We are IntechOpen, the world's leading publisher of Open Access books
Built by scientists, for scientists

6,900
Open access books available

186,000
International authors and editors

200M
Downloads

Our authors are among the

154
Countries delivered to

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

CLARIVATE ANALYTICS
BOOK CITATION INDEX
INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Numbers displayed above are based on latest data collected.
For more information visit www.intechopen.com

Continuous Schemes for Program Evolution

Cyril Fonlupt, Denis Robilliard and Virginie Marion-Poty

Additional information is available at the end of the chapter

1. Introduction

Genetic Programming (GP) is a technique aiming at the automatic generation of programs. It was successfully used to solve a wide variety of problems, and it can be now viewed as a mature method as even patents for old and new discovery have been filled, see e.g. [1, 2]. GP is used in fields as different as bio-informatics [3], quantum computing [4] or robotics [5], among others.

The most widely used scheme in GP was proposed by Koza, where programs are represented as Lisp-like trees and evolved by a genetic algorithm. Many other paradigms were devised these last years to automatically evolve programs. For instance, linear genetic programming (LGP) [6] is based on an interesting feature: instead of creating program trees, LGP directly evolves programs represented as linear sequences of imperative computer instructions. LGP is successful enough to have given birth to a derived commercial product named *discipulus*. The representation (or genotype) of programs in LGP is a bounded-length list of integers. These integers are mapped into imperative instructions of a simple imperative language (a subset of C for instance).

While the previous schemes are mainly based on discrete optimization, a few other evolutionary schemes for automatic programming have been proposed that rely on some sort of continuous representation. These include notably Ant Colony Optimization in AntTAG [7, 8], or the use of probabilistic models like Probabilistic Incremental Program Evolution [9] or Bayesian Automatic Programming [10].

In 1997, Storn and Price proposed a new evolutionary algorithm for continuous optimization, called Differential Evolution (DE) [11]. Another popular continuous evolution scheme is the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) that was proposed by Hansen and Ostermeier [12] in 1996. Differential Evolution differs from Evolution Strategies in the way it uses information from the current population to determine the perturbation brought to solutions (this can be seen as determining the direction of the search).

In this chapter, we propose to evolve programs with continuous representation, using these two continuous evolution engines, Differential Evolution and CMA Evolution Strategy. A

program is represented by a float vector that is translated to a linear sequence of imperative instructions, *a la* LGP.

The chapter is organized in the following way. The first section introduces the Differential Evolution and CMA Evolution Strategy schemes, focusing on the similarities and main differences. We then present our continuous schemes, LDEP and CMA-LEP, respectively based on DE and CMA-ES. We show that these schemes are easily implementable as plug-ins for DE and CMA-ES. In Section 4, we compare the performance of these two schemes, and also traditional GP, over a range of benchmarks.

2. Continuous evolutionary schemes

In this section we present DE and CMA-ES, that form the main components of the evolutionary algorithms used in our experiments.

2.1. Previous works on evolving programs with DE

To our knowledge O'Neill and Brabazon were the firsts to use DE to evolve programs within the well known framework of Grammatical Evolution (GE) [13]. In GE, a population of variable length binary strings is decoded using a Backus Naur Form (BNF) formal grammar definition into a syntactically correct program. The genotype-to-phenotype mapping process allows to use almost any BNF grammars and so to evolve programs in many different languages. GE has been applied to various problems ranging from symbolic regression problems or robot control [14] to physical-based animal animations [15] including neural network evolution, or financial applications [16]... In [13], Grammatical Differential Evolution is defined by retaining the GE grammar decoding process for generating phenotypes, with genotypes being evolved with DE. A diverse selection of benchmarks from the GP literature were tackled with four different flavors of GE. Even if the experimental results indicated that the grammatical differential evolution approach was outperformed by standard GP on three of the four problems, the results were somewhat encouraging.

More recently, Veenhuis also introduced a successful application of DE for automatic programming in [17], mapping a continuous genotype to trees, so called Tree based Differential Evolution (TreeDE). TreeDE improved somewhat on the performance of grammatical differential evolution, but it requires an additional low-level parameter, the tree depth of solutions, that has to be set beforehand. Moreover evolved programs do not include random constants.

Another recent proposal for program evolution based on DE is called Geometric Differential Evolution, and was issued in [18]. These authors introduced a formal generalization of DE to keep the same geometric interpretation of the search dynamic across diverse representations, either for continuous or combinatorial spaces. This scheme is interesting, although it has some limitations: it is not possible to model the search space of Koza style subtree crossover for example. Anyway, experiments on four standard benchmarks against Langdon's homologous crossover GP were promising.

Our proposal differs from these previous works by being based on Banzhaf's Linear GP representation of solutions. This allows us to implement real-valued constant management

inspired from the LGP literature, that are lacking in TreeDE. The tree-depth parameter from TreeDE is also replaced by the maximum length of the programs to be evolved: this is a lesser constraint on the architecture of solutions and it still has the benefit of limiting the well known bloat problem (uncontrolled increase in solution size) that plagues standard GP.

2.2. Differential evolution

This section only introduces the main Differential Evolution (DE) concepts. The interested reader might refer to [11] for a full presentation. DE is a population-based search algorithm that draws inspiration from the field of evolutionary computation, even if it is not usually viewed as a typical evolutionary algorithm.

DE is a real-valued, vector based, heuristic for minimizing possibly non-differentiable and non linear continuous space functions. As most evolutionary schemes, DE can be viewed as a stochastic directed search method. But instead of randomly mating two individuals (like crossover in Genetic Algorithms), or generating random offspring from an evolved probability distribution (like PBIL [19] or CMA-ES [20]), DE takes the difference vector of two randomly chosen population vectors to perturb an existing vector. This perturbation is made for every individual (vector) inside the population. A newly perturbed vector is kept in the population only if it has a better fitness than its previous version.

2.2.1. Principles

DE is a search method working on a set or population $X = (X_1, X_2, \ldots, X_N)$ of N solutions that are $d-$dimensional float vectors, trying to optimize a fitness (or objective) function $f(X_i)_{i \in [1,N]} : \mathbb{R}^d \to \mathbb{R}$.

DE can be roughly decomposed into an initialization phase and three very simple steps that are iterated on:

> 1- initialization
> 2- mutation
> 3- crossover
> 4- selection
> 5- end if termination criterion is fulfilled else
>   go to step 2

At the beginning of the algorithm, the initial population is randomly initialized and evaluated using the fitness function f. Then new potential individuals are created: a new trial solution is created for every vector X_j, in two steps called mutation and crossover. A selection process is triggered to determine whether or not the trial solution replaces the vector X_j in the population.

2.2.2. Mutation

Let t indicate the number of the current iteration (or generation), for each vector $X_j(t)$ of the population, a variant vector $V_j(t+1) = (v_{j1}, v_{j2}, \ldots, v_{jd})$ is generated according to Eq. 1:

$$V_j(t+1) = X_{r_1}(t) + F \times (X_{r_2}(t) - X_{r_3}(t)) \tag{1}$$

where:

- r_1, r_2 and r_3 are three mutually *different* randomly selected indices in the population that are also different from the current index j.
- the scaling factor F is a real constant which controls the amplification of differential evolution and avoids the stagnation in the search process — typical values for F are in the range $[0, 2]$.
- The expression $(X_{r_2}(t) - X_{r_3}(t))$ is referred to as the difference vector.

Many variants were proposed for equation 1, including the use of more than 3 individuals. According to [17, 21], the mutation method that is the more robust over a set of experiments is the method DE/best/2/bin, defined by Eq. 2:

$$V_j(t+1) = X_{\text{best}}(t) + F \times (X_{r_1}(t) + X_{r_2}(t) - X_{r_3}(t) - X_{r_4}(t)) \tag{2}$$

where $X_{\text{best}}(t)$ is the best individual in the population at the current generation. This method DE/best/2/bin is used throughout the chapter.

2.2.3. Crossover

As explained in [11], the crossover step ensures to increase or at least to maintain the diversity. Each trial vector is partly crossed with the variant vector. The crossover scheme ensures that at least one vector component will be crossovered.

The trial vector $U_j(t+1) = (u_{j1}, u_{j2}, \ldots, u_{jd})$ is generated using Eq. 3:

$$u_{ji}(t+1) = \begin{cases} v_{ji}(t+1) & \text{if} \quad (rand \leq CR) \quad \text{or } j = rnbr(i) \\ x_{ji}(t) & \text{if} \quad (rand > CR) \quad \text{and } j \neq rnbr(i) \end{cases} \tag{3}$$

where:

- $x_{ji}(t)$ is the jth component of vector $X_i(t)$;
- $v_{ji}(t+1)$ is the jth component of the current variant vector $V_j(t+1)$ (see above Eq. 1 and 2);
- *rand* is a random float drawn uniformly in the range $[0, 1[$;
- *CR* is the crossover rate in the range $[0, 1]$ which has to be determined by the user;
- $rnbr(i)$ is a randomly chosen index drawn in the range $[1, d]$ independently for each vector $X_i(t)$ which ensures that $U_j(t+1)$ gets at least one component from the variant vector $V_j(t+1)$.

2.2.4. Selection

The selection step decides whether the trial solution $U_i(t+1)$ replaces the vector $X_i(t)$ or not. The trial solution is compared to the target vector $X_i(t)$ using a greedy criterion. Here we assume a minimization framework: if $f(U_i(t+1)) < f(X_i(t))$, then $X_i(t+1) = U_i(t+1)$ otherwise the old value is kept: $X_i(t+1) = X_i(t)$.

2.2.5. Iteration and stop criterion

These three steps (mutation, crossover, selection) are looped over until a stop criterion is triggered: typically a maximum number of evaluations/iterations is allowed, or a given value of fitness is reached. Overall DE is quite simple, only needing three parameters: the population size N, the crossover rate CR, and the scaling factor F.

2.3. Covariance matrix adaptation evolution strategy

Among continuous optimization methods, DE was often compared (in e.g. [22, 23]) to the Covariance Matrix Adaptation Evolution Strategy (CMA-ES), initially proposed in [12]. The CMA Evolution Strategy is an evolutionary algorithm for difficult non-linear non-convex optimization problems in continuous domains. It is typically applied to optimization problems of search space dimensions between three and one hundred. CMA-ES was designed to exhibit several invariances: (a) invariance against order preserving (i.e. strictly monotonic) transformations of the objective function value; (b) invariance against angle preserving transformations of the search space (e.g rotation, reflection); (c) scale invariance. Invariances are highly desirable as they usually imply a good behavior of the search strategy on ill-conditioned and on non-separable problems.

In this section we only introduce the main CMA-ES concepts, and refer the interested reader to the original paper for a full presentation of this heuristic. An abundant literature has brought several refinements to this algorithm (e.g. [24] and [25]), and has shown its strong interest as a continuous optimization method.

2.3.1. Principles

The basic CMA-ES idea is sampling search points using a normal distribution that is centered on an updated model of the ideal solution. This ideal solution can be seen as a weighted mean of a best subset of current search points. The distribution is also shaped by the covariance matrix of the best solutions sampled in the current iteration. This fundamental scheme was refined mainly on two points:

- extracting more information from the history of the optimization run; this is done through the so-called accumulation path whose idea is akin to the momentum of artificial neural networks;

- allocating an increasing computational effort via an increasing population size in a classic algorithm restart scheme.

The main steps can be summed-up as:

1. sample points are drawn according to the current distribution

2. the sample points are evaluated

3. the probability distribution is updated according to a best subset of the evaluated points

4. iterate to step 1, until the stop criterion is reached

2.3.2. Sampling step

More formally, the basic equation for sampling the search points (step 1) is:

$$x_k^{(g+1)} \leftarrow m^{(g)} + \sigma^{(g)} N(0, C^{(g)}) \tag{4}$$

where:

- g is the generation number
- $k \in 1, ..., N$ is an index over the population size
- $x_k^{(g+1)}$ is the k-th offspring drawn at generation g + 1
- $m^{(g)}$ is the mean value of the search distribution at generation g
- $\sigma^{(g)}$ is the "overall" standard deviation (or step-size) at generation g
- $N(0, C^{(g)})$ is a multivariate normal distribution with zero mean and covariance matrix $C^{(g)}$ at generation g

2.3.3. Evaluation and selection step

Once the sample solutions are evaluated, we can select the current best μ solutions, where μ is the traditional parameter of Evolution Strategies. Then the new mean $m^{(g+1)}$, the new covariance matrix $C^{(g+1)}$ and the new step size control $\sigma^{(g+1)}$ can be computed in order to prepare the next iteration, as explained in the following section.

2.3.4. Update step

The probability distribution for sampling the next generation follows a normal distribution. The new mean $m^{(g+1)}$ of the search distribution is a weighted average of the μ selected best points from the sample $x_1^{(g+1)}, \ldots, x_N^{(g+1)}$, as shown in Eq. 5:

$$m^{(g+1)} = \sum_{i=1}^{\mu} w_i x_{i:N}^{(g+1)} \tag{5}$$

where:

- $\mu \leq N$, μ best points are selected in the parent population of size N.
- $x_{i:N}^{(g+1)}$, i-th best individual out of $x_1^{(g+1)}, \ldots, x_N^{(g+1)}$ from Eq. 4.
- $w_1 \geq \ldots \geq w_\mu$ are the weight coefficients with $\sum_{i=1}^{\mu} w_i = 1$

Thus the calculation of the mean can also be interpreted as a recombination step (typically by setting the weights $w_i = 1/\mu$). Notice that the best μ points are taken from the new current generation, so there is no elitism.

Adapting the covariance matrix of the distribution is a complex step, that consists of three sub-procedures: the rank-μ-update, the rank-one-update and accumulation. They are similar

to a Principal Component Analysis of steps, sequentially in time and space. The goal of the adaptation mechanism is to increase the probability of successful consecutive steps.

In addition to the covariance matrix adaptation rule, a step-size control is introduced, that adapts the overall scale of the distribution based on information obtained by the evolution path. If the evolution path is long and single steps are pointing more or less to the same direction, the step-size should be increased. On the other hand, if the evolution path is short and single steps cancel each other out, then we probably oscillate around an optimum, thus the step-size should be decreased.

For the sake of simplicity, the details of the update of the covariance matrix C and step-size control are beyond the scope of this chapter.

2.4. Main differences between DE and CMA-ES

The Differential Evolution method and the CMA Evolution Strategy are often compared, since they are both population-based continuous optimization heuristics. Unlike DE, CMA-ES is based on strong theoretical aspects that allow it to exhibit several invariances that make it a robust local search strategy, see [12]. Indeed it was shown to achieve superior performance versus state-of-the art global search strategies (e.g. see [26]). On the other hand and in comparison with most search algorithms, DE is very simple and straightforward both to implement and to understand. This simplicity is a key factor in its popularity especially for practitioners from other fields.

Despite or maybe thanks to its simplicity, DE also exhibits very good performance when compared to state-of-the art search methods. Furthermore the number of control parameters in DE remains surprisingly small for an evolutionary scheme (Cr, F and N) and a large amount of work has been proposed to select the best equation for the construction of the variant vector.

As explained in [27], the space complexity of DE is low when compared to the most competitive optimizers like CMA-ES. Although CMA-ES remains very competitive over problems up to 100 variables, it is difficult to extend it to higher dimensional problems due mainly to the cost of computing and updating the covariance matrix.

Evolving programs which are typically a mix of discrete and continuous features (e.g. regression problems) is an interesting challenge for these heuristics, since they were not designed for this kind of task.

3. Linear programs with continuous representation

We propose to use Differential Evolution and CMA Evolution Strategy to evolve float vectors, which will be mapped to sequences of imperative instructions in order to form linear programs, similar to the LGP scheme from [6]. For the sake of simplicity, these schemes are respectively denoted:

- LDEP, for Linear Differential Evolutionary Programming, when DE is used as the evolutionary engine;

- CMA-LEP, for Covariance Matrix Adaption Linear Evolutionary Programming, when the evolutionary engine is CMA-ES.

First we recall the basis of linear programs encoding, and execution, and then we explain the mapping process from continuous representation to imperative instructions. We conclude with some remarks on the integration of this representation and mapping with the DE and CMA-ES engines.

3.1. Linear sequence of instructions

In LGP a program is composed of a linear sequence of imperative instructions (see [6] for more details). Each instruction is typically 3-register instruction. That means that every instruction includes an operation on two operand registers, one of them could be holding a constant value, and then assigns the result to a third register:

$$r_i = \begin{cases} r_j \text{ op } (r_k|c_k) \\ (r_j|c_j) \text{ op } r_k \end{cases}$$

where *op* is the operation symbol, r_i is the destination register, r_j, r_k are calculation registers (or operands) and c_j, c_k are constant registers (only one constant register is allowed per instruction).

On the implementation level of standard LGP, each imperative instruction is represented by a list of four integer values where the first value gives the operator and the three next values represent the three register indices. For instance, an instruction like $r_i = r_j \times r_k$ is stored as a quadruple $< \times, i, j, k >$, which in turn is coded as four indices indicating respectively the operation number in the set of possible operations, and 3 indices in the set of possible registers (and/or constant registers). Of course, even if the programming language is basically a 3-register instruction language, it is possible to ignore the last index in order to include 2-register instructions like $r_i = \sin(r_k)$.

Instructions are executed by a virtual machine using floating-point value registers to perform the computations required by the program. The problem inputs are stored in a set of registers. Typically the program output is read in a dedicated register (usually named r_0) at the end of the program execution. These input and output registers are read-write and can serve for intermediate calculations. Usually, additional read-only registers store user defined constants, and extra read-write registers can be added to allow for complex calculations. The use of several calculation registers makes possible a number of different program paths, as explained in [6] and in [28].

3.2. Mapping a float vector to a linear program

Here we explain how a float vector (i.e. an individual of the population), evolved by either DE or CMA-ES, is translated to a linear program in the LGP form.

As explained in the previous section, we need 4 indices to code for the operation number and 3 registers involved. Thus we split the float vector individual into consecutive sequences of 4 floats $< v_1, v_2, v_3, v_4 >$, where v_1 encodes the operator number, and v_2, v_3, v_4 encode the

destination and operand registers. In order to convert a float v_i into an integer index, we apply one of the following computations:

- Conversion of the operator index:

$$\#\text{operator} = \lfloor (v_i - \lfloor v_i \rfloor) \times n_{\text{operators}} \rfloor \tag{6}$$

where $n_{\text{operators}}$ denotes the number of possible operators.

- Conversion of the destination register index:

$$\#\text{register} = \lfloor (v_i - \lfloor v_i \rfloor) \times n_{\text{registers}} \rfloor \tag{7}$$

where $n_{\text{registers}}$ denotes the number of possible read-write registers.

- The conversion of an operand register depends whether it is a constant or a read-write register. This is controlled by a user defined probability of selecting constant registers, denoted P_C in the following equation:

$$\begin{cases} \#\text{ read-write register} = \lfloor (\frac{v_i - \lfloor v_i \rfloor - P_c}{1 - P_c}) \times n_{\text{registers}} \rfloor & \text{if } (v_i - \lfloor v_i \rfloor) > P_C \\ \#\text{ constant register} \quad = \lfloor v_i \rfloor \bmod n_{\text{constants}} & \text{otherwise} \end{cases} \tag{8}$$

where $n_{\text{registers}}$ denotes the number of possible read-write registers, and $n_{\text{constants}}$ denotes the number of possible constant registers.

Example of a mapping process

Let us suppose we work with 6 read-write registers (r_0 to r_5), 50 constant registers, and the 4 following operators:

$$0 : + \quad 1 : - \quad 2 : \times \quad 3 : \div$$

We set up the constant register probability to $P_C = 0.1$ and we consider the following vector composed of 8 floats, to be translated into 2 imperative instructions ($< v_1, v_2, v_3, v_4 >$ and $< v_5, v_6, v_7, v_8 >$):

v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8
0.17	2.41	1.86	3.07	0.65	1.15	1.25	4.28

Value v_1 denotes one operator among the four to choose from. Applying Eq. 6, we get $\#\text{operator} = \lfloor (0.17 - \lfloor 0.17 \rfloor) \times 4 \rfloor = 0$, meaning that the first operator is $+$.

The second value $v_2 = 2.41$ is turned into a destination register. According to Eq. 7, we obtain $\#\text{register} = \lfloor (2.41 - \lfloor 2.41 \rfloor) \times 6 \rfloor = \lfloor 2.46 \rfloor = 2$, meaning that the destination register is r_2.

The next value $v_3 = 1.86$ gives an operand register. According to Eq. 8, it is a read-write register since $(1.86 - \lfloor 1.86 \rfloor) = 0.86 > P_C$. Thus the first operand register is: $\#\text{register} = \lfloor ((1.86 - \lfloor 1.86 \rfloor - 0.1)/0.9) \times 6 \rfloor = \lfloor 5.07 \rfloor = 5$, meaning read-write register r_5.

The last of the four first operands is decoded as a constant register since $(3.07 - \lfloor 3.07 \rfloor) = 0.07 \leq P_C$. The index is $\lfloor 3.07 \rfloor \bmod 50 = 3$, meaning constant register c_3.

So the 4 first values of the genotype are translated as:

$$r_2 = r_5 + c_3$$

The mapping process continues with the four next values, until we are left with the following program:

$$r_2 = r_5 + c_3$$
$$r_0 = r_1 \times r_1$$

3.3. Algorithm

To finalize the LDEP and CMA-LEP algorithms, the basic idea is to simply plug the float vector to program translation and the virtual machine program evaluation into the DE and CMA-ES schemes. However some technical points need to be taken into account to allow this integration and they are detailed below.

Initialization

We have to decide about the length of the individuals (float vectors) since we usually cannot extract this feature from the problem. This length will determine the maximum number of instructions allowed in the evolved programs.

Moreover we need to fix a range of possible initial values to randomly generate the components of the initial population $\{X_i\}_{1 \leq i \leq N}$, as typical in DE.

Constant registers are initialized at the beginning of the run, and then are only accessed in read-only mode. This means that our set of constants remains fixed and does not evolve during the run. The number and value range of constant registers are user defined, and the additional parameter P_C must be set to determine the probability of using a constant register in an expression, as explained above in Eq. 8.

Main algorithm iteration

For LDEP, we tried two variants of the iteration loop described in Section 2.2: either generational replacement of individuals as in the original Storn and Price paper [11], or steady state replacement, which seems to be used in [17]. In the generational case, newly created individuals are stored in a temporary set, and once the generation is completed, they replace their respective parent if their fitness is better. In the steady state scheme, each new individual is immediately compared with its parent and replaces it if its fitness is better, and thus it can be used in remaining crossovers for the current generation. Using the steady state variant seems to accelerate convergence, see Section 4.

During the iteration loop of either LDEP or CMA-LEP, the vector solutions are decoded using equations 6, 7 and 8. The resulting linear programs are then evaluated on a set of fitness cases (training examples). The fitness value is then returned to the evolution engine that continues the evolution process.

Heuristic	Problem	Pop.	Ind. size	# eval.	extra params
LDEP	Regressions	20	128	5E4	$F = 0.5$, $CR = 0.1$
	Ant	30	50	2E5	$F = 0.5$, $CR = 0.1$
CMA-LEP	Regressions	20	128	5E4	$\sigma \in \{1, 10\}$, $\lambda \in \{10, 100\}$,
	Ant	30	50	2E5	$\sigma \in \{1, 10\}$, $\lambda \in \{10, 100\}$
GP	Regressions	1000	N.A.	5E4	Elitism, max Depth=11, 80% Xover, 10% Mut, 10% Copy
	Ant	4000	N.A.	2E5	Elitism, max Depth=11, 80% Xover, 10% Mut, 10% Copy

Table 1. Main experimental parameters

4. Experiments

We use the same benchmark problems as in [17] (4 symbolic regressions and the Santa Fe artificial ant), and we also add two regression problems that include float constants.

Before listing our experimental parameters in Table 1, we explain some of our implementation choices:

- We run all standard GP experiments using the well-known ECJ library[1].

- For GP we use a maximum generation number of 50 and set the population size in accordance with the maximum number of evaluations. We keep the best (elite) individual from one generation to the next.

- We use the publicly available C language version of CMA-ES[2], with overall default parameters.

- For TreeDE we take the results as they are reported in [17]:
  - For regression, 1500 iterations on a population of 20 vectors were allowed, and runs were done for every tree depth in the range $\{1, \dots, 10\}$. It thus amounts to a total of $300,000$ evaluations. Among these runs, reference [17] reported only those associated to the tree depth that obtained the best result (which may well imply a favorable bias, in our opinion). As we could not apply this notion of best tree depth in our heuristic, we decided as a trade-off to allow $50,000$ evaluations for regression with both LDEP, CMA-LEP and GP.
  - For the Santa Fe Trail artificial ant problem, the same calculation gives a total of $450,000$ evaluations for TreeDE. We decided for a trade-off of $200,000$ evaluations for LDEP, CMA-LEP and GP.

4.1. Symbolic regression problems

The aim of these 1-dimensional symbolic regression problems is to find some symbolic mathematical expression (or program) that best approximates a target function that is known only by a set of examples, or fitness cases, $(x_k, f(x_k))$. In our case, 20 values x_k are chosen evenly distributed in the range $[-1.0, +1.0]$. The evaluation of programs (or *fitness*

---

[1] http://cs.gmu.edu/~eclab/projects/ecj/
[2] http://www.lri.fr/~hansen/cmaes_inmatlab.html

computation) is done according to the classic Koza's book [1], that is computing the sum of deviations by looping over all fitness cases:

$$fitness = \sum_{1 \leq k \leq N} |f(x_k) - P(x_k)|$$

where $P(x_k)$ is the value computed by the evolved program P on input x_k, f is the benchmark function and $N = 20$ is the number of (input, output) fitness cases. A *hit solution* means that the deviation is less than 10^{-4} on each fitness case.

The first 4 test functions are from [17]:

$$f_1(x) = x^3 + x^2 + x$$
$$f_2(x) = x^4 + x^3 + x^2 + x$$
$$f_3(x) = x^5 + x^4 + x^3 + x^2 + x$$
$$f_4(x) = x^5 - 2x^3 + x$$

While TreeDE benchmarks were run without constants in [17], we strongly believe that it is interesting to use benchmark problems that are expressed as functions both with and without float constants, in order to assess the impact of constant management by the heuristics. Moreover in the general case, especially on real world problems, one cannot know in advance whether or not float constants may be useful. For this reason we add two benchmarks:

$$f_5(x) = \pi \quad \text{(a constant function)}$$
$$f_6(x) = \tfrac{x}{\pi} + \tfrac{x^2}{\pi^2} + 2x\pi$$

The set of operators is $\{+, -, \times, \div\}$ with $\div$ being the protected division (*i.e.* $a \div b = a/b$ if $b \neq 0$ else $a \div b = 0$ if $b = 0$).

For LDEP and CMA-LEP, 6 read-write registers are used for calculation (from r_0 to r_5), with r_0 being the output register. For each fitness case $(x_k, f(x_k))$ that is submitted to the evolved program inside the evaluation loop, all 6 calculation registers are initialized with the same input value x_k. This standard LGP practice provides redundancy of the input value and thus more robustness to the run.

Runs without constants

In the first set of experiments, programs are evolved without constants. This unrealistic setting is proposed here only to allow a comparison of DE-based scheme, confronting LDEP versus Veenhuis's TreeDE, and excluding CMA-LEP. Results are reported in table 2, all three heuristics exhibit close results on the f_1, f_2, f_3, f_4 problems, with GP providing the overall most precise approximation, and LDEP needing the largest number of evaluations (notwithstanding the possible bias in the TreeDE figures, as mentioned at the beginning of Section 4). Note that the steady state variant of LDEP converges faster than the generational, as shown by the average number of evaluations for perfect solutions. It seems safe to conclude that this increased speed of convergence is the explanation for the better result of the steady

	generational LDEP			steady state LDEP			TreeDE		
Problem	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
f_1	0.0	100%	4297	0.0	100%	2632	0.0	100%	1040
f_2	0.0	100%	12033	0.0	100%	7672	0.0	100%	3000
f_3	0.28	72.5%	21268	0.08	85%	21826	0.027	98%	8440
f_4	0.20	62.5%	33233	0.13	75%	26998	0.165	68%	14600

	standard GP		
Problem	Fit.	% hits	Eval.
f_1	0.0	100%	1815
f_2	0.0	100%	2865
f_3	0.03	97%	6390
f_4	0.01	80%	10845

For each heuristic, over 40 independent runs, the column Fit. gives the average of the best fitness (taken from [17] for TreeDE), then we have the percentage of run reaching a hit solution, then the average number of evaluations to produce the first hit solution (if ever produced).

Table 2. Results for symbolic regression problems without constants.

state variant versus generational, in a limited number of evaluations. This steady state faster convergence may also benefit to TreeDE.

Runs with constants

In the second set of experiments, presented in Table 3, heuristics are allowed to evolve programs with constants, thus ruling out TreeDE from the comparison. All problems from f_1 to f_6 are tested, which means that heuristics manage float constants even on the first 4 problems when they are not needed. This simulates the frequent absence of background knowledge on a new problem and this also tests the robustness of heuristics.

- For LDEP and CMA-LEP, we add 50 constant registers, with a probability of occurrence $P_C = 0.05$, and initial values in the range $[-1.0, +1.0]$.
- For GP, we define 4 redundant input terminals reading the same input value x_k for each fitness case (x_k, y_k), against only one ephemeral random constant (ERC) terminal, that draws new random value instances when needed, in the range $[-1.0, +1.0]$. Thus the probability to generate a constant, e.g. during program initialization or in a subtree mutation, is much lower than the usual 50% when having only one x terminal. This is closer to the LDEP setting and it significantly improves the GP results.

In Table 3, we again observe that the steady state variant of LDEP is better than the generational. For its best version LDEP is comparable to GP, with a slightly higher hit ratio and better average fitness (except on f_6), with more evaluations on average. For CMA-LEP, two values for $\sigma \in \{1, 10\}$ and two values for $\lambda \in \{10, 100\}$ were tried with no significant

differences. In contrast with the other methods, CMA-LEP results are an order of magnitude worse. Tuning the CMA-ES engine to tackle the problem as separable did not improve the results. We think this behavior may result from the high dimensionality of the problem (N=128), that certainly disrupts the process of modeling an ideal mean solution from a comparatively tiny set of search points. This is combined to the lack of elitism, inherent to the CMA-ES method, thus when it comes to generate new test points, the heuristic is left solely with a probably imperfect model.

	generational LDEP			steady state LDEP		
Problem	Fit.	%hits	Eval.	Fit.	%hits	Eval.
f_1	0.0	100%	7957	0.0	100%	7355
f_2	0.02	95%	16282	0.0	100%	14815
f_3	0.4	52.5%	24767	0.0	100%	10527
f_4	0.36	42.5%	21941	0.278	45%	26501
f_5	0.13	2.5%	34820	0.06	15%	29200
f_6	0.59	0%	NA	0.63	0%	NA

	standard GP			CMA-LEP		
Problem	Fit.	%hits	Eval.	Fit.	%hits	Eval.
f_1	0.002	98%	3435	0.03	20%	6500
f_2	0.0	100%	4005	2.76	0%	NA
f_3	0.02	93%	7695	5.33	0%	NA
f_4	0.33	23%	24465	2.06	6%	10900
f_5	0.07	0%	NA	13.35	0%	NA
f_6	0.21	0%	NA	5.12	0%	NA

For each heuristic, over 40 independent runs, the column Fit. gives the average of the best fitness, then we have the percentage of run reaching a hit solution, then the average number of evaluations to produce the first hit solution (if ever produced or else NA if no run produced a hit solution).

Table 3. Results for symbolic regression problems with constants.

Overall, these results confirm that DE is an interesting heuristic, even when the continuous representation hides a combinatorial type problem, and thus the heuristic is used outside its original field. The LDEP mix of linear programs and constant management appears competitive with the standard GP approach.

4.2. Santa Fe ant trail

The Santa Fe ant trail is a famous problem in the GP field. The objective is to find a computer program that is able to control an artificial ant so that it can find all 89 pieces of food located on a discontinuous trail within a specified number of time steps. The trail is drawn on a discrete 32×32 toroidal grid illustrated in Figure 1. The problem is known to be rather hard, at least for standard GP (see [29]), with many local and global optima, which may explain why the size of the TreeDE population was increased to $N = 30$ in [17].

Only a few actions are allowed to the ant. It can turn left, right, move one square forward and it may also look into the next square in the direction it is facing, in order to determine if
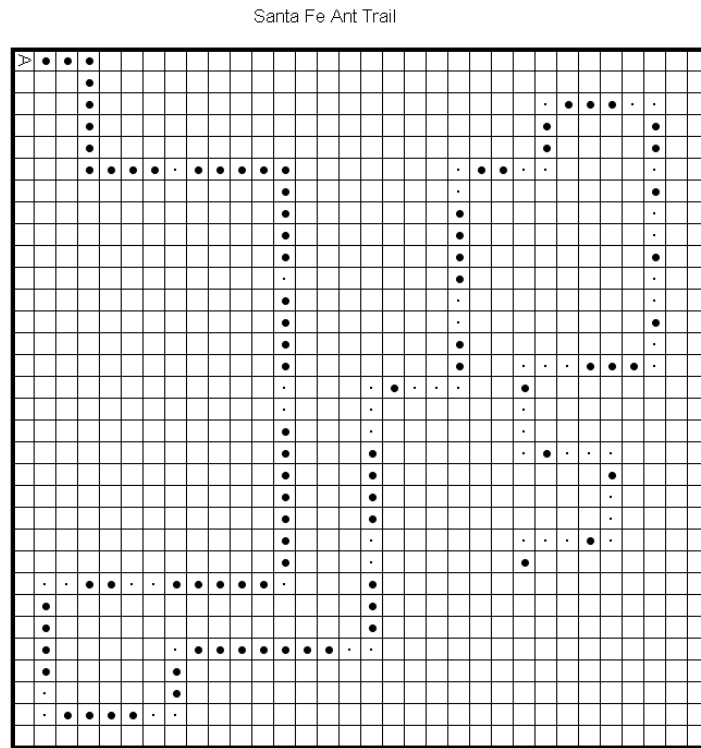
Figure 1. Illustration of the Santa Fe Trail (the ant starts in the upper left corner, heading to the east, large dots are food pellets, and small dots are empty cells on the ideal path).

it contains a piece of food or not. Turns and moves cost one time step, and a maximum time steps threshold is set at start (typical values are either 400 or 600 time steps). If the program finishes before the exhaustion of the time steps, it is restarted (which amounts to iterating the whole program).

We do not need mathematical operators nor registers, only the following instructions are available:

- MOVE: moves the ant forward one step (grid cell) in the direction the ant is facing, retrieving an eventual food pellet in the cell of arrival;
- LEFT: turns on place 45 degrees anti-clockwise;
- RIGHT: turns on place 45 degrees clockwise;
- IF-FOOD-AHEAD: conditional statement that executes the next instruction or group of instructions if a food pellet is located on the neighboring cell in front of the ant, else the next instruction or group is skipped;
- PROGN2: groups the two instructions that follow in the program vector, notably allowing IF-FOOD-AHEAD to perform several instructions if the condition is true (the PROGN2 operator does not affect *per se* the ant position and direction);
- PROGN3: same as the previous operator, but groups the three following instructions.
- Each MOVE, RIGHT and LEFT instruction requires one time step.

# steps	generational LDEP			steady state LDEP			standard GP		
	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
400	11.55	12.5%	101008	14.65	7.5%	46320	8.87	37%	126100
600	0.3	82.5%	88483	1.275	70%	44260	1.175	87%	63300

# steps	CMA-LEP			TreeDE		
	Fit.	% hits	Eval.	Fit.	% hits	Eval.
400	37.45	0%	NA	17.3	3%	24450
600	27.05	0%	NA	1.14	66%	22530

The 1st column is the number of allowed time steps, then for each heuristic, over 40 independent runs, we give the average of the best fitness (taken from [17] for TreeDE), then the percentage of run reaching a hit solution (solution that found all 89 food pellets), then the average number of evaluations to produce the first hit solution (if ever produced or else NA if no run produced a hit solution).

Table 4. Santa Fe Trail artificial ant problem.

Programs are again vectors of floating point values. Each instruction is represented as a single value which is decoded in the same way as operators are in the regression problems, that is using Eq. 6. Instruction are decoded sequentially, and the virtual machine is refined to handle jumps over an instruction or group of instructions, so that it can deal with IF-FOOD-AHEAD instructions. Incomplete programs may be encountered, for example if a PROGN2 is decoded for the last value of a program vector. In this case the incomplete instruction is simply dropped and we consider that the program has reached normal termination (and the program is iterated if there are remaining time steps).

The Santa Fe trail being composed of 89 pieces of food, the fitness function is the remaining food (89 minus the number of food pellets taken by the ant before it runs out of time). So, the lower the fitness, the better the program, a hit solution being a program with fitness 0, i.e. a program able to pick up all the food on the grid.

Results are summed-up in Table 4. Contrary to the regression experiment, the generational variant of LDEP is now better than the steady state. We think this behavior is explained by the hardness of the problem: more exploration is needed, and it pays no more to accelerate convergence.

GP gives the best results for 400 time steps, but it is LDEP that provides the best average fitness for 600 steps, at the cost of a greater number of evaluations, meaning LDEP is better at exploiting the available amount of computing time. LDEP is also better than TreeDE on both steps limits. For CMA-LEP, two values for $\sigma \in \{1, 10\}$ and two values for $\lambda \in \{10, 100\}$ were again tried, the best setting being $\sigma = 10$ and $\lambda = 100$ (whose results are reported here). CMA-LEP performed really poorly, and its first results were so bad that it motivated us to try this rather high initial variance level ($\sigma = 10$), which brought a sensible but insufficient improvement. We think that the lack of elitism is, here again, a probable cause of CMA-ES bad behavior, on a very chaotic fitness landscape with many neutral zones (many programs exhibit the same fitness).

```
If food{ Move } else {
  Progn3{
    Progn3{
      Progn3{ Right ;
        If food{ Right } else { Left } ;
        Progn2{ Left ;
          If food{ Progn2{ Move ; Move } }
             else { Right } } } ; // end Progn3
      Move ;
      Right } ; // end Progn3
    If food{ Move } else { Left } ; //end Progn3
  Move } }
```

Table 5. Example of a perfect solution for the Ant Problem found by LDEP in 400 time steps

Here again LDEP appears as a possible competitor to GP. Table 5 shows an example of a perfect solution found by LDEP for 400 time steps.

4.3. Evolving a stack

As the LDEP continuous approach for evolving programs achieved interesting results on the previous GP benchmarks, we decided to move forward and to test whether or not we were able to evolve a more complex data structure: a stack. Langdon successfully showed in [30] that GP was able to evolve not only a stack with its minimal set of operations (push, pop, makenull), but also two other optional operations (top, empty), which are considered to be inessential. We followed this setting, and the five operations to evolve are described in Table 6.

Operation	Comment
makenull	initialize stack
empty	is stack empty?
top	return top of stack
pop	return top of stack and remove it
push(x)	store x on top of stack

Table 6. The five operations to evolve

This is in our opinion a more complex problem than the previous ones, since the correctness of each trial solution is established using only the values returned by the stack operations and only pop, top and empty return values.

Choice of primitives

As explained in [30], the set of primitives that was chosen to solve this problem is a set that a human programmer might use. The set basically consists in functions that are able to read and write in an indexed memory, functions that can modify the stack pointer and functions that can perform simple arithmetic operations. The terminal set consists in zero-arity functions (stack pointer increment and decrement) and some constants.

The following set was available for LDEP:

- arg1, the value to be pushed on to the stack (read-only argument)

- aux, the current value of the stack pointer

- arithmetic operators $+$ and $-$

- constants 0, 1 and MAX (maximum depth of the stack, set to 10)

- indexed memory functions read and write. The write function is a two argument function arg1 and arg2. It evaluates the two arguments and sets the indexed memory pointed by arg1 to arg2 (i.e. stack[arg1] = arg2). It returns the original value of aux.

- functions to modify the stack pointer: inc_aux to increment the stack pointer, dec_aux to decrement it, write_aux to set the stack pointer to its argument and returns the original value of aux.

Algorithm and fitness function

We used a slightly modified version of our continuous scheme as the stack problem requires the simultaneous evolution of the five operations (`push`, `pop`, `makenull`, `top`, `empty`). An individual is composed of 5 vectors, one for each operation. Mutation and crossover are only performed with vectors of the same type (i.e. vectors evolving the `push` operation for example).

Programs are coded in prefix notation, that means that an operation like (`arg1 + MAX`) was coded as `+ arg1 MAX`. We did not impose any restrictions on each program's size except that each vector has a maximum length of 100 (this is several times more than sufficient to code any of the five operations needed to manipulate the stack).

In his original work, Langdon chose to use a population of size 1,000 individuals with 101 generations. In the DE case, it is known from experience that using large populations is usually inadequate. So, we fixed a population of 10 individuals with 10,000 generations for LDEP, amounting to about the same number of evaluations.

We used the same fitness function that was defined by Langdon. It consists in 4 test sequences, each one being composed of 40 stack operations. As explained in the previous section, the `makenull` and `push` operations do not return any value, they can only be tested indirectly by seeing if the other operations perform correctly.

Results

In Langdon's experiments, 4 runs out of 60 produced successful individuals (i.e. a fully operational stack). We obtained the same success ratio with LDEP: 4 out of the first 60 runs yielded perfect solutions. Extending the number of runs, LDEP evolved 6 perfect solutions out of 100 runs, providing a convincing proof of feasibility. Regarding CMA-LEP, results are less convincing, since only one run out of 100 was able to successfully evolve a stack.

An example of successful solution is given in table 7 with the raw evolved code and a simplified version where redundant code is removed.

Operation	Evolved operation	Simplified operation
push	write(1 ,write(dec_aux ,arg1))	stack[aux] = arg1 aux = aux - 1

pop	write(aux ,((aux + (dec_aux + inc_aux)) + read(inc_aux)))	aux = aux + 1
		tmp = stack[aux];
		stack[aux] = tmp + aux;
		return tmp

top	read(aux)	return sp[aux]
empty	aux	if (aux > 0) return true else return false
makenull	write((MAX - (0 + write_aux(1))),MAX)	aux = 1

Table 7. Example of an evolved push-down stack

5. Conclusions

This chapter explores evolutionary continuous optimization engines applied to automatic programming. We work with Differential Evolution (LDEP) and CMA-Evolution Strategy (CMA-LEP), and we translate the continuous representation of individuals into linear imperative programs. Unlike the TreeDE heuristic, our schemes include the use of float constants (e.g. in symbolic regression problems).

Comparisons with GP confirm that LDEP is a promising optimization engine for automatic programming. In the most realistic case of regression problems, when using constants, steady state LDEP slightly outperforms standard GP on 5 over 6 problems. On the artificial ant problem, the leading heuristic depends on the number of steps: for the 400 steps version GP is the clear winner, while for 600 steps generational LDEP yields the best average fitness. LDEP improves on the TreeDE results for both versions of the ant problem, without needing a fine-tuning of the solutions tree-depth.

For both regression and artificial ant, CMA-LEP performs poorly with the same representation of solutions than LDEP. This can be deemed not really surprising since the problems we tackle are clearly outside the domain targeted by the CMA-ES heuristic that drives evolution. Nonetheless it is also the case for DE, which still produces interesting solutions, thus this points to a fundamental difference in behavior between these two heuristics. We suspect that CMA-ES lack of elitism may be an explanation. It also points to a possible inherent robustness of the DE method, on fitness landscapes that are possibly more chaotic than the usual continuous benchmarks.

The promising results of LDEP on the artificial ant and on the stack problems are a great incentive to deepen the exploration of this heuristic. Many interesting questions remain open. In the beginnings of GP, experiments showed that the probability of crossover had to be set differently for internal and terminal nodes: is it possible to improve LDEP in similar ways? It is to be noticed that in our experiments the individual vector components take their values in the range $(-\infty, +\infty)$, since it is required by the standard CMA-ES algorithm. It could be interesting to experiment DE-based algorithms with a reduced range of vector component values, for example $[-1.0, 1.0]$, that would require to modify the mapping of constant indices.

Author details

Cyril Fonlupt, Denis Robilliard, Virginie Marion-Poty
LISIC, ULCO, Univ Lille Nord de France, France

6. References

[1] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza. *Genetic Programming IV Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[2] Sameer H. Al-Sakram, John R. Koza, and Lee W. Jones. Automated re-invention of a previously patented optical lens system using genetic programming. In *[31]*, pages 25–37, 2005.

[3] Kun-Hong Liu and Chun-Gui Xu. A genetic programming-based approach to the classification of multiclass microarray datasets. *Bioinformatics*, 25(3):331–337, 2009.

[4] Adrian Gepp and Phil Stocks. A review of procedures to evolve quantum algorithms. *Genetic Programming and Evolvable Machines*, 10(2):181–228, 2009.

[5] M. Szymanski, H. Worn, and J. Fischer. Investigating the effect of pruning on the diversity and fitness of robot controllers based on MDL2E during genetic programming. In *[33]*, pages 2780–2787, 2009.

[6] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, 2007.

[7] H.A. Abbass, NX Hoai, and R.I. Mckay. AntTAG: A new method to compose computer programs using colonies of ants. In *The IEEE Congress on Evolutionary Computation*, pages 1654–1659, 2002.

[8] Y. Shan, H. Abbass, RI McKay, and D. Essam. AntTAG: a further study. In *Proceedings of the Sixth Australia-Japan Joint Workshop on Intelligent and Evolutionary Systems, Australian National University, Canberra, Australia*, volume 30, 2002.

[9] R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.

[10] Evandro Nunes Regolin and Aurora Trindad Ramirez Pozo. Bayesian automatic programming. In *[31]*, pages 38–49, 2005.

[11] Rainer Storn and Kenneth Price. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization*, 11(4):341–359, 1997.

[12] Nikolaus Hansen and Andreas Ostermeier. Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation. In *International Conference on Evolutionary Computation*, pages 312–317, 1996.

[13] Michael O'Neill and Anthony Brabazon. Grammatical differential evolution. In *International Conference on Artificial Intelligence (ICAI'06)*, pages 231–236, Las Vegas, Nevada, USA, 2006.

[14] Michael O'Neill and Conor Ryan. Grammatical evolution. *ieeetec*, 5(4):349–357, aug 2001.

[15] James E. Murphy, Michael O'Neill, and Hamish Carr. Exploring grammatical evolution for horse gait optimisation. In *[32]*, pages 183–194, 2009.

[16] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Press, 2003.

[17] Christian B. Veenhuis. Tree based differential evolution. In *[32]*, pages 208–219, 2009.

[18] Alberto Moraglio and Sara Silva. Geometric differential evolution on the space of genetic programs. In Anna Isabel Esparcia-Alcázar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, editors, *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, volume 6021 of *LNCS*, pages 171–183, Istanbul, 7-9 April 2010. Springer. Best paper.

[19] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. In *Proceedings of the 12th International Conference on Machine Learning*, pages 38–46, Morgan Kaufmann Publishers, 1995.

[20] A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1769 – 1776 Vol. 2, September 2005.

[21] K. Price. Differential evolution: a fast and simple numerical optimizer. In *Biennial conference of the North American Fuzzy Information Processing Society*, pages 524–527, 1996.

[22] S. Rahnamayan and P. Dieras. Efficiency competition on n-queen problem: DE vs. CMA-ES. In *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*, pages 000033 –000036, May 2008.

[23] Carmen G. Moles, Pedro Mendes, and Julio R. Banga. Parameter Estimation in Biochemical Pathways: A Comparison of Global Optimization Methods. *Genome Research*, 13(11):2467–2474, 2003.

[24] N. Hansen and S. Kern. Evaluating the CMA evolution strategy on multimodal test functions. In X. Yao et al., editors, *Parallel Problem Solving from Nature PPSN VIII*, volume 3242 of *LNCS*, pages 282–291. Springer, 2004.

[25] A. Auger and N. Hansen. A restart CMA evolution strategy with increasing population size. In *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, volume 2, pages 1769 – 1776 Vol. 2, September 2005.

[26] Nikolaus Hansen and Stefan Kern. Evaluating the CMA evolution strategy on multimodal functions. In Springer-Verlag, editor, *Parallel Problem Solving from Nature, PPSN VIII*, volume 3242 of *lncs*, pages 282–291, 2004.

[27] Swagatam Das and Ponnuthurai Nagaratnam. Differential evolution: A survey of the state-of-the-art. *IEEE Transactions on Computers*, pages 4–31, feb 2011.

[28] Garnett Wilson and Wolfgang Banzhaf. A comparison of cartesian genetic programming and linear genetic programming. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Esparcia Alcázar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Genetic Programming*, volume 4971 of *Lecture Notes in Computer Science*, pages 182–193. Springer Berlin / Heidelberg, 2008.

[29] W. B. Langdon and R. Poli. Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science, January 1998. Presented at GP-98.

[30] William B. Langdon. *Genetic Programming and Data Structures = Automatic Programming !* Kluwer Academic Publishers, 1998.

[31] Maarten Keijzer, Andrea Tettamanzi, Pierre Collet, Jano van Hemert, and Marco Tomassini, editors. *8th European Conference, EuroGP 2005*, volume 3447 of *LNCS*, Lausanne, Switzerland, mar 2005.

[32] Leonardo Vanneschi, Steven Gustafson, Alberto Moraglio, Ivanoe De Falco, and Marc Ebner, editors. *12th European Conference, EuroGP 2009*, volume 5481 of *LNCS*, Tubingen, Germany, apr 2009.

[33] *Congress on Evolutionary Computation*, Trondheim, Norway, may 2009.