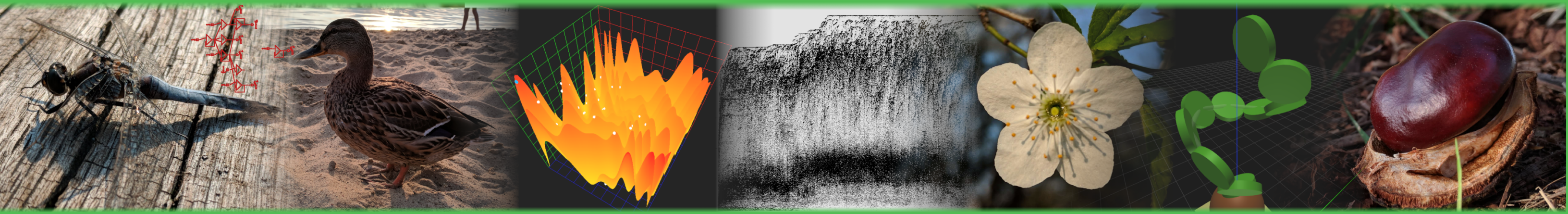
Artificial Life and Nature-Inspired Algorithms

[citing this script]

Maciej Komosinski

2025



Resources useful to recall and review most of the ideas presented during lectures on **Artificial Life**.

https://www.cs.put.poznan.pl/mkomosinski/site/?q=biologically-inspired-computing-and-artificial-life

Contents

Chapter 1

Artificial Life – introduction

Video for this chapter: https://youtu.be/4u75vgmIq-U

All videos for this script: https://www.youtube.com/playlist?list=PLWsgSwUaSMpE0KeeOWfVsxmfIN4RtfT

Lecture meetings:

- Oct. 2: video #1
- Oct. 30: questions (non-graded test) and discussion on videos #2–#8 you watched
- Nov. 27: questions and discussion on videos #9–#12 you watched
- Jan. 8: questions and discussion on all lectures (#1–#14)
- January 22 – the final test (the entire playlist).

The remaining odd-week Wednesdays (Oct. 16, Nov. 13, Dec. 11): optional consultations – if you need one, come to the lecture room.

1.1 Definition, methodology, goals

Discussion: “What is life?”

Artificial Life[1] (AL, ALife) [Sip95]:

- is an interdisciplinary research enterprise aimed at understanding ***life-as-it-is*** (***life-as-we-know-it***) on Earth and ***life-as-it-could-be*** (larger domain of “bio-logic” of possible life)
- synthesizes life-like phenomena (embodiment and physical constraints for the self-organization) in chemical (wetware), electronic (hardware) [AK09], software [KA09] (cf. movie “Her”, 2013; movie “Transcendence”, 2014), and other artificial media
- is devoted to understanding life by attempting to abstract the fundamental dynamical principles underlying biological phenomena, and recreating these dynamics in other physical media, such as computers, making them accessible to new kinds of experimental manipulation and testing [Lan97]
- redefines the concepts of artificial and natural, blurring the borders between traditional disciplines and providing new insights into the origin and principles of life.

Complementary research methods (Fig. 1.1):

- most research (biology and AI, too) is essentially *analytic*, breaking down complex phenomena into their basic components (which is not always possible),
- ALife is *synthetic*, attempting to construct phenomena from their elemental units – this is inevitable when trying to understand emergent phenomena.

Main goals of ALife:

- Increasing our understanding of nature by studying existing biological phenomena. Examples are provided in Sect. 5.5.
- Enhancing our insight into applicable artificial models (this requires studying complex systems) in order to improve their performance. Examples are software development through evolution (Genetic Programming, GP) or devising biologically-inspired optimization algorithms. This is where we will focus during this course.

Sample questions for ALife:

[1] http://en.alife.pl/main/e

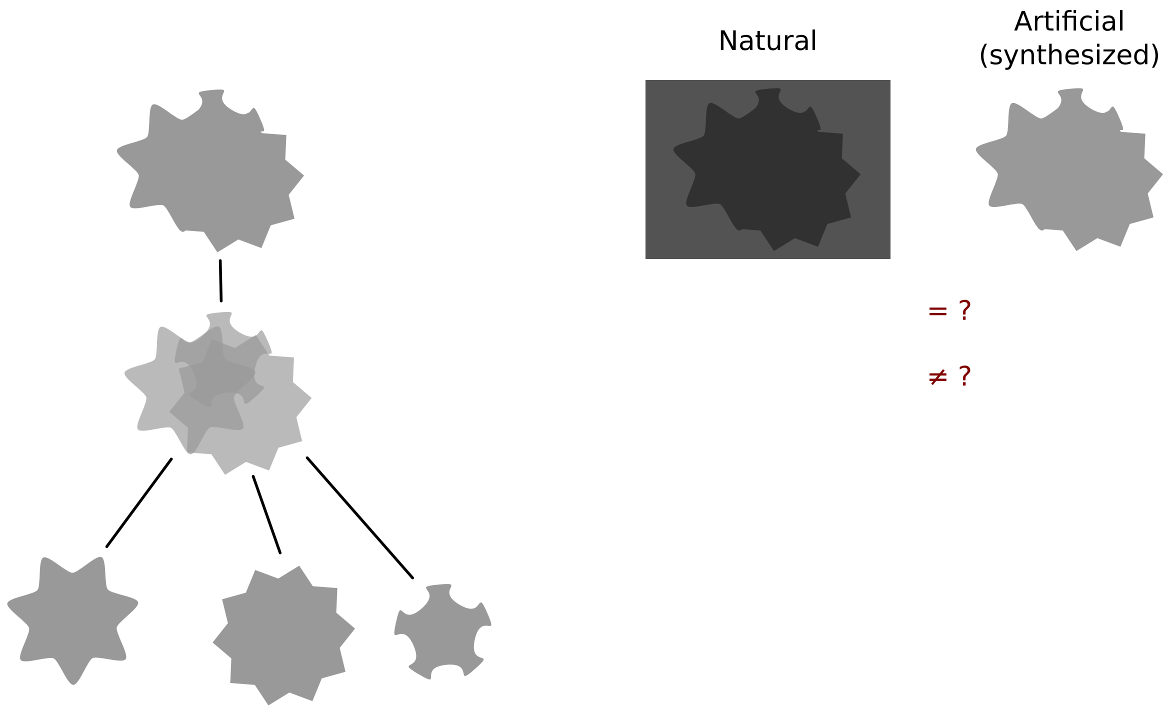


Figure 1.1: Left: analytic/synthetic research methods. Right: synthesis as a way of the inference about how complex systems are built and how they work. Consider two examples: neurons, brain, thoughts and companies, market, stock prices. Another use case: consider the vertical axis to be time (top=*present*) – sometimes time obscures the past and we cannot know what happened (e.g. how life emerged, how species evolved, how atypical supernovae were created [JMK20]), so to learn what might have happened we have to build models of changes in time, simulate them and compare the outcomes to the present state.

- Can a machine reproduce?[2] (John von Neumann, early 1950's – CA, Sect. 5.3)
- Can software be evolved? (John Koza – GP, Sect. 3.5)
- How are sophisticated robots built to function in a human environment?
- Can an ecological system be created within a computer?
- How do flocks of birds fly?

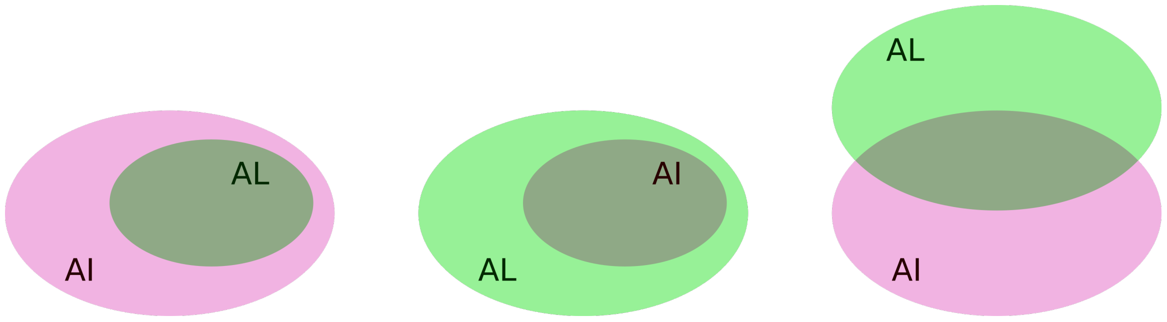


Figure 1.2: Three possible relationships between AL and AI. Left: AI-centric/traditional. Middle: common sense. Right: right.

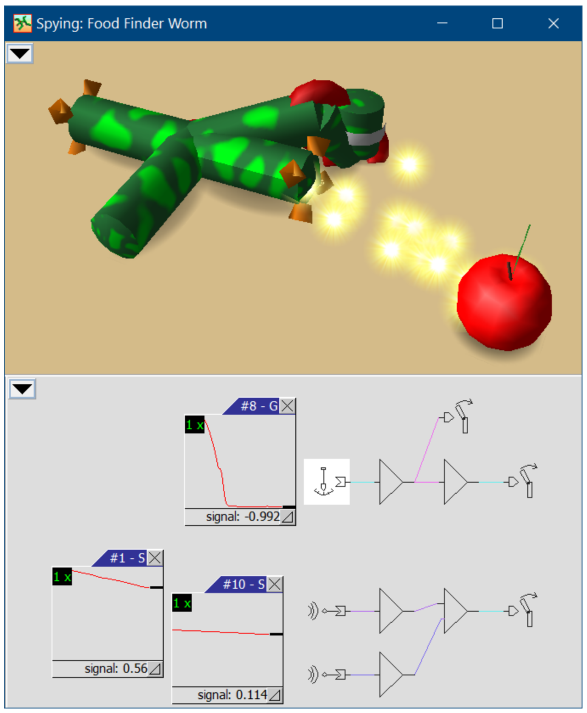


Figure 1.3: A sensory input space (here, 3D) is all a creature can sense. Contrary to what we, its observers, can sense...

1.2 Artificial life vs. artificial intelligence

A remark on notation: sometimes a difference between “Artificial Life” and “artificial life” is emphasized (same as with “Artificial Intelligence” and other domains/disciplines/fields) and such a distinction is useful, but capitalization rules say not to do it[3].

[2]The ability to repair is a part of reproduction, and repair may concern the https://en.wikipedia.org/wiki/Trolley_problem.

[3]https://english.stackexchange.com/questions/6246/capitalize-fields-of-study

A new paradigm of intelligence states that intelligence is not an abstract process, but it rather requires *situatedness* in the environment and *embodiment*. This allows agents to influence the data they perceive; an agent-environment interaction emerges, and *sensory-motor coordination* is required. Sometimes an agent is expected to react differently to the same stimuli (in robotics this is called the *perceptual aliasing problem*[4]: one should act differently in seemingly identical situations – situations that are perceived as identical, compare Fig. 1.3). Usually the solution is then to acquire more information or to transform/translate the information already possessed. Thanks to embodiment, an agent can do it (*active perception*), e.g. by changing the way they perceive objects, by reaching for additional data that will discriminate between agent's decisions, or even by self-observation and analyzing their interaction with the environment [NP99]. Example 1: a perfectly symmetrical football playing field and identical players: how do you tell your own goal from your opponent's goal? Example 2: a tiny bacteria and chemotaxis. Example 3: accidents of autonomous cars. Example 4: no eyesight.

1.3 What life is and what it is not: definitions of life

From [Ada98], for an extended discussion see [Life10]:

- **Physiological Definition**: Focuses on physiological functions such as breathing, moving, digesting, etc, to construct a list of requirements that will distinguish living from non-living. *Outdated.*
- **Metabolic Definition**: Centers on the exchange of materials between the organism and its surroundings as the only requirement for it to be alive. *Too narrow?* or *Too general?*
- **Biochemical Definition**: Classifies living systems by their capability to store hereditary information in nuclear acid molecules. *Focuses on DNA/RNA. Too narrow.*
- **Genetic Definition**: Focuses on the process of *evolution* as the central defining characteristic of living systems, without regard to *how* the information is coded (i.e., independently of substrate).
- **Thermodynamic Definition**: Describes systems in terms of their ability to maintain low levels of *entropy* (i.e., disorder) despite a noisy environment [Sch44]. *Too general?*

[4]Perception: receiving information from the environment (and from oneself) with senses or sensors. Cognition: receiving, processing and storing information in order to control one's behavior. More in Sect. 5.4.4.

- **Physics-based Definition**: Life is a property of an *ensemble* of units that *share information* coded in a physical substrate and which, in the presence of noise, manages to keep its entropy significantly lower than the maximal entropy of the ensemble, on timescales exceeding the "natural" timescale of *decay* of the (information-bearing) substrate by many orders of magnitude.

According to one theory, life is a perfect dissipator of energy[5]. The universe tends toward disorder, decay, and equilibrium (entropy grows). But life maintains low entropy due to a non-random, specific structure; the complexity of life on Earth increased during evolution, as if entropy decreased. However, Earth is not a closed system – Sun provides energy. Life decreases its own entropy by increasing the entropy of its surroundings (it absorbs order and ejects disorder), and so energy gradients (energy flow) facilitate the emergence of life[6]. The original source of the lowest entropy was the Big Bang, and life (as a local phenomenon of order) arises naturally as the energy is redistributed into the most random possible state.

Why these are alive?

- **Viruses:** which definitions support it, and which are against it?
- **Sand Dunes**: as they get blown through the desert, end up growing, and splitting off smaller dunes. Many think of this as a form of self-replication.
- **An Organized Religion**: a meme of sorts, with no physical representation. Certainly an 'idea' which can spread like wildfire as a parasite through a host population of 'people'. It can also be argued to be the people who are part of the religion, and then it would have a genetic basis.
- **Chain Letter**: This is another meme, only this one *does* certainly have a physical representation. And I know many people who wish Chain Letters would just die already...
- **Prions**: These proteins will infect a living organism and bend its proteins around to become more prions (e.g., Mad Cow and Creutzfeld-Jacob disease). It requires a very select environment to work (so do we. . .)
- A **Robot** which moves around collecting resources from the environment and returns them to a robotic 'hive' to build more robots just like it.

[5] https://www.youtube.com/watch?v=GcfLZSL7YGw

[6] https://www.sciencefocus.com/science/the-origin-of-life-a-new-theory-suggests-physics-holds-the-answer/

Epistemology (the study of the nature of knowledge, justification, and the rationality of belief). What do we commonly consider alive? How to identify life? What qualities living beings possess? [Dom99]

According to Farmer and Belin [FB90]: Life is a pattern in spacetime, self-reproduction, storage of a self-representation, a metabolism, functional interaction with the environment, interdependence of parts, stability under perturbations, the ability to evolve.

Mark Bedau, in turn, proposed flexible, *supple adaptation* [Bed96] as a property (a determinant) of life. This was because it seemed that to determine whether objects are alive, their own properties are not enough – one should also looked at these objects from the perspective of the system they constitute, its behavior and dynamics. To determine whether continuous, supple adaptation takes place, the evolution of organisms is compared in the tested model and in the additionally created *null model*, in which mutations occur as often as in the tested model. However, in the *null model*, genotypes have no effect on survival. The *traits* (characteristic properties, features) of organisms are studied: if they occur statistically significantly more frequently in the model under study than in the *null model*, it means that the system under study shows adaptive evolution, which, according to Bedau, is an indicator of life.

In the first artificial life models, we either do not notice such characteristic properties of simulated organisms, or they appear until the "task" posed by the environment is "solved" (then the evolution no longer progresses), and therefore statistically they do not last longer than in the *null model*. On the other hand, by using the property of supple adaptation as a definition of life, we come to the conclusion that individual people are less alive than the biosphere as a whole, and also less than some chemical or economic systems [Dom99].

The properties (determinants) proposed by Farmer and Belin and the feature of supple adaptation were criticized; Paul Domjan invented the "Romance Novel System" – a system in which romance writers are also readers, all novels are published, and the authors can read them and freely borrow ideas from themselves [Dom99]. Domjan believes that his system in a sense and with a proper interpretation has all the properties of life mentioned above, yet neither this system nor its components would be by common sense, or intuitively, considered alive.

More about the role of *null model*: if you are studying an evolutionary model in terms of adaptation (e.g., looking for the frequency of genes, traits, etc.), it is beneficial to create the corresponding *null model* ("*neutral shadow*") [RB99]. This new model is the 'shadow' of the examined model, because their evolutionary parameters, the rules of survival and reproduction, etc., are identical, but genotypes have no adaptive significance in the null model. Selection in the *null model* is purely random, while in the model under investigation it usually removes poor genotypes. Thus *null model* helps minimize the impact of evolutionary phenomena such as *chance* and *necessity*[7]. When comparing the examined model with its "shadow", the effect of adaptation becomes clearly distinguished, and the remaining evolutionary effects (potentially overlapping the adaptation process and hindering analysis) can be filtered out.

1.4 Research interests and applications

- Self-organization
- Chemical origins of life, Autocatalytic systems, Prebiotic evolution, RNA systems, Evolutionary/artificial chemistry
- Fitness landscapes
- Natural selection
- Artificial evolution
- Ecosystem evolution
- Multicellular development
- Natural and artificial morphogenesis
- Learning and development
- Bio-morphic and neuro-morphic engineering
- Artificial / Virtual worlds
- Simulation tools
- Artificial organisms
- Synthetic actors
- Artificial (virtual and robotic) humanoids
- Intelligent autonomous robots

[7]"Everything existing in the universe is the fruit of chance and necessity" is attributed to Democritus, an ancient Greek pre-Socratic philosopher (~400 BC). "Chance and Necessity: Essay on the Natural Philosophy of Modern Biology" is a 1970 book by a French biochemist, Nobel Prize winner (1965, for discoveries concerning genetic control of enzyme and virus synthesis) Jacques Monod. He interpreted the processes of evolution to show that life is only the result of natural processes by "pure chance" – https://en.wikipedia.org/wiki/Chance_and_Necessity.

- Evolutionary Robotics / Design
- Life detectors
- Self-repairing hardware
- Evolvable hardware (EHW)
- Emergent collective behaviors
- Swarm intelligence
- Evolution of social behaviors
- Evolution of communication
- Epistemology
- Artificial Life in Art. Evolutionary art (example); evolutionary music; creative evolutionary design; conceptual evolutionary design; strategy evolution; collaborative evolutionary systems; interactive evolutionary systems; evolutionary sculpture; evolutionary architecture.

Works on artificial life are not limited to evolving systems. Sometimes the goal is to simulate life focusing on the realism of behaviors. Realistic macro-simulations were created in this respect – one of the first ones were Artificial Fishes [TTG94] and the Humanoid and Humanoid-2 projects [Tha+95]. For such purposes, virtual reality (VR) techniques, interactive actors (*avatars*), or Lindenmayer systems (L-systems, Sect. 5.1) are used.

Selected applications of AL [KC06]: robotics, design, engineering and construction of three-dimensional objects; robots adapting to tasks, to environments and to their own damage; computer animations (movies, advertisements, games, simulations); medicine, therapy and physical therapy; study of group and social behavior, crowds, schools (fish, birds), ecosystems (forests, bacteria, viruses); studying the behavior of complex adaptive systems[8] (biology, economics, market and consumer models) and biological processes (e.g., building a spider's web, the structure and working principles of the eye); research on distributed knowledge and information, intelligence, communication ability and language evolution; creating robust algorithms and protocols for time-varying/mobile computer networks; integrated circuits adapting to computation.

Sample questions

What is perceptual aliasing? What is its cause and how to prevent it?

[8] https://en.wikipedia.org/wiki/Complex_adaptive_system

Chapter 2

Optimization

2.1 Single-solution neighborhood search

Before we start talking about biologically-inspired optimization algorithms (including evolutionary algorithms), we should learn some basics of optimization theory and the simplest optimization algorithms [url].

- `OptIntroduction.pdf` https://youtu.be/_EUoWFFA_mo
  Fundamentals of optimization – slides up to "Homework" and the red landscape at the end.
- `LS-en.pdf` https://youtu.be/b-gGAE0mP7U
  The idea behind local search – neighborhood and its size for a permutation and for a vector of numbers; the difference between *greedy* and *steepest*.
- `MetaheuristicsSummary.pdf` https://youtu.be/paY0XcrL08o
  The idea behind SA and TS; slides up to "But...".

Sample questions

- How is TSP defined?
- How many solutions are there in the TSP?
- Why is it hard to find global optimum in combinatorial problems?
- How many solutions must be checked in the TSP in the worst case to find the global optimum?
- How the exhaustive (a.k.a. brute force or full search) algorithm works?
- What is the difference between Simulated Annealing and Greedy Local Search?
- What is the difference between Tabu Search and Steepest Local Search?
- What is the role of "temperature" in Simulated Annealing? How the initial temperature value should be adjusted?
- What is stored in the "tabu" list in Tabu Search?
- Are TS and SA faster or slower than Greedy and Steepest?
- What is the stopping condition for LS, SA and TS?
- Do SA or TS always discover the global optimum? Why?

2.2 Adjusting parameter values; interactive and batch application

As in the case of many AI algorithms, the optimal values of parameters depend on the nature of the task being solved, which, however, a priori (and usually also a posteriori) is unknown. The selection of values also depends on the application – interactive (*on-line*) or batch (*off-line*). With a batch approach, we are interested in the best solution found during the algorithm's operation. With the on-line approach, we are interested in making the optimization algorithm work at its best all the time. To evaluate the operation in *on-line* and *off-line* modes one can employ various statistics such as `average` and `max`.

Chapter 3

Evolutionary algorithms

3.1 Classification

Evolutionary Computation (EC) / Algorithms (EA)

EA is based upon biological observations that date back to Charles Darwin's discoveries in the 19th century: the means of natural selection and the survival of the fittest, and theories of evolution.

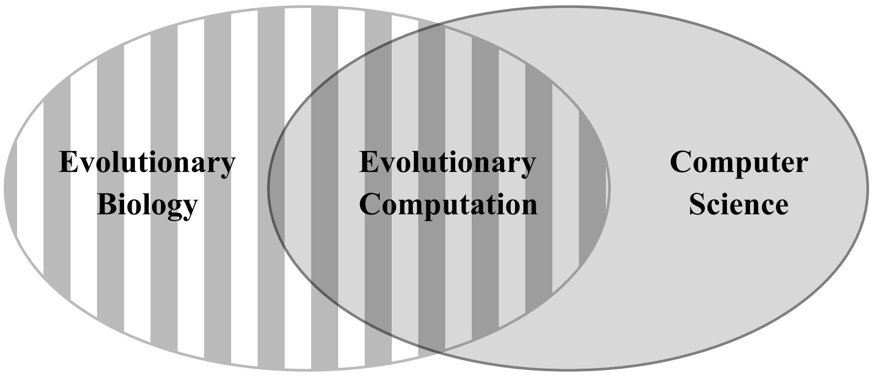


Figure 3.1: Evolutionary computation as a part of computer science and biology.

- Genetic Algorithms (GA)
- Evolution Strategies (ES)
- Evolutionary Programming (EP)
- Genetic Programming (GP)

- Classifier Systems (CFS), Genetic-Based Machine Learning (GBML)
- Various coevolutionary architectures
- ...

GA: created by John Holland (1973, 1975), made famous by David Goldberg (1989)
EP: created by Lawrence Fogel (1963), developed by his son, David Fogel (1992)
ES: created by Ingo Rechenberg (1973), promoted by Thomas Bäck (1996)
GP: developed by John Koza (1992)

Applications of EA:

- optimization of mathematical functions
- operational research – scheduling, optimization, ...
- multiple-criteria optimization and decision support
- image processing, pattern recognition
- adaptive algorithms in games
- control; robotics; evolutionary design
- biology – simulations (species, populations, ...)
- social sciences – simulations of groups
- artificial life
- ...

3.2 Genetic algorithms

All genotypes are binary vectors of the same, fixed length. If the optimization problem requires a different representation (e.g. a permutation), then solutions need encoding, decoding and sometimes repair. Genetic operators are unaware of the original representation of solutions, so they can be the same for every optimization problem. Compare this to nature... very different species, the same basic code.

3.2.1 Algorithm structure and parameters

Main loop:

```
$t := 0$
initialize $P(t)$
evaluate $P(t)$
**while** (**not** stopping-condition)
{
    $t := t + 1$
    select $P(t)$ from $P(t-1)$
    modify $P(t)$
    evaluate $P(t)$
}
```

Parameters:

- population size *POPSIZE*
- probability of crossing-over *PXOVER*
- probability of mutation *PMUT*

- choosing the stopping criterion
- choosing the selection mechanism (positive and possibly negative)
- adjusting parameter values of the selection mechanism

Simple demo: `http://en.alife.pl/opt/e/index.html`

Creating consecutive gene pools in GA: *steady state* (incremental) or *generational replacement*. In *steady state GA*, not all individuals undergo modification – some go to the next generation unchanged. In a special case, we change only one individual and the algorithm works smoothly, without clearly separated generations, and uses newly evolved ideas sooner (faster convergence is possible).

3.2.2 Selection

From the reproduction stage we expect good individuals to be multiplied. The stronger the dominance of better solutions over worse ones (higher selective pressure), the lower the diversity of the resulting population will be. These two aspects of selection (preference for better individuals over worse ones and maintaining diversity) are to a certain extent contradictory, although both are also desirable.

The amount of selective pressure can be expressed numerically, e.g. by dividing the probability of selecting the best individual in the population by the probability of selecting an average individual (having fitness equal to the median fitness in the population).

The selective pressure can be controlled, for example, by means of scaling of individual fitness values. Too high pressure will lead to premature convergence (to a local optimum), because the best individuals at the given time will gain preponderance and dominate the remaining solutions. On the other hand, low selective pressure will ensure a high diversity of individuals in populations, which may result in the inefficiency of the entire evolutionary process and make it similar to a random search.

Let f_i be the fitness of i-th individual ($i = 1..POPSIZE$), and e_i – the number of its expected copies in the new (consecutive) population, $e_i = POPSIZE \cdot f_i / \sum f_j$.

The most popular selection techniques are:

- Fitness proportionate random selection with replacement, commonly called the roulette wheel technique: individuals are assigned fields on the roulette wheel, the sizes of which are proportional to their fitness f_i. Then the roulette wheel is spun *POPSIZE* times, selecting the drawn individual for the new population. The same principle is implemented by the *stochastic universal sampling* method[1], which provides better properties of randomness during selection.
- Stochastic remainder selection without replacement: each individual gets as many copies in the new population as the integer part of its e_i. The remaining free places are filled by randomly deciding, for each individual with the probability being the fractional part of its e_i, whether it should go to the new population. Example: 4 individuals, $\boldsymbol{f} = [1,3,5,6]$.
- Selection according to random tournaments: k individuals are randomly drawn, and then the winner (the individual with the highest fitness) is placed in the new population. The process is repeated until all places are filled in the new population. A more careful variant of this technique ensures that each individual participates in the same number of tournaments.

Other selection techniques:

- Deterministic remainder-based selection: each individual gets as many copies in the new population as the integer part of its e_i, and the remaining free places in the population are filled in order of decreasing fractional parts of individual e_i.

[1] https://en.wikipedia.org/wiki/Stochastic_universal_sampling

- Stochastic remainder selection with replacement: each individual gets as many copies in the new population as the integer part of its expected number of copies (e_i). The remaining places are filled according to the roulette principle proportionally to the fractional part of e_i.
- Ordinal selection: individuals are assigned integer ranks that correspond to their position in ranking, from best to worst. The selection is based on the probability function that depends not on raw fitness values, but on individual positions in the ranking. Various probability functions are used – linear and non-linear, and the parameters of these functions allow one to adjust selective pressure.

Additional properties of selection:

- Elitism (elitist model): fulfills the expectation that the selection process should not cause the loss of the best individual found so far. If such an individual does not find its way to the next population in a natural way (resulting from the selection method used), it is included in it and thus the information about the best solution so far is always preserved.
- Crowding factor model: similar to nature, where species filling the ecological niche must fight for limited resources – in the crowding model, new individuals replace old individuals (from the previous population) taking into account their similarities, i.e., new individuals take the place of the old individuals most similar to them. The crowding factor (a parameter) affects the way individuals are replaced [DJ75; Mah92].

Meta-schemes of selection:

In the following selection methods, parts of the population (subpopulations) can be independently processed – these methods can therefore also act as a distribution and parallelization scheme for evolution.

- Island model: a population is split into subpopulations in which the chosen selection scheme operates (for example tournament, roulette or other). Evolution proceeds on each island independently, with periodic migration of some genotypes between islands. This model increases exploration capabilities.
- Convection selection: unlike in the traditional island model, the division into subpopulations follows the similarity of the value of the objective function of solutions. Convection selection improves the exploration ability of an EA by properly balancing selective pressure [KU17; KM18]. The way this selection method works is illustrated in animations here.

The selection techniques mentioned above have their pros and cons; in particular, the first of them – the roulette method – has a high variance that results in large differences between the actually achieved and the expected numbers of individuals. Hence many techniques were introduced (such as stochastic remainder selection without replacement) that overcome this drawback. The choice of the selection method has a big impact on the behavior of the algorithm, in particular on the ability to cross saddles[2] during optimization.

Sometimes (depending on the adopted GA architecture), in addition to using a positive selection, it is also necessary to employ a negative selection. Its role is to make room in the population for new genotypes – negative selection decides which genotypes to remove from the population. Similar mechanisms as for the positive selection can be used; two examples of naive methods are deleting the worst genotype and a random one.

Sample questions

- Enumerate and summarize the selection techniques you learned.
- What are the disadvantages and advantages of each technique?

3.2.3 Crossover

Discussion: is the crossover operator mandatory in the evolutionary algorithm?

Crossover: creates new solutions inheriting information from two (or more) "parent" solutions. For example a *single-point crossover* ("cutting" a genotype in a random location and swapping parts of the genotype)

```
a b c d e f|g h        a b c d e f G H

A B C D E F|G H        A B C D E F g h
```

or a *uniform crossover* (each bit in a child is chosen from a random parent).

3.2.4 Mutation

Discussion: is the mutation operator mandatory in the evolutionary algorithm?

Mutation: creates a new solution inheriting almost all information from one "parent" solution. A source of variability, prevents convergence. Analogous to the neighborhood operator in local optimization algorithms.

[2]https://en.wikipedia.org/wiki/Saddle_point

3.3 Evolutionary strategies

Similar to a GA, but the representation is suited for numerical optimization – it is a vector of real (i.e., floating-point) numbers. Mutation adds a normally distributed (with $\mu = 0$) random value to each gene – this is the "creep" mutation. Crossover can be uniform or arithmetic. Arithmetic crossover produces offspring that are a linear combination of parents (an average in a special case).

3.4 Evolutionary programming

EP, originally conceived[3] by Lawrence J. Fogel[4] in 1960, is a stochastic OPTIMIZATION strategy similar to GAs, but instead places emphasis on the behavioral linkage between PARENTs and their OFFSPRING, rather than seeking to emulate specific GENETIC OPERATORS as observed in nature.

Three ways in which EP differs from GAs:

1. There is no constraint on the representation. The typical GA approach involves encoding the problem solutions as a string of representative tokens, the GENOME. In EP, the representation follows from the problem. Example: a neural network can be represented in the same manner as it is implemented, because the mutation operation does not demand a linear encoding. (In this case, for a fixed topology, real-valued weights could be coded directly as their real values and mutation operates by perturbing a weight vector with a zero mean multivariate Gaussian perturbation. For variable topologies, the architecture is also perturbed).

2. The mutation operation simply changes aspects of the solution according to a statistical distribution: minor variations in the behavior of the offspring are highly probable, substantial variations are unlikely. The severity of mutations is often reduced in time.

3. EP typically does not use any CROSSOVER as a GENETIC OPERATOR.

Nowadays, "evolutionary programming" is a rarely used name. Instead we speak about an evolutionary algorithm – which generally means an algorithm adapted to the problem at hand. The degree of its adaptation varies; most often customizations involve the representation and operators.

Many representations of individuals are used: a set, list, permutation, tree, non-directed graph, directed graph, matrix, logical expressions, rules (as in GBML, Sect. 3.6), neural

[3] https://www.kanadas.com/whats-ep.html, email from https://en.wikipedia.org/wiki/David_B._Fogel

[4] https://en.wikipedia.org/wiki/Lawrence_J._Fogel

networks, automata, grammar expressions (e.g. stored as RPN[5]), expressions structured as trees, programs (as in GP, Sect. 3.5), ...

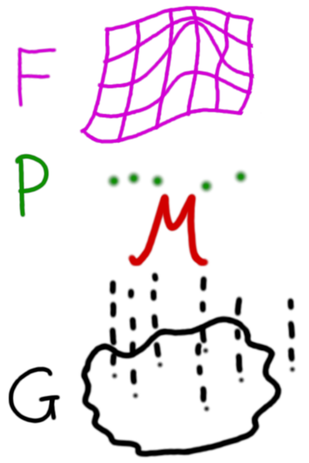
3.4.1 Crossover and mutation vs. smoothness of the fitness landscape

The way mutation and crossing over operators are defined determines which solutions are neighbors, and this determines the smoothness of the fitness landscape. The more smooth the fitness landscape, the better for optimization. "Smoothness" can be estimated by calculating *fitness–distance correlation* (FDC) – the higher, the better. The correlation C is calculated in a set of two properties of *pairs* of solutions. These two properties are the difference in fitness F of a pair of solutions and the difference in their genotypes ("distance" D). D = how different the two solutions are: the number of mutations needed to transform one into the other.

3.4.2 Embryogeny

Embryogeny: mapping genotype → phenotype. For simple representations and uniform, homogeneous spaces like the full space of bits, numbers or permutations, a trivial direct 1:1 mapping is the first (default) idea.

But is such a mapping the best choice?

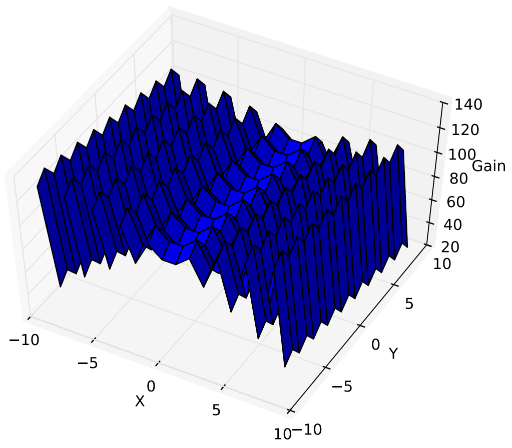


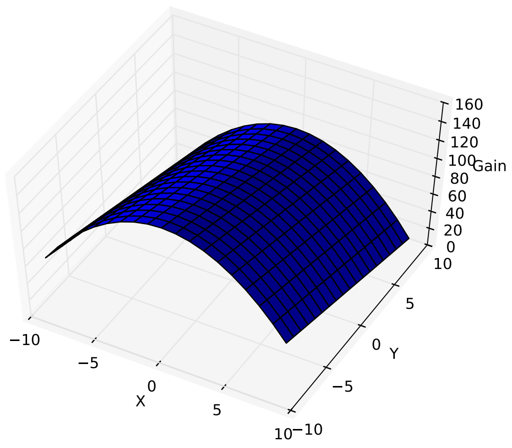
Recall RGB ↔ HSL, signal ↔ spectrum, water tap ↔ , ...

Consider in which situations the genotype → phenotype mapping should (or must?) be

[5] https://en.wikipedia.org/wiki/Reverse_Polish_notation

more sophisticated. What properties of the mapping should be provided by the procedure that maps the genotype space into the phenotype space?





Now think about the nature and the biological genotype → phenotype mapping. How it works and is it advantageous? Could this mapping be implemented better?

If the phenotype space is different from the genotype space (which is often the case – imagine the optimization of any highly complicated solution, for example a bridge, a car, a robot, . . .), then a procedure is needed to "map" one space to another (Fig. 3.2). In biology, this process is called embryogenesis (the development from the genotype to the embryo stage, i.e., building a body). But even for identical spaces, indirect mapping can be beneficial.

Embryogeny – choices and their consequences [Rot06]:

- redundancy: many genotypes → one phenotype
  - synonymous: genotypes that produce the same phenotype are neighbors
    - uniform: each phenotype is produced by the same number of genotypes
    - non-uniform: the opposite is true
  - non-synonymous: bad for optimization
- scaling of alleles: how uniformly alleles affect fitness
- locality: similarity (closeness) in genotypes correlated with similarity in their corresponding phenotypes
  - high: good! the mapping does not make the problem more difficult
  - low: adds difficulty to the problem

The above properties can be estimated numerically.

Possible reasons to use a non-trivial mapping [Ben99]:

- reduction of the search space (recursive, hierarchical etc.),
- better enumeration of the search space (resulting in a topology that increases FDC as described in Sect. 3.4.1),
- more complex solutions in the phenotype space ("growing instructions" in genotype),
- improved constraint handling (mapping **every** genotype into a valid phenotype),

and:

- compression: simple genotypes define complex phenotypes,
- repetition: genotypes can describe symmetry, segmentation, subroutines, etc.,
- adaptation: phenotypes can be grown "adaptively" (to satisfy constraints, or to correct errors),

but:

- experience is required to manually define an embryogeny that provides abovementioned benefits,
- it is hard to automatically evolve embryogeny (specific operators needed because of genetic and phenetic bloat, epistasis and excessive disruption of child solutions by genetic operators or poor inheritance of information).

In most applications, embryogeny is a set of fixed rules designed by a human that map genotypes into their meanings. More complex embryogenies are required in complicated optimization problems – an example is *evolutionary design* where robots, 3D structures, mechanical components, trusses and architectural designs, or analogue and digital circuits in electronics are optimized.

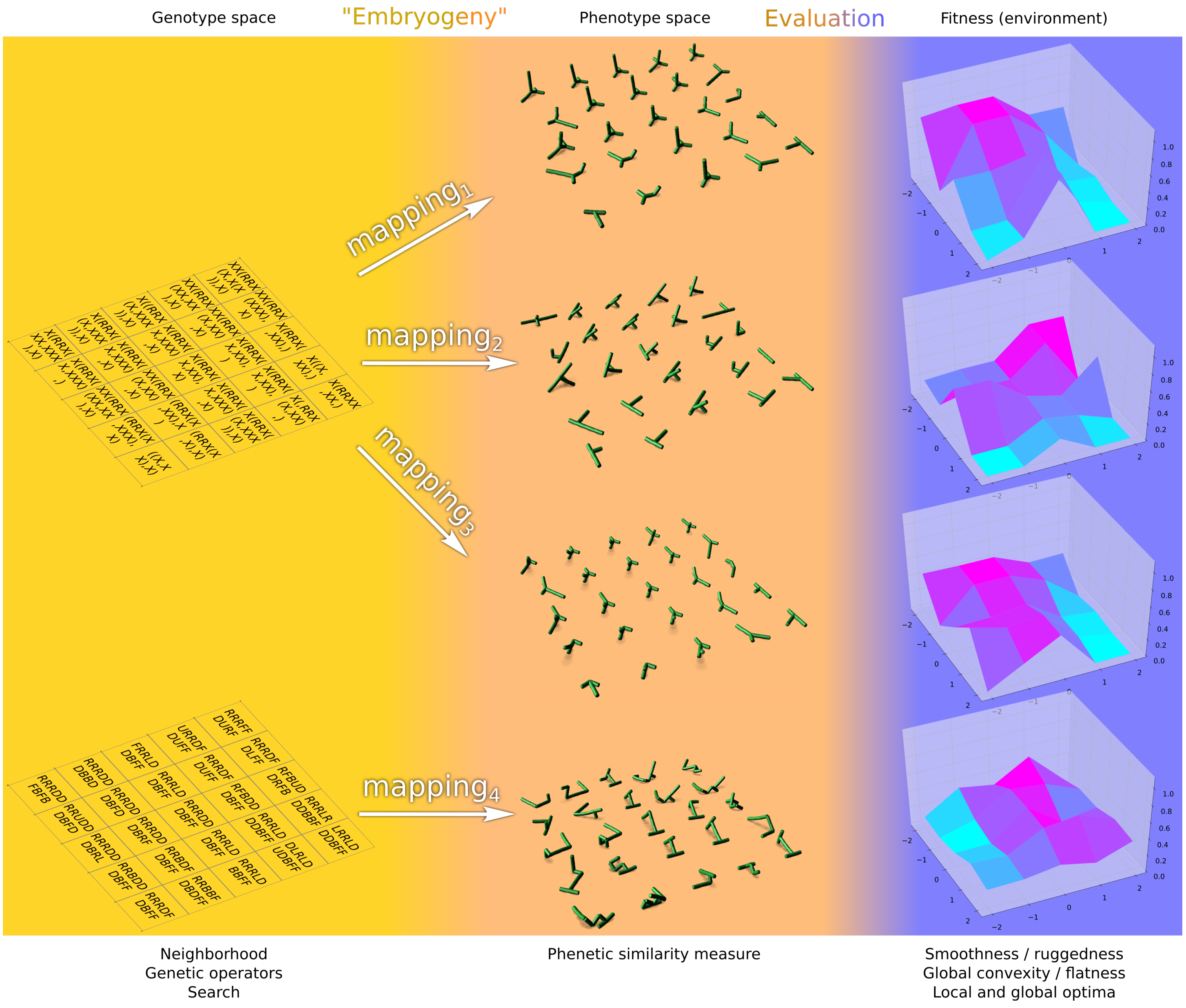


Figure 3.2: The relationship between the genetic space, the phenetic space, and the fitness landscape. Note that different embryogenies (and thus different sets of phenotypes, phenotypic topologies and fitness landscapes) may be the result of (1) different representations and their dedicated operators (two are shown), (2) different interpretations (three are shown) of genes within one representation, and (3) the same representation and the same interpretation of genes, but different mutation/neighborhood operators (not shown).

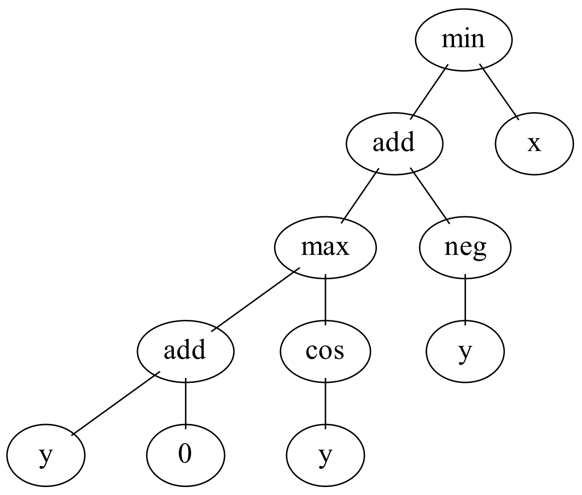


Figure 3.3: Expression `min(add(max(add(y, 0), cos(y)), neg(y)), x)` which is `min(max(y + 0, cos(y)) + (-y), x)` which is `min(x, max(y, cos(y)) - y)`.

3.5 Genetic programming

Genetic programming[6] is used to optimize expressions and programs. A characteristic property is a tree structure that represents solutions, so programs can be encoded – Fig. 3.3, although a less popular linear representation also exists.[7]

Expressions existing in a population consist of elements that belong to the set of functions $\boldsymbol{F}$ (tree nodes) and the set of terminals $\boldsymbol{T}$ (tree leaves). These sets can be composed as needed and adapted to the problem being solved. The solution space consists of all combinations of expressions composed of members of both sets.

Set of functions	
Type	Examples
Arithmetic	+, *, /
Math	sin, cos, exp
Logic	AND, OR, NOT
Conditional	IF-THEN-ELSE
Looping	FOR, REPEAT

Set of terminals	
Type	Examples
Variables	$\vec{x}$, y, x172
Constants	3, 0.45, π
Procedures	rand, go_left, read_proximity

“Procedures” can be functions or actions without arguments.

Two properties of the $\boldsymbol{F}$ and $\boldsymbol{T}$ sets are desirable:

1. *closure* – each function works for any values and types of arguments returned by any

[6]Free book: http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli08_fieldguide.pdf

[7]https://en.wikipedia.org/wiki/Linear_genetic_programming

function or terminal,

2. *sufficiency* – elements available in both sets allow one to construct a solution to the problem.

```
from deap import gp
# https://deap.readthedocs.io/en/master/tutorials/advanced/gp.html
# https://deap.readthedocs.io/en/master/examples/gp_symbreg.html

pset = gp.PrimitiveSet("MAIN", 2) # two arguments (x and y)
pset.addPrimitive(operator.add, 2)
pset.addPrimitive(operator.sub, 2)
pset.addPrimitive(operator.mul, 2)
pset.addPrimitive(operator.neg, 1)
pset.addPrimitive(min, 2)
pset.addPrimitive(max, 2)
pset.addPrimitive(math.cos, 1)
pset.addPrimitive(math.sin, 1)
pset.addEphemeralConstant("rand101", lambda: random.randint(-1,1))
pset.renameArguments(ARG0='x')
pset.renameArguments(ARG1='y')
```

Consider how the *closure* property can be ensured.

The *closure* property can be achieved by protecting functions (e.g. always calculating the absolute value of the square root argument) or penalizing invalid expressions (lowering their fitness value). Or set the CPU/program/operating system flags so that all operations do not cause exceptions... (mention here a long numerical simulation under linux and the difference of the same simulation under Windows).

```
def protectedDiv(left, right):
    try:
        return left / right
    except ZeroDivisionError:
        return 1

pset.addPrimitive(protectedDiv, 2)
```

If we don't provide *sufficiency*, GP will try to find the (best) approximation of the solution using available means.

Basic methods of creating the initial population:

- *Full*: randomly pick nodes from $\boldsymbol{F}$ if the depth is below the selected threshold, otherwise from $\boldsymbol{T}$. All trees will have the same depth – examples in Fig. 3.4.

- *Grow*: randomly pick nodes from $\boldsymbol{F} \cup \boldsymbol{T}$ if the depth is below the selected threshold, otherwise from $\boldsymbol{T}$. The trees will have different depth and shape – examples in Fig. 3.5.
- *Ramped half-and-half*: generate half of the population using the *full* method, and another half using the *grow* method – this ensures diversity in the initial population.

```
toolbox.register("expr", gp.genHalfAndHalf, pset=pset, min_=1, max_=2) #
    tree height range
toolbox.register("individual", tools.initIterate, creator.Individual,
    toolbox.expr)
toolbox.register("population", tools.initRepeat, list, toolbox.individual
    )
```

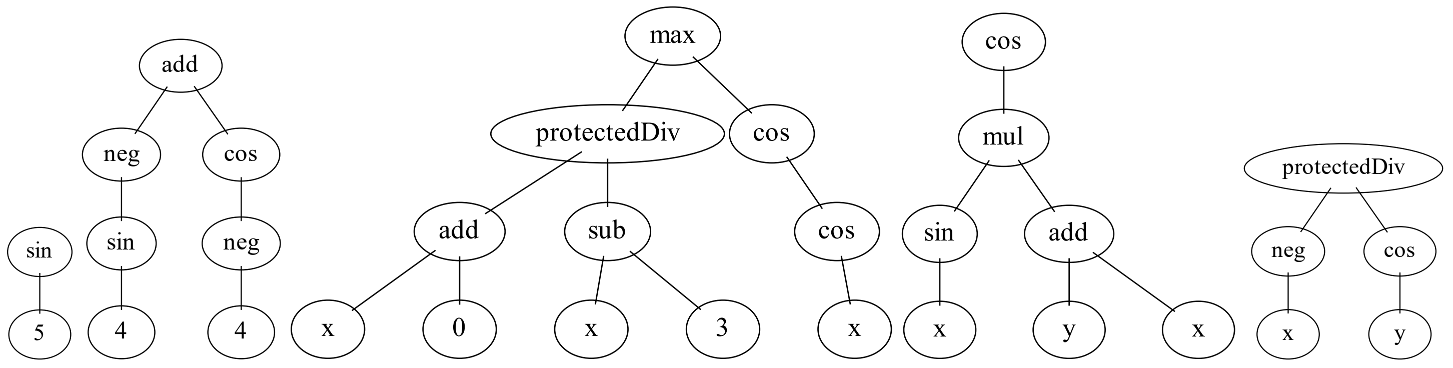


Figure 3.4: Five individuals generated using the *Full* method, `gp.genFull(pset,1,3)` (DEAP requires two parameters, not one) for $\boldsymbol{T}$={x, y, 0, 1, 2, 3, 4, 5}.

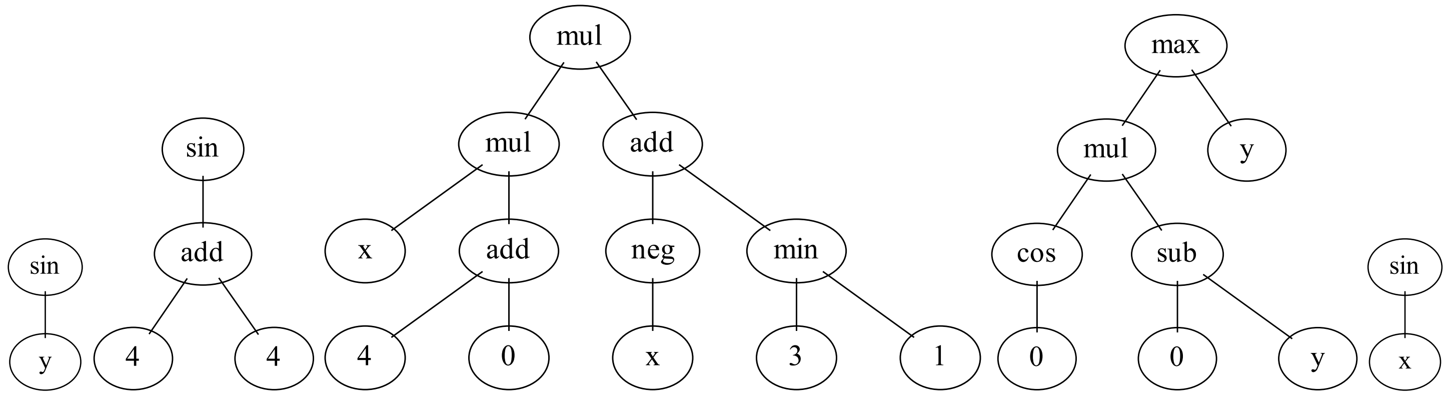


Figure 3.5: Five individuals generated using the *Grow* method, `gp.genGrow(pset,1,3)`, for $\boldsymbol{T}$={x, y, 0, 1, 2, 3, 4, 5}. In the DEAP's `genGrow()` method there is no point in setting the min_depth and max_depth arguments to the same value, because then the generated trees will have all the leaves at the same depth – as if the trees were generated using the `genFull()` method.

Crossing over in GP is usually implemented as swapping randomly selected subtrees of parent trees (Fig. 3.6).

```
toolbox.register("mate", gp.cxOnePoint)
```

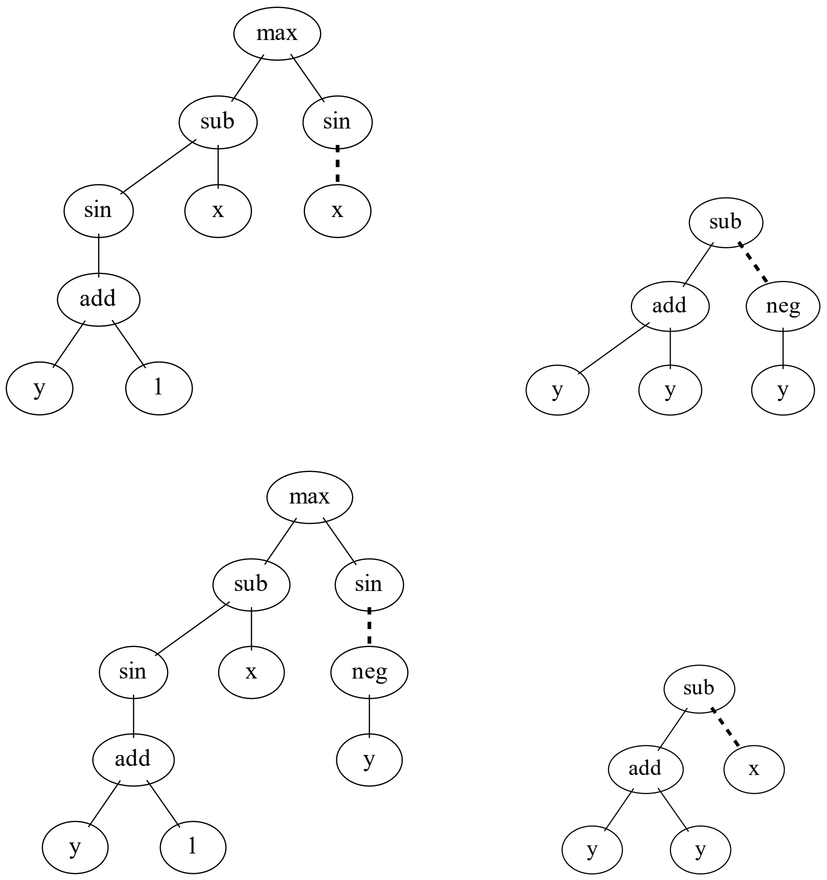


Figure 3.6: Crossing over in GP. Top: parents generated by the `gp.genGrow(pset,2,4)` method. Bottom: offspring generated using the `gp.cxOnePoint(parent1,parent2)` method.

A standard mutation is implemented as selecting a random location in the original tree and replacing the subtree with a newly generated one using one of the methods described above (Fig. 3.7).

```
toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)
toolbox.register("mutate", gp.mutUniform, expr=toolbox.expr_mut, pset=
    pset)
```

To protect against uncontrolled bloating of expressions, penalties for the size of expressions can be included in fitness, or limits of the depth of the tree can be introduced.

```
toolbox.decorate("mate", gp.staticLimit(key=operator.attrgetter("height")
    , max_value=13))
```

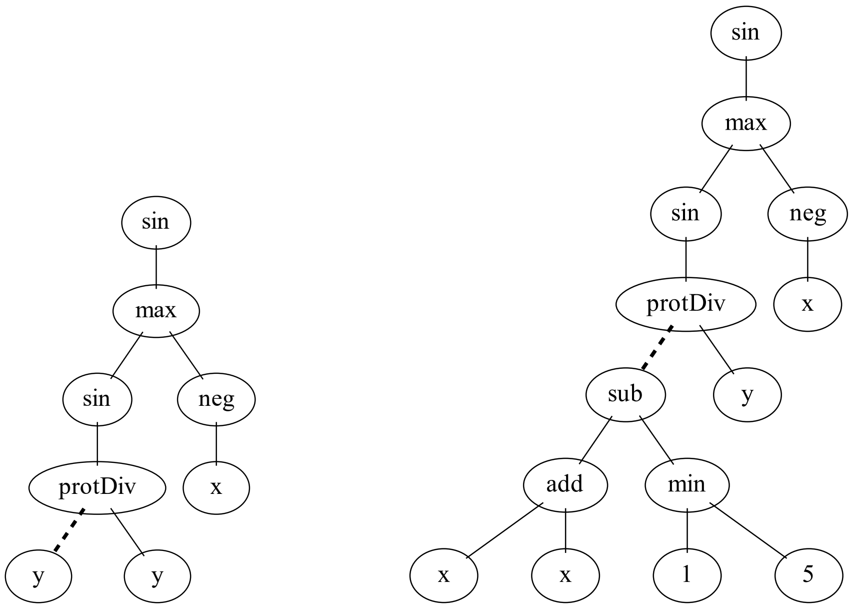


Figure 3.7: Mutation w GP. Left: original solution generated by the `gp.genGrow(pset,2,5)` method. Right: a mutant created using the `gp.mutUniform(parent, toolbox.expr_mut, pset=pset)` method, with earlier `toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)`.

```
toolbox.decorate("mutate", gp.staticLimit(key=operator.attrgetter("height
   "), max_value=11))
```

Since expressions or programs generated by GP are random in their character, it would be difficult to run them directly in the operating system – it is safer to interpret or evaluate them in a virtual environment (e.g. in a virtual machine or “sandbox”). The evaluation of the quality of a solution requires most often its calculation or its application in many situations (different argument values, different robot locations, etc.).

```
# Exception: MemoryError - Error in tree evaluation: Python cannot
   evaluate a tree higher than 90.
```

Discussion: fitness landscape, global convexity and optimization efficiency in GP.

3.5.1 Symbolic regression

Symbolic regression is a typical application of GP where we are looking for a function that describes (fits) as precisely as possible the given points. While in traditional regression methods the form of the function sought is fixed (we only look for coefficients), in GP it is easy to manipulate the form of the function[8] and even look for certain classes of functions or for any functions – hence this regression method is called *symbolic*.

```
def target_function(x):
   return x**2 - x  # in a real application, this is what we look for!

def eval_expr(individual, points):
   # transform the tree expression into a callable function
   func = toolbox.compile(expr=individual)
   # evaluate the mean squared error between the expression and the
  target function
   sqerrors = ((func(x) - target_function(x))**2 for x in points)
   return math.fsum(sqerrors) / len(points),

toolbox.register("evaluate", eval_expr, points=[x/10. for x in range
   (-10,11)])
```

The form of the expression that we look for is controlled by the appropriate selection of elements in the set of functions $\boldsymbol{F}$ and the set of terminal symbols $\boldsymbol{T}$, and by imposing potential restrictions on the tree depth, the number of occurrences of functions from the $\boldsymbol{F}$ set, etc.

Sample experiment #1: Find the expression that best describes the set of points belonging to the function $f(x) = x^2 - x$. Remember that in practice this function is unknown and we want to discover it! Available to GP are functions that can be seen in the example source codes above, i.e., x, also operators neg, $+$, $-$, $*$, $/$, max, min, sin, cos, and additionally, constants $-1, 0, 1$.

[8] http://en.alife.pl/genetic-programming

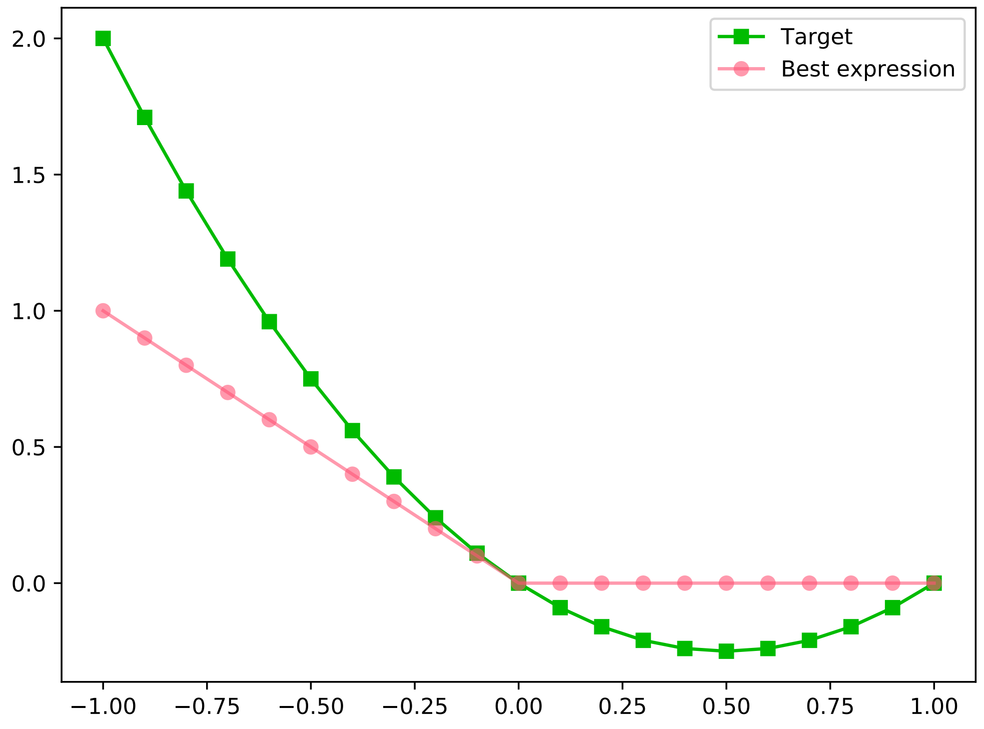


Figure 3.8: The best solution in the first generation (i.e., in a randomly generated population).

```
mul(min(0, x), neg(1))
```

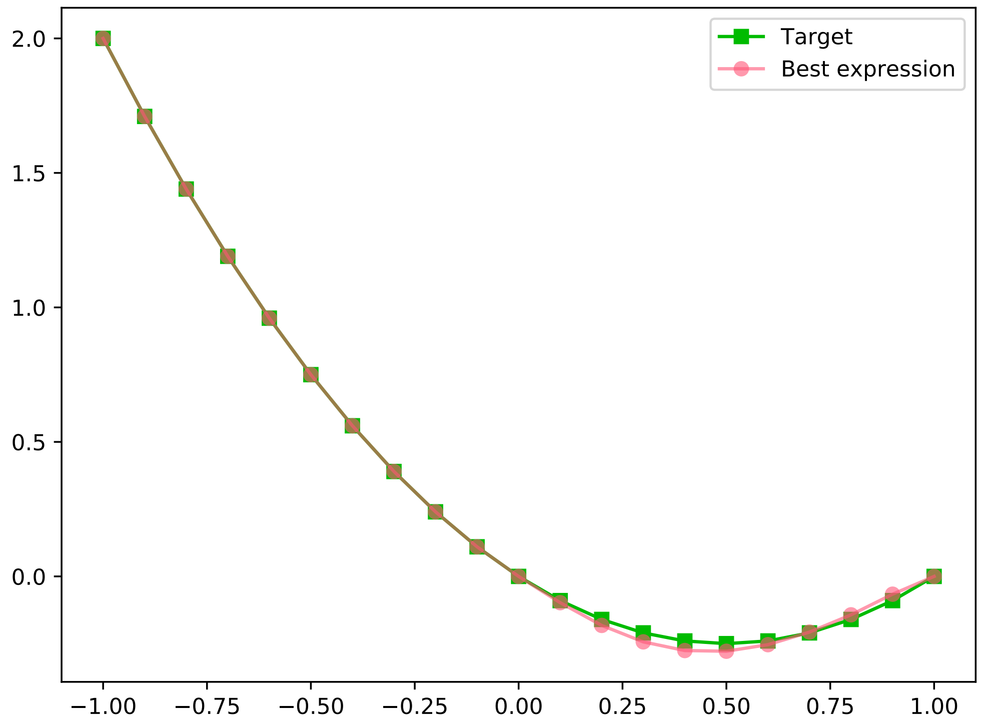


Figure 3.9: The best solution once the evolution finished.

```
sub(x, add(min(min(min(0, x), mul(0, add(0, max(1, 0)))),
add(x, max(x, mul(add(0, x), neg(x))))), max(add(min(min(x, 0),
add(min(sin(x), x), max(sin(x), add(add(0, 0), sin(sin(sin(x))))))),
max(sin(add(min(sin(x), sin(sin(sin(sin(x))))), max(sin(sin(x)), -1))),
x)), x)))
```

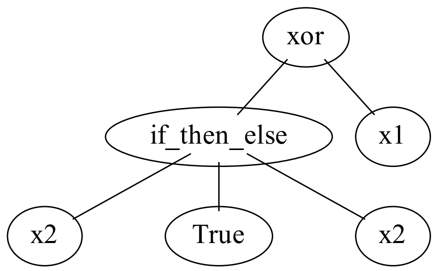
After increasing population size and the number of generations: `mul(add(-1, x), min(x, x))`. Similarly, after limiting the complexity of expressions (intensifies search among simple expressions): `mul(add(-1, x), protectedDiv(x, 1))`.

Sample experiment #2: Find a logic circuit that implements the XOR function, i.e., $\{x_1, x_2, y\} = \{(0,0,0); (0,1,1); (1,0,1); (1,1,0)\}$.

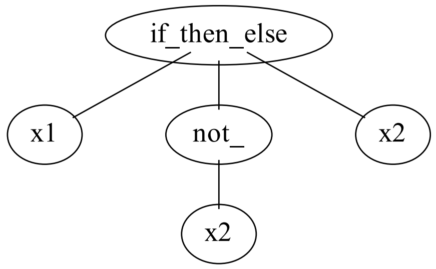
```
def nand(input1, input2):
    return not(input1 and input2)

def if_then_else(input, output1, output2):
    return output1 if input else output2

pset = gp.PrimitiveSetTyped("main", [bool, bool], bool) # let's use
   strongly-typed GP as an example
pset.addPrimitive(operator.xor, [bool, bool], bool)
pset.addPrimitive(operator.or_, [bool, bool], bool)
pset.addPrimitive(operator.and_, [bool, bool], bool)
pset.addPrimitive(operator.not_, [bool], bool)
pset.addPrimitive(nand, [bool, bool], bool) # custom
pset.addPrimitive(if_then_else, [bool, bool, bool], bool) # custom
pset.addTerminal(True, bool)

pset.renameArguments(ARG0="x1")
pset.renameArguments(ARG1="x2")

def eval_expr(individual):
    # transform the tree expression into a callable function
    func = toolbox.compile(expr=individual)
    # evaluate the error between the expression and the target function
    err = 0
    for x1 in (False,True):
        for x2 in (False,True):
            target = x1^x2
            actual = func(x1,x2)
            if target != actual:
               err += 1
    return err,
```

In this experiment, GENERATIONS=100 and POPSIZE=150, and in case of failure – another attempt with POPSIZE=1500.

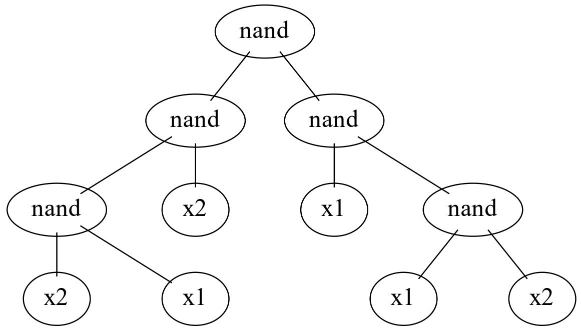
- All operators and the `True` constant as in the source code above:
  `xor(if_then_else(x2, True, x2), x1)`



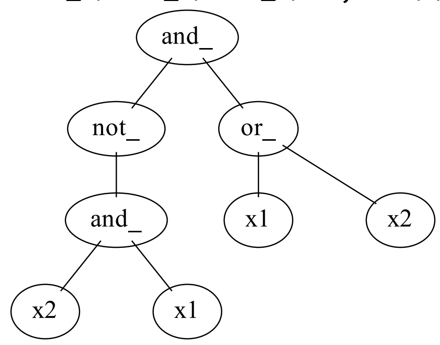
- Only `if-then-else`: no perfect solution found (lowest error = 1)
- Only `if-then-else` and `not`:
  `if_then_else(x1, not_(x2), x2)`



- Only `not` and `and`: no perfect solution found (lowest error = 1)
- Only `nand`:
  `nand(nand(nand(x2, x1), x2), nand(x1, nand(x1, x2)))`



- Trio `and`, `or`, `not`:
  `and_(not_(and_(x2, x1)), or_(x1, x2))`



Discussion: would it be beneficial to simplify expressions during evolution?

Discussion: in which areas does GP have a chance to compete with humans, in which it can surpass them, and in which it has no chance? Why?

Improving effectiveness: semantic GP (semantics = the set of results of an individual for the set of tests) and geometric semantic GP (genetic operators take into account the topology of the semantic space) [Bak+19].

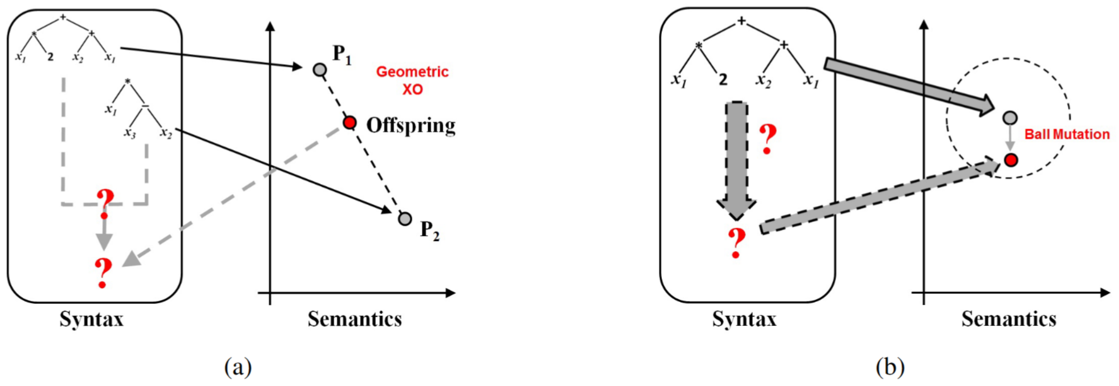
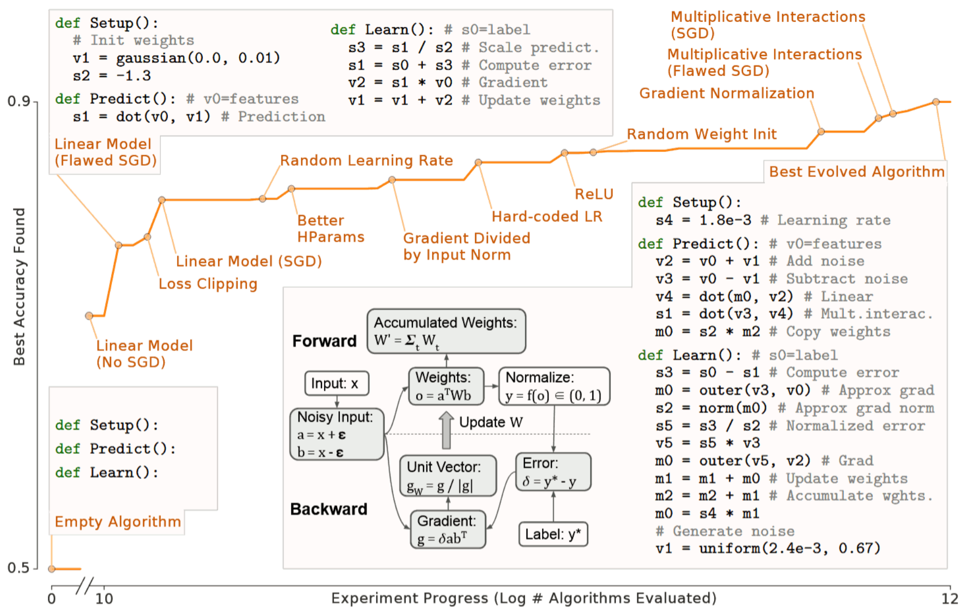


Figure 1: Geometric semantic crossover (plot (a)) (respectively geometric semantic mutation (plot (b))) performs a transformation on the syntax of the individual that corresponds to geometric crossover (respectively geometric mutation) on the semantic space. In this figure, the unrealistic case of a bidimensional semantic space is considered, for simplicity.

Cf. earlier reminder on FDC, DPX, "How to intentionally develop (design) effective crossover operators?", and the embryogeny/mapping.

Sample experiment #3: Find an algorithm that trains a neural network...

Evolutionary architecture: "regularized evolution" (Fig. 2) [Rea+20].
Discoveries of evolution – Fig. 6:



```
def Setup():
def Predict():
def Learn():
```

```
def Setup():
  # Init weights
  v1 = gaussian(0.0, 0.01)
  s2 = -1.3
def Predict(): # v0=features
  s1 = dot(v0, v1) # Prediction
def Learn(): # s0=label
  s3 = s1 / s2 # Scale predict.
  s1 = s0 + s3 # Compute error
  v2 = s1 * v0 # Gradient
  v1 = v1 + v2 # Update weights
```

```
def Setup():
  s4 = 1.8e-3 # Learning rate
def Predict(): # v0=features
  v2 = v0 + v1 # Add noise
  v3 = v0 - v1 # Subtract noise
  v4 = dot(m0, v2) # Linear
  s1 = dot(v3, v4) # Mult.interac.
  m0 = s2 * m2 # Copy weights
def Learn(): # s0=label
  s3 = s0 - s1 # Compute error
  m0 = outer(v3, v0) # Approx grad
  s2 = norm(m0) # Approx grad norm
  s5 = s3 / s2 # Normalized error
  v5 = s5 * v3
  m0 = outer(v5, v2) # Grad
  m1 = m1 + m0 # Update weights
  m2 = m2 + m1 # Accumulate wghts.
  m0 = s4 * m1
  # Generate noise
  v1 = uniform(2.4e-3, 0.67)
```

Figure 6: Progress of one evolution experiment on projected binary CIFAR-10. Callouts indicate some beneficial discoveries. We also print the code for the initial, an intermediate, and the final algorithm. The last is explained in the flow diagram. It outperforms a simple

---

If you have some time and you like SF, read https://www.teamten.com/lawrence/writings/coding-machines/.

3.5.2 Hyper-heuristics and self-programming algorithms

The structure of the evolutionary algorithm (the selection technique, crossing over, mutation, ...) may be controlled by GP (i.e., the structure may be subject to evolutionary improvement) [BT96; OG03; Olt05]. GP can "construct" the optimization algorithm from modules, including atypical architectures: many kinds of mutations, unusual operators that influence just a part of the population, multiple selection processes in one step, etc., depending on the degrees of freedom of GP.

Results are better than those produced by the traditional algorithm, but at a cost...

Compare: the *No Free Lunch* theorem and hyper-heuristics[9] that search through the space of heuristics and their combinations [Ros05; ÖBK08; Bur+10].

[9] http://en.wikipedia.org/wiki/Hyper-heuristic

Sample questions

Review the variants of evolutionary algorithms presented in earlier sections and their abbreviations, and make sure you can distinguish them from one another – recall their discriminating features.

3.6 Classifier systems (CFS/LCS/GBML)

LCS is a simple example of a *cognitive architecture*. A cognitive architecture can mean:

- a theory about the structure of the human mind,
- an implementation of such a theory (used in AI) – a cognitive system or agent.

Possible functional criteria of such architectures include flexible behavior, real-time operation, rationality, large knowledge base, learning, development, adaptation, modularity, linguistic abilities, and self-awareness. Competencies and behaviors demonstrated by such systems include – similarly to AGI – perception, memory, attention, actuation, social interaction, planning, motivation, emotion, development and using knowledge efficiently to perform new tasks. Components include memory storage, control components, data representation, and input/output devices [KT20]. More in Sect. 5.4.4.

John Holland envisioned a cognitive system [HR78] capable of classifying the goings on in its environment, and then reacting to these goings on appropriately[10]. To build such a system[11] (see Fig. 3.10) we need

(1) an environment;

(2) receptors/sensors that tell our system about the goings on;

(3) effectors/actuators that let our system manipulate its environment; and

(4) the system itself that has (2) and (3) attached to it, and "lives" in (1).

CFS is quite a general and versatile architecture – consider the following three examples:

- (4) can be a real or simulated robot or creature: (1) is a world with "food" (something beneficial, reward) and "poison" (something detrimental, penalty), and a robot walking (3) across this environment and trying to learn to distinguish (2) between these two items, and to survive while maximizing reward.

[10] With minor updates and corrections, from https://www.cs.cmu.edu/Groups/AI/html/faqs/ai/genetic/part2/faq-doc-5.html and from no longer available parts of online slides by Riccardo Poli.

[11] https://en.wikipedia.org/wiki/Learning_classifier_system

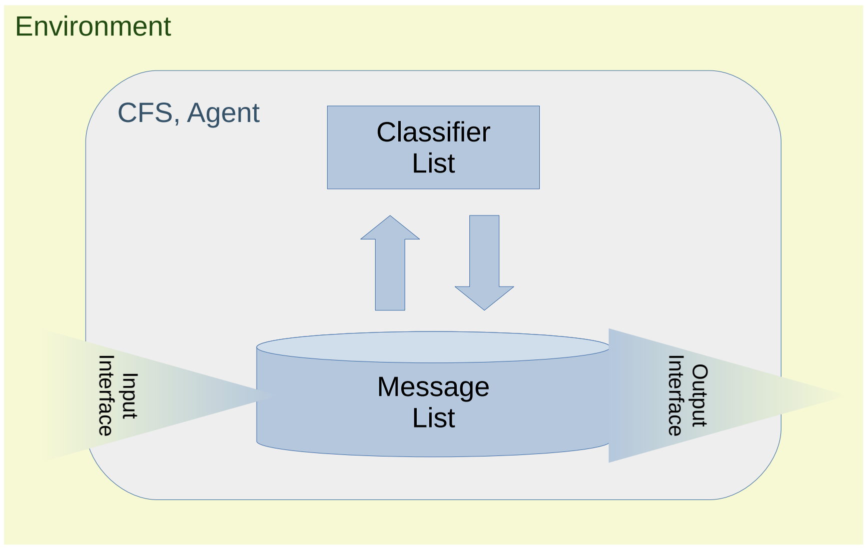


Figure 3.10: The architecture of the classifier system.

- (4) can be a computer. It has inputs (2), outputs (3), and a message passing system in-between, converting certain input messages into output messages, according to a set of rules, usually called a *program*.
- (4) can be a machine learning (ML) algorithm. Inputs (2) provide values of conditional attributes, outputs (3) send out a value of the decision attribute, and a message passing system in-between is the ML model that maps inputs to outputs (predicts outputs based in inputs).

The input interface (2) generates messages – strings of symbols that are written on the message list. Then these messages (internal and external) are matched against the condition-part of all classifiers ("if-then" rules), to find out which actions are to be triggered. The message list is then emptied, and the encoded actions, themselves just messages, are posted to the message list. Then, the output interface (3) checks the message list for messages concerning the effectors. Then the cycle restarts.

You may start from scratch (from tabula rasa – without any knowledge) using a randomly generated classifier population, and let the system learn its program by induction. The input stream are input patterns that must be repeated over and over again, to enable the agent to classify its current situation/context and react on the goings on appropriately, as in the example below:

```
IF small, flying object to the left THEN send @
IF small, flying object to the right THEN send %
IF small, flying object centered THEN send $
```

```
    IF large, looming object THEN send !
    IF no large, looming object THEN send *
    IF * and @ THEN move head 15 degrees left
    IF * and % THEN move head 15 degrees right
    IF * and $ THEN move in direction head pointing
    IF ! THEN move rapidly away from direction head pointing
```

Classifier list is a list of classifiers:

```
    IF cond-1 AND cond-2 ... AND cond-N THEN action

    cond-1, cond-2, ... cond-N / action
```

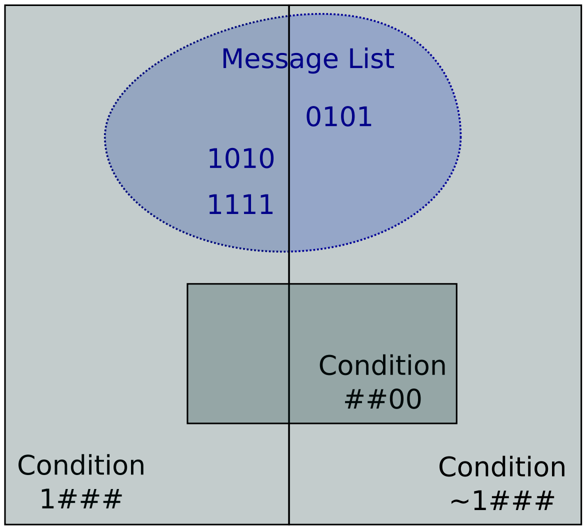
(we use a shorter notation, “,” means “AND”).

For now, let us assume the simplest language for messages and conditions – they will be encoded by {0, 1, #}. In a real application, we would use the natural language (set of symbols) – this is analogous to the GA/EP difference.

A message matches a condition if all its 0's and 1's are in the same positions as in the condition string. A negated condition is satisfied if no message in the message list matches it:

Message List

Label	**Message**
a	0101
b	1010
c	1111



Condition	**Matched by**	**Satisfied**	**Negation satisfied**
0101	a	Yes	No (∼0101)
1101		No	Yes (∼1101)
#101	a	Yes	No (∼#101)
1###	b, c	Yes	No (∼1###)
##00		No	Yes (∼##00)
####	a, b, c	Yes	No (∼####)

In CFSs actions are strings of fixed length built from characters in the alphabet {0, 1, #}; their length is usually the same as that of messages. Action strings can be interpreted as parameterized assertions (messages) that go into the message list. When a classifier is activated, a message is built using the following procedure:

- 0's and 1's in the action string are simply copied in the message
- #'s are substituted by the corresponding characters in the message that matches the first condition in the condition part. For this reason the # character is also called the pass-through operator.

Example:

Message List	
Label	**Message**
a	0101
b	1010
c	1111

Classifier List	
Label	**Classifier**
i	#11#, ~#110 / 00##
ii	###1, ~#110 / ###0
iii	##1#, ~1110 / 0##0

The following set of messages will be produced:

Message	**Reason**
0011	Posted by i, c matches cond-1
0100	Posted by ii, a matches cond-1
1110	Posted by ii, c matches cond-1
0010	Posted by iii, b matches cond-1
0110	Posted by iii, c matches cond-1

The only actions allowed in the basic CFS are assertions, so messages (facts) cannot be explicitly deleted. However, as the message list is of finite size, old messages can be overwritten. Many classifiers can be activated in parallel by the messages in the message list. A classifier can post as many messages as the number of messages matching its first condition. Conflict resolution is only necessary if the active classifiers can produce more messages than entries in the message list.

3.6.1 Input and output interfaces

The input interface can be thought of as a mechanism by which the CFS can obtain information about the environment. The messages posted by the input interface are often descriptions of the state of a set of (binary) detectors that can sense various features of the environment.

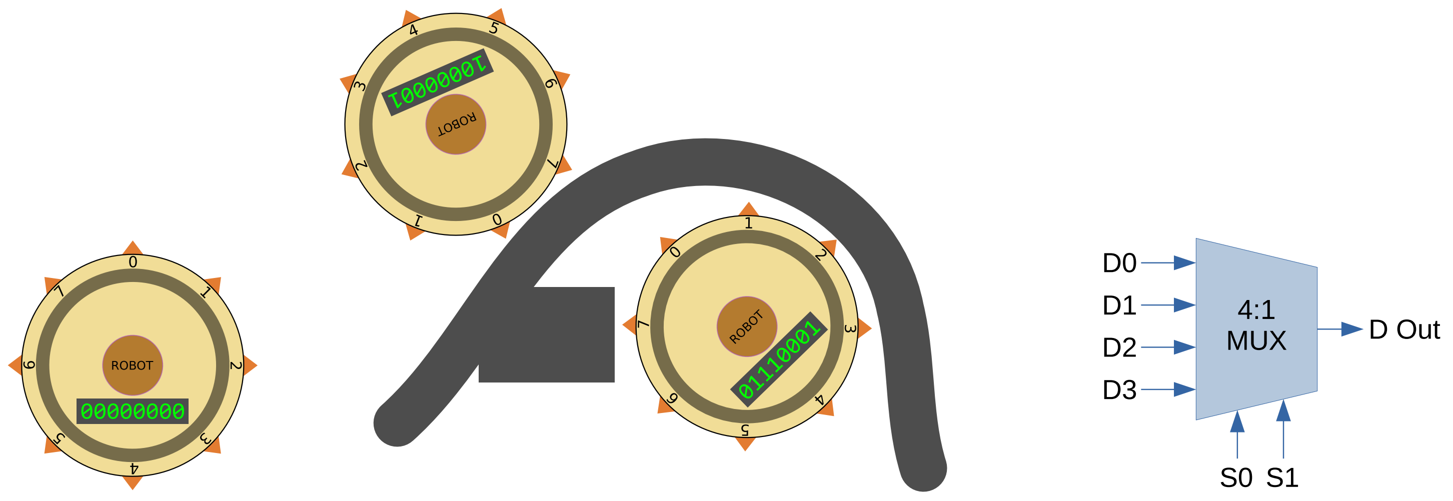


Figure 3.11: Simple examples of CFS control: a robot with eight proximity sensors and a 4:1 multiplexer.

The output interface can be realized by any procedure capable of selecting (deleting) and using some messages in the message list. Usually we imagine that the bits in a message picked up by the output interface represent (and control) the state of a set of effectors which act on the environment, for example to control the actions of a robot.

In any case the output interface must be able to recognize which messages are input messages posted by the input interface, which are internal messages posted by classifiers, and messages meant to be output messages. This is obtained by using tags usually consisting of two additional bits in the messages that are interpreted in a special way. An example convention:

Tag	**Interpretation**
01	Internal Message
11	Internal Message
00	Output Message
10	Input Message

Or:

bit 1: Input/Output (from/to)
bit 2: Inside/Outside

For example, a CFS which controls a robot (Fig. 3.11, left) might have some classifiers devoted to obstacle avoidance, like

```
10 1####### / 00 0100 0000
10 ##1##### / 00 0001 0000
```

```
10 ####1### / 00 1000 0000
10 ######1# / 00 0010 0000
```

The conditions of these simple classifiers just check whether there is an obstacle in front, on the right, on the back, on the left, respectively. The action-strings have the following interpretation:

- The first two bits indicate that the messages are for the effectors in the output interface
- The following four bits indicate which movement (forward, backward, right, left) is appropriate to avoid the obstacle
- The other bits are padding.

In the case of the 6-bit 4:1 multiplexer (Fig. 3.11, right), the following classifier list would provide the correct behavior:

```
10 00 0### / 00 0 00000
10 00 1### / 00 1 00000
10 01 #0## / 00 0 00000
10 01 #1## / 00 1 00000
10 10 ##0# / 00 0 00000
10 10 ##1# / 00 1 00000
10 11 ###0 / 00 0 00000
10 11 ###1 / 00 1 00000
```

- The first two bits of the conditions mean “input message”
- The next two bits of the conditions are interpreted as address (i.e., selector) bits
- The other characters of the conditions as checks for the state of the data lines
- The first two bits of the action are interpreted as the “output message” tags
- The third bit as the output of the multiplexer
- The remaining bits as padding.

3.6.2 Main cycle

1. Activate the input interface and post the input messages it generates to the message list.
2. Perform the matching of all the conditions of all classifiers against the message list.

3. Activate the fireable classifiers (those whose conditions are satisfied) and add the messages they generate to the message list.
4. Activate the output interface, i.e. remove the output messages from the message list and perform the actions they describe; go to 1.

In the previous two examples we have considered non-overlapping rulesets, i.e. sets of classifiers in which one and only one classifier is active in the presence of a given message in the message list. An alternative, more parsimonious way of using classifiers is to organize them to form a default hierarchy, in which some very general classifiers provide the default behavior for the system. Other, more specific classifiers refine such a behavior in the presence of messages improperly handled by the default ones.

For example, for the 6-bit multiplexer we could use:

```
10 00 0### / 00 0 00000
10 01 #0## / 00 0 00000
10 10 ##0# / 00 0 00000
10 11 ###0 / 00 0 00000
10 ## #### / 00 1 00000
```

“Unless there is a 0 on the data line currently addressed, set the output to 1”

Default hierarchies are not only more parsimonious ways of programming CFSs, they also make the “search” for a good program much easier (for example we can successively refine the hierarchy). This is very important when we introduce learning and rule discovery in CFSs.

3.6.3 *Learning* Classifier Systems (LCS)

The real power of CFSs derives from the possibility of adding adaptation mechanisms to the basic architecture, so that they can learn to behave appropriately in the environment[12] or, more simply, to perform useful tasks. There are two ways in which we can adapt (i.e. improve the performance of) a CFS:

- Adaptation by **credit assignment**: changing the way existing classifiers are used.
- Adaptation by **rule discovery**: introducing new classifiers in the system.

[12] http://en.alife.pl/learning-classifier-system

3.6.4 Good and bad classifiers

Not all the classifiers active at a given cycle will in general produce a message which will lead (directly or after additional processing) to a good action. For example, if a CFS controls an autonomous agent who has to find a source of energy (i.e. food) to survive, some classifiers will post actions which will lead to get some food; others will post actions which will delay the search for food. In summary, in a CFS there are usually some classifiers which are better than the others.

3.6.5 The need for competition

Unfortunately, the basic CFS is absolutely, blindly fair and gives to all the fireable classifiers the same chances of posting messages, and therefore of influencing the overall behavior of the system. To maximize the performance of a CFS it would be nice to give higher priority to the messages posted by good classifiers and low priority to the others. Even better – to prevent low quality classifiers from posting their messages at all if other, better classifiers are fireable. This behavior could be obtained if the classifiers had to compete to post their messages, basing on some measure of quality of classifiers.

3.6.6 Quality of classifiers

There may be several properties of classifiers on which a quality measure can be based. The two most important ones are:

- The usefulness of the classifier in determining the good performance of the whole system: *strength*.
- The relevance of a classifier in a particular situation: the *specificity* of the classifier = its (length – number_of_#'s) / length.

Strength and *specificity* are usually combined into a single measure, the *bid* a classifier makes in the auction (competition):

$$bid = k \cdot strength \cdot specificity \qquad (k \text{ is a constant} \approx 0.1)$$

- To maintain parallelism, in the auction there must be more than one winner.
- To avoid premature convergence, we use a noisy (probabilistic) auction, in which classifiers have a bid-proportionate winning probability.

- As a classifier's specificity is a constant, the strengths associated to classifiers are the only quantities that can be varied to influence the auction and therefore the behavior of a CFS.

3.6.7 Adaptation by credit assignment

A learning algorithm for CFS should be capable of modifying the *strengths* of classifiers to optimize the behavior of the system as a whole. To do that, the algorithm will need to have some kind of information about the quality of the behavior of the system. This information will come from the environment, e.g. from an external observer (a teacher), or from some other part of the system, e.g. from an internal variable representing the level of energy of the system. The simplest (more biologically plausible) form of behavioral-quality information would be a scalar value, termed *reward*, whose sign tells the learning algorithm whether the actions of the system are good (positive reward) or bad (negative reward or punishment) and whose magnitude may be fixed (e.g. $+1$, -1, 0) or variable. If rewards only are available, the learning algorithm will have to solve the so-called credit assignment problem: which classifiers are responsible (and to which extent) for the good or bad overall behavior of the system?

3.6.8 The Bucket Brigade algorithm

The bucket brigade (pol. *brygada kubełkowa/wiaderkowa*) algorithm is a parallel, domain-independent, local credit-assignment-based learning algorithm:

1. If there is a reward (or punishment), add it to the strength of all the classifiers active in the current major cycle.
2. Make each active classifier pay its bid to the classifiers that prepared the stage for it (i.e. posted messages matched by its conditions).

The stage-preparing classifiers had to pay (invest) their bids in the previous cycle (when they were active). In this cycle they get back their "money". In turn, the classifiers that prepared the stage for the stage-preparing classifier received some money two cycles earlier, and so on. Good classifiers are rewarded often so their strengths tend to grow. So, they will make bigger bids, and so they will pay more to their stage-preparing classifiers. In turn those classifiers will be able to pay more to their stage-preparing classifiers, and so on. During time, strength is propagated backwards, and each classifier receives the correct share of credit for the good (or bad) behavior of the system as a whole. With time, strengths reach a (nearly) constant equilibrium value.

Components of the classifier evaluation

The strength of the classifier in the next step $t+1$ can be expressed as a function of its current strength $S(t)$, its bid B (payments to other rules), taxes T, and reward R (income):

$$S(t+1) = S(t) + R(t) - T(t) - B(t)$$

The amount of R is one of the parameters of the system, and the initial value of $S(0)$ is a property of each rule initially included in the classifier set. The bid depends on strength:

$$B = C_{bid} \cdot S$$

C_{bid} is a system parameter and determines the fraction of strength that becomes the bid (e.g. 10%). Taxes are required to remove unproductive classifiers. Such classifiers are never activated because their conditions are not matched, but given some initial strength they would exist in the system forever. The fact that they never match any messages may mean that they are redundant. There are two types of taxes – turnover and fixed (paying just for existing). The first type is only paid by rules that were activated. The second type is paid by all rules. If a rule pays the tax all the time and it cannot increase its strength (by receiving a part of the offer of the winning classifier, or the reward from the environment), then its strength will decrease to 0. Turnover and fixed taxes are a fraction of the rule strength, S. The specific fraction size is the system parameter.

3.6.9 Adaptation by rule discovery

In addition to credit assignment, in order to learn we need a way to introduce new classifiers to the system. Evolutionary algorithm can be used to optimize and adapt a CFS in two ways:

- Considering the classifier list as a single individual whose chromosome is obtained by concatenating the conditions and actions of all classifiers (the "Pittsburgh" ("Pitt") approach, De Jong)
- Considering each classifier as a separate individual (the "Michigan" approach, Holland).

In the Pittsburgh approach, the fitness of each CFS is determined by observing the behavior of the system for a certain amount of time or on some test data. The EA optimizes the CFS by breeding and making *compete* different sets of classifiers. The Michigan approach requires a fitness measure for each classifier. If used with the bucket brigade algorithm, the strength of the classifier can be taken as its fitness. In this case, the EA optimizes the CFS by breeding and making *compete* and *co-operate* different classifiers.

3.6.10 Summary

The description above concerned the classic GBML idea (analogous in its simplicity to a GA). In practice, the components of the system and its entire architecture are adapted to the problem at hand. Modifications and improvements of LCS systems concern language (using a large number of symbols increases the expressive power of a grammar), representation (the complexity of rules), genetic operators, reward mechanisms (e.g., integration with reinforcement learning algorithms), etc. Applications are very diverse due to the universality of the idea itself – for example, the exploration of an unknown environment by a robot, control systems, difficult games (e.g. poker), the construction of semantic networks, learning the rules of conduct in medical diagnosis and treatment, and many others.

In general, for complex problems, LCS/GBML systems give better results than traditional classification systems and traditional machine learning algorithms. In their operation, we see analogies to deep and recursive neural networks, but contrary to NNs, LCSs offer a symbolic form of knowledge which is in some applications easier to interpret.

Chapter 4

Other nature-inspired optimization techniques

4.1 Ant systems, ant colony optimization (AS, ACO) and swarm intelligence

The behavior of social insects in general, and of ant colonies in particular, has since long time fascinated researchers in ethology and animal behavior, who have proposed many models to explain their capabilities. Ant algorithms have been proposed as a novel computational model that replaces the traditional emphasis on control, preprogramming, and centralization with designs featuring autonomy, emergence, and distributed functioning. These designs are proving flexible and robust, able to adapt quickly to changing environments and to continue functioning even when individual elements fail.

Ant algorithms are a part of Swarm Intelligence (pol. *inteligencja grupowa/zbiorowa/roju*). A particularly successful research direction in ant algorithms is known as "ant colony optimization"[1] (ACO). ACO has been applied successfully to a large number of difficult combinatorial problems like the quadratic assignment (QAP) and the traveling salesman (TSP) problems, to routing in telecommunications networks, to scheduling problems. In ACO, the discrete optimization problem is mapped onto a graph called *construction graph* in such a way that feasible solutions to the original problem correspond to paths in the construction graph.

An "Ant System" (AS) – a particular ant colony optimization algorithm – was introduced

[1]https://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms

in 1992, and was inspired by the observation of the behavior of ant colonies: emergence[2] of global properties following the mutual interaction among many elementary agents performing simple actions. The algorithm is easy to parallelize. How do ants find the shortest route between two points, being almost blind and having very simple individual capacities? By pheromone deposition[3]: each ant moves at random unless they feel the pheromone – then they follow the marked path and leave new pheromone. The collective effect is a positive feedback. Two aspects are present: exploration and exploitation. Pheromone is simulated by a global knowledge structure, which is updated by "ants". This knowledge is used in the construction of solutions during the iterative optimization process.

Swarm intelligence $\supset$ ant algorithms $\supset$ ACO $\supset$ $\{$AS, Ant-Q, Max-Min-AS, Ant Colony System, $\ldots\}$

Applying AS to solve TSP: at any given time, each city has a certain number of ants, which choose another unvisited city with a probability that is a function of **distance** to that city (the smaller, the higher the probability) and the **amount of pheromone** between the cities (the higher, the higher the probability). There are several mechanisms for modifying the amount of pheromone [DCG99]: *ant-density* – while building a solution, a constant amount, *ant-quantity* – while building a solution, a variable amount depending on the quality of the added fragments, and the most effective: *ant-cycle* – variable amount, after completing the entire solution.

An example of applying AS to solve a problem that does not resemble TSP – the attribute selection problem [SB06]: for example, cities are attributes and paths are subsets of attributes; in this approach, we do not have to visit all the cities, and their order does not matter. So in general, instead of speaking about the "distance to some city", we speak about the attractiveness of adding some part of the solution.

Some parameters in ACO:

- m: number of ants (agents) – population size,
- pheromone persistence $\rho < 1$ (evaporation is $1 - \rho$),
- α: pheromone importance,
- $\ldots$

[2]https://en.wikipedia.org/wiki/Emergence

[3]This is a form of **stigmergy**: https://en.wikipedia.org/wiki/Stigmergy

4.2 Particle swarm optimization (PSO)

Introduced in 1995, it is quite similar to EAs. A group ("population" or "swarm") of solutions ("particles") moves ("flies") in the space of solutions seeking better and better areas.[4] There is no selection or crossing over. Each particle remembers the best solution it has discovered so far, and knows the best solution found by its neighbors (or the entire swarm). The movement of each particle in each step depends on its speed, whose vector is changed in the direction of (randomly weighted) best remembered solutions[5]. PSO usually converges faster than EAs.

4.3 Other swarm-intelligent optimization algorithms

Other algorithms inspired by nature are proposed (and some animals are still unused):

- Artificial immune systems – AIS, 1994
- Artificial bee colony – ABC, 2005
- Glowworm swarm optimization – GSO, 2005
- Firefly algorithm – FA, 2008
- Cuckoo search – CS, 2009
- Gravitational search algorithms – GSA, 2009; charged system search – CSS, 2010
- Krill herd algorithm – KH, 2012
- and more: https://en.wikipedia.org/wiki/List_of_metaphor-based_metaheuristic

...but these algorithms are all about the exploration vs. exploitation tradeoff.

Sample questions

- What is emergence? Give examples of natural occurrences (physics, biology) and artificial implementations (engineering).
- What is stigmergy? Give examples of natural occurrences (biology) and artificial implementations (engineering).

[4] https://en.wikipedia.org/wiki/Particle_swarm_optimization

[5] http://en.alife.pl/particle-swarm-optimization

Chapter 5

Remaining aspects of artificial life

5.1 Modeling plants using *L-systems*

An L-system (a Lindenmayer system) is a type of a formal grammar, where all possible rules are applied in each step of development. L-systems can be deterministic or stochastic, context-insensitive or sensitive, and parametric or not.

- Basic information: `https://en.wikipedia.org/wiki/L-system`
- Comprehensive book [PL96] – first published in 1990, Lindenmayer[1] & Prusinkiewicz[2]
- Basic 2D demo: `http://en.alife.pl/lsyst/e/index.html`
- Basic 3D demo: `http://en.alife.pl/lsyst/e/ls_3d.html`
- More advanced: modeling development [JPM00] and climbing [Knu09]; can also be used to model differences in development in response to various environmental conditions: temperature, humidity, sunlight, fertilization…
- Used not just for modeling plants [Bou+12], but also for robot morphologies, architectural design (buildings, cities) [PM01], games [EE17], generating music [WS05], and in other applications, e.g. [Bie+18]. Can be evolved – available as 'fL' genetic encoding in Framsticks[3].

5.2 Emergence in *Boids*

Boids are a simple example of emergent phenomena. From [Sip95]:

> Another process predominating ALife systems is that of emergence (pol. *emergencja*), where phenomena at a certain level arise from interactions at lower levels. In physical systems, temperature and pressure are examples of emergent phenomena. They occur in large ensembles of molecules and are due to interactions at the molecular level. An individual molecule possesses neither temperature nor pressure, which are higher-level, emergent phenomena.
>
> ALife systems consist of a large collection of simple, basic units whose interesting properties are those that emerge at higher levels (with no central controller). One example is von Neumann's model, where the basic units are grid cells (a CA, cellular automaton, Sect. 5.3) and the observed phenomena involve composite objects consisting of several cells (for example, the universal

[1] `https://en.wikipedia.org/wiki/Aristid_Lindenmayer`

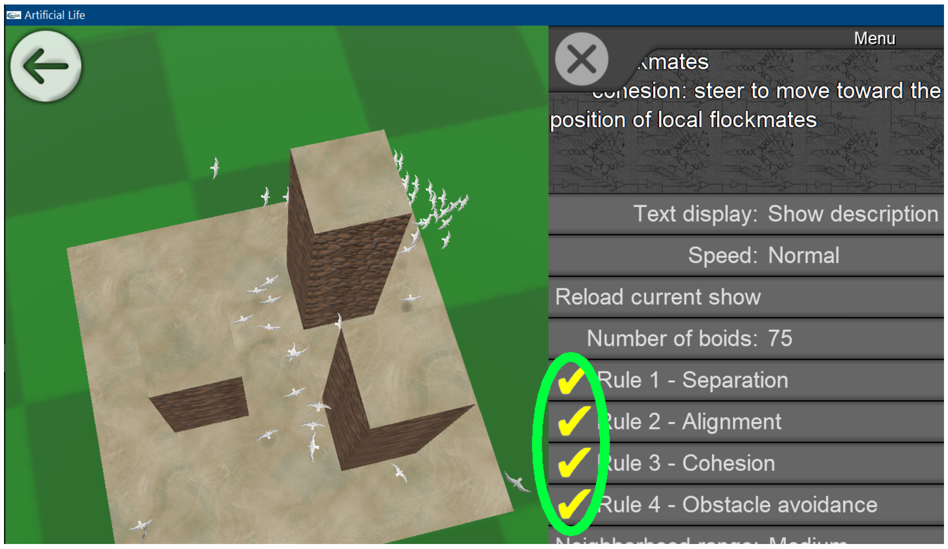
[2] `https://en.wikipedia.org/wiki/Przemys%C5%82aw_Prusinkiewicz`

[3] `http://www.framsticks.com/a/al_genotype`

constructing machine). Another example is Craig Reynolds' work on flocking behavior.

Reynolds wished to investigate how flocks of birds fly, without central direction (that is, a leader). He created a virtual bird with basic flight capability, called a "boid". The computerized world was populated with a collection of boids, flying in accordance with the following three rules:

- Collision Avoidance: Avoid collisions with nearby flock-mates.
- Velocity Matching: Attempt to match velocity with nearby flock-mates.
- Flock Centering: Attempt to stay close to nearby flock-mates.



mates
esion: steer to move toward the position of local flockmates

Text display: Show description

Speed: Normal

Reload current show

Number of boids: 75

✓ Rule 1 - Separation

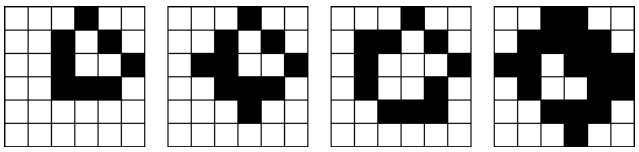
✓ ule 2 - Alignment

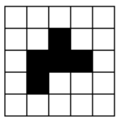
✓ ule 3 - Cohesion

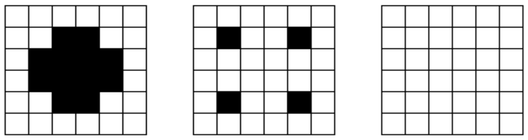
✓ Rule 4 - Obstacle avoidance

Each boid comprises a basic unit that "sees" only its nearby flock-mates. The three rules served as sufficient basis for the emergence of flocking behavior. The boids flew as a cohesive group, and when obstacles appeared in their way they spontaneously split into two subgroups, without any central guidance, rejoining again after clearing the obstruction (as observed in nature). The boids algorithm has been used to produce photorealistic imagery of bat swarms for the classical motion pictures *Batman Returns* and *Cliffhanger*.

5.3 Spatio-temporal dynamics in *Cellular Automata*







From [Sip95]:

> A *machine* in the cellular automata model is a collection of cells that can be regarded as operating in unison. For example, if a square configuration of four black cells exists, that appears at each time step one cell to the right, then we say that the square acts as a machine moving right.
>
> Von Neumann used this simple model to describe a universal constructing machine, which can read assembly instructions of any given machine, and construct that machine accordingly. These instructions are a collection of cells of various colors, as is the new machine after being assembled – indeed, any compound element on the grid is simply a collection of cells.
>
> Von Neumann's universal constructor can build any machine when given the appropriate assembly instructions. If these consist of instructions for building a universal constructor, then the machine can create a duplicate of itself; that is, reproduce. Should we want the offspring to reproduce as well, we must *copy* the assembly instructions and attach them to it. Von Neumann showed that a reproductive process is possible in artificial machines. (...)
>
> One of von Neumann's main conclusions was that the reproductive process uses the assembly instructions in two manners: as interpreted code (during assembly), and as uninterpreted data (copying of assembly instructions to offspring). During the following decade, it became clear that nature had "adopted" von Neumann's conclusions. The process by which assembly instructions (that is, DNA) are used to create a working machine (that is, proteins), indeed makes dual use of information: as interpreted code (translation) and as uninterpreted data (transcription).

Based on this video: `https://www.youtube.com/watch?v=xP5-iIeKXE8`, what can you tell about the CA and the "game of life" rules?

Complexity of self-replicating systems estimated in [bits] despite different environments [Mer97]:

- 800, C-program: more...
  ```
  main(){char q=34,n=10,*a="main()
  {char q=34,n=10,*a=%c%s%c;printf(a,q,a,q,n);}%c"; printf(a,q,a,q,n);}
  ```
- 500 000, Von Neumann's universal constructor about...

- 500 000, Internet worm (1988) about...
- 8 000 000, *Mycoplasma capricolum* about...
- 100 000 000, Drexler's assembler about...
- 6 400 000 000, Human
- 100 000 000 000, NASA Lunar Manufacturing Facility

Related topics (optional):

- `https://en.wikipedia.org/wiki/Garden_of_Eden_(cellular_automaton)`
- `https://en.wikipedia.org/wiki/Boolean_network`
- `https://en.wikipedia.org/wiki/Gene_regulatory_network#Boolean_network`

5.4 Agent and environment

5.4.1 Complex Adaptive Systems (CAS), Multi-Agent Systems (MAS)

CAS: `https://en.wikipedia.org/wiki/Complex_adaptive_system`

MAS: `https://en.wikipedia.org/wiki/Multi-agent_system`

Presentations or experiments during laboratory classes: *StarLogo*[4], *NetLogo*[5], *Repast*[6].

5.4.2 Robotics: hierarchical control with layers

When designing robots operating in a (noisy) environment, e.g. walking around the building and collecting garbage, the hierarchical structure of their brain (control system) is used. It consists of "layers". Each layer performs more complex functions than the lower layer. For example: the first layer is responsible for avoiding obstacles. The second one – for movement (random motion in the environment – a room, a building, etc.), and no longer deals with avoiding obstacles. Higher layers can overrule the function of the lower layers by overriding their operation, although the lower layers still work when we add the higher ones. Such an architecture roughly resembles the structure of the human brain, in which primitive layers

[4] `https://education.mit.edu/project/starlogo-tng/`
[5] `https://ccl.northwestern.edu/netlogo/`
[6] `https://repast.github.io/`

correspond to basic functions (e.g. breathing), and higher layers – to more complex functions (e.g. abstract thinking). The hierarchical approach allows for incremental creation of robots and their incremental optimization (by successively adding layers).

Each layer consists of behavior modules that communicate asynchronously without a central controller.[7] For example, the collision detection layer has sensor modules, danger detection modules, and an engine system – and these modules communicate with each other to agree on a decision that affects the behavior.

This method demonstrates the *bottom-up* approach typical of artificial life and situated, embodied AI: starting with simple, elementary modules, gradually building up using evolution, emergence, and development[8]. Traditional AI employs the *top-down* methodology: a complex behavior (e.g., playing chess) is analyzed and divided to build a system that will ultimately reflect the details of this behavior.

5.4.3 Levels of autonomy

In many applications of agents (Internet, simulations, robotics), the optimal level of their autonomy is different [MKT99] – depending on the application and requirements: speed of operation, low cost, realism, safety, etc. Autonomy is influenced by independent behavior, the range of allowed actions, perception, memory, reasoning, self-control, etc.

Level of autonomy	**Agent goes to a specific location**	**Agent applies a specific action**
Guided	Agent needs to receive a list of collision-free positions (external control)	Agent needs to receive information about the action to be applied (external control)
Programmed	Agent is programmed to follow a path while avoiding collisions	Agent is programmed to apply the action in appropriate circumstances
Autonomous$_1$	Agent chooses a path to follow to reach the goal	Agent decides how to apply the action
Autonomous$_2$	Agent decides if...	Agent decides if...
Autonomous$_3$	Agent decides what to do	Agent decides what to do

It is agreed that we cannot yet construct robots with a sufficiently high level of autonomy for certain applications, and progress is slow here. But do we really want it? "Just because you can doesn't mean you should".

[7] Cf. Elira's control architecture from a short story [Kom19].

[8] https://en.wikipedia.org/wiki/Artificial_intelligence,_situated_approach

5.4.4 Cognitive architectures and artificial general intelligence

So far we encountered two very simple examples of cognitive architectures: a recurrent neural network in an agent (Fig. 1.3) when we discussed situatedness and embodiment, and a set of rules – LCS (Sect. 3.6).

Now let's broaden our view by briefly reviewing the properties of more elaborate cognitive architectures [KT20]. [youtube lecture video: pay particular attention to and learn about the concepts marked in yellow]

5.5 Models of biological life – selected examples

During this course, we focused only on one of the two main goals of artificial life (as mentioned in Sect. 1.1) – on "enhancing our insight into applicable artificial models in order to improve their performance", as this goal is more important for computer science. We did not talk much about the other goal – "increasing our understanding of nature by studying existing biological phenomena", so this section is left out.

Bibliography

[Ada98] Christoph Adami. *Introduction to Artificial Life*. Springer, 1998. ISBN: 9780387946467. URL: https://books.google.pl/books?id=2wouAc-WOnYC.

[AK09] Andrew Adamatzky and Maciej Komosinski, eds. *Artificial Life Models in Hardware*. London: Springer, 2009, p. 270. ISBN: 978-1-84882-529-1. DOI: 10.1007/978-1-84882-530-7. URL: http://www.springer.com/978-1-84882-529-1.

[Bak+19] Illya Bakurov et al. "A regression-like classification system for geometric semantic genetic programming". In: *Proceedings of the 11th International Joint Conference on Computational Intelligence (IJCCI)*. Vol. 1. 2019, pp. 40–48. URL: https://run.unl.pt/bitstream/10362/87064/1/Regression_like_Classification_System_Geometric_Semantic_Genetic.pdf.

[Bed96] Mark A. Bedau. "The nature of life". In: *The philosophy of artificial life* (1996), pp. 332–357.

[Ben99] Peter Bentley. *Evolutionary design by computers*. Morgan Kaufmann, 1999.

[Bie+18] Dongyang Bie et al. "Parametric L-systems-based modeling self-reconfiguration of modular robots in obstacle environments". In: *International Journal of Advanced Robotic Systems* 15.1 (2018). URL: https://journals.sagepub.com/doi/full/10.1177/1729881418754477.

[Bou+12] Frédéric Boudon et al. "L-Py: an L-system simulation framework for modeling plant architecture development based on a dynamic language". In: *Frontiers in plant science* 3 (2012). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC3362793/.

[BT96] Andreas Bölte and Ulrich Wilhelm Thonemann. "Optimizing simulated annealing schedules with genetic programming". In: *European Journal of Operational Research* 92.2 (1996), pp. 402–416. ISSN: 0377-2217. DOI: 10.1016/0377-2217(94)00350-5.

[Bur+10] E. K. Burker et al. "A classification of hyper-heuristic approaches". In: *Handbook of Metaheuristics* (2010), pp. 449–468.

[DCG99] Marco Dorigo, Gianni Di Caro, and Luca M. Gambardella. "Ant algorithms for discrete optimization". In: *Artificial life* 5.2 (1999), pp. 137–172. URL: https://web2.qatar.cmu.edu/~gdicaro/Papers/ArtificialLife-original.pdf.

[DJ75] Kenneth Alan De Jong. "Analysis of the behavior of a class of genetic adaptive systems". PhD thesis. University of Michigan, 1975. URL: https://deepblue.lib.umich.edu/bitstream/handle/2027.42/4507/bab6360.0001.001.pdf.

[Dom99] Paul Domjan. "Are Romance Novels Really Alive? A Discussion of the Supple Adaptation View of Life". In: *Advances in Artificial Life*. Ed. by Dario Floreano, Jean-Daniel Nicoud, and Francesco Mondada. Springer, 1999, pp. 21–25. DOI: 10.1007/3-540-48304-7_6.

[EE17] Gustavo Santarsiere Etchebehere and Maria Amelia Eliseo. “L-Systems and Procedural Generation of Virtual Game Maze Sceneries”. In: *Proceedings of Brazilian Symposium on Computer Games and Digital Entertainment*. 2017, pp. 602–605. URL: https://www.sbgames.org/sbgames2017/papers/ComputacaoShort/174978.pdf.

[FB90] J. Doyne Farmer and Alletta Belin. *Artificial life: The coming evolution*. Tech. rep. SFI working paper 1990–003. Santa Fe Institute, 1990. URL: https://www.santafe.edu/research/results/working-papers/artificial-life-the-coming-evolution.

[HR78] John H. Holland and Judith S. Reitman. “Cognitive systems based on adaptive algorithms”. In: *Pattern-directed inference systems*. Elsevier, 1978, pp. 313–329.

[JMK20] Anders Jerkstrand, Keiichi Maeda, and Koji S. Kawabata. “A type Ia supernova at the heart of superluminous transient SN 2006gy”. In: *Science* 367.6476 (2020), pp. 415–418. DOI: 10.1126/science.aaw1469.

[JPM00] Catherine Jirasek, Przemyslaw Prusinkiewicz, and Bruno Moulia. “Integrating biomechanics into developmental plant models expressed using L-systems”. In: *Plant biomechanics* (2000), pp. 615–624. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.364.585&rep=rep1&type=pdf.

[KA09] Maciej Komosinski and Andrew Adamatzky, eds. *Artificial Life Models in Software*. 2nd. London: Springer, 2009, p. 442. ISBN: 978-1-84882-284-9. DOI: 10.1007/978-1-84882-285-6. URL: http://www.springer.com/978-1-84882-284-9.

[KC06] Kyung-Joong Kim and Sung-Bae Cho. “A comprehensive overview of the applications of artificial life”. In: *Artificial Life* 12.1 (2006), pp. 153–182.

[KM18] Maciej Komosinski and Konrad Miazga. “Comparison of the tournament-based convection selection with the island model in evolutionary algorithms”. In: *Journal of Computational Science* 32 (2018), pp. 106–114. ISSN: 1877-7503. DOI: 10.1016/j.jocs.2018.10.001. URL: http://www.framsticks.com/files/common/ConvectionSelectionVsIslandModel.pdf.

[Knu09] Johan Knutzen. “Generating climbing plants using L-systems”. MA thesis. 2009. URL: http://www.cse.chalmers.se/~uffe/xjobb/climbingplants.pdf.

[Kom19] Maciej Komosinski. *Humann3ss*. 2019. URL: http://www.mooncoder.com/humann3ss.

[KT20] Iuliia Kotseruba and John K. Tsotsos. “40 years of cognitive architectures: core cognitive abilities and practical applications”. In: *Artificial Intelligence Review* 53.1 (2020), pp. 17–94. DOI: 10.1007/s10462-018-9646-y.

[KU17] Maciej Komosinski and Szymon Ulatowski. “Multithreaded computing in evolutionary design and in artificial life simulations”. In: *The Journal of Supercomputing* 73.5 (2017), pp. 2214–2228. ISSN: 1573-0484. DOI: 10.1007/s11227-016-1923-4. URL: http://www.framsticks.com/files/common/MultithreadedEvolutionaryDesign.pdf.

[Lan97] Christopher G. Langton. *Artificial life: An overview*. MIT Press, 1997.

[Life10] Jean Gayon et al., eds. *Defining life*. Vol. 40. Origins of Life and Evolution of Biospheres 2. Springer, 2010, pp. 119–244. URL: https://cache.media.eduscol.education.fr/file/Formation_continue_enseignants/52/2/Jean_Gayon_2_292522.pdf.

[Mah92] Samir W. Mahfoud. “Crowding and preselection revisited”. In: *Parallel problem solving from nature*. Ed. by R. Männer and B. Manderick. Vol. 2. Elsevier, 1992, pp. 27–36. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.52.3943&rep=rep1&type=pdf.

[Mer97] Ralph C. Merkle. “It’s a small, small, small, small world”. In: *MIT Technology Review* (1997). URL: https://www.researchgate.net/profile/Ralph_Merkle/publication/228378235_It's_a_small_small_small_small_world/links/0a85e53a224a7d10bd000000.pdf.

[MKT99] Soraia Raupp Musse, Marcelo Kallmann, and Daniel Thalmann. “Level of autonomy for virtual human agents”. In: *European Conference on Artificial Life*. Springer, 1999, pp. 345–349.

[NP99] Stefano Nolfi and Domenico Parisi. “Exploiting the power of sensory-motor coordination”. In: *European Conference on Artificial Life*. Springer, 1999, pp. 173–182.

[ÖBK08] E. Özcan, B. Bilgin, and E. E. Korkmaz. “A comprehensive analysis of hyper-heuristics”. In: *Intelligent Data Analysis* 12.1 (2008), pp. 3–23.

[OG03] Mihai Oltean and Crina Groşan. “Evolving evolutionary algorithms using multi expression programming”. In: *European Conference on Artificial Life*. Springer. 2003, pp. 651–658. URL: https://www.researchgate.net/profile/Mihai_Oltean2/publication/226167912_Evolving_Evolutionary_Algorithms_Using_Multi_Expression_Programming/links/55dac32308aed6a199aaf916.pdf.

[Olt05] Mihai Oltean. “Evolving evolutionary algorithms using linear genetic programming”. In: *Evolutionary Computation* 13.3 (2005), pp. 387–410. URL: https://mihaioltean.github.io/oltean_mit_draft_2005.pdf.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The algorithmic beauty of plants*. Springer, 1996. URL: http://algorithmicbotany.org/papers/abop/abop.pdf.

[PM01] Yoav I. H. Parish and Pascal Müller. “Procedural modeling of cities”. In: *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*. 2001, pp. 301–308. URL: https://dl.acm.org/doi/abs/10.1145/383259.383292.

[RB99] Andreas Rechtsteiner and Mark A. Bedau. “A generic neutral model for quantitative comparison of genotypic evolutionary activity”. In: *European Conference on Artificial Life*. Springer, 1999, pp. 109–118.

[Rea+20] Esteban Real et al. “AutoML-Zero: Evolving machine learning algorithms from scratch”. In: *International Conference on Machine Learning*. PMLR. 2020, pp. 8007–8019. URL: https://proceedings.mlr.press/v119/real20a/real20a.pdf.

[Ros05] P. Ross. “Hyper-heuristics”. In: *Search Methodologies* (2005), pp. 529–556.

[Rot06] Franz Rothlauf. *Representations for genetic and evolutionary algorithms*. Springer, 2006. DOI: 10.1007/3-540-32444-5.

[SB06] Christine Solnon and Derek Bridge. “An ant colony optimization meta-heuristic for subset selection problems”. In: *System engineering using particle swarm optimization* (2006), pp. 7–29. URL: https://liris.cnrs.fr/Documents/Liris-2279.pdf.

[Sch44] Erwin Schrödinger. *What is life? The physical aspect of the living cell and mind*. https://en.wikipedia.org/wiki/What_Is_Life%3F. Cambridge University Press, 1944. URL: http://old.biovip.com/UpLoadFiles/Aaron/Files/2005051204.pdf.

[Sip95] Moshe Sipper. “An introduction to artificial life”. In: *Explorations in Artificial Life (special issue of AI Expert)* (1995), pp. 4–8.

[Tha+95] Daniel Thalmann et al. “Virtual and real humans interacting in the virtual world”. In: *Proc. International Conference on Virtual Systems and Multimedia95*. 1995, pp. 48–57. URL: https://infoscience.epfl.ch/record/102037/files/Thalmann_and_al_VSMM_95.pdf.

[TTG94] Demetri Terzopoulos, Xiaoyuan Tu, and Radek Grzeszczuk. “Artificial fishes: Autonomous locomotion, perception, behavior, and learning in a simulated physical world”. In: *Artificial Life* 1.4 (1994), pp. 327–351. URL: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.8131&rep=rep1&type=pdf.

[WS05] Peter Worth and Susan Stepney. “Growing music: musical interpretations of L-systems”. In: *Workshops on Applications of Evolutionary Computation*. Springer. 2005, pp. 545–550. URL: https://www-users.cs.york.ac.uk/susan/bib/ss/nonstd/eurogp05.pdf.

Citing this script:

```
@booklet{MK-ALIFEscript,
  title  = {Artificial Life and Nature-Inspired Algorithms},
  author = {Maciej Komosinski},
  year   = {2025},
  note   = {Lecture script},
  url    = {https://www.cs.put.poznan.pl/mkomosinski/lectures/MK\_ArtLife.pdf}
}
```