary algorithms, swarm intelligence, ...
4. Perform the search in several places in the same time.
   - population-based optimization algorithms (Nelder-Mead, evolutionary algorithms, swarm intelligence, ...)

Taboo Search

Algorithm 5: Taboo Search

```
begin
    x ← Initialize()
    y ← x
    M ← ∅
    while not TerminationCondition() do
        y ← BestOfNeighborhood(N(y, d) − M)
        M ← UpdateMemory(M, y)
        if BetterThan(y, x) then
            x ← y
```

Meaning of symbols:

- M — memory holding already visited points that become taboo.
- $N(\mathbf{y}, d) - M$ — set of states which would arise by taking back some of the previous decisions

Features:

- The canonical version of TS is based on LS with best-improving strategy.
- First-improving can be used as well.
- It is difficult to use in real domain, usable mainly in discrete spaces.

Stochastic Hill-Climber

Assuming minimization:

Algorithm 6: Stochastic Hill-Climber

```
begin
    x ← Initialize()
    while not TerminationCondition() do
        y ← Perturb(x)
        p = 1 / (1 + e^((f(y) − f(x)) / T))
        if rand() ≤ p then
            x ← y
```

Features:

- It is possible to move to a worse point *anytime*.
- T is the algorithm parameter and stays constant during the whole run.
- When T is low, we get local search with first-improving strategy
- When T is high, we get random search

Probability of accepting a new point y when $f(y) - f(x) = -13$:

T	$e^{-\frac{13}{T}}$	p
1	0.000	1.000
5	0.074	0.931
10	0.273	0.786
20	0.522	0.657
50	0.771	0.565
10^{10}	1.000	0.500

Probability of accepting a new point y when $T = 10$:

$f(y) - f(x)$	$e^{\frac{f(y)-f(x)}{10}}$	p
-27	0.067	0.937
-7	0.497	0.668
0	1.000	0.500
13	3.669	0.214
43	73.700	0.013

Simulated Annealing

Algorithm 7: Simulated Annealing

```
begin
    x ← Initialize()
    T ← Initialize()
    while not TerminationCondition() do
        y ← Perturb(x)
        if BetterThan(y,x) then
            x ← y
        else
            p = e^(−(f(y) − f(x)) / T)
            if rand() < p then
                x ← y
        if InterruptCondition() then
            T ← Cool(T)
```

Very similar to stochastic hill-climber

Main differences:

- If the new point y is better, it is *always* accepted.
- Function `Cool`(T) is the *cooling schedule*.
- SA changes the value of T during the run:
  - T is high at beginning: SA behaves like random search
  - T is low at the end: SA behaves like deterministic hill-climber

Issues:

- How to set up the initial tempereature T and the cooling schedule `Cool`(T)?
- How to set up the interrupt and termination condition?

Evolutionary Algorithms

Evolutionary algorithms

- are population-based counterpart of single-state local search methods (more robust w.r.t. getting stuck in LO).
- Inspired by
  - Mendel's theory of inheritance (transfer of traits from parents to children), and
  - Darwin's theory of evolution (random changes of individuals, and survival of the fittest).

Difference from a mere parallel hill-climber: candidate solutions affect the search of other candidates.

Originally, several distinct kinds of EAs existed:

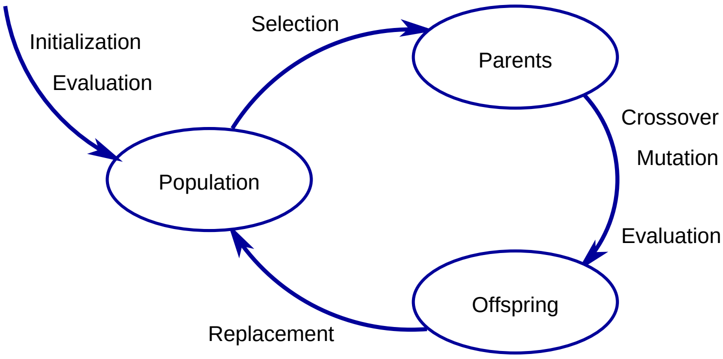
- **Evolutionary programming, EP** (Fogel, 1966): real numbers, state automatons
- **Evolutionary strategies, ES** (Rechenberg, Schwefel, 1973): real numbers
- **Genetic algorithms, GA** (Holland, 1975): binary or finite discrete representation
- **Genetic programming, GP** (Cramer, Koza, 1989): trees, programs

Currently, the focus is on emphasizing what they have in common, and on exchange of ideas among them.

Inspiration by biology

individual	a candidate solution
fitness	quality of an individual
fitness function (landscape)	objective function
population	a set of candidate solutions
selection	picking individuals based on their fitness
parents	individuals chosen by selection as sources of genetic material
children (offspring)	new individuals created by breeding
breeding	the process of creating children from a population of parents
mutation	perturbation of an individual; asexual breeding
recombination or crossover	producing one or more children from two or more parents; sexual breeding
genotype	an individual's data structure as used during breeding
phenotype	the meaning of genotype, how is the genotype interpreted by the fitness function
chromosome	a special type of genotype – fixed-length vector
gene	a variable or a set of variables in the genotype
allele	a particular value of gene
generation	one cycle of fitness assessment, breeding, and replacement

Evolutionary cycle



Algorithm

Algorithm 8: Evolutionary Algorithm

```
begin
    X ← InitializePopulation()
    f ← Evaluate(X)
    x_BSF, f_BSF ← UpdateBSF(X, f)
    while not TerminationCondition() do
        X_N ← Breed(X, f)                    // e.g., using the pipeline below
        f_N ← Evaluate(X_N)
        x_BSF, f_BSF ← UpdateBSF(X_N, f_N)
        X, f ← Join(X, f, X_N, f_N)          // aka ''replacement strategy''
    return x_BSF, f_BSF
```

BSF : Best So Far

Algorithm 9: Canonical GA Breeding Pipeline

```
begin
    X_S ← SelectParents(X, f)
    X_N ← Crossover(X_S)
    X_N ← Mutate(X_N)
    return X_N
```

Other different `Breed()` pipelines can be pluged in the EA.

Initialization

Initialization is a process of creating individuals from which the search shall start.

- **Random:**
  - No prior knowledge about the characteristics of the final solution.
  - No part of the search space is preferred.
- **Informed:**
  - Requires prior knowledge about where in the search space the solution can be.
  - You can directly *seed* (part of) the population by solutions you already have.
  - It can make the computation faster, but *it can unrecoverably direct the EA to a suboptimal solution!*
- **Pre-optimization:**
  - (Some of) the population members can be set to the results of several (probably short) runs of other optimization algorithms.

Selection

Selection is the process of choosing which population members shall become parents.

- Usually, the better the individual, the higher chance of being chosen.
- A single individual may be chosen more than once; better individuals influence more children.

Selection types:

- **No selection:** all population members become parents.
- **Truncation selection:** the best n % of the population become parents.
- **Tournament selection:** the set of parents is composed of the winners of small tournaments (choose n individuals uniformly, and pass the best of them as one of the parent).
- **Uniform selection:** each population member has the same chance of becoming a parent.
- **Fitness-proportional selection:** the probability of being chosen is proportional to the individual's fitness.
- **Rank-based selection:** the probability of being chosen is proportional to the rank of the individual in population (when sorted by fitness).
- ...

Mutation

Mutation makes small changes to the population members (usually, it iteratively applies *perturbation* to each individual). It

- promotes the population diversity,
- minimizes the chance of loosing a useful part of genetic code, and
- performs a local search around individuals.

Selection + mutation:

- Even this mere combination may be a powerfull optimizer.
- It differs from several local optimizers run in parallel.

Types of mutation:

- For binary representations: bit-flip mutation
- For vectors of real numbers: Gaussian mutation, ...
- For permutations: 1-opt, 2-opt, ...
- ...

Crossover

Crossover (xover) combines the traits of 2 or more chosen parents.

- Hypothesis: by combining features of 2 (or more) good individuals we can maybe get even better solution.
- Crossover usually creates children in unexplored parts of the search space, i.e., promotes diversity.

Types of crossover:

- For vector representations: 1-point, 2-point, uniform
- For vectors of real numbers: geometric xover, simulated binary xover, parent-centric xover, ...
- For permutations: partially matched xover, edge-recombination xover, ...
- ...

Replacement

Replacement strategy (the `join()` operation) implements the *survival of the fittest* principle. It determines which of the members of the old population and which new children shall survive to the next generation.
Types of replacement strategies:

- **Generational:** the old population is thrown away, new population is chosen just from the children.
- **Steady-state:** members of the old population may survive to the next generation, together with some children.
- Similar principles as for selection can be applied.

Why EAs?

EAs are popular because they are

- easy to implement,
- robust w.r.t. problem formulations, and
- less likely to end up in a local optimum.

Some of the application areas:

- automated control
- planning
- scheduling
- resource allocation
- design and tuning of neural networks
- signal and image processing
- marketing
- ...

Evolutionary algorithms are best applied in areas where we have no idea about the final solution. Then we are often surprised what they come up with.

Learning outcomes: Prerequisities

Before entering this course, a student shall be able to

- define an optimization task in mathematical terms; explain the notions of search space, objective function, constraints, etc.; and provide examples of optimization tasks;
- describe various subclasses of optimization tasks and their characteristics;
- define exact methods, heuristics, and their differences;
- explain differences between constructive and generative algorithms and give examples of both.

Learning outcomes: This lecture

After this lecture, a student shall be able to

- describe and explain what makes real-world search and optimization problems hard;
- describe black-box optimization and the limitations it imposes on optimization algorithms;
- define a neighborhood and explain its importance to local search methods;
- describe a hill-climbing algorithm in the form of pseudocode; and implement it in a chosen programming language;
- explain the difference between best-improving and first-improving strategy; and describe differences in the behaviour of the resulting algorithm;
- enumerate and explain the methods for increasing the chances to find the global optimum;
- explain the main difference between single-state and population-based methods; and name the benefits of using a population;
- describe a simple EA and its main components; and implement it in a chosen programming language.