Biologically-inspired algorithms and models

3. Variants and specializations of evolutionary algorithms

Maciej Komosinski

The "messy" approach is intended to improve the properties of the genetic algorithm through a more effective use and processing of schemata. The same purpose is pursued by the inversion operator, which will be covered in the next presentation on nature-inspired mechanisms. A messy genetic algorithm [Gol+93] uses a specific representation of individuals: genotypes are of variable length, composed of pairs (*label*, *value*). The label represents the meaning of a gene – as in the inversion operator, the label can be the initial number of the gene.

Incomplete (underspecified) genotypes, i.e., genotypes that do not specify the values of all genes, are allowed. A genotype may also contain redundant or even contradictory genes.

Messy genetic algorithm: operation

Three genetic operators are used: cutting, splicing, and mutation. The cutting operator cuts a string of bits with some probability, at a random point. The splicing operator merges two genotypes with some probability. Mutation is identical to a simple mutation.

Tournament selection is used. The optimization process consists of two phases (possibly repeatedly performed): the selection of building blocks and the application of operators. The population size is variable during the course of the algorithm.

Redundant (overspecified) genotypes can be easily handled by considering the first encountered value of a given gene in the genotype, but other methods exist – such as averaging all the values of a gene or using some kind of voting to determine which value to choose. Underspecified genotypes, unless they are acceptable in a given optimization problem, are resolved by filling in the missing genes with the best known value of a given gene from an earlier phase of the algorithm.

Messy genetic algorithms used for deceptive problems performed several times better than classical genetic algorithms with point crossover.

Discussion: is there some way to “detect” epistasis?

As with messy genetic algorithms, the motivation for the development of H-GA was the desire to automatically discover the degree of interdependence of solution parts (*linkage*, not to be confused with genetic linkage) in order to decompose the problem. By sampling specially constructed solutions, the dependence or independence of genes and groups of genes can be determined with some probability, and then independent optimization can be performed for the detected independent groups (modules) [JTW04].

To thoroughly investigate the independence of two genes from the rest of the solution, it would be necessary to generate a set of solutions in which all possible pairs of values of these two genes were surrounded by all possible values of the remaining genes (which would constitute "the context"). Then one would need to evaluate all these solutions and determine the relationship between the values of the genes and the value of the objective function. This would be very computationally expensive, and yet it would only be a test for one pair of genes! This is why sampling is used – it allows to estimate the potential independence for all subsets of genes in a solution (cf. GOMEA: Gene-pool Optimal Mixing EA [Thi18]).

Detecting dependencies: statistical and empirical techniques

The prospect of automatic decomposition of the optimization problem is very attractive – what remains is the matter of methods and their efficiency, so this issue is actively researched.

Methods for detecting dependencies between genes are sometimes divided into statistical (such as H-GA or the GOMEA family of methods) and empirical (such as DLED: Direct Linkage Empirical Discovery). The former are based on individuals in the population and their evaluations, and from this they statistically estimate the independence of genes. The latter sample and evaluate the complete neighborhood of a particular individual, so in its local neighborhood and for its particular set of gene values, they acquire complete information about (in)dependence.

Based on this information, we discover if and how the problem can be decomposed – which can take place just as the algorithm is running [PKF21, Sect. 5]. The algorithm can thus appropriately manage subpopulations optimizing potentially independent subproblems, adapt the crossover operator, the mutation operator, etc.

Suppose we have an individual whose genotype consists of at most five elements. We represent the existence of each element by one bit (for example, four out of five elements would be e.g. the 10111 individual).

Interdependencies between genes are more pronounced in local optima. Therefore, if we so desire, individuals that are local optima can be subjected to analysis – for example, they can be optimized beforehand using the Greedy method (with a random order of neighbors–genes) by swapping individual $1 \rightarrow 0$ and $0 \rightarrow 1$.

Suppose we have an individual whose genotype consists of at most five elements. We represent the existence of each element by one bit (for example, four out of five elements would be e.g. the 10111 individual).

Interdependencies between genes are more pronounced in local optima. Therefore, if we so desire, individuals that are local optima can be subjected to analysis – for example, they can be optimized beforehand using the Greedy method (with a random order of neighbors–genes) by swapping individual $1 \to 0$ and $0 \to 1$.

We perform the DLED-type decomposition on the analyzed genotype of the individual [PKF21]:

1. For each gene A, we introduce a perturbation (change its value to the opposite).
2. We check all the other genes, how for gene A after perturbation, the value of another gene B affects fitness if it remains unchanged and if it is changed.
3. The decision about dependency is binary and follows from satisfying the condition(s) listed on the next slide.

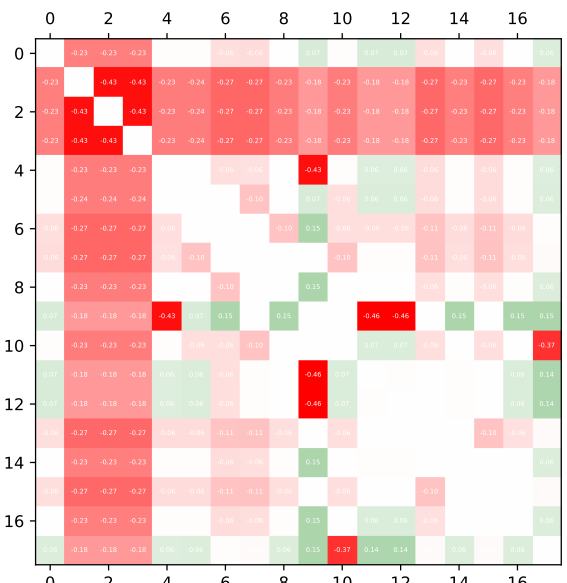
The DLED technique: conditions for the dependence of genes

The conditions from [PTK23]:

> The situation changed when the Direct Empirical Linkage Discovery (DLED) was proposed [17]. To check the dependency between genes g, h, DLED requires computing $f(\boldsymbol{x})$, $f(\boldsymbol{x}, g)$, $f(\boldsymbol{x}, h)$, $f(\boldsymbol{x}, g, h)$ values, where $(\boldsymbol{x}, g)$ and $(\boldsymbol{x}, h)$ are the genotypes of individual $\boldsymbol{x}$ with genes g and h flipped, respectively. Finally, $f(\boldsymbol{x}, g, h)$ is the genotype of $\boldsymbol{x}$ with both genes flipped. In DLED, genes g and h are considered dependent if at least one of the conditions holds:
> C1. $f(\mathbf{x}) < f(\mathbf{x}, g) \;\;\&\;\; f(\mathbf{x}, h) \geq f(\mathbf{x}, g, h)$
> C2. $f(\mathbf{x}) = f(\mathbf{x}, g) \;\;\&\;\; f(\mathbf{x}, h) \neq f(\mathbf{x}, g, h)$
> C3. $f(\mathbf{x}) > f(\mathbf{x}, g) \;\;\&\;\; f(\mathbf{x}, h) \leq f(\mathbf{x}, g, h)$
> C4. $f(\mathbf{x}) < f(\mathbf{x}, h) \;\;\&\;\; f(\mathbf{x}, g) \geq f(\mathbf{x}, g, h)$
> C5. $f(\mathbf{x}) = f(\mathbf{x}, h) \;\;\&\;\; f(\mathbf{x}, g) \neq f(\mathbf{x}, g, h)$
> C6. $f(\mathbf{x}) > f(\mathbf{x}, h) \;\;\&\;\; f(\mathbf{x}, g) \leq f(\mathbf{x}, g, h)$
> The above conditions can be interpreted as the following statement. If the modification of one gene changes the fitness relations for the values of the other gene, then these two genes are dependent. DLED is an ELL technique proven to report only the direct dependencies.

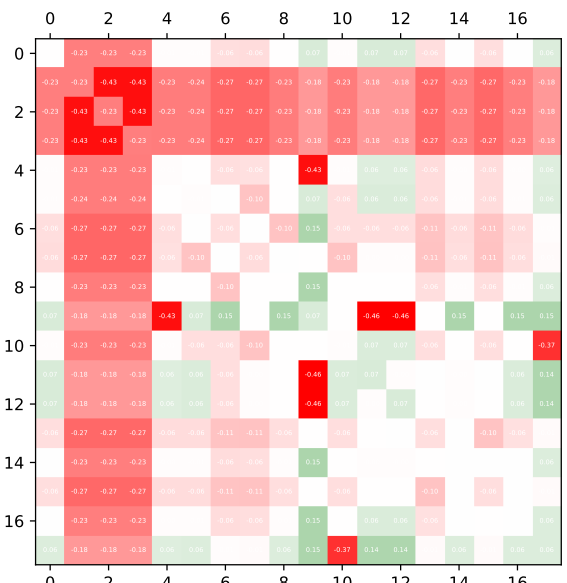
Epistasis estimation and decomposition – a specific example

Changes in fitness after pairs of genes are turned off in a sample individual



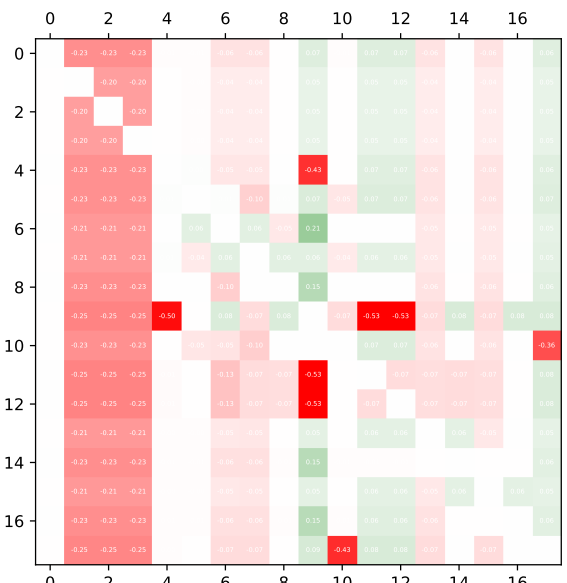
Epistasis estimation and decomposition – a specific example

Changes in fitness after pairs of genes are turned off; the diagonal shows the effect of turning off one gene



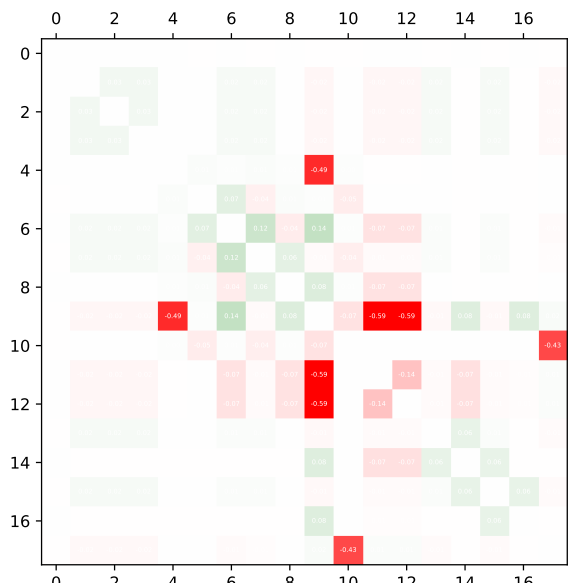
Epistasis estimation and decomposition – a specific example

Subtracting values on the diagonal from the rows: how to interpret the result?



Epistasis estimation and decomposition – a specific example

Additivity of the influence or lack thereof



What should constitute rows and columns (in this example it was "turning off genes", but is that always the most appropriate action?)

What should constitute rows and columns (in this example it was "turning off genes", but is that always the most appropriate action?)

How to use this information in the algorithm during optimization?

Epistasis estimation and decomposition

What should constitute rows and columns (in this example it was "turning off genes", but is that always the most appropriate action?)

How to use this information in the algorithm during optimization?

- optimize independent subsets of genes separately → decrease computational complexity
- design crossover and mutation to preserve beneficial epistatic interactions → treat co-adapted epistatic groups of genes as units → respect and effectively propagate "building blocks" of good solutions [GT12; TB13]

Epistasis estimation and decomposition

What should constitute rows and columns (in this example it was "turning off genes", but is that always the most appropriate action?)

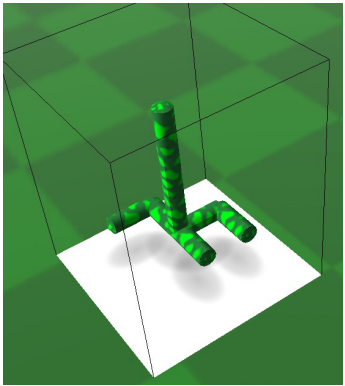
How to use this information in the algorithm during optimization?

- optimize independent subsets of genes separately → decrease computational complexity
- design crossover and mutation to preserve beneficial epistatic interactions → treat co-adapted epistatic groups of genes as units → respect and effectively propagate "building blocks" of good solutions [GT12; TB13]

How to employ an analogous approach to detect third-order interdependencies? (between triplets of genes?)

Genotype: /*9*/UDDDLFBFBRFBBFBFBR

Fitness: elevation of the center of mass of the structure
(0.547 for the original genotype – a good genotype, but not locally optimal)



Evolutionary strategies: origins

Evolutionary strategies (ES) were developed for some time independently from GAs as methods for numerical optimization. Many aspects distinguish them from GAs; what is common is the use of evolutionary mechanisms during optimization.

Evolutionary strategies (ES) were developed for some time independently from GAs as methods for numerical optimization. Many aspects distinguish them from GAs; what is common is the use of evolutionary mechanisms during optimization.

ES – natural origin: one of the first applications (1964) was *evolutionary design*, i.e., the engineering of structures (we will talk about this problem later and will experiment in lab classes). In order to minimize the water flow drag and optimize pipe shapes, to evaluate a design or a pipeline, it was not simulated, but actually built [Rec84, see figure on p. 123]; changes in design corresponded to "mutations". It was therefore a way of proceeding that implemented an optimization algorithm.

Early evolutionary strategies used only the mutation operator, which modified the only individual that was processed. Unlike in genetic algorithms, the individual was a pair consisting of a vector of variable values and a vector of standard deviations (which was constant throughout the process of evolution). Mutation consisted of changing each variable in the vector of values by a random factor generated according to the normal distribution with the corresponding standard deviation (specified in the vector of standard deviations). The individual after mutation replaced its ancestor only if it was better than it and feasible.

Such a strategy has been named two-membered (because at any given time, there exists one descendant and one ancestor) and is denoted (1+1)-ES. Its operation is similar to *Local Search*.

An improvement of the two-membered strategy is the multi-membered strategy, in which, as in GAs, a population of individuals exists. In addition, a uniform crossover is introduced, but it is not applied to all individuals, only to two of them – so that a single offspring is produced that replaces the worst individual (one new individual – thus analogously to the *steady-state* evolutionary algorithms).

Another improvement was the use of crossover many times in one step (many descendants were created), followed by the selection of *POPSIZE* individuals from the ancestors and descendants (the so-called *plus-selection*). Another approach selects individuals for the next generation only from the group of descendants (the so-called *comma-selection*), which is advantageous in problems with a moving optimum.

A general and concise notation of the ES architecture is $(\mu/\rho, \lambda)$-ES or $(\mu/\rho + \lambda)$-ES, where μ denotes the number of parents, $\rho \leq \mu$ is the number of parents from which offspring are produced, λ is the number of offspring, the comma symbol is used to specify selection only from the set of descendants, and the plus symbol – from both the set of parents and the set of descendants.

An improved mutation operator is used, which modifies not only the value of the variable, but also the standard deviation of these changes, which is also subject to evolution. In addition to the variable values and standard deviations, information about the preferred deviation angle during the search process can be introduced into the representation of the individual, this way improving the rate of convergence of evolutionary strategies. A variable is then represented by its value, standard deviation and deviation angle, and all these quantities are subject to evolution allowing for self-adaptation and enabling precise local fine-tuning.

Arithmetic crossover (the weighted average of parents) can also be used.

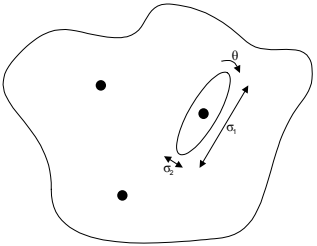


Figure: Mutation parameters in evolutionary strategies.

The evolutionary strategy that is known for its effectiveness is the strategy that adaptats the mutation covariance matrix, CMA-ES,* whose implementation is available for example in the DEAP library.** This strategy is also suitable for ill-conditioned problems.

The CMA-ES method has many parameters, and there are many alternative mechanisms for each step of the method. Default values can be used, as well as multiple run policies that free the user from having to specify values for any parameters.

*https://en.wikipedia.org/wiki/CMA-ES
**https://deap.readthedocs.io/en/master/examples/cmaes.html

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,
- sample solutions from a multivariate (n) normal distribution (specified by a single parameter – isometric, or n parameters – scaling parallel to the axes, or $\binom{n}{2}$ parameters that is the covariance matrix – allows for rotation),

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,
- sample solutions from a multivariate (n) normal distribution (specified by a single parameter – isometric, or n parameters – scaling parallel to the axes, or $\binom{n}{2}$ parameters that is the covariance matrix – allows for rotation),
- evaluate all solutions,

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,
- sample solutions from a multivariate (n) normal distribution (specified by a single parameter – isometric, or n parameters – scaling parallel to the axes, or $\binom{n}{2}$ parameters that is the covariance matrix – allows for rotation),
- evaluate all solutions,
- move the center of the population: set it in the location of the average weighted by the (ranking) fitness of the best individuals. Using ranking enables insensitivity to minor disturbances in the fitness value ("roughness" of the landscape) and its curvature – the degree of convexity,

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,
- sample solutions from a multivariate (n) normal distribution (specified by a single parameter – isometric, or n parameters – scaling parallel to the axes, or $\binom{n}{2}$ parameters that is the covariance matrix – allows for rotation),
- evaluate all solutions,
- move the center of the population: set it in the location of the average weighted by the (ranking) fitness of the best individuals. Using ranking enables insensitivity to minor disturbances in the fitness value ("roughness" of the landscape) and its curvature – the degree of convexity,
- the dispersion of new (sampled) individuals is proportional to the speed at which the center of the population moves: slower movement $\rightarrow$ less dispersion,

Evolutionary strategies: the main idea behind CMA-ES

- initialize the center of the population,
- sample solutions from a multivariate (n) normal distribution (specified by a single parameter – isometric, or n parameters – scaling parallel to the axes, or $\binom{n}{2}$ parameters that is the covariance matrix – allows for rotation),
- evaluate all solutions,
- move the center of the population: set it in the location of the average weighted by the (ranking) fitness of the best individuals. Using ranking enables insensitivity to minor disturbances in the fitness value ("roughness" of the landscape) and its curvature – the degree of convexity,
- the dispersion of new (sampled) individuals is proportional to the speed at which the center of the population moves: slower movement $\rightarrow$ less dispersion,
- update the covariance matrix to slightly stretch the multivariate normal distribution in the direction of the movement of the population center. By doing so, we will continue to follow the approximated gradient of the expected fitness of solutions.

Differential evolution (pol. *ewolucja różnicowa*)

A characteristic aspect of differential evolution is differential mutation [SP97]. In each iteration of evolution, for each individual o from the population of N individuals, repeat:

- randomly select n distinct individuals from N individuals, pick the base individual β and the difference individual δ (for $n = 3$, β can be picked randomly, and δ can be the difference of the two remaining individuals),
- create a temporary ("donor") individual $\omega = \beta + F\delta$ (F – constant),
- crossover ω with o,
- decide if the crossover outcome should replace the original o or not (selection).

DE is known for its simplicity, small number of parameters (sample implementation) and fast convergence. It does not require specifying a separate, independent probability distribution for mutation – mutation results from the state of the population. DE variants are competitive to other algorithms in annual optimization competitions.

Creating a temporary individual, ω: compare the simplex crossover discussed later.

Evolutionary programming (pol. *programowanie ewolucyjne*)

Three main differences between EP and GAs:

1. The representation of the solution does not have to be binary – it follows naturally from the problem.
2. Mutation changes parts of the solution, with small changes being more frequent and large changes – less frequent.
3. Crossover may be absent.

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, undirected graph, directed graph, matrix, logical expressions, rules (as in genetic-based machine learning, LCS/GBML), neural networks, automata, grammar expressions (e.g. stored as RPN), expressions structured as trees, programs (as in GP discussed later), ...

*Crossover for permutations: OX, PMX, ERO, others: https://hrcak.srce.hr/file/163313

Real numbers representation – operators

It is very common in EAs (and in optimization in general!) to use the representation of continuous values. Genes encode real numbers in the format that is standard on processors (variable precision depending on the absolute value of the number).

Question: what crossover and mutation operators (in addition to the usual ones, such as gene exchange or multi-point) can be proposed for a vector of numbers? When proposing operators, keep in mind the purpose of crossover and mutation.

Crossover: for example, the average of the parents, or a weighted average to get two different offspring. The weighted average is an **arithmetic** crossover – the offspring are a linear combination of the parents: $d_1 = r_1 \cdot a + r_2 \cdot (1 - a)$, $d_2 = \ldots$

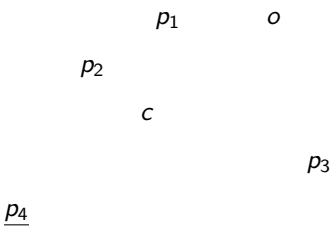
The weight a can be a drawn randomly on each execution.

Simplex crossover

We calculate the centroid c of the parents p.

A variant without accessing the fitness of solutions – SPX [TYH99]: we randomly pick an offspring from the (expanded) space of linear combinations of the parents (the parents get offset from c by ε – *the expanding rate*).

A variant involving fitness: we create an offspring o as an offset from the worst individual/parent further through the point c.



- *uniform random* (**Flat**) – set the gene to a random value from the allowed range
- *creep* – change the gene by a value drawn from some distribution (e.g., normal or uniform – e.g., $-3..+3$, etc.)

In order to achieve independence from the "axes" (i.e., to avoid the mutation being only parallel to the axes – only affecting individual parameters, which would be disadvantageous if the objective function were, for example, a rotated version of the function that directly depended on the parameters), all genes are mutated at once (and then we use the normal distribution of the random change rather than the uniform distribution – figure out why).

To ensure that such a mutation of n elements of the vector at once will move the current solution by the same distance in the n-dimensional space as a mutation of only one dimension would move, by what value should each of the n random values of the change in the n-dimensional vector be divided (normalized)?

What should we do if the value of a gene after mutation falls out of the allowed range? What methods can be proposed to solve this problem? [Bul99; Bul01]

Real numbers representation – mutation

Methods of handling mutants out of the allowed range

- **Absorb**: Illegal mutant values are truncated to the nearest boundary – the well-known trick with max(min()).
- **Repeat**: Mutant values are repeatedly generated, until a legal value is obtained.
- **Replace**: Any offspring for which illegal trait values are generated is replaced by a new offspring, re-choosing parents.

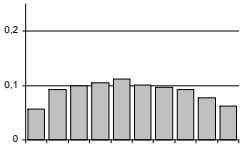
- **Ignore**: Mutation events which transgress legal bounds are ignored. Rather than inherit an illegal mutant value, offspring inherit the parental value.
- **Reflect**: Mutant values lying a distance of d above (or below) the legal range are replaced by vales a distance of d below (or above) the nearest boundary.
- **Wrap**: The trait is treated as if it were periodic. The edges of its legal range "wrap" around. Mutant values are calculated modulo the trait's range.

Does it matter which method we will choose? Yes! [discussion of desirable mutation characteristics and selection of the winner method].

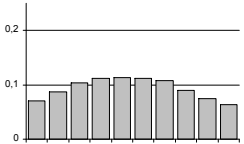
Experimental results: how often did certain values of a gene occur after mutation and "repair" by various methods? The ancestor had an equal chance of each value. Horizontal axis – the range of gene variability, vertical axis – the frequency of offspring values in subintervals:

Absorb

Repeat



Replace



Ignore, Reflect, Wrap, Flat – similar to:

The fact that in the **Ignore**, **Reflect**, **Wrap** methods, the same distribution was obtained as in **Flat** does not mean that these methods perform in the same way. Actually, the principle of their operation is quite different. Example: the **Ignore** method. Near the boundaries of the range, more mutations will be invalid and ignored. Since they will be ignored, values close to the boundaries will also occur less often overall. When they eventually occur, they will rarely result in a valid mutation...

Consequently, the fact that the distribution is the same does not yet guarantee that effective (i.e., actually changing the value of the gene) mutations in particular subintervals occur with the same frequency. Hence, in addition to the frequency distribution of gene values for different methods, it is also necessary to compare other parameters – e.g., how many numbers are moved from each subinterval (before mutation) to every other one (after mutation), what is the relationship between such pairs, etc. In this regard, the **Ignore**, **Reflect**, **Wrap** methods are different, and none of them performs like **Flat**.

We considered here a straightforward type of mutation and simple mechanisms, yet these small elements have a big impact on the evolutionary process. For example, **Absorb** guides (introduces bias to) the genetic drift **continuously** towards the extreme values of the allowed ranges. Without being aware of it, one can conclude that such values are optimal and evolution favors them – meanwhile, it is a continuous influence of mutation.

A review of mutation and crossover operators for numerical problems is provided by books–collections [Gwi07a; Gwi07b].

Reminder from earlier studies (and, possibly, a supplement)

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?

Reminder from earlier studies (and, possibly, a supplement)

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?
- How to intentionally develop (design) effective crossover operators?

Reminder from earlier studies (and, possibly, a supplement)

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?
- How to intentionally develop (design) effective crossover operators?
- What is the characteristic of DPX – *distance-preserving crossover*?

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?
- How to intentionally develop (design) effective crossover operators?
- What is the characteristic of DPX – *distance-preserving crossover*?
- How does the expertise from the above questions translate to the mutation operator?

Reminder from earlier studies (and, possibly, a supplement)

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?
- How to intentionally develop (design) effective crossover operators?
- What is the characteristic of DPX – *distance-preserving crossover*?
- How does the expertise from the above questions translate to the mutation operator?
- What is *global convexity* and how to identify it?

- Having several ideas for crossover operators, how do we know which one is probably going to perform better?
- How to intentionally develop (design) effective crossover operators?
- What is the characteristic of DPX – *distance-preserving crossover*?
- How does the expertise from the above questions translate to the mutation operator?
- What is *global convexity* and how to identify it?
- What does the FDC measure evaluate?

Quantitative evaluation of similarity/distance between solutions

The measure of similarity of solutions has numerous applications – among others, it is useful for:

- testing ideas for the crossover operator – different properties of solutions and FDCs,
- performing crowding model selection – discussed earlier,
- estimating diversity in the population and assessing convergence,
- analyzing population structure; analyzing clusters in a set of solutions,
- maintaining "species" during evolution – these methods will be discussed later,

and wherever there is a need to determine the difference between two solutions, such as in the already presented: differential evolution and simplex crossover.

If solutions have a simple representation (consider a few examples), then ideas for similarity measures may come to mind naturally. For complex representations (consider a few examples), the concepts of edit distance* and earth mover's distance** may be useful.

*https://en.wikipedia.org/wiki/Edit_distance
**https://en.wikipedia.org/wiki/Earth_mover%27s_distance

Embryogeny

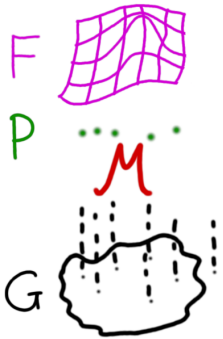
Embryogeny: mapping genotype → phenotype. For simple representations and uniform, homogeneous spaces like the full space of bits, numbers or permutations, a trivial direct 1:1 mapping is the first (default) idea.

But is such a mapping the best choice?

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.

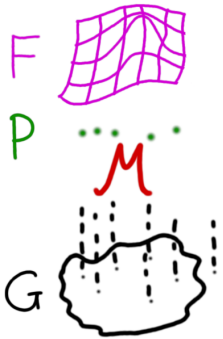
But is such a mapping the best choice?



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?

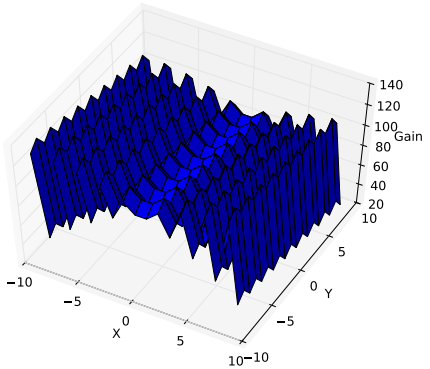


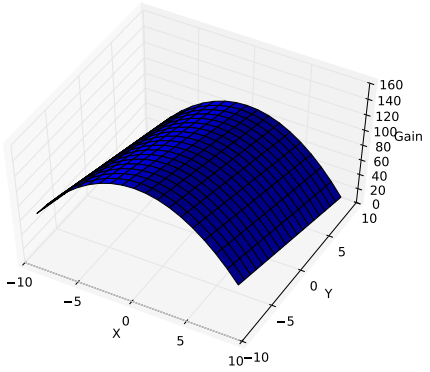
Recall RGB ↔ HSL, signal ↔ spectrum, ...

Embryogeny – what for?

Consider in which situations the genotype → phenotype mapping should (or must?) be more sophisticated. What properties of the mapping should be provided by the procedure that maps the genotype space into the phenotype space?

Consider in which situations the genotype → phenotype mapping should (or must?) be more sophisticated. What properties of the mapping should be provided by the procedure that maps the genotype space into the phenotype space?





Embryogeny – what for?

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

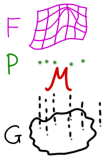
Embryogeny – what for?

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

Embryogeny – properties and benefits

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),
- more complex solutions in the phenotype space ("growing instructions" in genotype),
- improved constraint handling (mapping **every** genotype into a valid phenotype),

and:

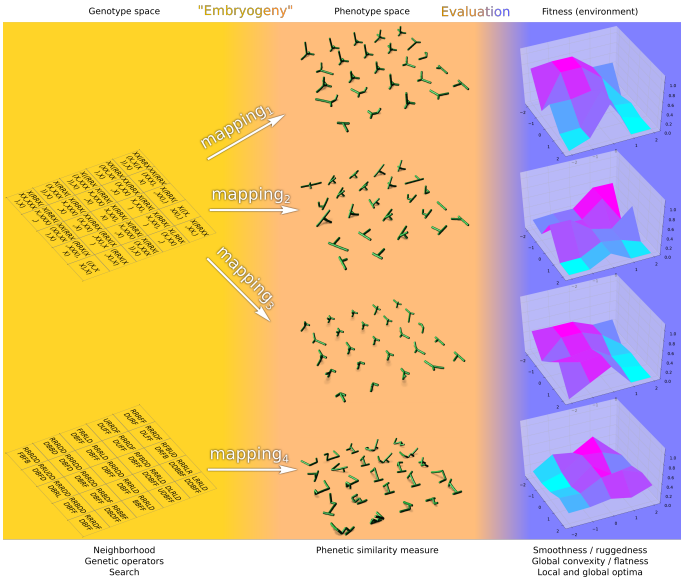
- compression: simple genotypes define complex phenotypes,
- repetition: genotypes can describe symmetry, segmentation, subroutines, etc.,
- adaptation: phenotypes can be grown "adaptively" (to satisfy constraints, or to correct errors).

Embryogeny – challenges

- 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 (*external embryogeny*). For more information, see later lecture on structure optimization/evolutionary design.

Embryogeny – illustration



The diagram on the previous slide shows 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,
3. the same representation and the same interpretation of genes, but different mutation/neighborhood operators (not shown).

Genetic programming (pol. *programowanie genetyczne*)

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

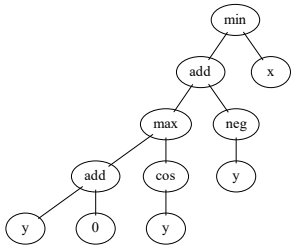


Figure: Expression `min(add(max(add(y, 0), cos(y)), neg(y)), x))` which is `min(max(y + 0, cos(y)) + (-y), x)` which is `min(x, max(y, cos(y)) - y)`.

*Free book:
http://www0.cs.ucl.ac.uk/staff/W.Langdon/ftp/papers/poli08_fieldguide.pdf
**https://en.wikipedia.org/wiki/Linear_genetic_programming

Expressions existing in a population consist of elements that belong to the set of functions ***F*** (tree nodes) and the set of terminals ***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.

Genetic programming – an example of using the DEAP library

```
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')
```

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

Consider how the *closure* property can be ensured.

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.

- *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.
- *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. 4.
- *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.

Generating individuals using the *Full* method

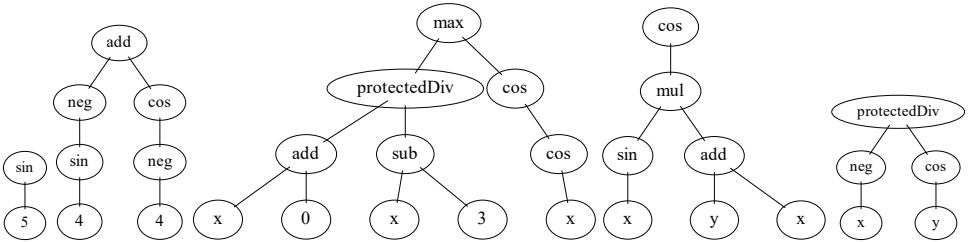


Figure: Five individuals generated using the *Full* method, `gp.genFull(pset,1,3)` (DEAP requires two parameters, not one) for $\boldsymbol{T}=\{x, y, 0, 1, 2, 3, 4, 5\}$.

Generating individuals using the *Grow* method

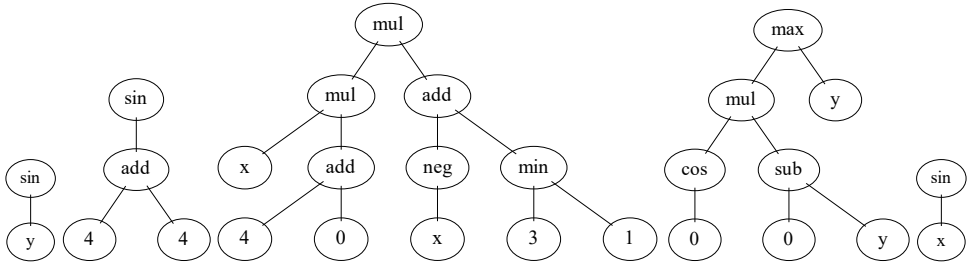


Figure: Five individuals generated using the *Grow* method, `gp.genGrow(pset,1,3)`, for $\boldsymbol{T}=\{x, y, 0, 1, 2, 3, 4, 5\}$. In the DEAP's `genGrow()` method there is no point in setting the min_depth and max_depth arguments to the same value, because then the generated trees will have all the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

Crossing over

Crossing over

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

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

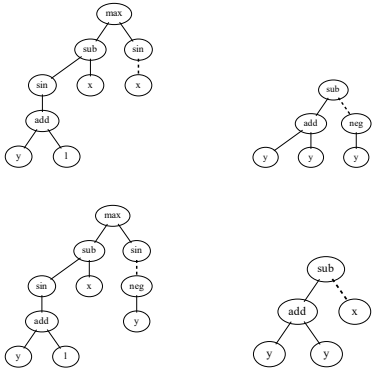


Figure: Crossing over in GP. Top: parents generated by the `gp.genGrow(pset,2,4)` method. Bottom: offspring generated using the `gp.cxOnePoint(parent1,parent2)` method.

Mutation

Mutation

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.

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

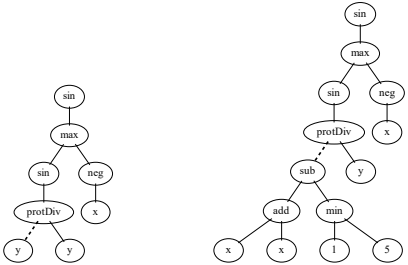


Figure: Left: original solution generated by the `gp.genGrow(pset,2,5)` method. Right: a mutant created using the `gp.mutUniform(parent, toolbox.expr_mut, pset=pset)` method, with earlier `toolbox.register("expr_mut", gp.genFull, min_=0, max_=2)`.

Bloating of solutions

Bloating of solutions

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))
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.
```

While the standard selection methods discussed earlier can be used, Lexicase selection often delivers better results. In this method, we do not aggregate the errors of each solution on all tests into a single value. Instead, to select one individual from the population, we first randomize the order of the tests, and then select those individuals that scored the best in the population on the first test (from this randomized order). If there is more than one equally best individual, we also compare their performance on the second test, then possibly the third, and so on.

Discussion: how does this approach differ from selection methods used in evolutionary multi-criteria optimization such as NSGA or SPEA?

The effectiveness of GP

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

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 and even look for certain classes of functions or for any functions – hence this regression method is called *symbolic*.

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.

Symbolic regression – looking for $f(x) = x^2 - x$

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
$\text{neg}, +, -, *, /, \max, \min, \sin, \cos$, and additionally, constants $-1, 0, 1$.

Symbolic regression – an example of using the DEAP library


```
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)])
```

Looking for $f(x) = x^2 - x$. First generation

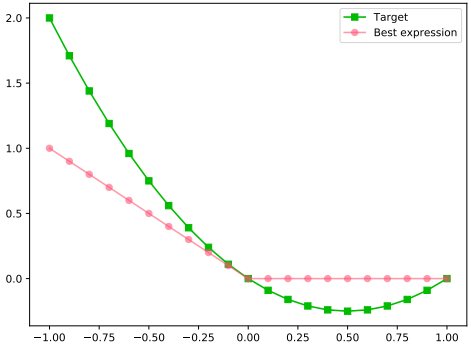


Figure: The best solution in the first generation (i.e., in a randomly generated population).

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

Looking for $f(x) = x^2 - x$. Last generation

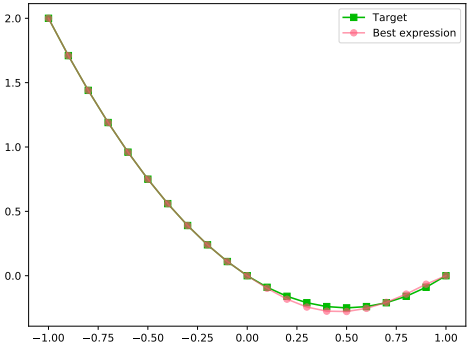


Figure: The best solution once the evolution finished.

Looking for $f(x) = x^2 - x$. Last generation

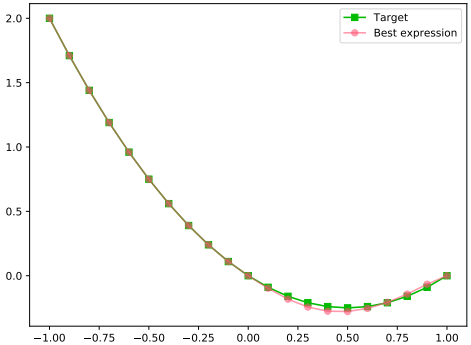


Figure: The best solution once the evolution finished.

```
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., $\{x_1, x_2, y\} = \{(0,0,0); (0,1,1); (1,0,1); (1,1,0)\}$.

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

Symbolic regression – looking for XOR: functions and terminals

```
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")
```

Symbolic regression – looking for XOR: evaluation


```
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,
```

Symbolic regression – looking for XOR: results

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

Symbolic regression – looking for XOR: results

- 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 $= 1$)

Symbolic regression – looking for XOR: results

- 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)`

Symbolic regression – looking for XOR: results

- 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)

Symbolic regression – looking for XOR: results

- 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)))`

Symbolic regression – looking for XOR: results

- 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: 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?

Genetic programming – 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].

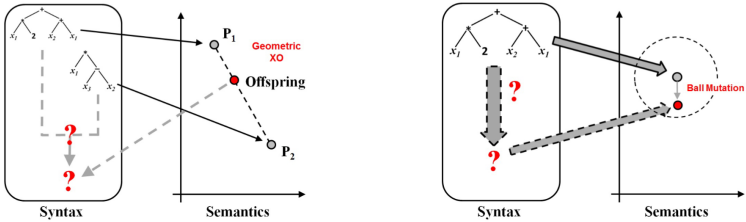


Figure 1: Geometric semantic crossover (plot (a)) (respectively geometric semantic mutation (plot (b))) performs a transformation on the syntax of the individual that corresponds to geometric crossover (respectively geometric mutation) on the semantic space. In this figure, the unrealistic case of a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

Looking for a training algorithm for a neural network

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

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

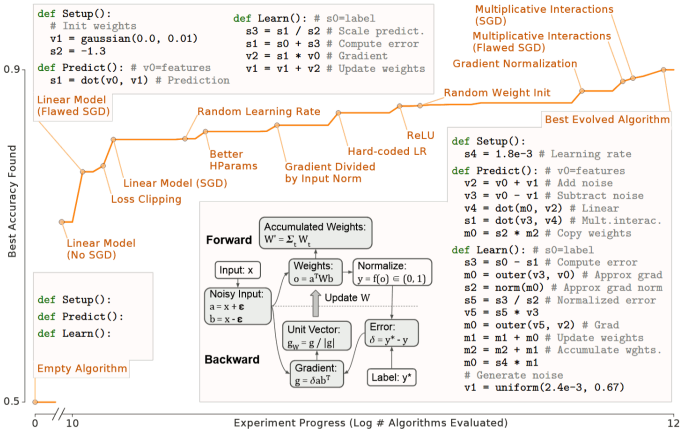


Figure 6: Progress of one evolution experiment on projected binary CIFAR-10. Callouts indicate some beneficial discoveries. We also print the code for the initial, an intermediate, and the final algorithm. The last is explained in the flow diagram. It outperforms a simple

Hyper-heuristics and self-programmable algorithms

*http://en.wikipedia.org/wiki/Hyper-heuristic

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* that search through the space of heuristics and their combinations [Ros05; ÖBK08; Bur+10].

*http://en.wikipedia.org/wiki/Hyper-heuristic

And what if. . .

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

References I

[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.

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

[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.

[Bul01] Seth Bullock. "Smooth operator? Understanding and visualising mutation bias". In: *European Conference on Artificial Life*. Springer. 2001, pp. 602–612. DOI: 10.1007/3-540-44811-X_68.

[Bul99] Seth Bullock. "Are artificial mutation biases unnatural?" In: *European Conference on Artificial Life*. Springer. 1999, pp. 64–73. DOI: 10.1007/3-540-48304-7_11. URL: https://eprints.soton.ac.uk/261452/1/10.1.1.40.2753.pdf.

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

[Gol+93] David E. Goldberg et al. *Rapid, Accurate Optimization of Difficult Problems Using Fast Messy Genetic Algorithms*. Tech. rep. 93004. 1993, pp. 1–16. URL: http://repository.ias.ac.in/81677/1/110-a.pdf.

[GT12] Brian W. Goldman and Daniel R. Tauritz. "Linkage tree genetic algorithms: variants and analysis". In: *Proceedings of the 14th annual conference on Genetic and evolutionary computation*. 2012, pp. 625–632. URL: http://www.cmap.polytechnique.fr/~nikolaus.hansen/proceedings/2012/GECCO/proceedings/p625.pdf.

References II

[Gwi07a] Tomasz Dominik Gwiazda. *Algorytmy genetyczne – kompendium. Tom I. Operator krzyżowania dla problemów numerycznych*. PWN, 2007.

[Gwi07b] Tomasz Dominik Gwiazda. *Algorytmy genetyczne – kompendium. Tom II. Operator mutacji dla problemów numerycznych*. PWN, 2007.

[JTW04] E. D. de Jong, D. Thierens, and R. A. Watson. "Hierarchical genetic algorithms". In: *Lecture notes in computer science* (2004), pp. 232–241. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=20bde9fe796a5e15c9007c7dc770b35aa731751d.

[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.

[PKF21] Michal W. Przewozniczek, Marcin M. Komarnicki, and Bartosz Frej. "Direct linkage discovery with empirical linkage learning". In: *Proceedings of the Genetic and Evolutionary Computation Conference*. 2021, pp. 609–617. URL: https://www.cs.put.poznan.pl/mkomosinski/lectures/optimization/extras/DLED-epistasis-GECCO2021.pdf.

References III

[PTK23] Michal W. Przewozniczek, Renato Tinós, and Marcin M. Komarnicki. "First Improvement Hill Climber with Linkage Learning – on Introducing Dark Gray-Box Optimization into Statistical Linkage Learning Genetic Algorithms". In: *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '23*. ACM, 2023, pp. 946–954. DOI: 10.1145/3583131.3590495.

[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.

[Rec84] Ingo Rechenberg. "The Evolution Strategy. A Mathematical Model of Darwinian Evolution". In: *Synergetics – From Microscopic to Macroscopic Order*. Ed. by Eckart Frehland. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 122–132. DOI: 10.1007/978-3-642-69540-7_13.

[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.

[SP97] Rainer Storn and Kenneth Price. "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces". In: *Journal of Global Optimization* 11.4 (1997), pp. 341–359. ISSN: 1573-2916. DOI: 10.1023/A:1008202821328.

[TB13] Dirk Thierens and Peter A. N. Bosman. "Hierarchical problem solving with the linkage tree genetic algorithm". In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. 2013, pp. 877–884. URL: https://homepages.cwi.nl/~bosman/publications/2013_hierarchicalproblemsolving.pdf.

[Thi18] Dirk Thierens. *Model-Based Evolutionary Algorithms, Part 2: Linkage Tree Genetic Algorithm*. 2018. URL: https://www.cs.uu.nl/docs/vakken/ea/slides/LTGA_GOMEA.pdf.

References IV

[TYH99] Shigeyoshi Tsutsui, Masayuki Yamamura, and Takahide Higuchi. "Multi-parent recombination with simplex crossover in real coded genetic algorithms". In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation – Volume 1*. 1999, pp. 657–664. URL: https://www.researchgate.net/profile/Shigeyoshi-Tsutsui/publication/243776468_Multi-parent_recombination_with_simplex_crossover_in_real-coded_genetic_algorithms/links/00463531fa92bd4738000000/Multi-parent-recombination-with-simplex-crossover-in-real-coded-genetic-algorithms.pdf.