Population-based optimisation methods

Differential Evolution and **Swarm Intelligence** (Artificial Bee Colony, Particle Swarm, Ant Colony Optimization)

Marco S. Nobile, Ph.D. – nobile@disco.unimib.it

University of Milano-Bicocca, Department of Informatics, Systems and Communication

PhD program in Computer Science

Outline

- Recap on optimization problems, local search, genetic algorithms
- Differential Evolution (DE)
- Variants of basic DE
- Self-adaptive DE (JADE, SHADE, DISH)
- Swarm Intelligence
- Particle Swarm Optimization (PSO)
- Variants of PSO
- Fuzzy Self-Tuning PSO
- Artificial Bee Colony
- Ant Colony Optimization

Optimization problems

- Given an objective (or fitness) function f and a space of feasible solutions $\mathcal{S}$, we want to identify a solution $\mathbf{o} \in \mathcal{S}$ such that

$$f(\mathbf{o}) \leq f(\mathbf{x}), \qquad \forall\, \mathbf{x} \neq \mathbf{o}$$

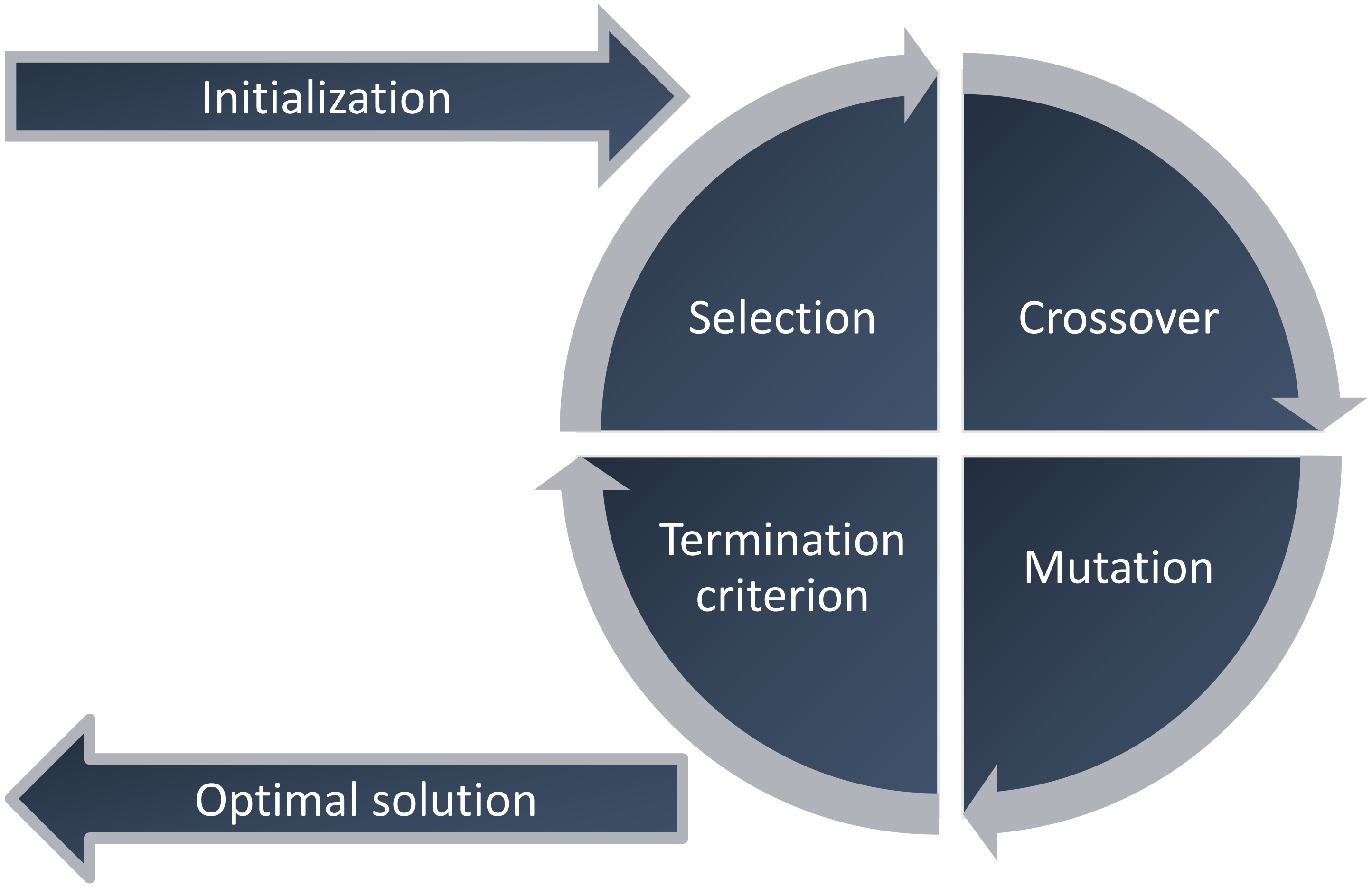
- The fitness landscape induced by the fitness function can be noisy, multi-modal, rugged, not convex, etc.
  - «Local search» (e.g., hill climbing, gradient descent) cannot be used: it gets trapped by local minima
  - **«Global search» population-based meta-heuristics** can be used (Global Optimization, GO)

- Genetic Algorithms (GA): evolutionary meta-heuristic for GO
  - Based on selection + crossover + mutation
  - Created for combinatorial optimization (e.g., binary) can be extended to real valued problems

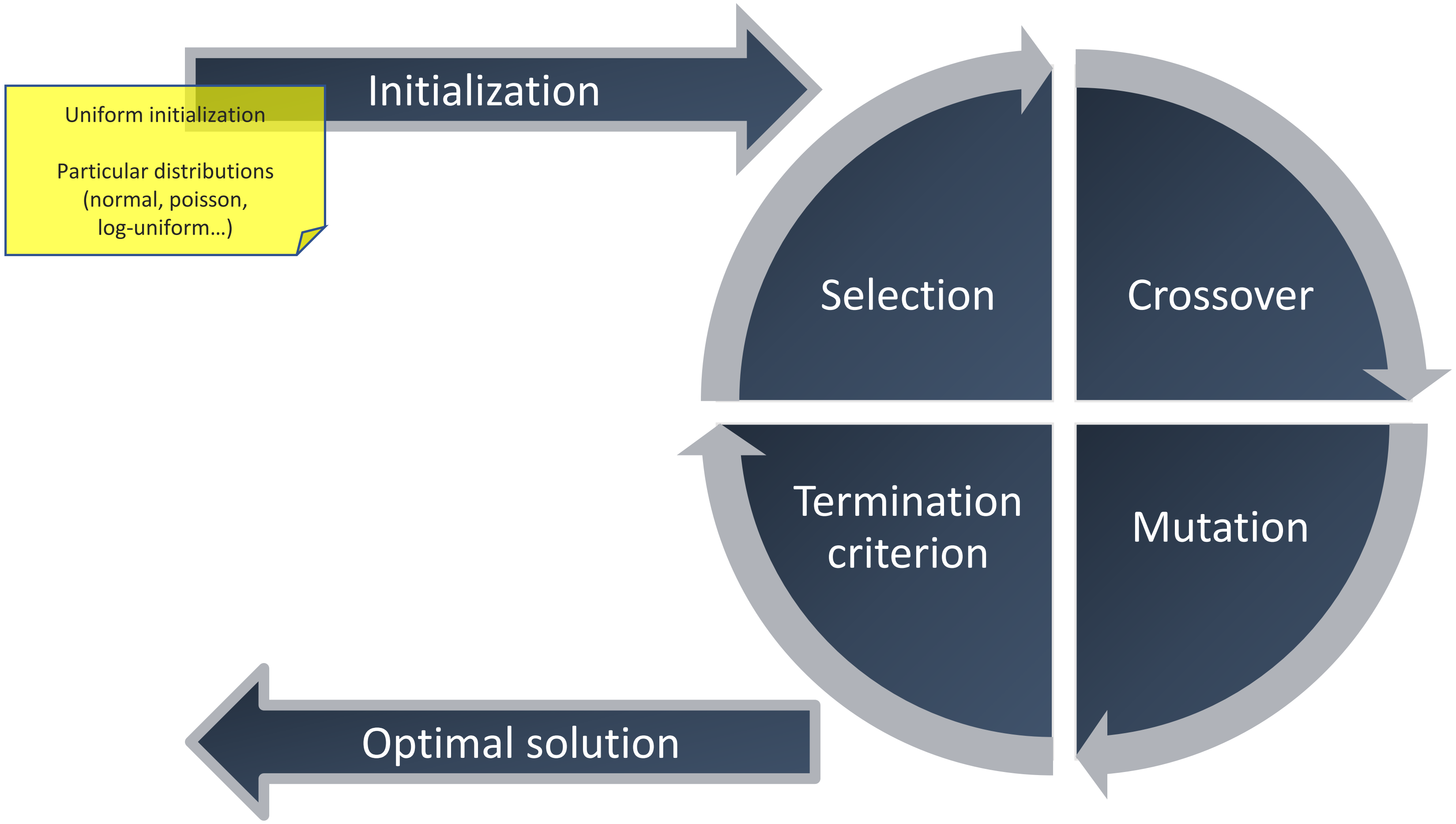
Additional definitions

- From now on, we denote by
  - D is the number of **dimensions of the search space** $\mathcal{S}$ (i.e., variables of the problem)
  - $\mathcal{P}$ denotes a **population** of (randomly initialized) candidate solutions
  - N is the number of **candidate solutions** in $\mathcal{P}$ (i.e., population size)
  - $\mathbf{x}_i \in \mathbb{R}^D$ denotes the i-th **candidate solution** in $\mathcal{P}$
  - $\mathbf{x}_i^G \in \mathbb{R}^D$ denotes the i-th candidate solution in $\mathcal{P}$ during **generation** G
  - $x_{i,d}^G \in \mathbb{R}$ denotes the value of the d-th **component** of the i-th candidate solution during generation G
  - $f\colon \mathbf{x} \subseteq \mathbb{R}^D \to \mathbb{R}$ denotes the **fitness function** to be optimized

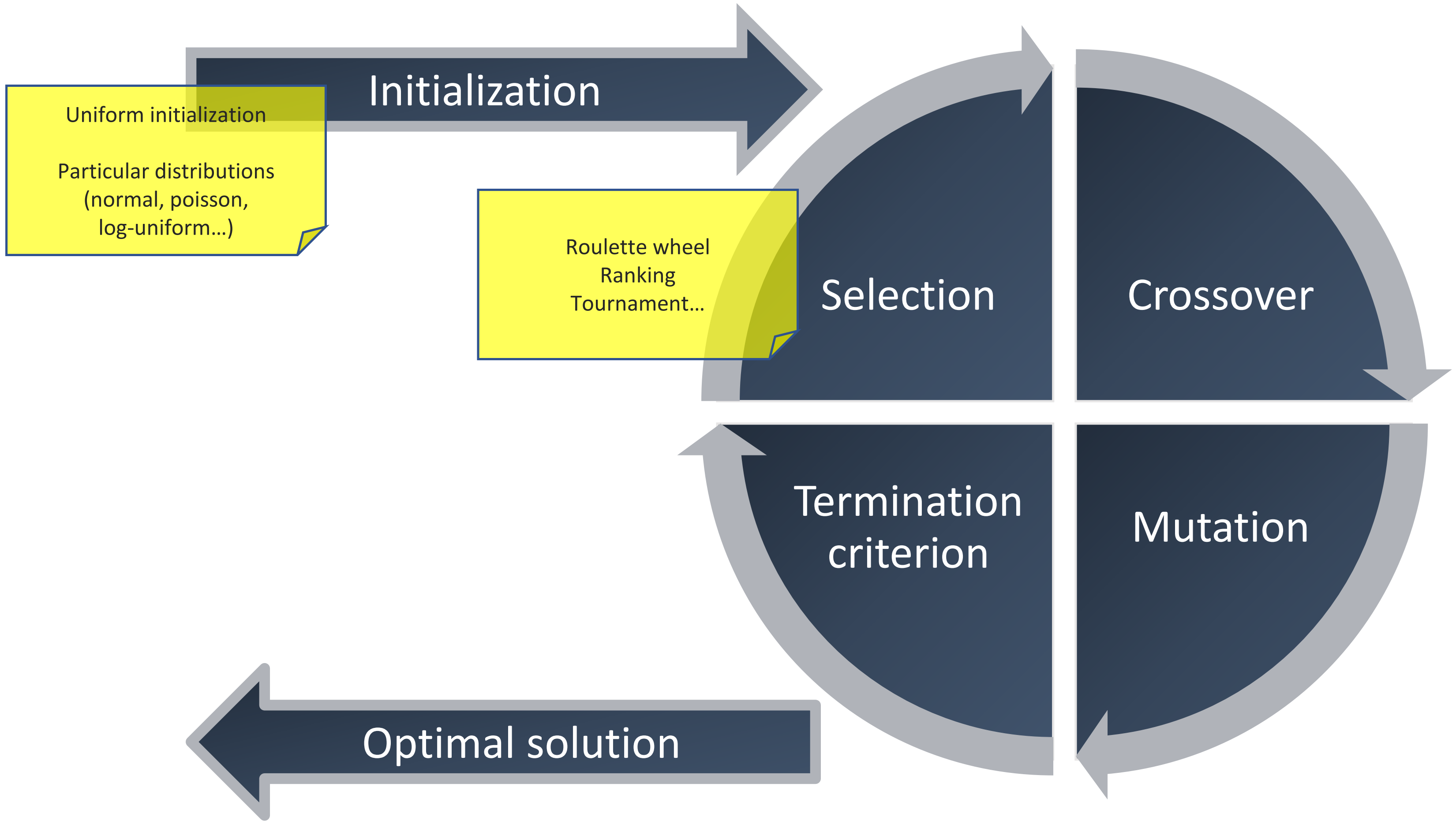
Genetic Algorithms in a nutshell



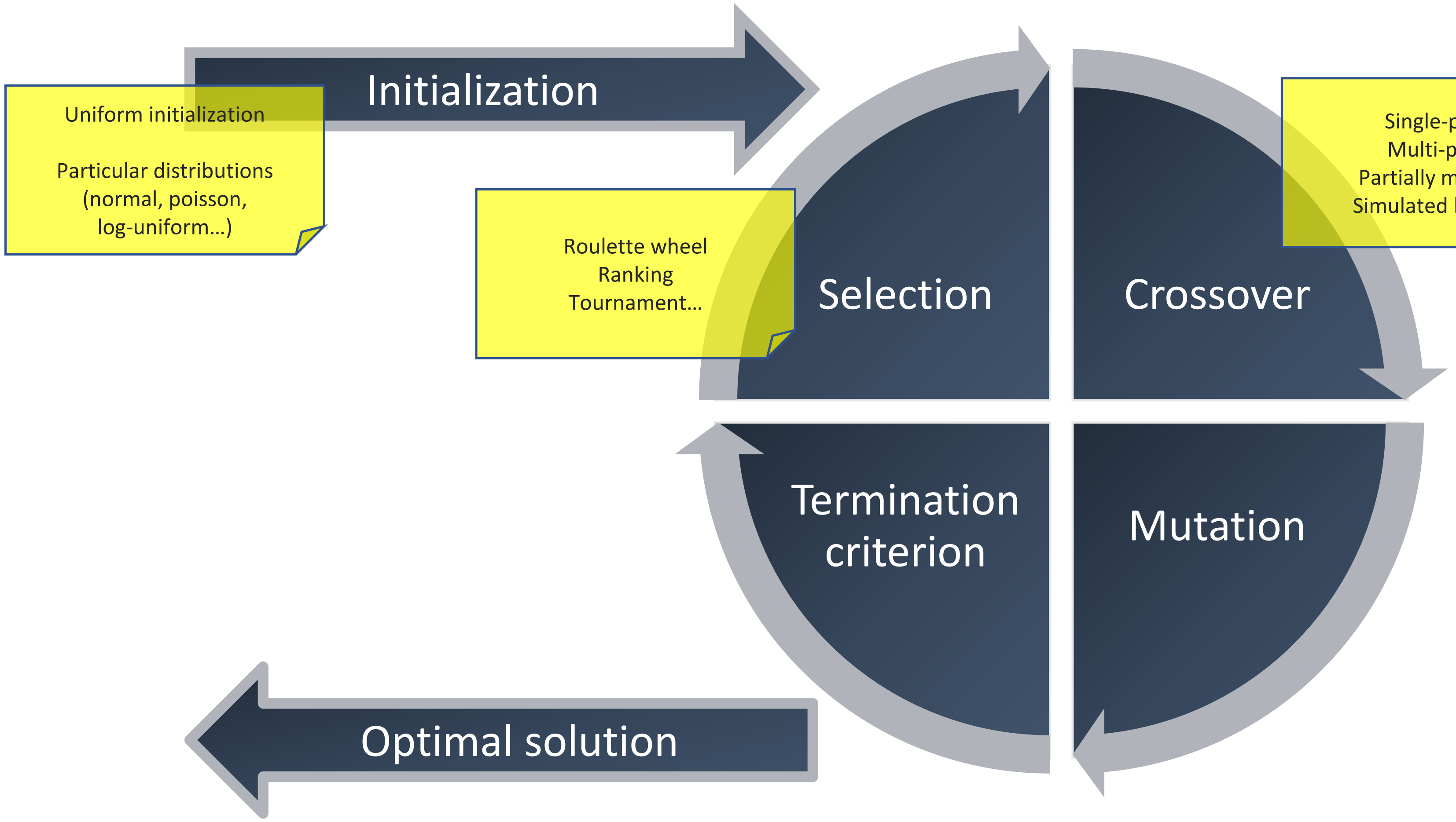
Genetic Algorithms in a nutshell



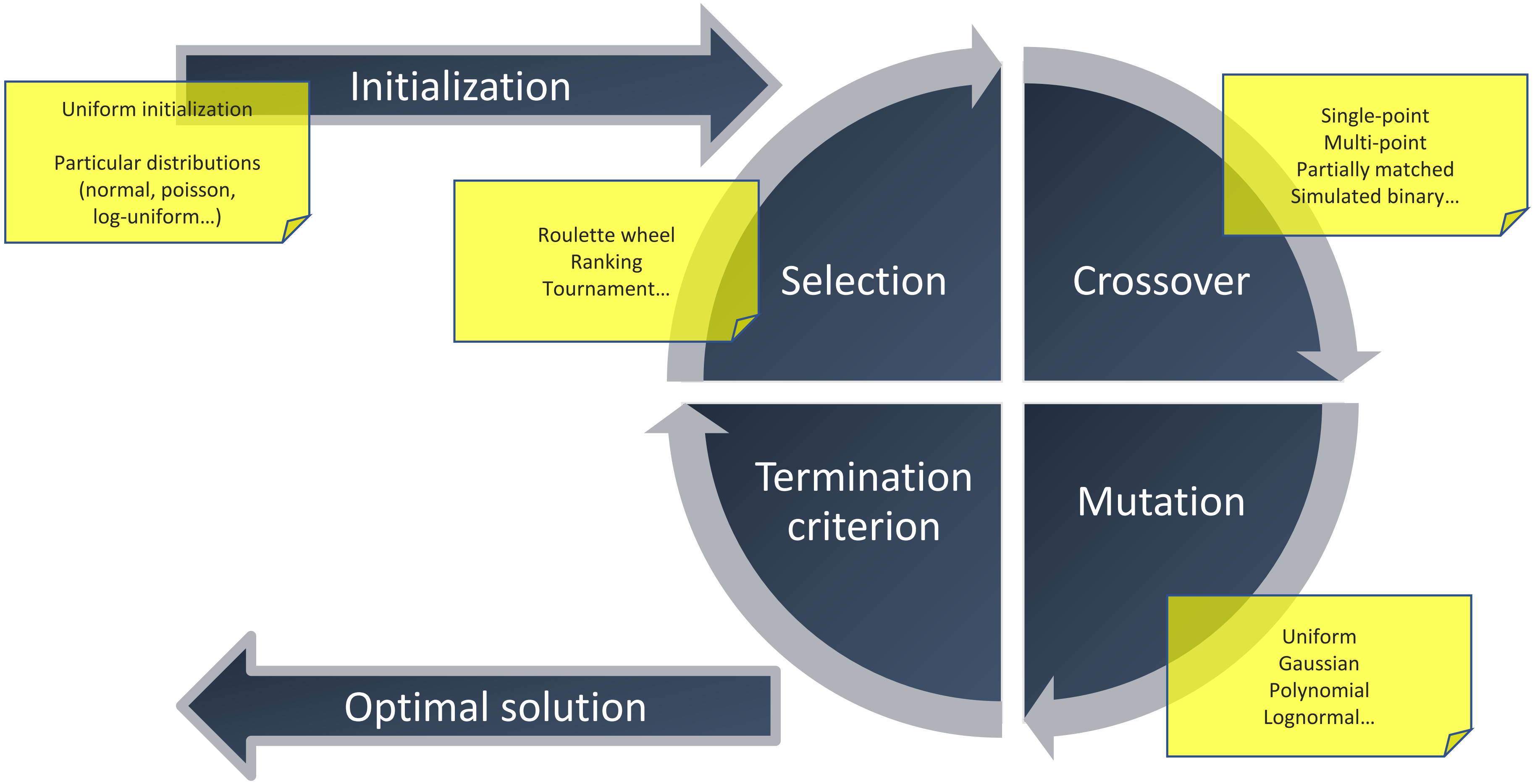
Genetic Algorithms in a nutshell



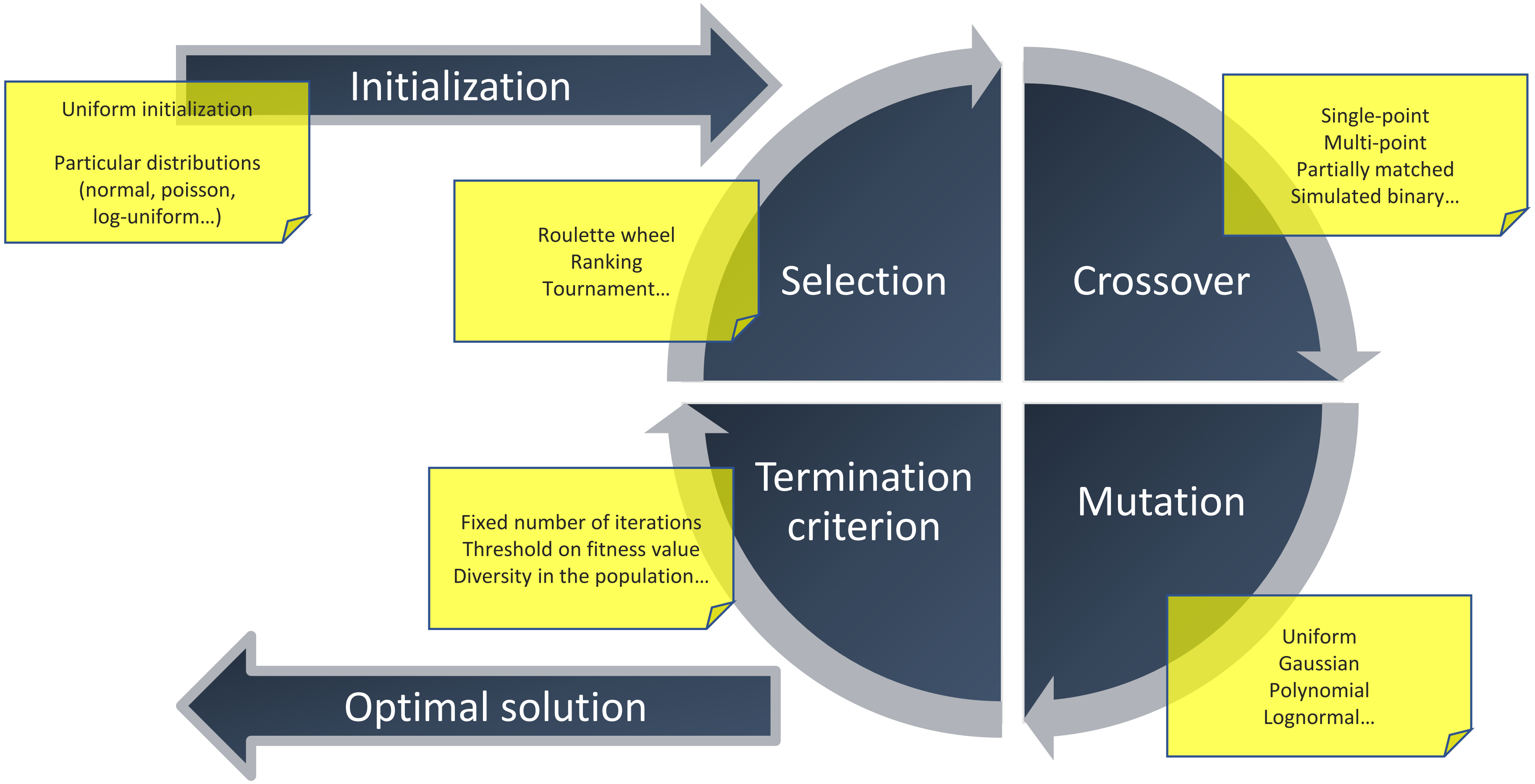
Genetic Algorithms in a nutshell



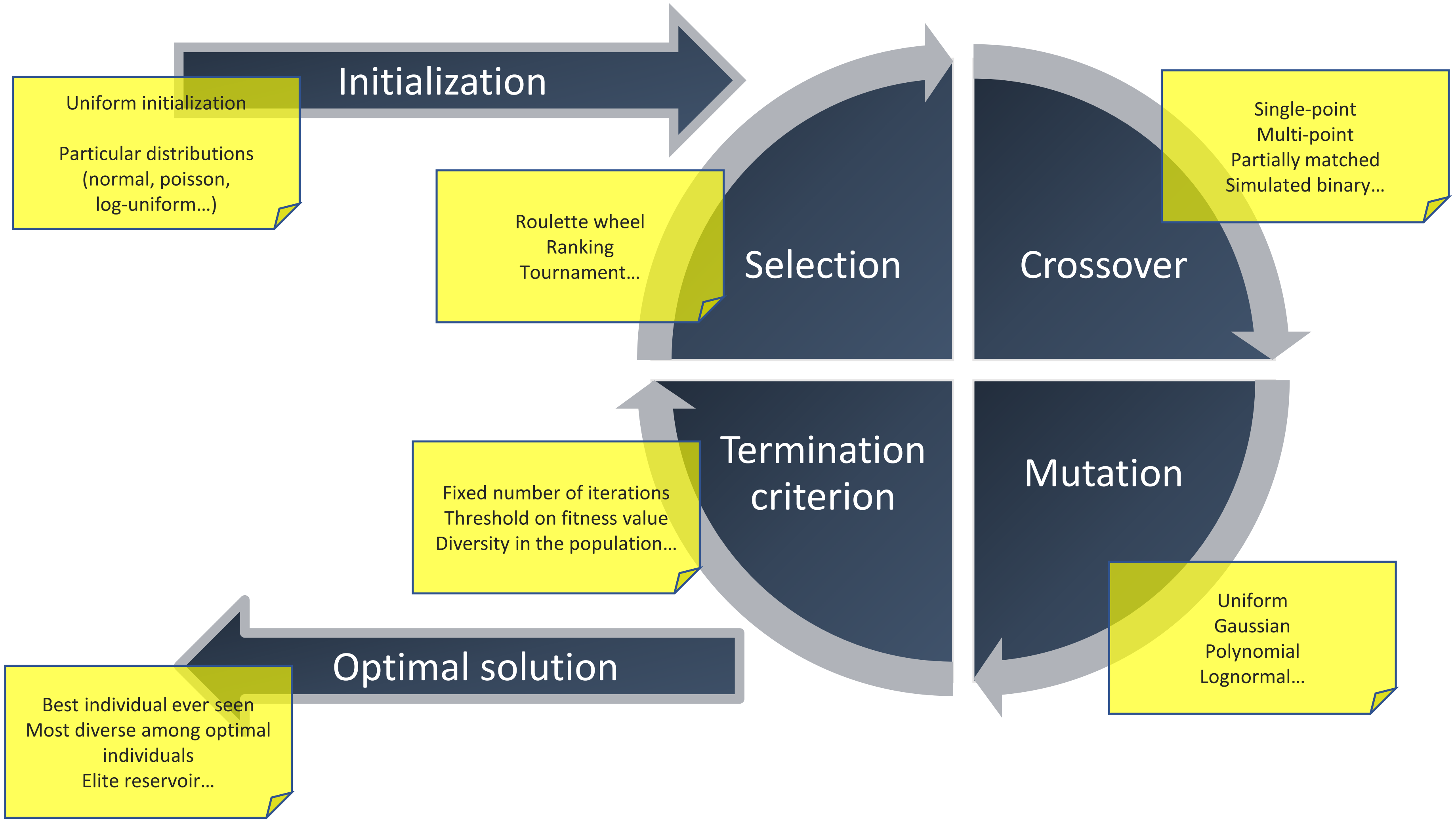
Genetic Algorithms in a nutshell



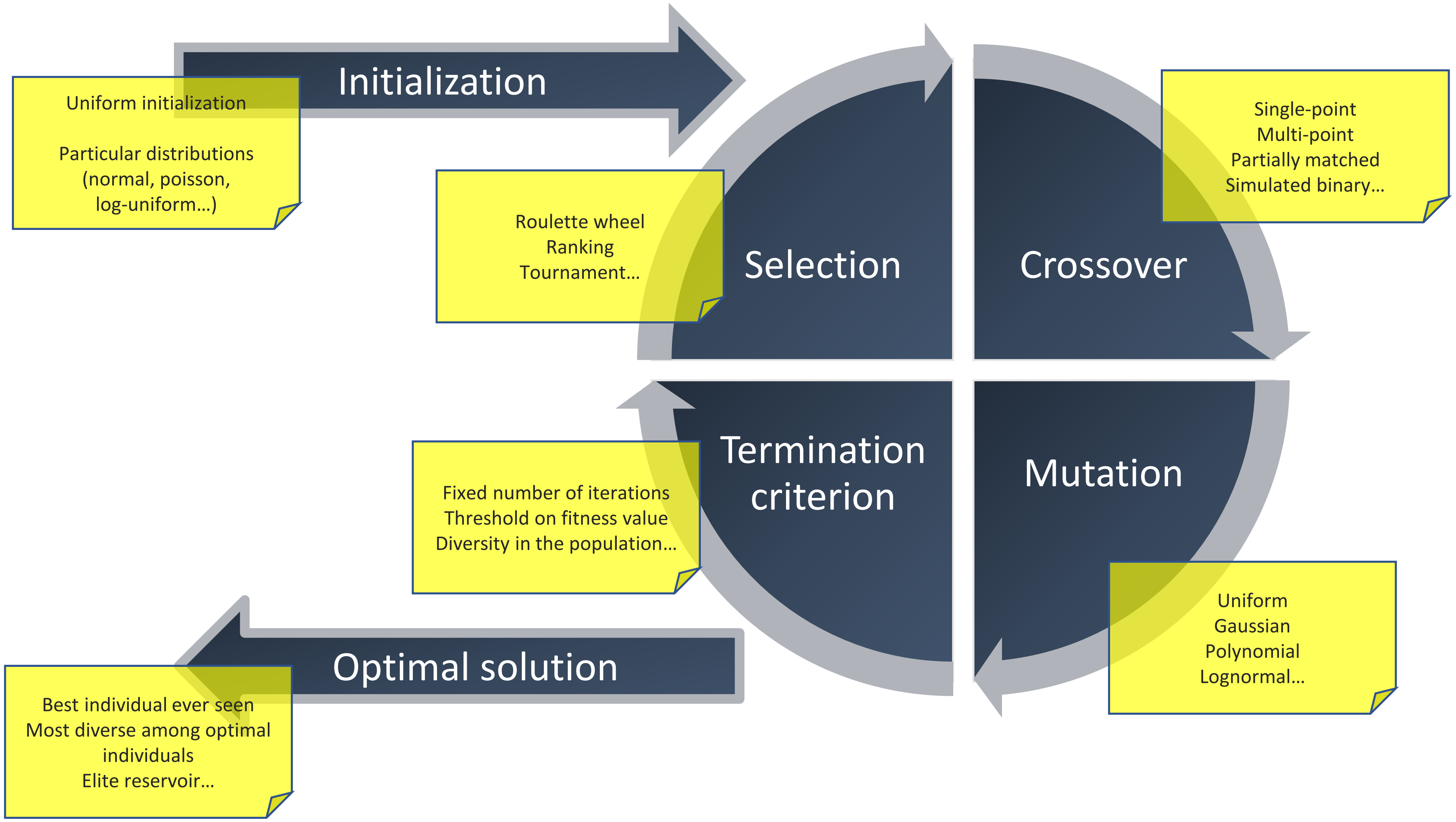
Genetic Algorithms in a nutshell



Genetic Algorithms in a nutshell



Genetic Algorithms in a nutshell

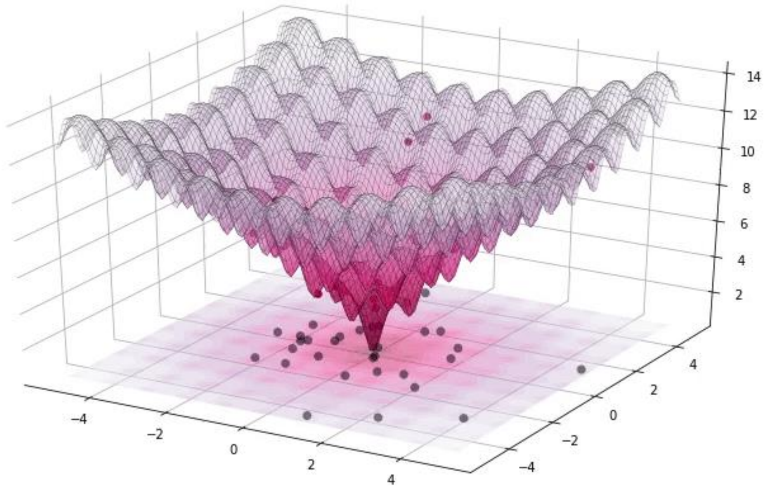


Hyper-parameters

- Crossover probability (and crowding degree)
- Mutation probability (and crowding degree)
- Tournament size
- Population size
- Iterations
- Threshold for stopping criterion

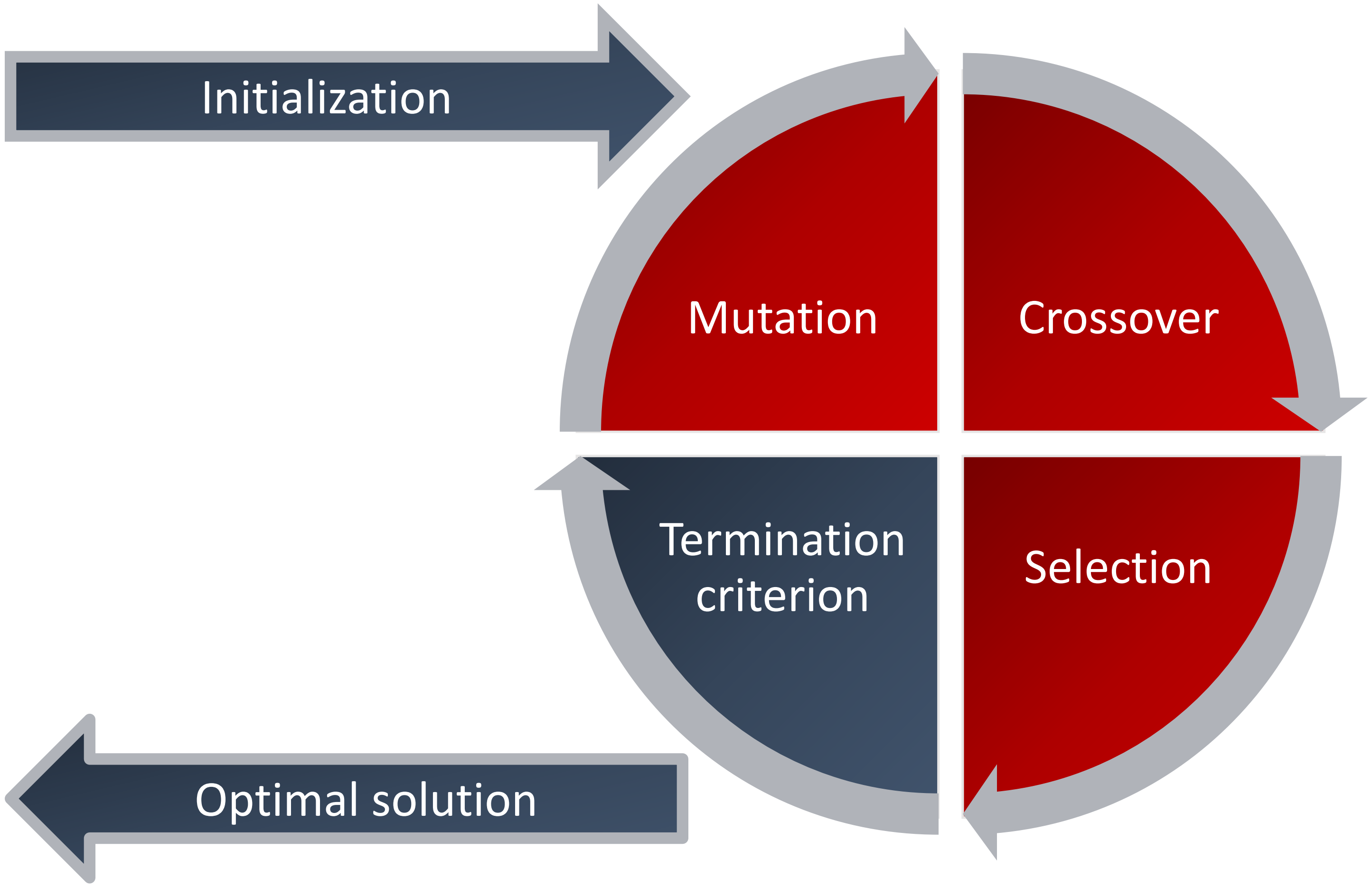
Differential Evolution

- **Differential Evolution** (DE) was introduced by Storn and Price in 1997
  - [Storn and Price, J Glob Optim 1997]
  - Defined «a parallel direct search method based on parameters vectors» for **real-valued GO**
  - **Evolutionary Computation (EC) approach**: a population of solutions «evolves» throughout the generations
  - Main difference wrt GAs: the scheme for the **generation of new individuals**, i.e., «the generation of new parameter vectors by adding a **weighted difference vector** – between two population members – to a **third** member»
  - Second difference wrt GAs is the **selection**: if the offspring vector yields a better objective function than the parent then the newly generated individual **replaces the parent**

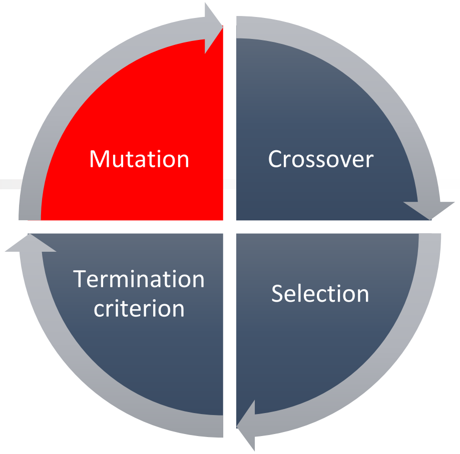


DE optimizing a 2D Ackley benchmark function
(figure by Pablormier, CC BY-SA 4.0)

Differential Evolution in a nutshell



Differential Evolution – Mutation



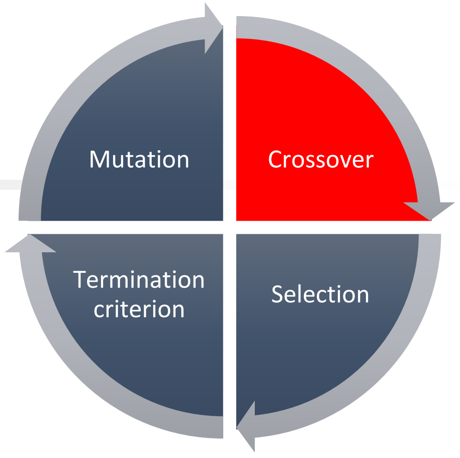
- Select one «designated» candidate solution $\mathbf{x}_i^G$
- Select **three** further random candidate solutions $\mathbf{a}, \mathbf{b}, \mathbf{c}$
- In the classic version of DE $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{x}_i$ are **distinct**

- The so-called **«donor» vector** $\mathbf{v}_i$ is calculated as

$$\mathbf{v}_i = \mathbf{a} + F \cdot (\mathbf{b} - \mathbf{c})$$

- $F \in [0,2]$ is called the **mutation factor**
- The operation is also known as **differential mutation**

Differential Evolution – Crossover



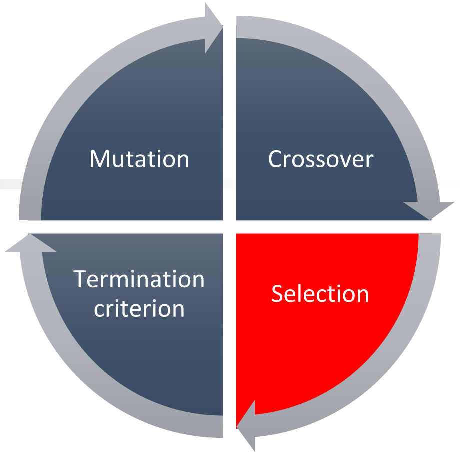
- The designated solution $\mathbf{x}_i^G$ and the donor vector $\mathbf{v}_i$ are **recombined** into a **«trial» vector $\mathbf{u}_i$**
  - The trial vector is mainly created using the elements of the designated solution $\mathbf{x}_i^G$
  - Elements of the donor vector $\mathbf{v}_i$ are selected for the trial vector with a **crossover probability** $\mathrm{CR} \in [0,1]$

$$u_{i,d} = \begin{cases} v_{i,d} & \text{if } rnd_{i,d} < CR \vee I_{rnd} = d \\ x_{i,d}^G & \text{otherw}ise \end{cases}$$

for all $d = 1, \dots, D$

- In the formula
  - $rnd_{i,d}$ is a random number sampled with uniform distribution in $[0,1]$
  - I_{rnd} is a random integer number chosen from $[1, \dots, D]$
  - This is sometimes called «binomial crossover» (see [Qin et al., IEEE Tran Evol Comp 2009])

Differential Evolution – Selection



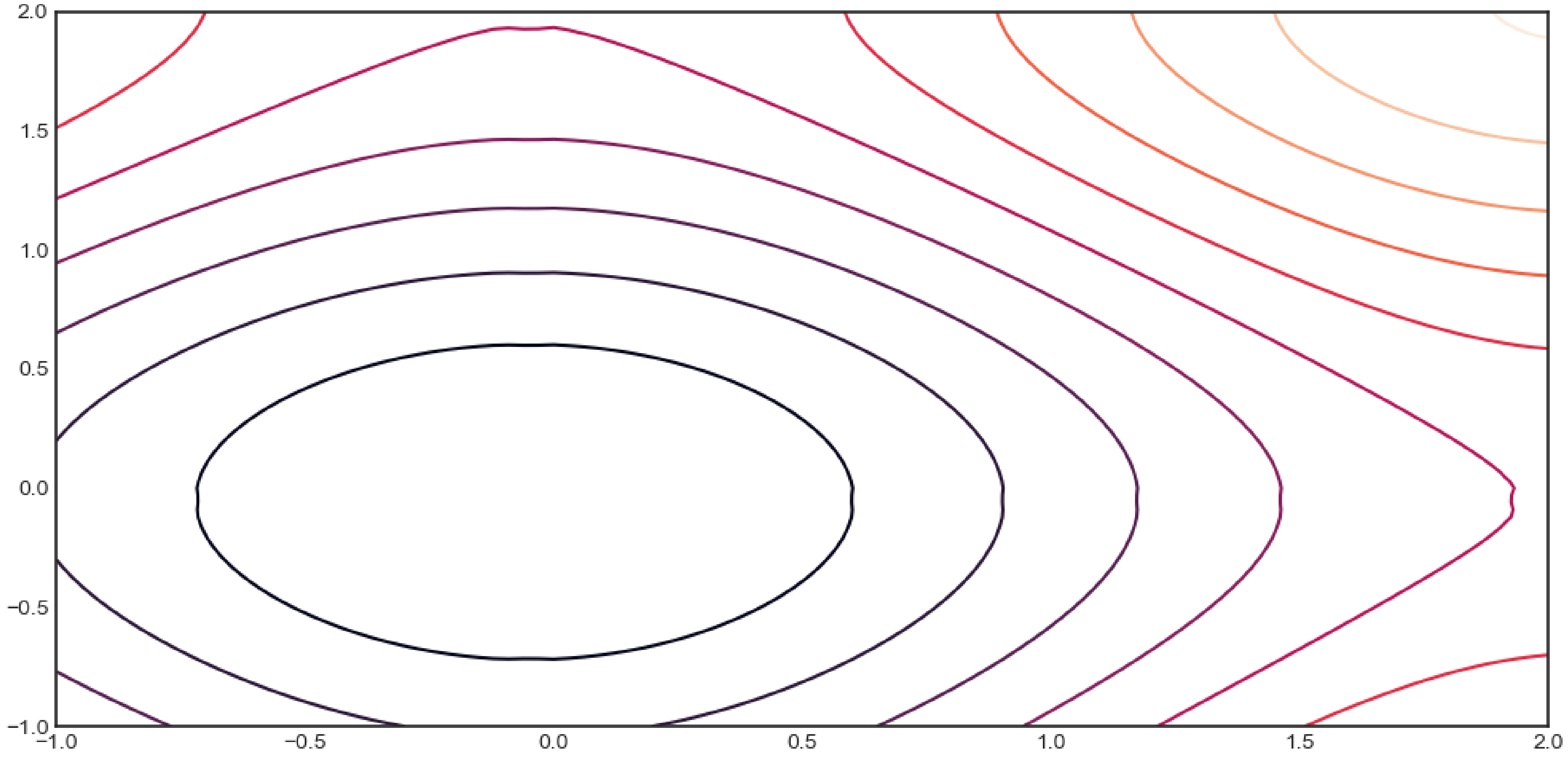
- The designated solution $\mathbf{x}_i$ is **compared** with the trial vector $\mathbf{u}_i$
  - **The best** of the two solutions (wrt the fitness function) is **selected**
  - E.g., in the case of minimization:

$$\mathbf{x}_i = \begin{cases} \mathbf{u}_i & \text{if } f(\mathbf{u}_i) < f(\mathbf{x}_i) \\ \mathbf{x}_i & \text{otherwise} \end{cases}$$

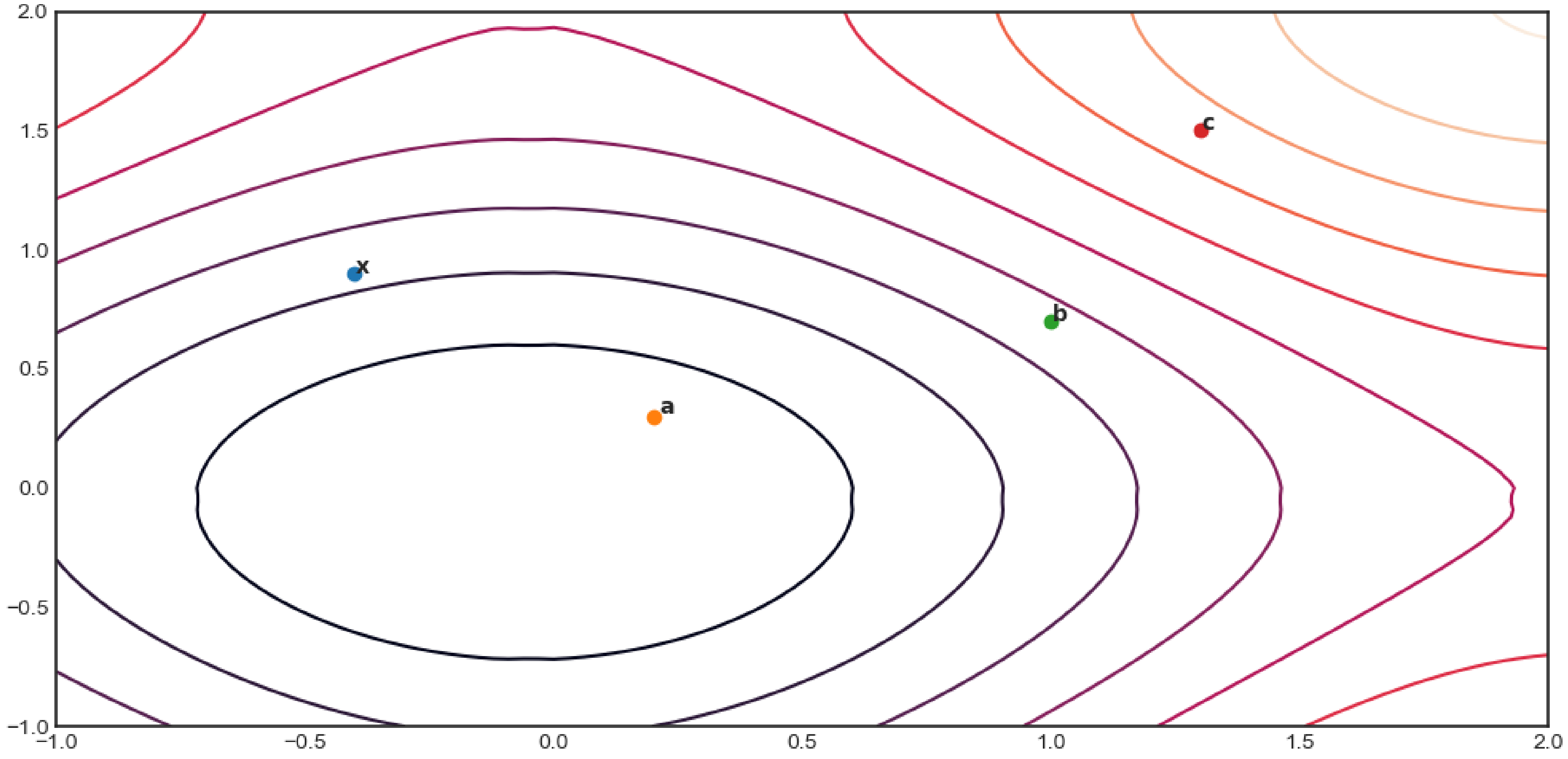
- The mutation-crossover-selection process is repeated for all $i = 1, \dots, N$ individuals in order to create a **new generation** of candidate solutions

- Generation after generation, DE converges to an optimal solution wrt to f and the algorithm halts when a **termination criterion** is met

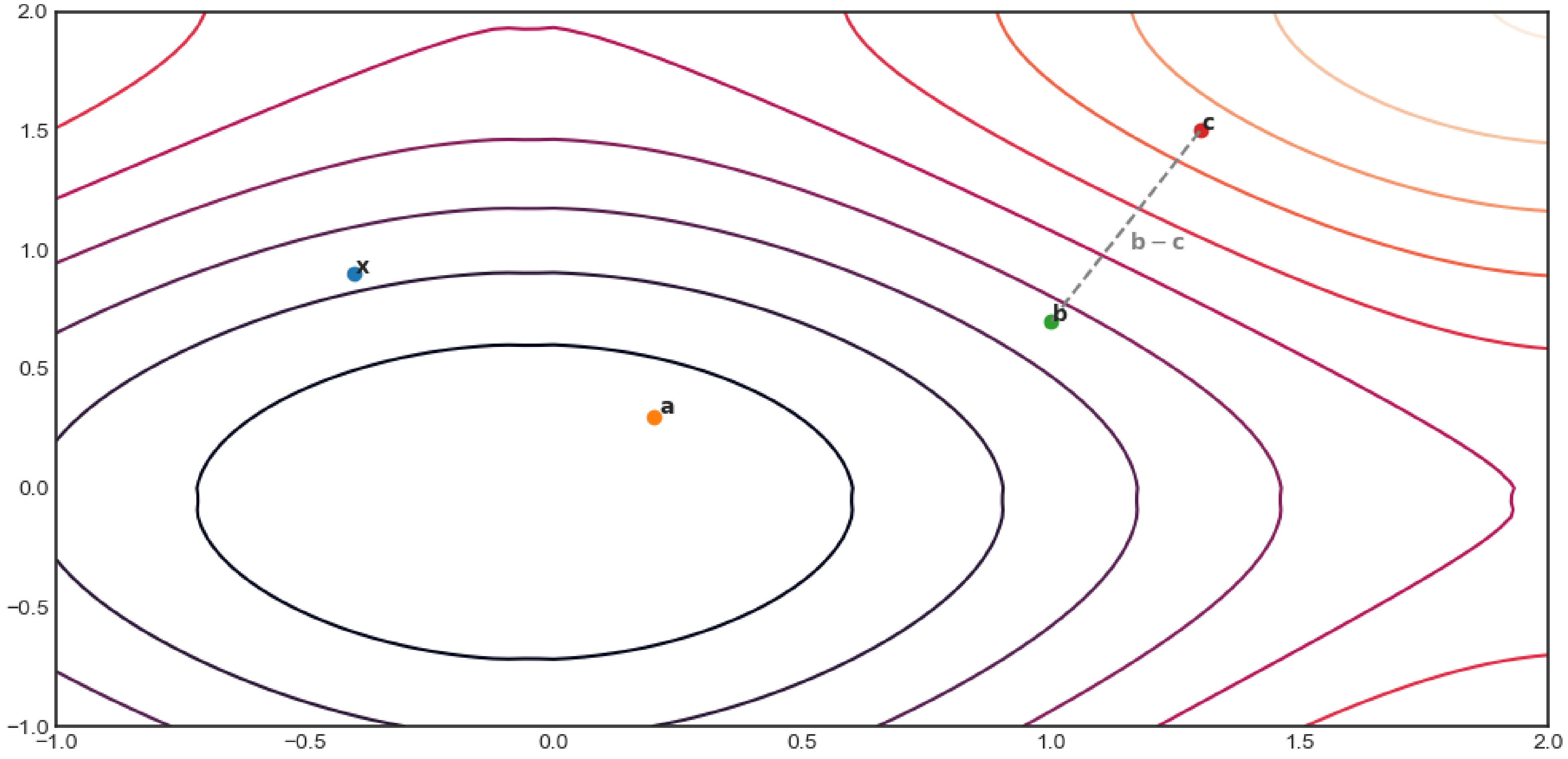
Example of DE



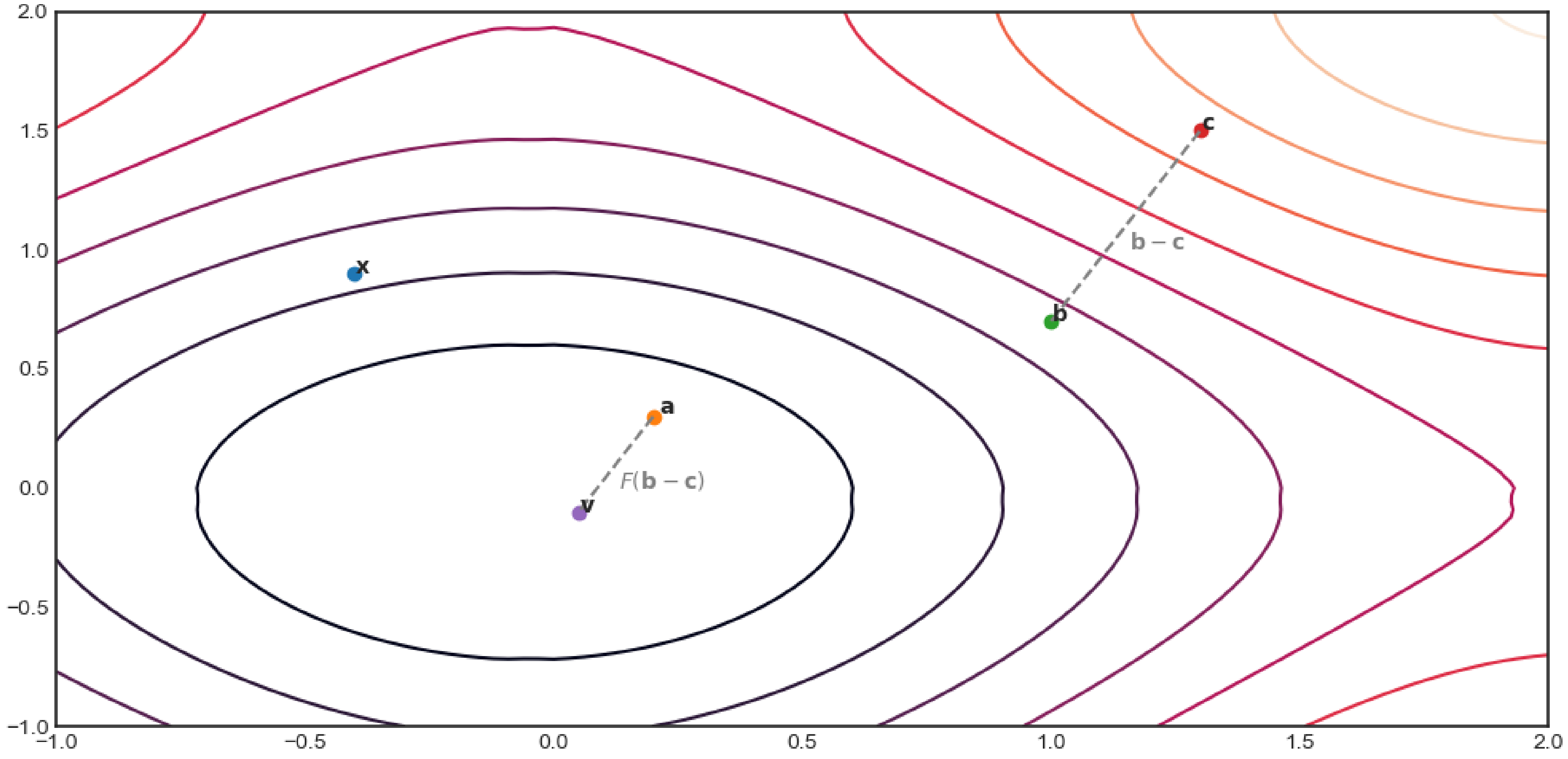
Example of DE



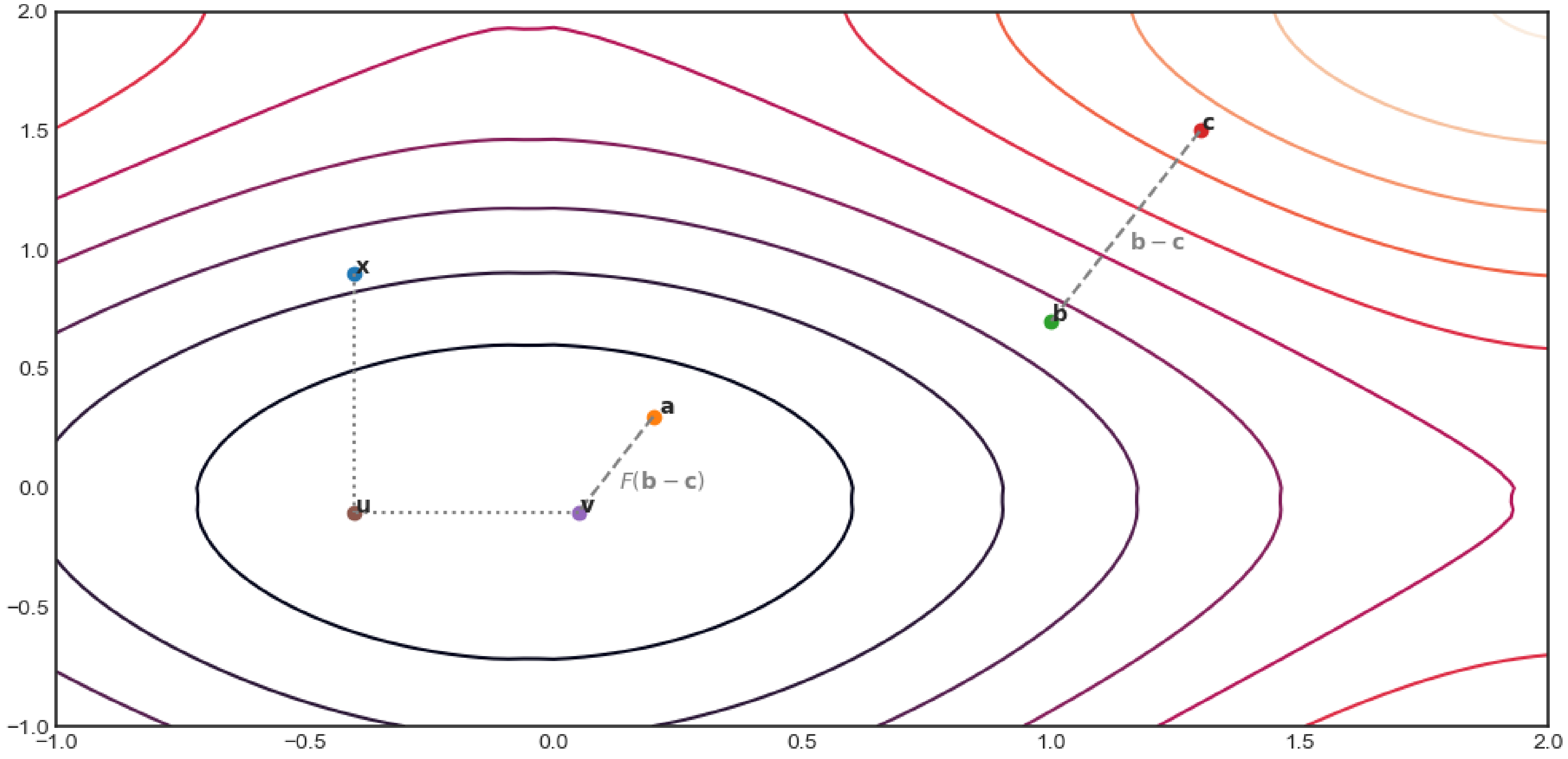
Example of DE



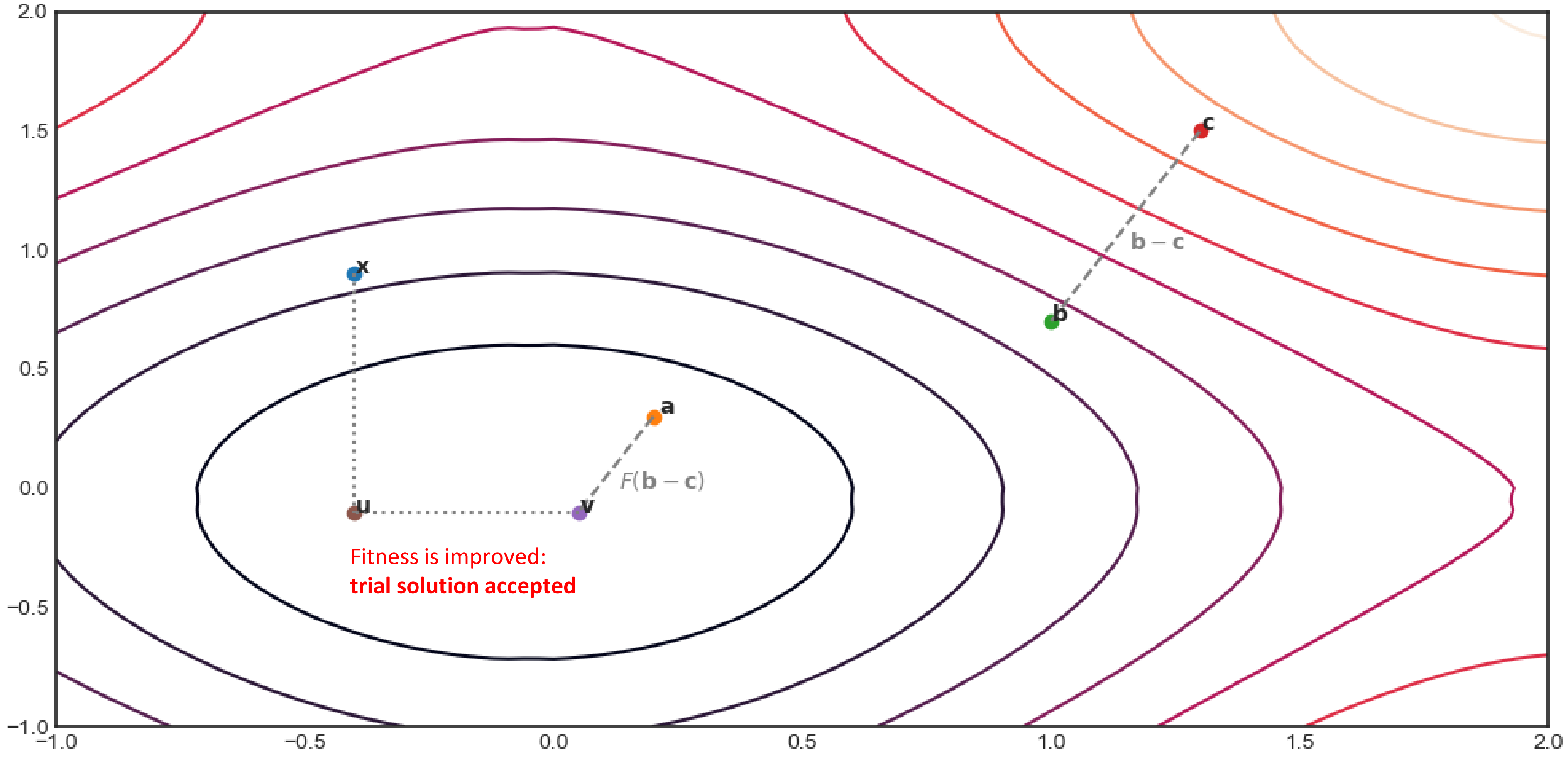
Example of DE



Example of DE



Example of DE



DE taxonomy

- The DE scheme just described is known as *DE/rand/1*
  - *DE* is self-explanatory
  - *rand* means that we pick a random individual as first vector for differential mutation
  - *1* means that we create the donor vector by using a single differential mutation

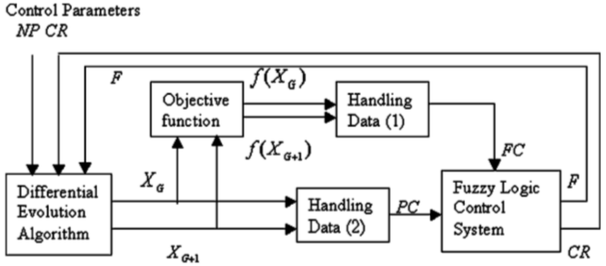
- There are some alternative schemes, most notably:

Name	Differential mutation equation
DE/best/1	$\mathbf{v}_i = \mathbf{x}_{best} + F \cdot (\mathbf{b} - \mathbf{c})$
DE/current-to-best/1	$\mathbf{v}_i = \mathbf{x}_i + F \cdot (\mathbf{x}_{best} - \mathbf{x}_i) + F \cdot (\mathbf{b} - \mathbf{c})$
DE/rand/2	$\mathbf{v}_i = \mathbf{a} + F \cdot (\mathbf{b} - \mathbf{c}) + F \cdot (\mathbf{d} - \mathbf{e})$
DE/rand-to-best/1	$\mathbf{v}_i = \mathbf{x}_i + F \cdot (\mathbf{x}_{best} - \mathbf{x}_i) + F \cdot (\mathbf{b} - \mathbf{c})$

$\mathbf{x}_{best}$ is the best candidate solution found so far
$\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}, \mathbf{e}$ are randomly selected individuals

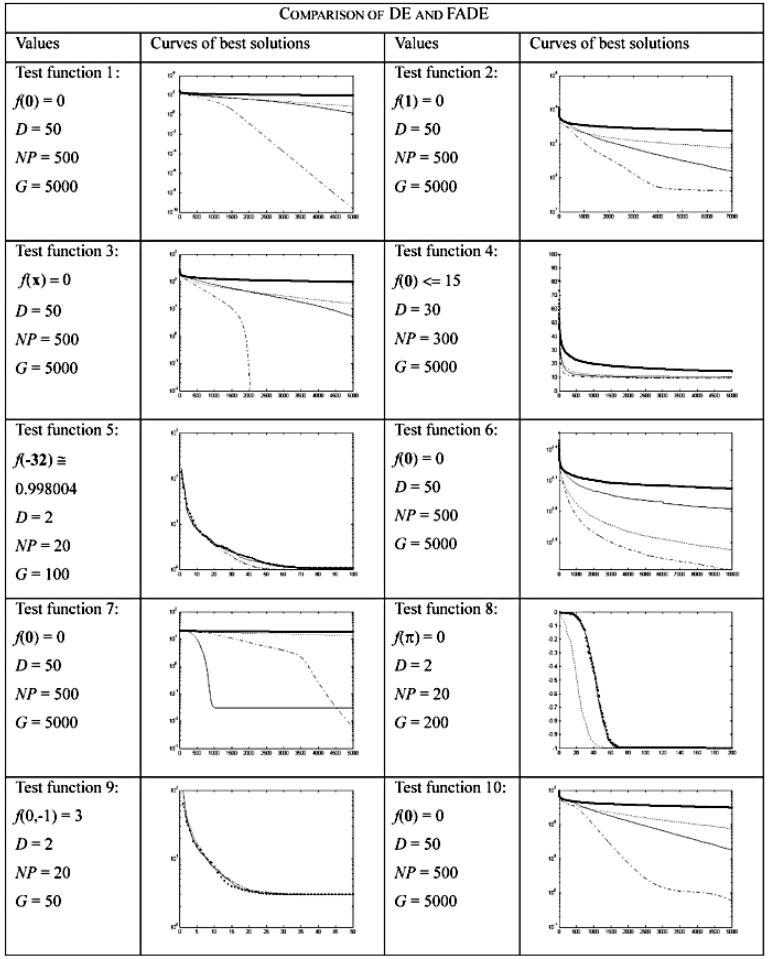
Advanced versions of DE – FADE

- **Fuzzy Adaptive DE** (FADE)
  - **Adapts the hyper-parameters** (F, CR) at run-time using a **Fuzzy Logic Control System** (FLC)
  - The FCL takes **two inputs**
    - The fitness values $FC = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\sum_{d=1}^{D}\left(x_{i,j}^{G} - x_{i,j}^{G-1}\right)^2}$
    - The population change $PC = \sqrt{\frac{1}{N}\sum_{n=1}^{N}\left(f\left(\mathbf{x}_i^{G}\right) - f\left(\mathbf{x}_i^{G-1}\right)\right)^2}$
  - The fuzzy reasoner controls **two outputs:** F, CR



  - **Settings-free**
  - [Liu and Lampinen, Soft Comput 2015]

Table 5 Results of experiments



Comparison of DE and FADE			
Values	Curves of best solutions	Values	Curves of best solutions
Test function 1: $f(\mathbf{0}) = 0$, $D = 50$, $NP = 500$, $G = 5000$		Test function 2: $f(\mathbf{1}) = 0$, $D = 50$, $NP = 500$, $G = 5000$	
Test function 3: $f(\mathbf{x}) = 0$, $D = 50$, $NP = 500$, $G = 5000$		Test function 4: $f(\mathbf{0}) <= 15$, $D = 30$, $NP = 300$, $G = 5000$	
Test function 5: $f(\mathbf{-32}) \cong$ 0.998004, $D = 2$, $NP = 20$, $G = 100$		Test function 6: $f(\mathbf{0}) = 0$, $D = 50$, $NP = 500$, $G = 5000$	
Test function 7: $f(\mathbf{0}) = 0$, $D = 50$, $NP = 500$, $G = 5000$		Test function 8: $f(\pi) = 0$, $D = 2$, $NP = 20$, $G = 200$	
Test function 9: $f(0,-1) = 3$, $D = 2$, $NP = 20$, $G = 50$		Test function 10: $f(\mathbf{0}) = 0$, $D = 50$, $NP = 500$, $G = 5000$	

Note: $f(\mathbf{x}^*)$ = the optimum of the specified function. Figure legend: (•) DE; (–) DE adapting CR; (—) DE adapting F; (— –) DE adapting F and CR

Advanced versions of DE – jADE [1/4]

- **Adaptive Differential Evolution** (jADE)
  - **New mutation strategy**
  - **External archive** of sub-optimal solutions
  - **Dynamic update** of hyper-parameters
  - [Zhang and Sanderson, IEEE Tran Evol Comp 2009]

Advanced versions of DE – jADE [2/4]

- jADE uses a **new mutation strategy**: *DE/current-to-$p$best*
  - Generalization of *DE/current-to-best*
  - The parameter $p \in (0,1]$ determines the **ratio** of solutions used to choose a $\mathbf{x}_{best}^{G}$
- jADE exploits an **external archive of sub-optimal solutions** to «steer» the population towards **new promising directions**
  - Archive composed of **individuals discarded** during selection procedure
  - **Archive limited** to a maximum of N solutions

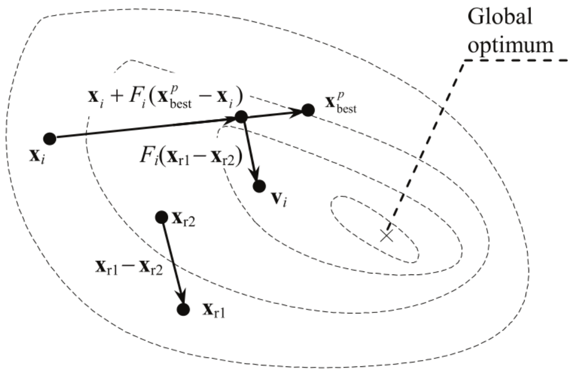


Fig. 1. Illustration of the DE/current-to-$p$best/1 mutation strategy adopted in JADE. The dashed curves display the contours of the optimization problem. $\mathbf{v}_i$ is the mutation vector generated for individual $\mathbf{x}_i$ using the associated mutation factor F_i.

Advanced versions of DE – jADE [2/4]

- jADE uses a **new mutation strategy**: *DE/current-to-pbest*
  - Generalization of *DE/current-to-best*
  - The parameter $p \in (0,1]$ determines the **ratio** of solutions used to choose a $\mathbf{x}_{best}^{G}$

- jADE exploits an **external archive of sub-optimal solutions** to «steer» the population towards **new promising directions**
  - Archive composed of **individuals discarded** during selection procedure
  - **Archive limited** to a maximum of N solutions

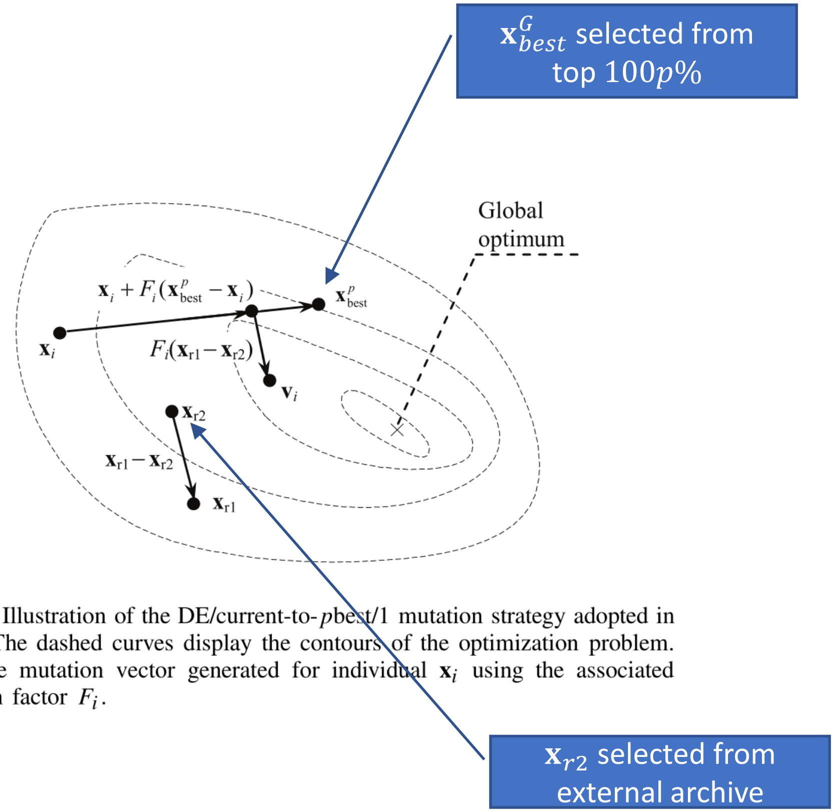


Fig. 1. Illustration of the DE/current-to-$p$best/1 mutation strategy adopted in JADE. The dashed curves display the contours of the optimization problem. $\mathbf{v}_i$ is the mutation vector generated for individual $\mathbf{x}_i$ using the associated mutation factor F_i.

Advanced versions of DE – jADE [3/4]

- CR and F are re-calculated **for each candidate solution** during each generation

  - Crossover probability calculated as $CR_i = \mathcal{N}\left(\mu_{CR}^{G}, 0.1\right)$
    - $\mathcal{N}(,)$ is a **normal** distribution truncated in $[0,1]$
    - $\mu_{CR}^{G} = (1-c) \cdot \mu_{CR}^{G-1} + c \cdot M_{CR}$
    - $\frac{1}{c} \in [5,20]$ is a **hyper-parameter** controlling the **adaptation rate**
    - $\mu_{CR}^{0} = 0.5$ is the **initial crossover probability**
    - M_{CR} is the **arithmetic mean** value of all **successful crossover probabilities** used so far

  - Similarly, the mutation factor is calculated as $F_i = \mathcal{C}(\mu_F, 0.1)$
    - $\mathcal{C}(,)$ is a Cauchy distribution bounded in $[0,1]$ (truncation and resampling)
    - $\mu_F = (1-c) \cdot \mu_F + c \cdot M_F$
    - M_F is the **Lehmer mean** value of all **successful mutation factors** used so far

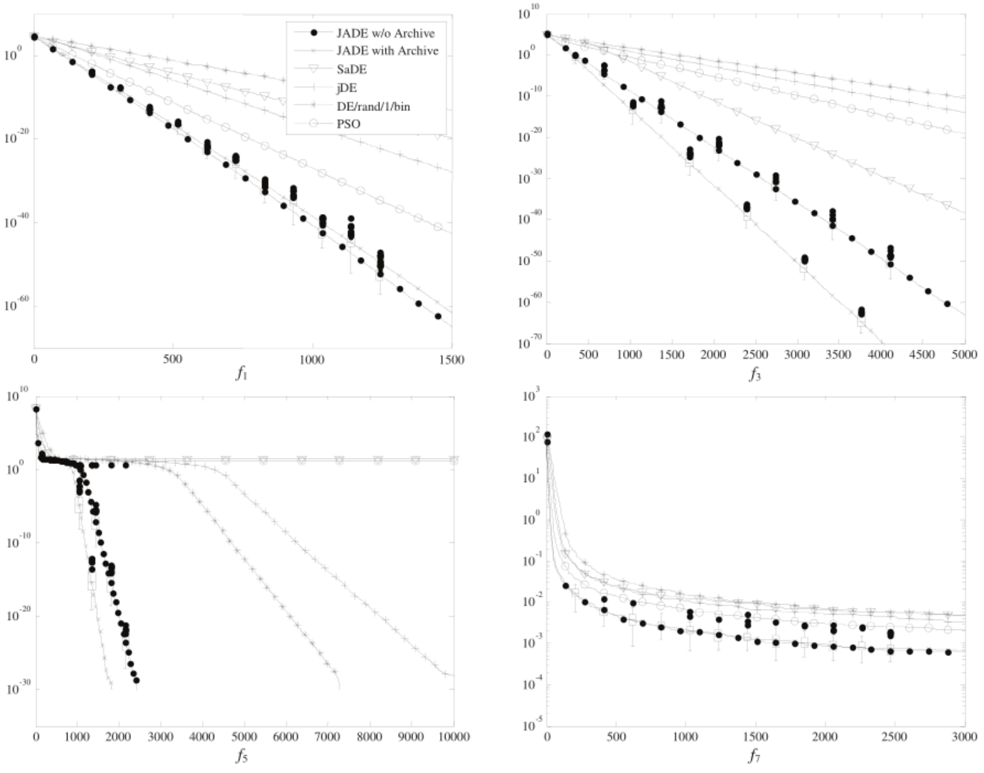
Advanced versions of DE – jADE [4/4]

- jADE outperforms classic DE/rand/1
- On higher dimensions (D=100) the **archive helps the optimization**

TABLE VII

EXPERIMENTAL RESULTS OF THE 100-DIMENSIONAL PROBLEMS f_1–f_{13}, AVERAGED OVER 50 INDEPENDENT RUNS. NOTE THAT THE BEST AND THE SECOND BEST AMONG JADE AND OTHER COMPETITIVE ALGORITHMS (EXCEPT JADE TWO VARIANTS) ARE MARKED IN BOLDFACE AND ITALIC, RESPECTIVELY

Functions		f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}	f_{11}	f_{12}	f_{13}
JADE (w/o archive)	SR	100	100	100	0	86	84	100	100	100	100	96	100	100
	FESS	*2.0E+5*	*2.9E+5*	*1.3E+6*	–	*2.2E+6*	*8.8E+4*	*2.3E+5*	*1.3E+6*	*1.5E+6*	*2.9E+5*	*1.9E+5*	*1.6E+5*	*1.9E+5*
JADE (with archive)	SR	100	100	100	100	90	100	100	100	100	100	98	100	100
	FESS	**1.6E+5**	**2.7E+5**	**9.6E+5**	**7.7E+5**	**1.5E+6**	**6.2E+4**	**2.0E+5**	1.4E+6	*1.5E+6*	**2.4E+5**	**1.7E+5**	**1.4E+5**	**1.6E+5**
jDE	SR	100	100	0	100	2	100	96	100	100	100	100	100	100
	FESS	5.5E+5	7.6E+5	–	*5.7E+6*	7.7E+6	2.2E+5	2.0E+6	**9.5E+5**	**1.4E+6**	8.0E+5	5.4E+5	5.7E+5	5.8E+5
SaDE	SR	36	0	100	0	0	100	54	100	100	0	100	100	100
	FESS	7.8E+5	–	2.1E+6	–	–	2.6E+5	1.6E+6	1.6E+6	1.8E+6	–	8.0E+5	9.2E+5	9.2E+5
DE/rand /1/bin	SR	0	0	0	0	0	0	0	0	0	0	0	0	0
	FESS	–	–	–	–	–	–	–	–	–	–	–	–	–
PSO	SR	100	10	0	0	0	2	70	0	0	2	36	28	54
	FESS	6.2E+5	1.1E+6	–	–	–	5.0E+5	1.9E+6	–	–	1.1E+6	6.0E+5	9.5E+5	8.7E+5
rand-JADE (w/o archive)	SR	0	0	0	0	0	74	0	100	100	0	0	0	0
	FESS	–	–	–	–	–	5.8E+5	–	2.5E+6	2.5E+6	–	–	–	–
nona-JADE (w/o archive)	SR	100	100	0	0	90	100	100	0	0	100	100	100	100
	FESS	2.0E+5	3.3E+5	–	–	5.0E+6	7.3E+4	3.5E+5	–	–	3.0E+5	2.0E+5	1.7E+5	1.9E+5



Advanced versions of DE – SHADE

- **Success History Adaptive Differential Evolution** (SHADE)
  - Improved version of jADE (*DE/current-to-pbest/1*)
  - Similarly to jADE, SHADE keeps an **«historic» archive** (circular buffer of size $H \approx 100$)
  - During each generation, the archive is updated with the information about M_{SC} and M_F
  - CR and F, for all individuals, are calculated using the same formulas seen in jADE, except that **random elements of the historic archive** are used for M_{SC} and M_F
  - $H = \infty$ yields **worse results** than limited buffer size

TABLE VI: Evaluation of infinite memory size SHADE variants (compared to $H = 100$)

H	100	∞-Random	∞-Time	∞-Fitness
	+	3	3	4
	-	14	8	18
	$\approx$	11	17	6

  - **SHADE outperforms jADE** on several benchmark functions [Tanabe and Fukunaga, IEEE CEC 2013]
  - **L-SHADE**: improves SHADE with **linearly decreasing population** [Tanabe and Fukunaga, IEEE CEC 2014]

TABLE IV: Comparison of L-SHADE with state-of-the-art DE algorithms on the CEC2014 benchmarks ($D = 10, 30, 50, 100$ dimensions). The aggregate results of statistical testing ($+, -, \approx$) on 30 functions are shown. The symbols $+, -, \approx$ indicate that a given algorithm performed significantly better ($+$), significantly worse ($-$), or not significantly different better or worse ($\approx$) compared to L-SHADE using the Wilcoxon rank-sum test (significantly, $p < 0.05$). The maximum number of objective function evaluations is $D \times 10,000$. All results are based on 51 runs.

vs. L-SHADE		$D = 10$	$D = 30$	$D = 50$	$D = 100$
SHADE	$+$ (better)	0	3	5	7
	$-$ (worse)	16	18	19	17
	$\approx$ (no sig.)	14	9	6	6
CoDE	$+$ (better)	6	4	6	4
	$-$ (worse)	12	19	23	22
	$\approx$ (no sig.)	12	7	1	4
EPSDE	$+$ (better)	4	5	6	7
	$-$ (worse)	20	22	21	23
	$\approx$ (no sig.)	6	3	3	0
JADE	$+$ (better)	1	2	3	6
	$-$ (worse)	20	22	23	20
	$\approx$ (no sig.)	9	6	4	4
SaDE	$+$ (better)	4	1	0	1
	$-$ (worse)	17	25	27	27
	$\approx$ (no sig.)	9	4	3	2
dynNP-jDE	$+$ (better)	6	3	5	7
	$-$ (worse)	16	20	20	18
	$\approx$ (no sig.)	8	7	5	5

Advanced versions of DE – Distance-based DE

- **Distance-based parameter adaptation** (Db-SHADE)
  - Extends SHADE
  - Promotes diversity in the population by adapting CR and F according to the «**Euclidean distance** between the trial and the original individual» **instead of fitness improvement**
  - **DbL-SHADE** extends Db-SHADE with **linearly decreasing population**
  - See [Viktorin *et al.*, CSOC 2018]

- **Distance Based Parameter Adaptation Success History DE** (DISH)
  - Improved version of Dbl-SHADE
  - Novel **distance-based parameter adaptation** to prevent **premature convergence**
  - Idea: the individual that moved the furthest has a highest importance
  - Significantly outperforms SHADE and L-SHADE on CEC's benchmark functions
  - [Viktorin *et al.*, Swarm Evol Comp 2018]

Friedman ranks for algorithms over CEC2015 (aggregated all 10*D*, 30*D*, 50*D*, and 100*D* functions).

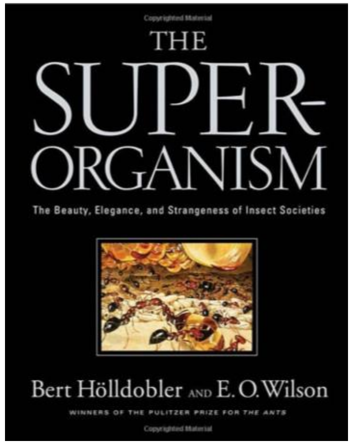
Rank	Name	F-rank
0	DISH	2.9
1	DbL-SHADE	3.2
2	jSO	3.2
3	Db_SHADE	3.7
4	L-SHADE	3.7
5	SHADE	4.3

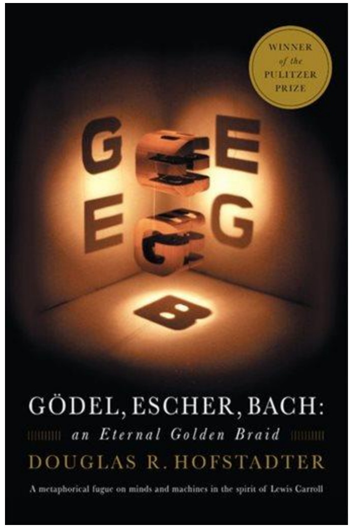
Swarm Intelligence

From evolution to superorganisms

- Swarm Intelligence is a class of **bio-inspired** global optimization methods
  - Instead of evolution, they rely on the intelligence of **superorganisms**
  - *“Tightly knit colony of individuals, formed by altruistic cooperation, complex communication, and division of labor, which represent one of the basic stages of biological organization, midway between the organism and the entire species”*
  - **Individuals are simple agents** with limited “intelligence”, limited capabilities, limited memory, limited possible behaviors
  - **An intelligent behavior emerges** from the collective effort of multiple similar agents

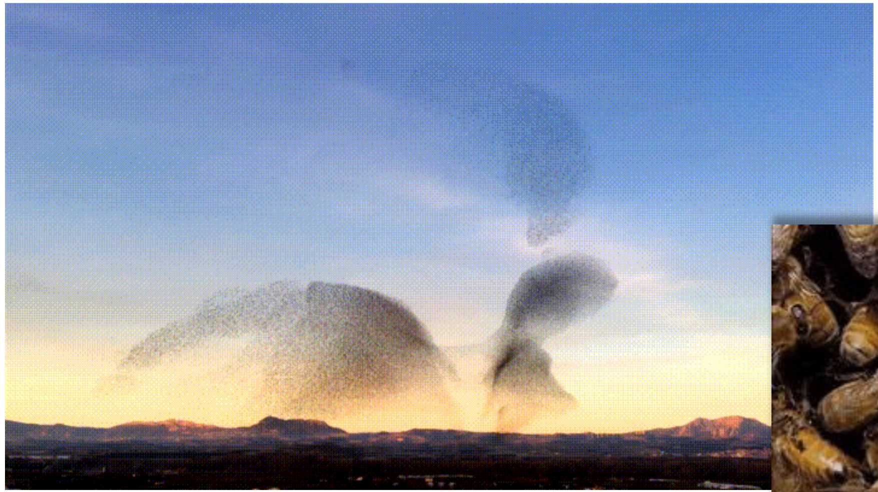
- Examples in nature: ant *vs* ant colony, bee *vs* bee colony, fish *vs* fish school, bird *vs* bird flock...





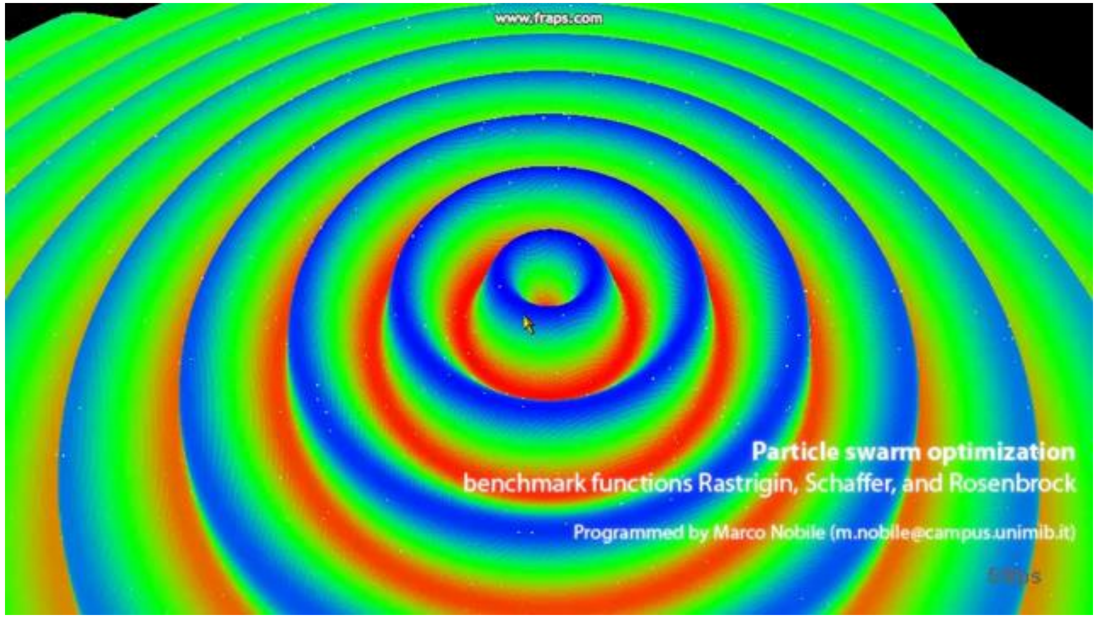
Swarm Intelligence

- Swarm Intelligence exploits the **emergent intelligence of groups of agents**
  - The group spontaneously **converges to the optimal solution** of a complex problem
  - We will see three algorithms: **particle swarm optimization**, **artificial bee colony**, **ant colony optimization**
  - The algorithms are based on **information exchange** about promising solutions (e.g., inter-swarm communication, waggle dance, pheromones traces)
  - [Eberhart *et al*., Swarm Intelligence, 2001]



Particle Swarm Optimization [1/3]

- Particle Swarm Optimization (PSO) is a Swarm Intelligence meta-heuristic for GO
  - Inspired by the **collective movement of animals** (e.g., flocks of birds, schools of fish)
  - PSO is based on a population (the **swarm**) of N individuals (the **particles**)
  - [Kennedy and Eberhart, Proc IEEE Int Conf Neural Networks 1995]

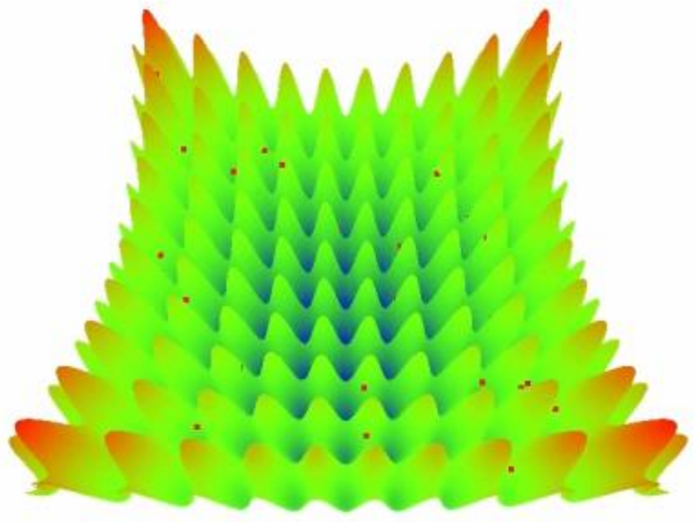


Particle Swarm Optimization [2/3]

- Particles represent candidate solutions
  - **They move** inside a M-dimensional bounded search space
  - **Particles cooperate** to identify the optimal solution

- i-th particle is characterized by **two vectors**:
  - $\mathbf{x}_i(t) \in \mathbb{R}^M$ is the **position** of i-th particle in the search space during iteration t
  - $\mathbf{v}_i(t) \in \mathbb{R}^M$ is the **velocity** of i-th particle during iteration t

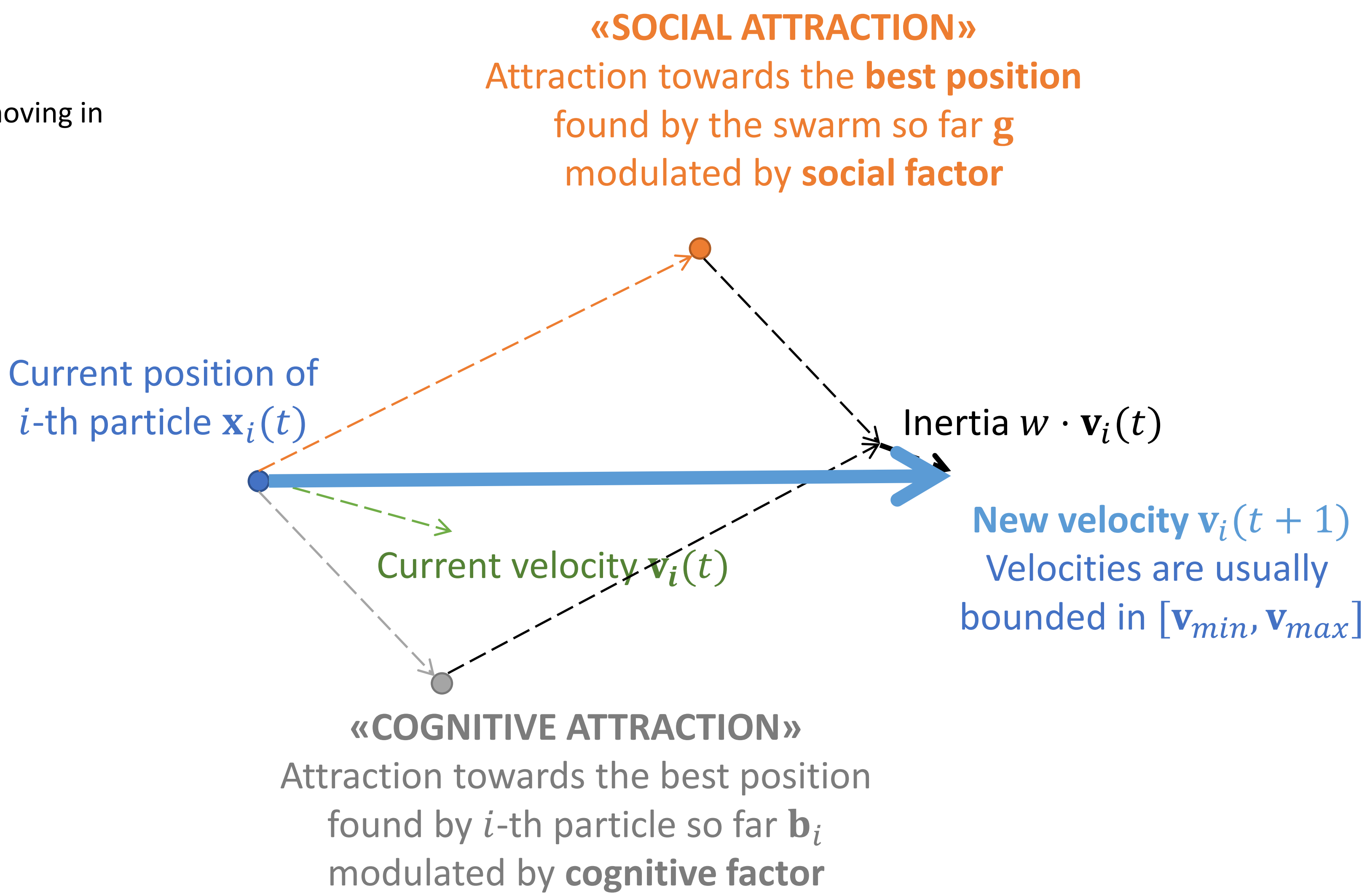
- Velocity is used to **update** particle's position
  - i.e., $\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t)$
  - Velocities change as a result of **two attractions**: social and cognitive



PSO optimizing a 2D Rastrigin benchmark function

Particle Swarm Optimization [3/3]

A swarm of particles moving in the search space…



Velocity and constriction factor

- The final formula for the velocity update is the following:

$$\mathbf{v}_i(t+1) = w \cdot \mathbf{v}_i(t) + c_{soc} \cdot \mathbf{r}_1 \otimes \big(\mathbf{g} - \mathbf{x}_i(t)\big) + c_{cog} \cdot \mathbf{r}_2 \otimes (\mathbf{b}_i - \mathbf{x}_i(t))$$

where $\mathbf{r}_1$ and $\mathbf{r}_2$ are vectors or random numbers sampled with uniform distribution in $[0,1]$ and $\otimes$ denotes the Hadamard product

- Following some theoretical studies about PSO's convergence (not theorem exist, yet) an alternative formulation based on a **constriction factor** was proposed:

$$\mathbf{v}_i(\mathrm{t}+1) = \chi(\mathbf{v}_i(t) + c_{soc} \cdot \mathbf{r}_1 \otimes \big(\mathbf{g} - \mathbf{x}_i(t)\big) + c_{cog} \cdot \mathbf{r}_2 \otimes (\mathbf{b}_i - \mathbf{x}_i(t))$$

$$\chi = \frac{2k}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}$$

where $k \in [0,1], \varphi = c_{soc} + c_{cog}$ and $\varphi > 4$

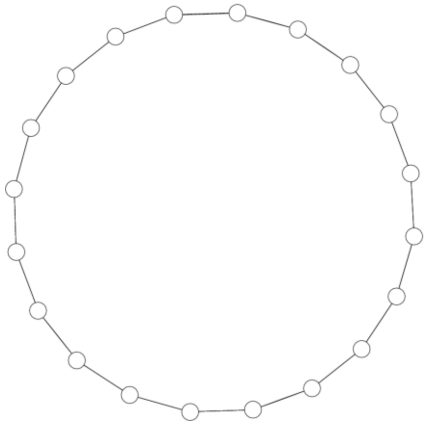
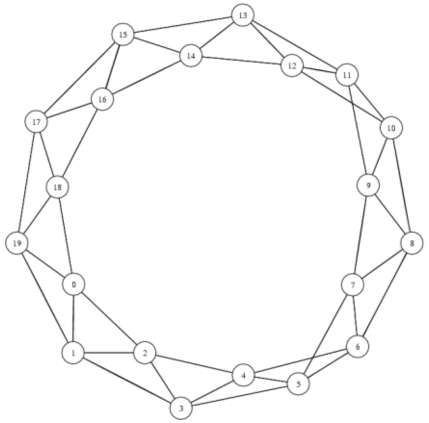
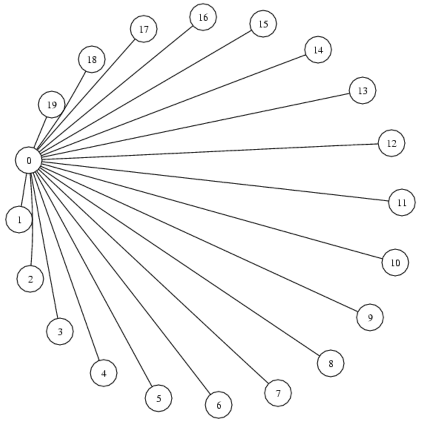
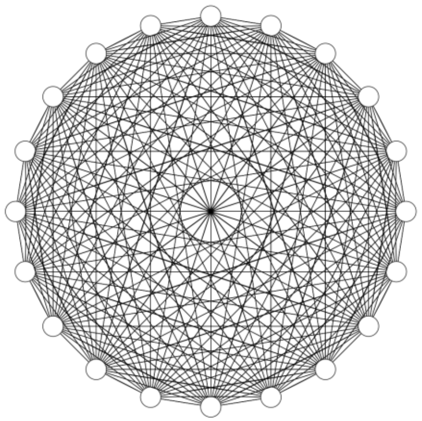
Controlling the swarm: PSO's hyper-parameters

- PSO user must choose **multiple settings**
  - The **social factor** $c_{soc} \in \mathbb{R}^+$, which modulates the attraction towards the «global best» («local exploitation» of the search space). Traditionally, $c_{soc} = 1.49445$
  - The **cognitive factor** $c_{cog} \in \mathbb{R}^+$, which modulates the attraction towards the «personal best» («global exploration» of the search space). Traditionally, $c_{soc} = 1.49445$
  - The **inertia weight** $w \in \mathbb{R}^+$, which balances the aforementioned settings: low inertia helps the local search, high inertia helps global search. Traditionally, w linearly decrements from 0.9 to 0.4
  - The **swarm size** N
  - The **maximum velocity** of particles $\mathbf{v}_{\max} \in \mathbb{R}^M$
  - Also the **minimum velocity** of particles $\mathbf{v}_{\min} \in \mathbb{R}^M$ (do not lose momentum!)
  - Moreover…
    - A strategy for **handling particles going outside** the search space must be selected
    - An algorithm for the **initial distribution** of particles in the search space must be selected (e.g., uniform)

- These hyper-parameters should be carefully chosen, since they have a **relevant impact** on the optimization: a bit annoying for people without expertise

Topologies

- Intra-swarm **communication topology** (i.e., how to choose **g**) has an impact on PSO's performance
  - Several topologies (fixed or dynamic) can be used
  - E.g., local, local with extended neighborhood, wheel, fully connected («gbest», the most widespread)

  - Unfortunately, in most cases the optimal topology is **problem dependent**
  - **Fully Informed PS**: particles attracted towards the center of gravity of the best positions of neighboors
  - See [Mendes *et al.*, IEEE Tran Evol Comp 2004]

Boundary conditions

- During their «flight», particles could go **outside the search space**
  - Those regions correspond to **unfeasible solutions**
  - We could just **ignore those particles**: they become «invisible» for the swarm, but we are losing exploration capability
  - We could assign a **bad fitness value** (hopefully they will return spontaneusly to the feasible region)

- We can use **boundary conditions**
  - Move back the particle ON the boundary (absorbing): particles lose momentum
  - Particles could «bounce» back (reflecting): better results
  - Damping (i.e., stochastic «bounce») seems to yield the best results
  - [Xu *et al.*, IEEE Tran Antennas Propag 2007]

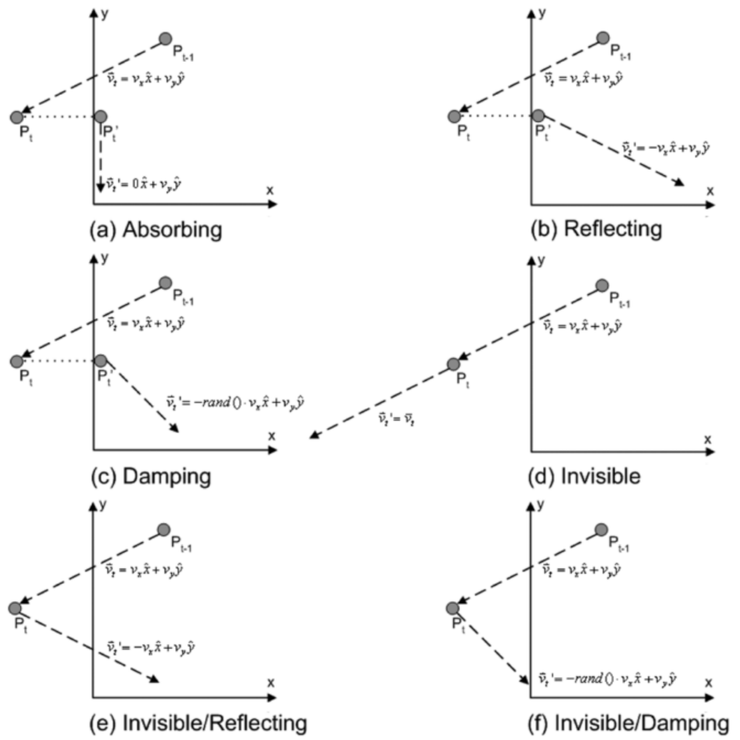


Fig. 3. Six different boundary conditions for a two-dimensional problem. P' and $\vec{v}'$ represent the modified position and velocity, respectively, after the errant particle is treated by boundary conditions.

Selection of PSO settings

- The optimal settings for PSO are **problem-dependent**
  - Problem of **meta-optimization**: optimize the settings of the optimizer
  - **Issues** related to poor choice: divergence, cyclic trajectories, premature convergence, lost momentum…
  - Possible approach 1: **differential investigation** of the impact of the settings
  - Possible approach 2: use **heuristic values** which are «good enough» in any situation (99% of papers)
  - Possible approach 3: implement a **self-tuning** PSO, in which the **parameters are automatically adjusted** by means of some rules or algorithm

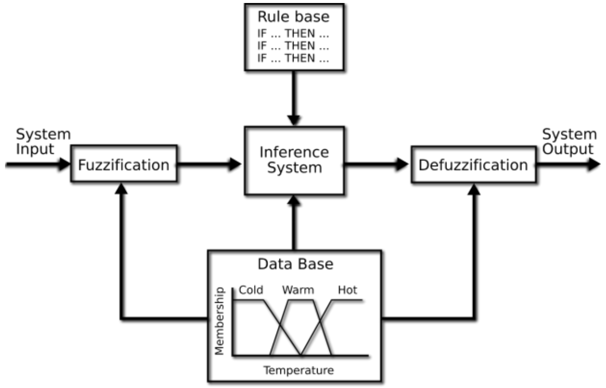
- For instance, one could be interested in implementing rules like

«when a particle gets too close to the best individual of the swarm, increase its cognitive factor»

Fuzzy PSO

- Similarly to DE, also PSO has been **hybridized** with Fuzzy Rule-Based Systems (FRBS) for the automatic selection of functioning settings

Authors	Settings calculated	Input variables
Shi and Eberhart (2001)	Inertia w	1) current best performance evaluation 2) current inertia value
Abraham and Liu (2009)	Minimum velocity $\mathbf{v}_{\min}$	1) current best performance evaluation 2) current velocity of particles
Tian and Li (2009)	Inertia w, «learning coefficient»	1) improvement of the global best 2) standard deviation of particles' fitness
Olivas *et al.* (2015)	c_{cog}, c_{soc}	1) diversity measure (based on distance) 2) error measure (diversity based on fitness)
Nobile *et al.* (2015)	$w_i, c_{cog_i}, c_{soc_i}$	1) difference in fitness value 2) distance from **g**



Fuzzy Self-Tuning PSO

- FST-PSO is a **settings-free** meta-heuristic for GO
    - Based on PSO
    - Relies on Fuzzy Logic to give particles **a «glimpse» of additional intelligence**
    - **No hyper-parameters**: completely automatic
    - FST-PSO calculates **inertia**, **social factor**, **cognitive factor**, **minimum velocity**, and **maximum velocity** at run-time
    - FST-PSO sets these values for **each particle** according to **current performance** and **distance from the best particle**
    - [Nobile *et al*., Swarm Evol Comp, 39:70–85, 2018]

Swarm and Evolutionary Computation 39 (2018) 70–85

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

ELSEVIER

Fuzzy Self-Tuning PSO: A settings-free algorithm for global optimization

Marco S. Nobile [a,c,*], Paolo Cazzaniga [b,c], Daniela Besozzi [a,c], Riccardo Colombo [a,c], Giancarlo Mauri [a,c], Gabriella Pasi [a]

[a] Department of Informatics, Systems and Communication, University of Milano-Bicocca, Milano, Italy
[b] Department of Human and Social Sciences, University of Bergamo, Bergamo, Italy
[c] SYSBIO.IT Centre of Systems Biology, Milano, Italy

ARTICLE INFO

Keywords:

ABSTRACT

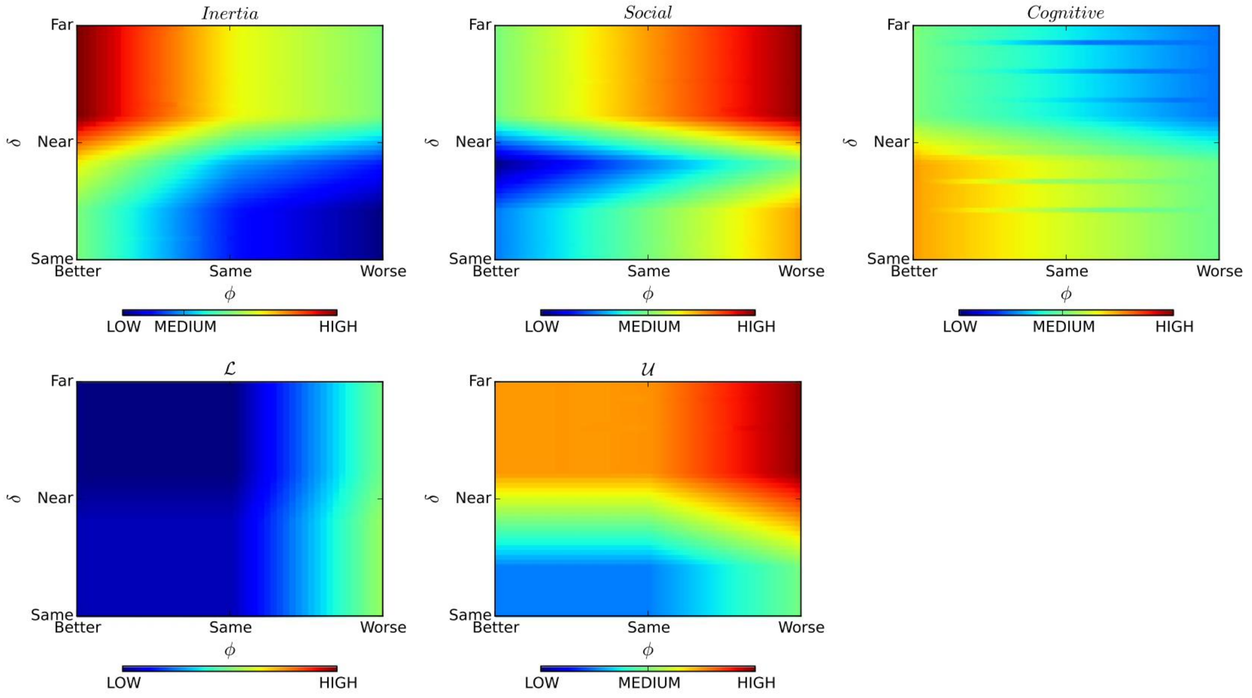
Among the existing global optimization algorithms, Particle Swarm Optimization (PSO) is one of the most

Functioning of FST-PSO

- Each particle dynamically adapts **its own** hyper-parameters according to two information
  - The normalized **fitness improvement** with respect to the previous iteration ϕ
  - The **distance from the global best** δ

Fuzzy rules used by FST-PSO.

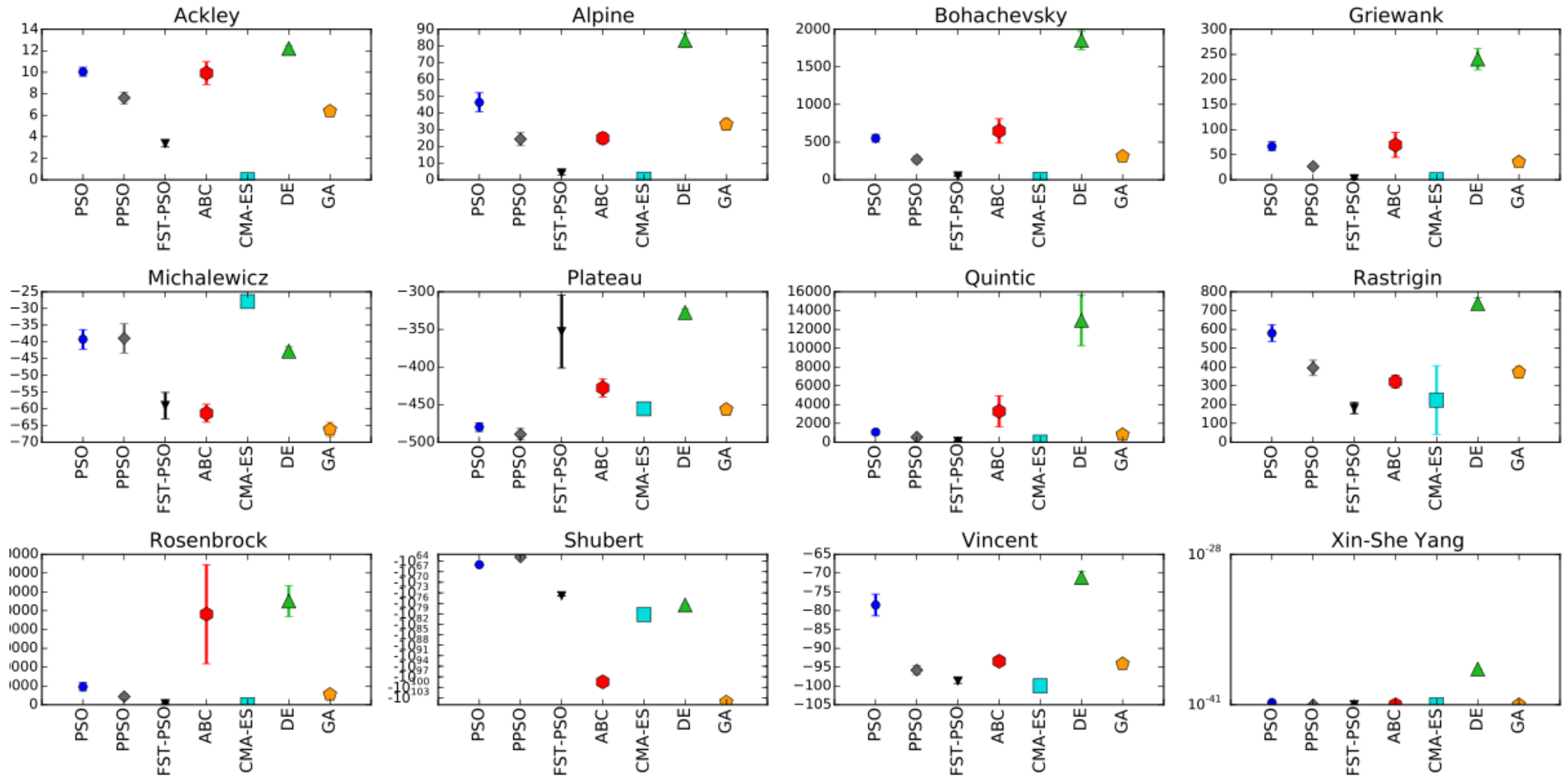
Rule no.	*Rule definition*
1	if (ϕ is *Worse* or δ is *Same*) then (*Inertia* is *Low*)
2	if (ϕ is *Same* or δ is *Near*) then (*Inertia* is *Medium*)
3	if (ϕ is *Better* or δ is *Far*) then (*Inertia* is *High*)
4	if (ϕ is *Better* or δ is *Near*) then (*Social* is *Low*)
5	if (ϕ is *Same* or δ is *Same*) then (*Social* is *Medium*)
6	if (ϕ is *Worse* or δ is *Far*) then (*Social* is *High*)
7	if (δ is *Far*) then (*Cognitive* is *Low*)
8	if (ϕ is *Worse* or ϕ is *Same* or δ is *Same* or δ is *Near*) then (*Cognitive* is *Medium*)
9	if (ϕ is *Better*) then (*Cognitive* is *High*)
10	if (ϕ is *Same* or ϕ is *Better* or δ is *Far*) then ($\mathcal{L}$ is *Low*)
11	if (δ is *Same* or δ is *Near*) then ($\mathcal{L}$ is *Medium*)
12	if (ϕ is *Worse*) then ($\mathcal{L}$ is *High*)
13	if (δ is *Same*) then ($\mathcal{U}$ is *Low*)
14	if (ϕ is *Same* or ϕ is *Better* or δ is *Near*) then ($\mathcal{U}$ is *Medium*)
15	if (ϕ is *Worse* or δ is *Far*) then ($\mathcal{U}$ is *High*)



Effects of rules to particles according to δ and ϕ (calculated with Sugeno inference engine)

Performances of FST-PSO

- FST-PSO is **competitive with state-of-the-art methods**
  - **FST-PSO ▼ outperforms** classic PSO, PPSO, DE, GA, and artificial bee colony (next topic)
  - **Tied to CMA-ES ■** (which is more computationally demanding)



Using FST-PSO

- Implemented in Python and available on PyPI, FST-PSO can be easily installed with pip:

```
pip install fst-pso
```

- FST-PSO is settings-free. The user must provide only two information:

1. The **search space** defined as a $2 \times D$ list or numpy array, containing with the minimum and maximum values for each coordinate
2. The **fitness function**, defined as a python function taking as input a particle and returning the corresponding fitness value

FST-PSO applied to 3D Alpine benchmark function

C:\Users\aresi\Desktop\example-fstpso.py - Sublime Text (UNREGISTERED)

```python
from fstpso import FuzzyPSO
from math import fabs, sin

def Alpine( particle ):
    acc = 0.0
    for i in range(len(particle)): acc += fabs(particle[i]*sin(particle[i])+0.1*particle[i])
    return acc

if __name__ == '__main__':
    D = 3
    boundaries = [[-10,10]]*D
    F = FuzzyPSO()
    F.set_search_space(boundaries)
    F.set_fitness(Alpine)
    F.solve_with_fstpso()
```

```
 * Max distance: 34.641016
 * Search space boundaries set to: [[-10, 10], [-10, 10], [-10, 10]]
 * Max velocities set to: [20.0, 20.0, 20.0]
 * Number of particles automatically set to 13
 * Testing fitness evaluation
 * Test successful
 * Max iterations set to 100
 * Settings: ['cognitive', 'social', 'inertia', 'minvelocity', 'maxvelocity']
 * Launching optimization
 * 13 particles created, verifying local and global best
new global best 0 has fitness 9.04877246671
new global best 1 has fitness 1.58516149381
new global best 4 has fitness 1.37500661984
new global best 7 has fitness 1.16331359268
new global best 9 has fitness 1.00008209755
new global best 6 has fitness 0.942663660408
new global best 11 has fitness 0.281185681136
new global best 3 has fitness 0.132149713257
new global best 9 has fitness 0.120003328827
new global best 9 has fitness 0.116738735803
```

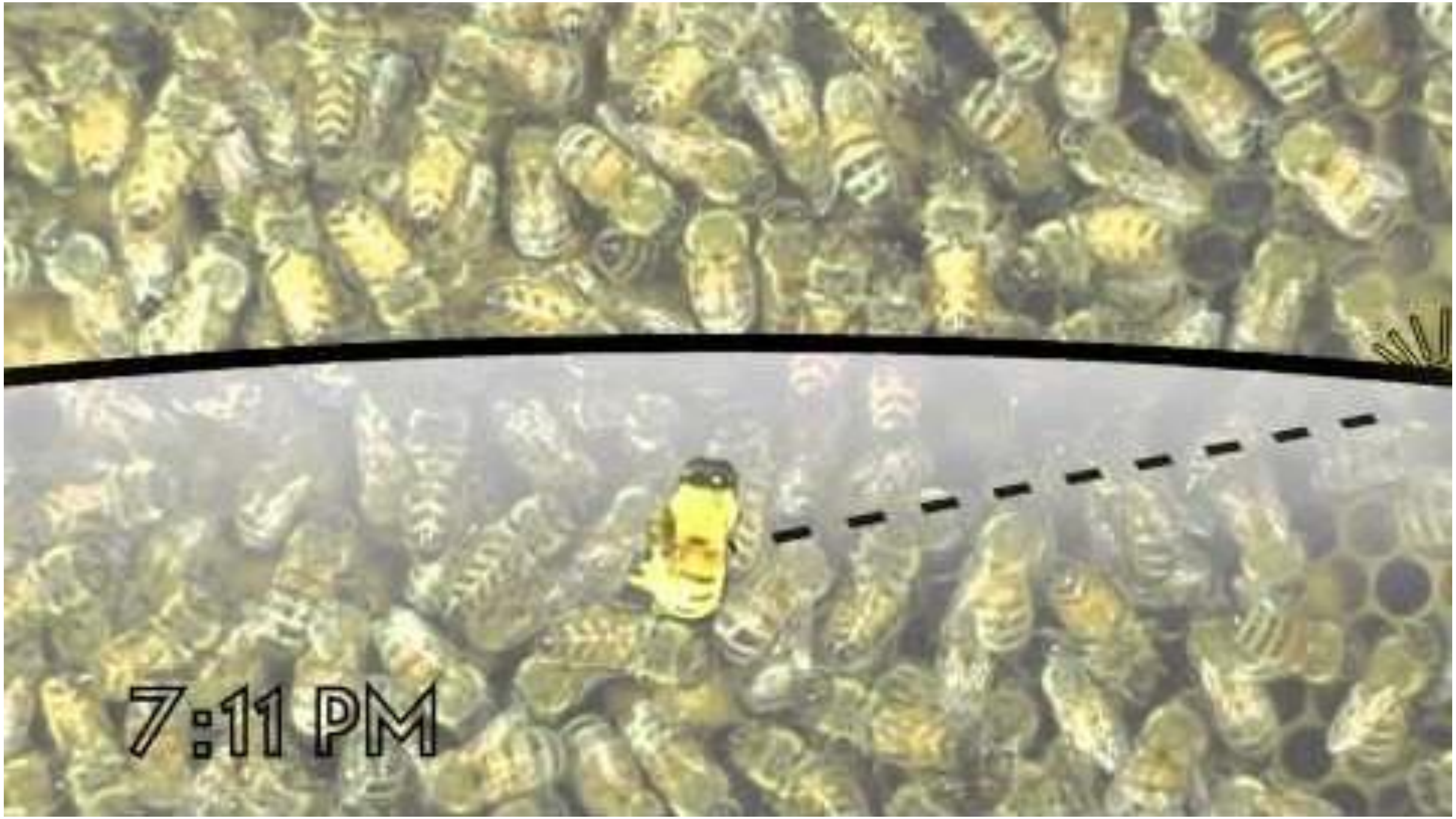
Line 14, Column 25 Tab Size: 4 Python

Fitness function

Search space

Exercise: implement your own fitness function (or use one from this list) and **optimize it with FST-PSO**

Artificial Bee Colony: the inspiration



Video courtesy of [Georgia Tech College of Computing]

Artificial Bee Colony

- Artificial Bee Colony (ABC) is a **Swarm Intelligence** meta-heuristic for GO
  - Inspired by the **foraging behavior** of **honey bees**
  - «Food» and «nectar» are metaphors for candidate solutions and fitness values

- In ABC, bees can have tree roles: **employed**, **onlookers**, and **scouts**
  - **Employed** bees are assigned to food sources and gather information about that region
  - **Onlooker** bees are distributed to the food sources, proportionally to nectar amount
  - **Scouts** perform **random search** in the search space, looking for new food sources

- By iterating the aforementioned process, the colony identifies the optimal food sources (i.e., optimal solutions)
- [Karaboğa and Basturk, J Glob Optim 2007]

General functioning of the ABC algorithm

- Each onlooker bee watches the waggle dances and then **chooses probabilistically** one food source to visit and «exploit»
  - The probability is proportional to the nectar (i.e., to the fitness values). For instance, **roulette wheel**
  - Onlookers exploit the food sources by using a **neighboorhood function** (e.g., a random perturbation of the candidate solution coordinates)

- **Employed bees** whose solutions cannot be improved for a **predetermined number of trials** (sometimes called the «limit») become **scout bees**

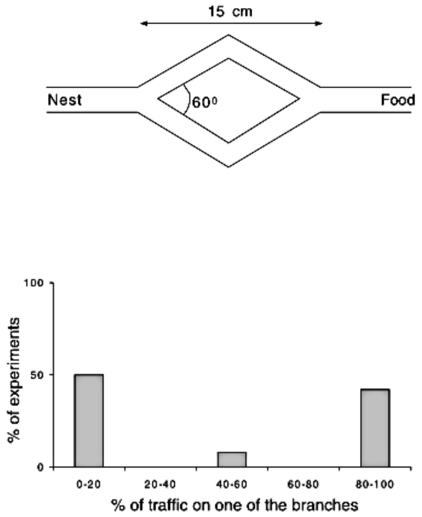
- The are some hidden **hyperparameters** in the ABC algorithm:
  - the **proportion** of employed, onlooker, and scout bees
  - the parameters governing the **neighborhood function**
  - the **number of trials**

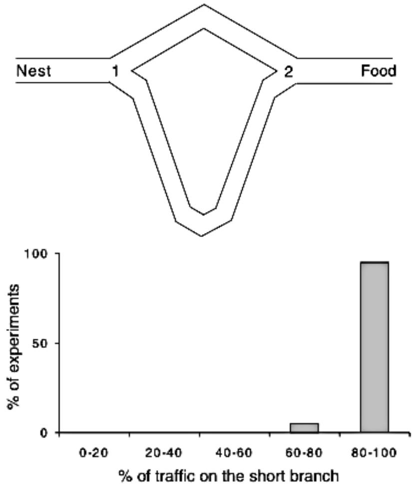
Stigmergy

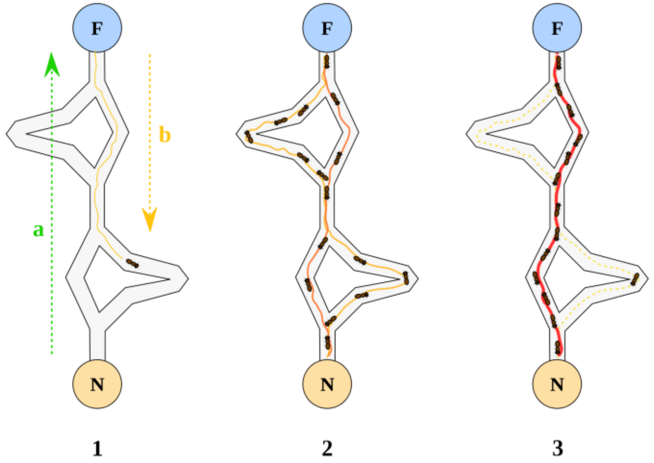
- **Stigmergy** is a mechanism of **indirect coordination** between agents mediated by the **environment**
  - Stigmergy is a form of **self-organization**: it produces complex, seemingly intelligent structures, without need for any planning, control, or even direct communication between the agents
  - Stigmergy supports efficient collaboration between extremely simple agents, who lack any memory, intelligence or even individual awareness of each other (see [Heylighen, Cogn Syst Res 2016])
- A **stigmergic system** displays three characteristics:
  1. **A global environment** (comprised of multiple local environments) **partially perceivable** through an internal dynamics that governs its temporal evolution
  2. **A set of agents** populating the environment with **no single agent** (nor cluster of agents) **having global knowledge** of the system's state. Rationality is bounded, behavior self-organizes and emerges (through adaptation and stochasticity)
  3. **Novel features arise** from interactions of 1 and 2, features that are neither predictable nor reducible to simpler constituents (complexity)
  - See [Marsh and Onof, Cogn Syst Res 2007]
- Classic example of stigmergy in Nature: **ants' pheromone trails**

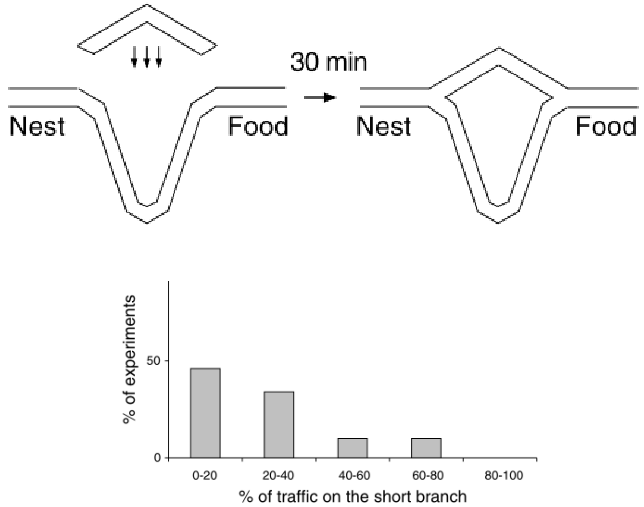
Double bridge experiments

- Ants must find the **shortest route** between nest and food (a) and back (b)
  - Ants wander **randomly** and leave pheromone on the ground
  - Shortest path is **reinforced** with more pheromone: the ants on that path lay pheromone trails faster
  - The most reinforced path is **more attractive for ants**; pheromone trails on longer paths **evaporate**
  - If we remove the shortest bridge, **the colony adapts** to the new environment by using the longest bridge
  - If the path is re-attached after 30’, **due to stigmergy** the ants **keep on using the longer route**
  - See [Goss *et al*., Naturwissenschaften 1999]









Figures taken from [Dorigo and Stützle, Ant Colony Opimization 2004]

Ant Colony Optimization

- **Ant Colony Optimization** (ACO)
  - GO meta-heuristic inspired by the collective behavior of ants
  - Designed for **combinatorial optimization** (applied to TSP, graph coloring, knapsack problems…)
  - [Dorigo *et al*., Fut Gen Comp Syst 2000]

- ACO relies on **stigmergy** and **simulated pheromone trails**
  - Pheromone is **left by ants** in the search space and **evaporates over time**
  - Pheromone traces **guide the behavior** of the ants and, unless it is **reinforced**, it disappears throughout the iterations
  - Ants are **stochastic solution building procedures** exploiting both simulated pheromone and available **heuristic information** about the problem being solved

- The possible paths between nest and food can be formalized using a graph $G = (V, A)$
  - V is a set of nodes, while A is the set of arcs connecting the nodes, representing **possible paths**
  - Each arc (i, j) has a **heuristic information** $\eta_{i,j}$ and a **pheromone** trail $\tau_{i,j}$

ACO algorithm

- The ACO algorithm is pretty simple:

```
while not termination_criterion():
      build_ant_solutions()
      update_pheromones()
      daemon_actions()  # optional
return best solution
```

ACO algorithm

- The ACO algorithm is pretty simple:

```
while not termination_criterion():
      build_ant_solutions()
      update_pheromones()
      daemon_actions()  # optional
return best solution
```

During this phase, **each ant builds a candidate solution**

For instance, **a random path on the graph G**

All solutions are evaluated using the **fitness function**

ACO algorithm

- The ACO algorithm is pretty simple:

```
while not termination_criterion():
      build_ant_solutions()
      update_pheromones()
      daemon_actions()  # optional
return best solution
```

During this phase, the pheromone trails are **updated** in the environment

Existing pheromone traces **evaporate**

New pheromone is deposited by ants according to the **visited paths**

ACO algorithm

- The ACO algorithm is pretty simple:

```
while not termination_criterion():
      build_ant_solutions()
      update_pheromones()
      daemon_actions()  # optional
return best solution
```

All **non-stigmergic actions** (i.e., centralized actions) are performed during this phase

For instance, **local search** steps to **fix/improve** the candidate solutions

Can be used to implement **elitism**

Simple example of ACO

- **Traveling Salesman Problem** (TSP)
  - Find the minimum Hamiltonian circuit
  - Provably NP-completed problem [Garey and Johnson, 1979]

- In ACO terms
  - Cities == Nodes (i.e., ants build paths between cities)
  - *Desiderability* of visiting city j after city i == Pheromone trails on arc (i, j)
  - The heuristic information is the distance between cities, i.e., $\eta_{i,j} = \frac{1}{d(i,j)}$ where $d(i, j)$ is the distance between the cities i and j
  - The goal is to find a solution (i.e., path) that minimizes the sum of the distances
  - $\mathcal{N}_i^k$ is the valid neighboorhood of city i for ant k (i.e., cities that are connected to i and were not added to i yet)

Ants and path building

- In classic ACO for TSP, the probability for ant k of selecting a city j after city i is calculated as

$$p_{i,j}^{k}(t) = \frac{\left[\tau_{i,j}(t)\right]^{\alpha} \cdot \left[\eta_{i,j}\right]^{\beta}}{\sum_{l \in \mathcal{N}_i^k} \left[\tau_{i,l}(t)\right]^{\alpha} \cdot \left[\eta_{i,l}\right]^{\beta}}$$

provided that $j \in \mathcal{N}_i^k$

- Please note that:
  - $\alpha = 0$ implies that pheromone traces are neglected, the heuristic information dominates and the algorithm becomes a **greedy search**
  - $\beta = 0$ implies that the **heuristic information is discarded**, and the construction of solutions is only based on the pheromone traces

Pheromone update

- **Positive feedback**: the better solutions (or just the best solution found, see MAXMIN ACO) are used to increase the pheromone traces
- Existing pheromone slowly **evaporates**
  - Avoid saturation and «forget» bad paths
  - **Pheromone evaporation** is updated with an additional hyper-parameter $0 < \rho < 1$
- In classic «Ant Systems» the **pheromone update formula** is

$$\tau_{i,j}(t) = (1 - \rho) \cdot \tau_{i,j}(t-1) + \sum_{k=1}^{N} \Delta\tau_{i,j}^{k}$$

where N is the number of ants in the colony and $\Delta\tau_{i,j}^{k} = \frac{1}{L_k}$ where L_k is the length of ant k's path (provided that arc (i,j) belongs to ant k's solution)

Elitism and exploration

- We can implement **elitism** by forcing ACO to «remember» the best solution meet so far, by adding a further term:

$$\tau_{i,j}(t) = (1-\rho)\cdot\tau_{i,j}(t-1) + \sum_{k=1}^{N}\Delta\tau_{i,j}^{k} + \mathrm{e}\cdot\Delta\tau_{i,j}^{best}$$

- We can also do the opposite! The ants **«consume» pheromone** when they visit a position. This strategy promotes explorative behavior

$$\tau_{i,j}(t) = (1-\xi)\cdot\tau_{i,j}(t-1) + \xi\cdot\tau_0$$

where τ_0 is a small constant value

Summary

- Computational Intelligence global optimization methods can be roughly subdivided into two categories: Evolutionary approaches (e.g., genetic algorithms, differential evolution) and Swarm Intelligence methods (e.g., particle swarm optimization, artificial bee colony, ant colony)
- Adaptive methods can be used to avoid the fine-tuning of hyper-parameters, improving the performances on real-world problems. Adaptive methods generally exploit information about the diversity in the population or the improvement of the fitness values
- Swarm Intelligence methods are based on the cooperation between multiple simple and «stupid» agents. From their collective effort an «intelligent» behavior emerges. This effort can be direct (through communication) or be mediated by the enviroment (stigmergy)
- So far we assumed that a single fitness function must be optimized. When this is not the case, multi-objective optimization (MOO) methods must be employed: this will be the topic of the next lecture