Bio-Inspired Algorithms: An Introduction

Heitor Baldo

IMECC - Winter School
University of Campinas

July 11th - 15th, 2016

Course Summary

- Lecture 1 – Introduction, Evolutionary Computing & Genetic Algorithms
- Lecture 2 – Collective Intelligence
- Lecture 3 – Cellular Automata
- Lecture 4 – Artificial Immune Systems
- Lecture 5 – Neural Computation
- References

Lecture 1 – Introduction, Evolutionary Computing & Genetic Algorithms

What is Bio-Inspired Computing?

Bio-inspired computing (a sub-field of *natural computing*) is the design of computational models, algorithms and architectures whose structure or behaviour is abstracted from biological systems and natural processes. We have three complementary study areas [13, 16]:

- **Nature as inspiration**: bio-inspired mechanisms/algorithms (evolution, swarming, immunity) to solve real-world problems.
- **Nature as substrate**: compute *with* natural media (DNA [7], molecules, membranes [11]).
- **Nature as computation**: model and understand biological systems as information processing systems [5].

Here we will focus on **nature as inspiration**. The main idea is that computational algorithms can emerge from *simple local rules* and *populations of interacting agents*.

Why Look to Nature?

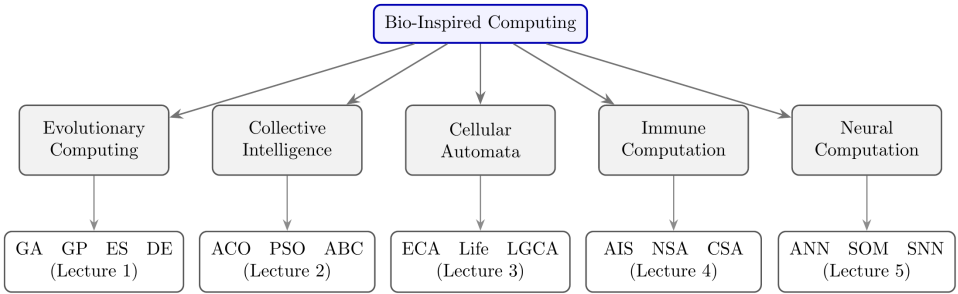
Natural systems routinely solve problems that are hard for classical algorithms:

- High-dimensional, non-linear, multimodal landscapes;
- Noisy, dynamic and uncertain environments;
- No gradient, no convexity, no closed form;
- Objective functions available only as a simulation or a physical experiment (*black-box optimisation*).

Desirable emergent properties [10, 15]:

- **Self-organisation** and adaptation;
- **Robustness** to failures;
- **Scalability** and parallelism;
- **Emergence** from local interactions.

Bio-Inspired Algorithms – Map of the Course



GA: Genetic Algorithms; GP: Genetic Programming; ES: Evolution Strategies; DE: Differential Evolution; ACO: Anti Colony Optimization; PSO: Particle Swarm Optimization; ABC: Artificial Bee Colony; ECA: Elementary Cellular Automata; LGCA: Lattice-Gas CA; AIS: Artificial Immune Systems; ANN: Artificial Neural Network; SOM: Self-Organising Map; SNN: Spiking Neural Network.

The Optimization Problem

Let S be a **search space** (binary strings, trees, graphs, etc.), $f : S \to \mathbb{R}$ be the **objective** function, and Ω be the **feasible region**. Every bio-inspired algorithm in this course is, formally, a stochastic process on S whose stationary behaviour we would like to concentrate on $\arg\min f$.

$$\text{minimize}_{\mathbf{x}\in\Omega} f(\mathbf{x}), \qquad \Omega = \{\mathbf{x} \in S \ : \ g_i(\mathbf{x}) \leq 0, \ h_j(\mathbf{x}) = 0\}.$$

What makes a problem hard: combinatorial explosion: $|S| = 2^n$, $n!$, etc.; multimodality: many local optima; noise: f observed as $f + \varepsilon$; dynamics: $f = f(\mathbf{x}, t)$.

What a metaheuristic assumes: f can be *evaluated*, not differentiated; a *neighbourhood* or *variation operator* exists on S; near-optimal is good enough.

Fitness Landscapes; Exploration vs Exploitation

A **fitness landscape** is the triple $(S, \mathcal{N}, f)$ where S is a search space, $\mathcal{N}$ is a neighbourhood structure, and f is an objective function [12, 14]. The *same* f on a *different* neighbourhood is a different landscape (this is why representation matters more than the choice of algorithm).

Exploration (diversification): sample unvisited regions; protects against premature convergence, wastes evaluations.

Exploitation (intensification): refine the best-so-far; converges fast, risks being trapped.

Examples: Genetic algorithm (exploration by mutation, large population; exploitation by selection pressure, crossover); Ant colony (exploration by pheromone evaporation; exploitation by pheromone reinforcement).

Evolutionary Computing

A **population** of candidate solutions evolves under *selection*, *variation* and *inheritance* [3, 12].

Generic evolutionary loop:

1. Initialize population randomly;
2. Evaluate fitness of each individual;
3. Select parents;
4. Recombine (crossover) and mutate;
5. Replace / form new generation;
6. Repeat until stopping criterion.

Mathematical modeling: Population: *multiset of points in* S; Fitness: *objective function* f; Natural selection: *biased sampling*; Recombination: *crossover*; Mutation: *random local perturbation*; Generations: *iteration*.

The Generic Evolutionary Algorithm (EA)

The generic evolutionary algorithm is the basis for several bio-inspeired algorithms (from genetic to swarm algorithms), and they are particular instantiations of the following five choices:

The five design decisions: 1. Representation (how a solution is encoded); 2. Fitness (how quality is measured); 3. Variation (crossover and mutation); 4. Selection (who reproduces, who survives); 5. Population model (size, structure, replacement).

Algorithm 1: Generic evolutionary algorithm

Data: fitness f, population size μ, operators
Result: best individual found

1 $t \leftarrow 0$; initialise P_0 at random
2 evaluate f on P_0
3 **while** *stopping criterion not met* **do**
4 $\quad Q \leftarrow \textsc{ParentSelection}(P_t)$
5 $\quad Q \leftarrow \textsc{Recombine}(Q)$ with probability p_c
6 $\quad Q \leftarrow \textsc{Mutate}(Q)$ with probability p_m
7 $\quad$ evaluate f on Q
8 $\quad P_{t+1} \leftarrow \textsc{SurvivorSelection}(P_t, Q)$
9 $\quad t \leftarrow t + 1$
10 **return** $\arg\max_{x \in \bigcup_t P_t} f(x)$

Genetic Algorithms (GA)

Introduced by **Holland** [3] and popularised by **Goldberg** [4].

- **Representation:** Fixed-length binary strings (*chromosomes*), $\mathbf{x} \in \{0,1\}^n$.
- **Parent selection:** Fitness proportionate (roulette wheel selection).
- **Crossover:** One-point, applied with probability $p_c \approx 0.6$–0.9.
- **Mutation:** Independent bit flips with $p_m \approx 1/n$.
- **Survivor selection:** Generational replacement, often with elitism.

Selection I – Fitness Proportionate Selection

Roulette wheel: Individual i is chosen with probability

$$p_i = \frac{f_i}{\sum_{j=1}^{\mu} f_j}.$$

However, we have the following problems [4, 12]:

- **Premature convergence**: One super-individual with $f_i \gg \bar{f}$ takes over the population in a few generations.
- **Stagnation**: When f values become similar, $p_i \to 1/\mu$ and selection becomes a random walk.
- **Not invariant under** $f \mapsto f + c$: The algorithm's behaviour changes if you add a constant to the objective.

Selection II – Ranking, Truncation, Tournament

- **Linear ranking**: Sort the population, then

$$p_i = \frac{1}{\mu}\Big(s - (2s-2)\frac{i-1}{\mu-1}\Big), \quad 1 < s \leq 2,$$

where $i = 1$ is the best. Depends only on order, not on fitness magnitude.

- **Exponential ranking**: $p_i \propto c^{\,i-1}$, $0 < c < 1$.
- **Truncation** (μ, λ): Keep the best μ of λ offspring; standard in evolution strategies.
- k-**tournament**: Draw k individuals uniformly at random, keep the best.
  - selection pressure tuned by k ($k = 1$: random; $k = \mu$: elitist);
  - no global sorting, no global sums $\Rightarrow$ trivially parallel;
  - invariant under monotone transformations of f;
  - probability that the best individual is chosen: $1 - (1 - 1/\mu)^k$.

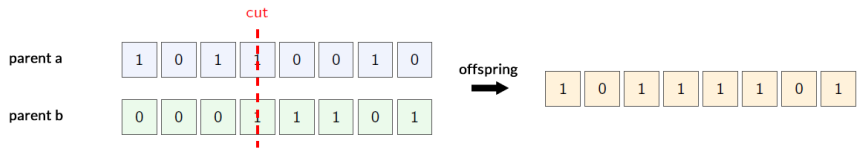
Variation I – Crossover (Binary and Integer Strings)

The purpose of crossover has been explained by two competing accounts: the *building-block hypothesis*, which posits that crossover assembles good partial solutions, and the *macromutation* view, which characterizes crossover as a large, population-directed jump [12].

Crossover methods for **binary and integer strings**:

- **One-point**: Cut at $c \sim \mathcal{U}\{1, n-1\}$; strong *positional bias* (nearby genes stay together).
- ***k*-point**: k cuts; less positional bias.
- **Uniform**: Each gene taken from either parent with probability $\frac{1}{2}$; maximal mixing, disrupts long schemata.

Example (one-point crossover):



Variation I – Crossover (Real vectors and Permutations)

Crossover methods for **real vectors and permutations**:

Real vectors:

- **Arithmetic**: $\mathbf{x}' = \alpha \mathbf{x}_a + (1 - \alpha)\mathbf{x}_b$ (contracting, reduces variance).
- **Blend BLX-α**: sample uniformly in an interval extended by α beyond the parents (non-contracting).

Permutations:

- **PMX** (partially mapped), *OX* (order), *CX* (cycle): all designed so the offspring remains a valid permutation.
- **Edge recombination**: preserves adjacency, which is what matters for the TSP.

Variation II – Mutation

Mutation guarantees **ergodicity**: with $p_m > 0$ every point of S is reachable, which is what makes asymptotic convergence proofs possible. Mutation methods:

- **Bit flip mutation** (rate p_m): Expected flips per string $= np_m$; the classical choice $p_m = 1/n$ flips one bit on average.
- **Random reset**: One or more positions are randomly selected and assigned new values from the allowed domain, such as the positive integers.
- **Gaussian**: $x_i' = x_i + \sigma\,\mathcal{N}(0,1)$; σ is the single most important parameter of a real-coded evolutionary algorithm.
- **Inversion mutation**: A substring of the chromosome is selected, reversed, and replaced.
- **Swap mutation**: Two genes are randomly selected and their values are exchanged.

Furthermore, the mutation rate plays a critical role in evolutionary search. If it is **too small**, the population may drift and eventually stall; if it is **too large**, the algorithm can degenerate into random search, reflecting the *error threshold* known from quasi-species theory. For a string of length n, the probability that a given individual remains unchanged is $(1 - p_m)^n \approx e^{-np_m}$; thus, with $p_m = 1/n$, this probability is approximately 0.37.

Population Models and Elitism

Replacement schemes

- *Generational*: $\lambda = \mu$ offspring replace the whole population.
- *Steady-state*: one or two offspring per iteration replace the worst (*generation gap* $G = \lambda/\mu$ small).
- $(\mu + \lambda)$: parents and offspring compete (elitist by construction).
- (μ, λ), $\lambda > \mu$: parents die (allows temporary fitness decrease, better for self-adaptation).

Elitism

- Copy the best e individuals unchanged. Guarantees monotone best-so-far and is required by most convergence proofs.
- Too much elitism $\Rightarrow$ loss of diversity.

Structured populations

- *Island model*: subpopulations evolve separately and exchange migrants every τ generations.
- *Cellular EA*: individuals live on a grid and mate only with neighbours (a cellular automaton whose states are solutions (see Lecture 3)).

Example: 0/1 Knapsack Problem

Suppose we have $n = 5$ items with weights $\mathbf{w} = [2, 3, 4, 5, 1]$ and values $\mathbf{v} = [3, 4, 5, 6, 1]$, with capacity $W = 8$. Let's endoce each chromosome $\mathbf{x} \in \{0, 1\}^5$, where $x_i = 1$ means item i is selected.

Fitness:

$$f(\mathbf{x}) = \sum_{i=1}^{5} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{5} w_i x_i \leq W, \quad f(\mathbf{x}) = 0 \text{ if constraint violated.}$$

Sample chromosomes:

Chromosome	Items selected	Total weight	Fitness
$[1, 1, 0, 0, 1]$	1, 2, 5	6	8
$[1, 0, 1, 0, 0]$	1, 3	6	8
$[0, 1, 1, 0, 1]$	2, 3, 5	8	**10**
$[1, 1, 0, 1, 0]$	1, 2, 4	10	0 (infeasible)

The chromosome $[0, 1, 1, 0, 1]$ achieves the best feasible fitness of 10.

The Schema Theorem

A **schema** H is a string over $\{0, 1, *\}$. It denotes the subset of the search space matching it, e.g. $H = 1{*}{*}0$ matches $\{1000, 1010, 1100, 1110\}$. We denote by $o(H)$ the number of fixed positions (*order*) and by $\delta(H)$ the distance between the outermost fixed positions (*defining length*).

Theorem (Holland's schema theorem [3, 4]):

$$\mathbb{E}[m(H, t+1)] \;\geq\; m(H, t)\,\frac{\hat{f}(H, t)}{\bar{f}(t)}\left[1 - p_c\frac{\delta(H)}{n-1} - o(H)\,p_m\right],$$

where $m(H, t)$ is the number of representatives of H at generation t, $\hat{f}(H, t)$ its average fitness, and $\bar{f}(t)$ the population average.

It is an inequality on *one* generation, it ignores the gain from crossover, and it says nothing about convergence to the optimum.

Building Blocks, Linkage and Deception

Building-block hypothesis [4]: GAs work by discovering, preserving and recombining short, low-order, high-fitness schemata.

Linkage: Genes that must be co-adapted should be close together on the chromosome, otherwise crossover keeps breaking them apart.

Deceptive problems: A problem is *deceptive* when low-order schemata guide the search away from the global optimum. An example is the trap function:

$$f_{\text{trap}}(u) = \begin{cases} n & u = n \\ n - 1 - u & u < n \end{cases}$$

where u represents the number of ones. The optimum is $11\ldots1$.

Deception, epistasis, and rugged landscapes constitute precisely the regime in which the No-Free-Lunch theorem applies [9].

Diversity: Preserving It, Measuring It

Premature convergence occurs when the population collapses toward a single basin of attraction before the budget is exhausted. Diversity is the resource that is being consumed.

Measures: Mean pairwise Hamming / Euclidean distance; population entropy per locus; fitness variance, number of distinct genotypes.

Restoring diversity: Increase p_m, restart, random immigrants; larger μ, weaker selection pressure.

Niching techniques:

- *Fitness sharing*: $f_i' = f_i / \sum_j \mathrm{sh}(d_{ij})$ with $\mathrm{sh}(d) = 1 - (d/\sigma_{\mathsf{share}})^{\alpha}$ for $d < \sigma_{\mathsf{share}}$.
- *Crowding* / *deterministic crowding*: offspring replace the most similar parent.
- *Clearing*: only the best few individuals per niche keep their fitness.
- *Island models*: geographic isolation.

The immune algorithms of Lecture 4 achieve niching *implicitly*, through affinity-proportional cloning.

Setting the Parameters

Parameter tuning (before the run): Factorial designs, racing, meta-evolutioinary algorithms. The best values are problem- and budget-dependent.

Parameter control (during the run):

- *Deterministic*: $p_m(t) = p_0/\sqrt{t}$, cooling schedules.
- *Adaptive*: Feedback from the search (e.g. Rechenberg's 1/5 success rule [2]).
- *Self-adaptive*: Parameters encoded in the genome and evolved with it.

Rechenberg's 1/5 rule [2]: Measure the fraction p_s of successful mutations over a window:

$$\sigma \leftarrow \begin{cases} \sigma/c & p_s > 1/5 \quad \text{(expand)} \\ \sigma \cdot c & p_s < 1/5 \quad \text{(contract)} \\ \sigma & p_s = 1/5 \end{cases}$$

with $c \approx 0.817$. Derived for the sphere and corridor models.

Beyond GA: Evolution Strategies (ES)

Developed by Rechenberg and Schwefel for experimental optimization of physical designs [2]. The following notation is adopted:

- $(1 + 1)$: one parent, one offspring, elitist
- $(\mu + \lambda)$: elitist survivor selection
- (μ, λ): non-elitist, $\lambda \geq 7\mu$ typical
- $(\mu/\rho \overset{+}{,} \lambda)$: ρ parents recombined

Self-adaptive mutation:

$$\sigma_i' = \sigma_i \exp\big(\tau' \mathcal{N}(0,1) + \tau \mathcal{N}_i(0,1)\big)$$
$$x_i' = x_i + \sigma_i' \mathcal{N}_i(0,1)$$

with $\tau \propto 1/\sqrt{2\sqrt{n}}$ and $\tau' \propto 1/\sqrt{2n}$.

(μ, λ) **for self-adaptation**: An individual carrying a bad σ must be allowed to die even if its objective value is momentarily good, otherwise poor strategy parameters survive on lucky fitness.

Beyond GA: Evolutionary Programming and Differential Evolution

Evolutionary programming (EP) [1]:

- Originally: Evolve finite-state machines for sequence prediction.
- No recombination: Mutation only; each parent produces exactly one offspring.
- Survivor selection by stochastic round-robin tournament.
- Modern EP is close to $(\mu + \mu)$ self-adaptive ES.

Differential evolution (DE) [8]:

For each target $\mathbf{x}_i$, choose distinct a, b, c:

$$\mathbf{v}_i = \mathbf{x}_a + F(\mathbf{x}_b - \mathbf{x}_c), \qquad F \in (0, 2)$$

binomial crossover with rate CR, then *greedy* one-to-one replacement: $\mathbf{x}_i \leftarrow \mathbf{u}_i$ iff $f(\mathbf{u}_i) \leq f(\mathbf{x}_i)$.

Key insight: The population itself determines the mutation scale. Initially, difference vectors are long (exploration); as the population contracts, they decrease (exploitation).

Beyond GA: Genetic Programming (GP)

Genetic programming evolves computer programs, represented as syntax trees over a *function set* $\mathcal{F}$ and a *terminal set* $\mathcal{T}$ [6].

Requirements on $\mathcal{F}$ and $\mathcal{T}$:

- *Closure*: Every function must accept every possible argument value (hence *protected* division, protected $\log$).
- *Sufficiency*: The target expression must be constructible.

Operators:

- Crossover: Exchange randomly chosen subtrees;
- Mutation: Replace a subtree by a new random one; point mutation of a node;
- Initialisation: *Grow*, *full*, or *ramped half-and-half*.

Convergence and the No-Free-Lunch Theorem

Convergence results:

- A GA with $p_m > 0$ and elitism is a Markov chain that visits the optimum with probability $\to 1$ as $t \to \infty$. Without elitism, it does not necessarily converge.
- Such results say nothing about *speed* (exhaustive search converges too).
- Useful runtime results exist only for simple functions (OneMax, LeadingOnes) and simple algorithms $(1+1)$ EA.

Theorem (No Free Lunch [9]): Averaged over *all* objective functions $f : S \to Y$ with S, Y finite, every algorithm that never re-evaluates a point has the same expected performance:

$$\sum_f P(\mathbf{y} \mid f, m, a_1) = \sum_f P(\mathbf{y} \mid f, m, a_2).$$

The theorem doesn't say that all algorithms are equally good on the problems we care about; real problems are a vanishingly small, highly structured subset. It *does* say that performance claims must be tied to a problem class, and that any bias built into an operator is a hypothesis about that class.

Python – Example 1: A Genetic Algorithm for the 0/1 Knapsack

```python
import numpy as np
rng = np.random.default_rng(0)

n, C = 40, 500                                  # items, knapsack capacity
w = rng.integers(10, 60, n)                     # weights
v = rng.integers(10, 90, n)                     # values

def fitness(P):                                 # P: (mu, n) binary population
    load, val = P @ w, P @ v
    return np.where(load <= C, val, val - 5.0 * (load - C))   # penalty

def tournament(P, f, k=3):
    idx = rng.integers(0, len(P), (len(P), k))
    return P[idx[np.arange(len(P)), np.argmax(f[idx], axis=1)]]

def uniform_crossover(P, pc=0.9):
    Q = P.copy()
    for i in range(0, len(P) - 1, 2):
        if rng.random() < pc:
            m = rng.random(P.shape[1]) < 0.5
            Q[i], Q[i + 1] = np.where(m, P[i], P[i + 1]), np.where(m, P[i + 1], P[i])
    return Q

def mutate(P, pm=None):
    pm = 1.0 / P.shape[1] if pm is None else pm
    return np.where(rng.random(P.shape) < pm, 1 - P, P)
```

Python – Example 1: A Genetic Algorithm for the 0/1 Knapsack (cont.)

The Evolutionary Loop:

```
mu, gens = 100, 200
P = rng.integers(0, 2, (mu, n))
best = None
for g in range(gens):
    f = fitness(P)
    elite = P[np.argmax(f)].copy()              # elitism: keep the champion
    if best is None or f.max() > fitness(best[None])[0]:
        best = elite
    P = mutate(uniform_crossover(tournament(P, f)))
    P[0] = elite
print("best value =", best @ v, " weight =", best @ w, " / C =", C)
```

Output: `best value = 1281, weight = 499, C = 500`. A greedy value/weight heuristic gives a strictly worse packing on this instance. Note the vectorised tournament: draw a $\mu \times k$ index matrix, then take the row-wise `argmax` of the fitness.

Python – Example 2: Differential Evolution on Rastrigin

```python
import numpy as np
rng = np.random.default_rng(1)

def rastrigin(X):                       # global minimum f(0)=0, many local minima
    return 10 * X.shape[1] + np.sum(X**2 - 10 * np.cos(2 * np.pi * X), axis=1)

def de(f, d=10, mu=60, F=0.5, CR=0.9, gens=1500, lo=-5.12, hi=5.12):
    P = rng.uniform(lo, hi, (mu, d))            # initial population
    fit = f(P)
    for g in range(gens):
        for i in range(mu):
            a, b, c = rng.choice(np.delete(np.arange(mu), i), 3, replace=False)
            V = P[a] + F * (P[b] - P[c])        # DE/rand/1  mutation
            V = np.clip(V, lo, hi)
            j = rng.integers(d)                 # ensure ≥ 1 inherited gene
            m = (rng.random(d) < CR); m[j] = True
            U = np.where(m, V, P[i])            # binomial crossover
            fu = f(U[None])[0]
            if fu ≤ fit[i]:                     # greedy one-to-one selection
                P[i], fit[i] = U, fu
    k = np.argmin(fit)
    return P[k], fit[k]

x, fx = de(rastrigin)
print("f* = %.6f   ||x*|| = %.3e" % (fx, np.linalg.norm(x)))
```

Output: `f* = 0.000007`, $\|\mathbf{x}^*\| = 1.8 \times 10^{-4}$. In $d = 10$ the Rastrigin function has of the order of 10^{10} local minima on $[-5.12, 5.12]^{10}$.

Homework (Lecture 1)

1. On the genetic algorithm:
   - Replace the penalty by a *repair* operator that drops the item with the worst value/weight ratio until the load is feasible. Which converges faster?
   - Set $k = 1$ (no selection pressure), then $k = \mu$ (extreme pressure). Observe stagnation and premature convergence respectively.
   - Plot the mean pairwise Hamming distance per generation (diversity curve).

2. On the differential evolution algorithm:
   - Sweep $F \in [0.3, 1.0]$ and $CR \in \{0.1, 0.5, 0.9\}$ on Rastrigin and on the sphere. Separable problems prefer small CR.
   - Implement `DE/best/1/bin` by replacing $\mathbf{x}_a$ with the current best. Faster, but check how often it now gets trapped.
   - Reduce μ to 10: the difference vectors lose diversity and the search stalls (the population *is* the step-size distribution).

References I

[1] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution.* Wiley, 1966.

[2] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Frommann-Holzboog, 1973.

[3] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[4] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[5] C. G. Langton (ed.). *Artificial Life.* Addison-Wesley, 1989.

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[7] L. M. Adleman. *Molecular computation of solutions to combinatorial problems.* Science, 266:1021–1024, 1994.

[8] R. Storn and K. Price. *Differential Evolution – a simple and efficient heuristic for global optimization over continuous spaces.* Journal of Global Optimization, 11:341–359, 1997.

[9] D. H. Wolpert and W. G. Macready. *No free lunch theorems for optimization.* IEEE Trans. Evolutionary Computation, 1(1):67–82, 1997.

References II

[10] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press, 1999.

[11] G. Păun. *Computing with membranes.* Journal of Computer and System Sciences, 61(1):108–143, 2000.

[12] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing.* Springer, 2003.

[13] L. N. de Castro. *Fundamentals of Natural Computing: Basic Concepts, Algorithms, and Applications.* Chapman & Hall/CRC, 2006.

[14] A. P. Engelbrecht. *Computational Intelligence: An Introduction.* 2nd ed., Wiley, 2007.

[15] D. Floreano and C. Mattiussi. *Bio-Inspired Artificial Intelligence: Theories, Methods, and Technologies.* MIT Press, 2008.

[16] L. Kari and G. Rozenberg. *The many facets of natural computing.* Communications of the ACM, 51(10):72–83, 2008.