Linear Imperative Programming with Differential Evolution

Cyril Fonlupt Denis Robilliard Virginie Marion-Poty
LISIC — ULCO
Univ Lille Nord de France
BP 719, F-62228 Calais Cedex, France
Email: fonlupt,robilliard,poty@lisic.univ-littoral.fr

***Abstract*—Differential Evolution (DE) is an evolutionary approach for optimizing non-linear continuous space functions. This method is known to be robust and easy to use. DE manipulates vectors of floats that are improved over generations by mating with best and random individuals. Recently, DE was successfully applied to the automatic generation of programs by mapping real-valued vectors to full programs trees - Tree Based Differential Evolution (TreeDE). In this paper, we propose to use DE as a method to directly generate linear sequences of imperative instructions, which we call Linear Differential Evolutionary Programming (LDEP). Unlike TreeDE, LDEP incorporates constant management for regression problems and lessens the constraints on the architecture of solutions since the user is no more required to determine the tree depth of solutions. Comparisons with standard Genetic Programming and with the CMA-ES algorithm showed that DE-based approach are well suited to automatic programming, being notably more robust than CMA-ES in this particular context.**

I. 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 [1], [2]. GP is used in fields as different as bio-informatics [3], quantum computing [4] or robotics [5].

The most widely used scheme in GP is the tree paradigm developed by Koza where each evolved program is coded as a Lisp-like tree. Many other paradigms were devised these last years to automatically evolve programs. For instance, Linear Genetic Programming [6] (LGP) 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).

A few other evolutionary schemes for automatic programming were proposed these last years 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 called Differential Evolution (DE) [11] for continuous optimization. DE is a stochastic search method, it can be described as an Evolution Strategy variant that uses information from the current vector population to determine the perturbation brought to solution (this can be seen as determining the direction of the search).

To our knowledge O'Neill and Brabazon were the first to use DE to evolve programs with the use of the well known grammatical evolution engine [12]. A diverse selection of benchmarks from the literature on genetic programming were tackled with four different flavors of DE. 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. Recently, Veenhuis [13] also introduced a successful application of DE for automatic programming, mapping a continuous genotype to trees: Tree based Differential Evolution (TreeDE).

TreeDE improves somewhat on the performance of grammatical differential evolution, but it requires an additional low-level parameter, the number of tree levels, that has to be set beforehand, and it does not provide constants.

In this paper, we propose to use a Differential Evolution engine in order to evolve not program trees but directly linear sequences of imperative instructions, *à la* LGP. We can thus implement real-valued constants management inspired by the LGP literature. The tree-depth parameter from TreeDE is now replaced by the maximum length of the programs to be evolved: this is a lesser constraint on the architecture of solutions and it also has the benefit of avoiding the well known bloat problem (uncontrolled increase in solution size) that plagues standard GP.

Among optimization methods, DE was often compared(e.g. [14], [15]) to the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [16]. We use the same representation for linear programs with CMA-ES and compare its results to those obtained from DE. For all our experiments, we observe that the DE engine is superior to CMA-ES.

This paper is organized in the following way. Section II presents the main DE concepts. In section III, our scheme Linear Differential Evolutionary Programming (LDEP) is introduced while section IV gives some experimental results and some comparisons with TreeDE, standard GP and CMA-ES. Future works and conclusions are discussed in section V.

II. Covariance Matrix Adaptation Evolution Strategy and Differential Evolution

This section only introduces the main CMA-ES and DE concepts, the interested reader being respectively referred to [16] and [11] for a full presentation of these two heuristics. We first draw a quick overview of CMA-ES, before covering DE in more detail, as it is the base of our algorithm.

A. Covariance Matrix Adaptation Evolution Strategy

CMA-ES was proposed by Hansen and Ostermeier in 1996 [16]. Since that, an abundant literature has brought refinements to the algorithm (e.g. [17]), and has shown its interest as a continuous optimization method.

The basic CMA-ES idea is sampling (hopefully new) search points using a normal distribution that is centered on an updated model of the ideal solution, which can be seen as a weighted mean or recombination of a best subset of current search points. The distribution is also shaped by the covariance matrix of those best solutions sampled in the current iteration. This fundamental scheme was refined mainly to extract more information from the history of the optimization run, introducing the so-called cumulation path whose idea is akin to the momentum of artificial neural networks, and also to allocate 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

CMA-ES does not use elitism (i.e. the best search point is not preserved from one iteration to the next).

B. Differential Evolution

Differential Evolution [11] is a population-based search algorithm. The algorithm 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 nonlinear and non-differentiable continuous space functions. As most evolutionary schemes, DE can be viewed as a stochastic directed search method but instead of generating random offspring from an evolved probability distribution (PBIL [18] or CMA-ES), or randomly mating two individuals (crossover in Genetic Algorithms), 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. The newly created vector is inserted into the population only if its fitness is better than the current vector.

Principles: DE is a parallel search method working on a set of N $d-$dimensional vector solutions X_i, f being the fitness function.

$$X_i = (x_{i1}, x_{i2}, \dots, x_{id}) \quad i = 1, 2, \dots, N \tag{1}$$

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 with a Gaussian law 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.

If the trial vector performs better (better fitness value), a selection process is triggered to determine whether or not the trial solution replaces the vector X_j in the population.

Mutation: In the initial DE [11], a so-called variant vector $V_j = (v_{j1}, v_{j2}, \dots, v_{jd})$ is generated for each vector $X_j(t)$ of the population, according to Equation 2:

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

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

Many variants were proposed for Equation 2, including the use of three or more individuals. According to [19], [13], the mutation method that leads to the best results is the method DE/best/2/bin, with:

$$V_j(t+1) = X_{\text{best}}(t) + F \times (X_{r_1}(t) + X_{r_2}(t) - X_{r_3}(t) - X_{r_4}(t)) \tag{3}$$

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

crossover: As explained in [11], the crossover step ensures to increase or at least 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 crossed over.

The trial vector $U_j = (u_{j1}, u_{j2}, \ldots, u_{jd})$ is generated using Equation 4:

$$u_{ji}(t+1) = \begin{cases} v_{ji}(t+1) & \text{if} \quad (rand \leq CR) \quad \text{or } j = rnbr(i) \\ x_{ji}(t) & \text{if} \quad (rand > CR) \quad \text{and } j \neq rnbr(i) \end{cases} \tag{4}$$

where x_{ji} is the jth component of vector x_i; v_{ji} is the jth component of the variant vector; $rand$ is drawn from a uniform random number generator 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 in the range $[1,d]$ which ensures that $U_j(t+1)$ gets at least one parameter from the variant vector $V_j(t+1)$, and t is the number of the current iteration (or generation).

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 the greedy criterion. If $f(U_i(t+1)) < f(X_i(t))$, then $X_i(t+1) = U_i(t+1)$ otherwise the old value $X_i(t)$ is kept.

These four steps are looped over until the maximum number of evaluations/iterations is reached or until a fit enough solution is found. DE is quite simple as it only needs three parameters, the population size (N), the crossover rate (CR), and the scaling factor (F).

III. Linear Differential Evolutionary Programming

We propose to generate linear programs from DE, which we call Linear Differential Evolutionary Programming (LDEP). The real-valued vectors that are evolved by the DE engine are mapped to sequences of imperative instructions. Contrary to the usual continuous optimization case, we can not deduce the vector length from the problem, so we have to set this parameter quite arbitrarily. This length will determine the maximum number of instructions allowed in the evolved programs.

A. Linear sequence of instructions

For the representation of programs, our work is based on LGP [6], with slight modifications.

Each imperative instruction is a 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:

$$r_i = r_j \text{ op } (r_k|c_k)$$

where r_i is the destination register, r_j and r_k are calculation registers (or operands) and c_k is a constant.

On the implementation level, each imperative instruction is a list of four values where the first value gives the operator and the three next values represent the three register indices. For instance, an instruction like $r_i = r_j \times r_k$ would be coded as four real values, to be casted into four indices (as explained in the next section), then translated to: $< op(\times), i, j, k >$. Of course, even if the programming language is basically a 3-register instruction language, it is possible to drop the last index in order to include 2-register instructions like $r_i = \sin(r_j)$.

Instructions are executed by a virtual machine with floating-point value registers. A subset of the registers contains the inputs to our problem. Besides this required minimal number of registers, we use an additional set of registers for calculation purpose and for storing constants. In the standard case, one of the calculation registers (usually named r_0) is used for holding the output of the program after execution. Evolved programs can read or write into these registers, with the exception of the read-only constant registers. The use of calculation registers allows a number of different program paths, as explained in [6].

Constants are initialized at the beginning of the run with values in a range defined by the user, then stored once for all in registers that will be accessed in read-only mode from now on. This means that our set of constants remains fixed and does not evolve during the run. Each instruction can either manipulate two registers or one register and one constant (register) but neither two constants (registers), as in LGP. The number of constants has to be set by the user, and we will also use a constant probability parameter to control the probability of occurrences of constants, as explained below.

B. Mapping vectors to programs

As sketched above, LDEP splits the population vectors into lists of 4 floating point-values (v_1, v_2, v_3, v_4). v_1 will encode the operator, v_2, v_3 and v_4 will encode the operands.

So for instance, an instruction like $r_i = r_j + r_k$ reduces to 4 floating point values $< v_1(+), v_2(i), v_3(j), v_4(k) >$. Value v_1 will be turned into an integer value that will index the operators list. Value v_i $(2 \leq i \leq 4)$ will be turned into an integer value to index the registers list. In order, for each element to be turned into an operator or an operand we apply the following equation which delivers a floating point value in the range $[0.0, 1.0[$:

$$(v_j - \lfloor v_j \rfloor) \tag{5}$$

(for example,if $v_j = 1.87$, then we will get $(v_j - \lfloor v_j \rfloor) = 0.87$).

Let $n_{\text{operators}}$ be the number of operators and $n_{\text{registers}}$ be the number of registers. The operator index will be equal to:

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

The same kind of transformation applies for computing the number of the register:

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

If the value returned by Equation 5 for an operand is less or equal to a fixed constant probability parameter, then a constant must be used (if no other constant has been used for this expression). The equation used here is different from the previous ones, as we directly take v_j to index the array of constant registers (else keeping only the fractional part of v_j would induce a strong linkage between the constant probability and the constant register index).

$$\#\ \text{constant} = (\lfloor (v_j) \times C) \rfloor \bmod C \quad (8)$$

where C is the number of constants.

Example: The following paragraph describes an example of the mapping process:

Let us suppose LDEP works with the 4 following operators:

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

and that 6 registers (r_0 to r_5) are available.

If we consider the following vector that is made of 8 values (2 imperative instructions):

0.17	2.41	1.84	1.07	0.65	1.22	1.22	4.28

The first vector value denotes one operator among the four to choose from. To make this choice, this first value is turned into an integer using Equation 6, operator $= \lfloor 0.17 \times 4 \rfloor = 0$, meaning that the first operator will be $+$.

The second value $v_2 = 2.41$ is turned into an operand. According to Equation 7, operand $= \lfloor 0.41 \times 6 = 2.4 \rfloor = 2$, meaning that r_2 will be used as the leftmost operand.

The next value $v_3 = 1.84$ will determine what operand will be used. If Equation 5 returns a value less or equal to the constant probability v_3 will be used as a constant otherwise it is considered as a register. Let us assume that this value is greater than the constant probability $\lfloor 0.84 \times 6 \rfloor = 5$, r_5 will be the second operand.

The last operand will be decoded as a constant as we suppose for the clarity of this example that the returned value $(v_4 - \lfloor v_4 \rfloor) = 0.07$ is less than the constant probability, so v_4 will be used to index a constant. The index number will be $\lfloor 1.07 \times 50 \rfloor \bmod 50 = 3$.

So with the 4 first values of the genotype, we now have the following:

$$r_2 = r_5 + c_3$$

Let us assume c_3 holds the value 1.87. The mapping process continues with the four next values, until we are left with the following program:

$$r_2 = r_5 + 1.87$$
$$r_1 = r_1 \times r_1$$

C. Algorithms

The rest of LDEP follows the DE general scheme:

Initialization: During the initialization phase, 50 values in the range $[-1.0, +1.0]$ are randomly initialized and will be used as constants. In DE, an upper and a lower bounds are used to generate the components of the vector solution X_i. LDEP uses the same bounds.

Iteration: We tried two variants of the iteration loop described in Section II-B: either generational replacement of individuals as in the original Storn and Price paper [11], or steady state replacement, which seems to be used by Veenhuis [13]. In the generational case, new individuals are stored upon creation in a temporary, and once creation is done, 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 as it is reported in the results section IV.

During the iteration loop, the DE vector solutions are decoded using Equations 6 and 7. The resulting linear programs are then evaluated on a set of fitness cases (training examples).

The representation and mapping method described in this section can be used to translate vectors of floats to linear programs, whatever the way how vectors are generated. So, we will use that same representation to evolve vectors with DE (LDEP) and CMA-ES.

IV. Experiments

In order to validate our scheme against TreeDE [13], we used the same problems as benchmarks (4 symbolic regression and the artificial ant problems), and we also added two regression problems with constants. We also ran all problems with standard GP, using the well-known ECJ library (http://cs.gmu.edu/~eclab/projects/ecj/). For all problems we measured the average best fitness of 40 independent runs. We also computed the ratio of so-called "hits", i.e. perfect solutions, and average number of evaluations to reach a hit. These last two figures are less reliable than the average fitness, as shown in [20], however we included them to allow a complete comparison with [13] that used these indicators.

- We used the standard values for the control parameters for DE namely : $F = 0.5, CR = 0.1$. We work with a set of $N = 20$ vectors (population size) for regression and for the artificial ant. As said before, the update method for DE is the so-called DE/best/2/bin. Results are shown for both generational and steady state LDEP variant.
- The size of vector (individual) that DE works with was set to 128 for regression, meaning that each program is equal to $128 \div 4 = 32$ imperative instructions. It is reduced to 50 for the artificial ant, since in that

case we need only one float to code one instruction, as explained in Section IV-B.

- When needed in the symbolic regression problems, the constant probability was set to 0.05.
- For regression 6 read/write registers were used for calculation (from r_0 to r_5), r_0 being the output register. They were all initialized for each training case (x_k, y_k) with the input value x_k.
- 1500 iterations on a population of 20 vectors were allowed for regression in the TreeDE experiments [13]. Runs were done for every tree depth in the range $\{1, \ldots, 10\}$, thus amounting to a total of $300,000$ evaluations, among these only the runs with the best tree depth were used to provide the figures given in Veenhuis paper. We could not apply this notion of best tree depth in our heuristic, and thus decided as a trade-off to allow $50,000$ evaluations.
- For the Santa Fe Trail artificial ant problem, the same calculation gives a total of $450,000$ evaluations in [13]. We decided as a trade-off to allow $200,000$ evaluations.
- the GP parameters were set to 50 generations and respectively 1000 individuals for the regression problems, and 4000 individuals for the artificial ant, in order to have the same maximum number of evaluations than LDEP. Genetic operator rates were tuned according to the usual practice: 80% for crossover, 10% for sub-tree mutation and 10% for duplication. The maximum tree depth was set to 11, and we kept the best (elite) individual from one generation to the next. For the regression problems, we defined 4 "input" terminals (reading the input value x_k for each training case (x_k, y_k)) against only one ephemeral random constant (ERC) terminal, thus the probability to generate a constant was lower than the usual 50% and thus closer to LDEP (this improves sensibly the GP results in Table II).
- we used the publicly available C language version of CMA-ES (downloaded at http://www.lri.fr/~hansen/cmaes_inmatlab.html). We kept the overall default parameters, setting the maximum number of function evaluations in accordance to the LDEP limit and the length of vectors to 128. We tried two values for $\sigma \in \{1, 10\}$ and two values for $\lambda \in \{10, 100\}$: there was no significant difference for the regression problems, but for the Artificial Ant the best setting was $\sigma = 10$ and $\lambda = 100$.

A. Symbolic Regression problems

The aim of a symbol regression problem is to find some mathematical expression in symbolic form that associates input and output on a given set of training pairs. In our case, 20 evenly distributed data points x_k in the range $[-1.0, +1.0]$ are chosen as inputs, the outputs being given by the following test functions from [13]:

$$\begin{aligned} f_1 &= x^3 + x^2 + x \\ f_2 &= x^4 + x^3 + x^2 + x \\ f_3 &= x^5 + x^4 + x^3 + x^2 + x \\ f_4 &= x^5 - 2x^3 + x \end{aligned}$$

As TreeDE benchmarks were run without constants in [13], we run LDEP both without and with constants. While this allows us to assess the impact of constant management, anyway we strongly believe that constants should be included in any regression problem, since in the general case one can not know in advance whether or not they are useful. For that same reason we add two benchmarks:

$$\begin{aligned} f_5 &= \pi \quad \text{(constant function)} \\ f_6 &= \tfrac{x}{\pi} + \tfrac{x^2}{\pi^2} + 2x\pi \end{aligned}$$

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 $b = 0$).

Evaluation (or fitness computation) is done in the typical way, that is computing the sum of deviations over all points, i.e $fitness = \sum_k |f(x_k) - P(x_k)|$ where P is the evolved program and k the number of input/output pairs. A hit means that the fitness function is less than 10^{-4} on each training pair.

As it can be seen in Table I, all three heuristics LDEP, TreeDE and GP 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 (however the TreeDE figures are taken from [13] where they are given only for the best tree depth). 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 state variant versus generational, in this limited number of evaluations framework.

When running the heuristics with constants (thus ruling out TreeDE) on all problems f_1 to f_6 in Table II, 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. In contrast, CMA-ES results are an order of magnitude worse: we think this 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 the heuristic is left with its only imperfect model (in our case) to generate new test points.

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

Table I
Results for symbolic regression problems without constants.

For each heuristic, the column Fit. gives the average of the best fitness over 40 independent runs (taken from [13] for TreeDE); then we have the percentage of hits, then the average number of evaluations for a hit.

	`generational LDEP`			`steady state LDEP`			`TreeDE`			`standard GP`		
Problem	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
f_1	0.0	100%	4297	0.0	100%	2632	0.0	100%	1040	0.0	100%	1815
f_2	0.0	100%	12033	0.0	100%	7672	0.0	100%	3000	0.0	100%	2865
f_3	0.28	72.5%	21268	0.08	85%	21826	0.027	98%	8440	0.03	97%	6390
f_4	0.20	62.5%	33233	0.13	75%	26998	0.165	68%	14600	0.01	80%	10845

Table II
Results for symbolic regression problems with constants.

For each heuristic, the column Fit. gives the average of the best fitness over 40 independent runs; then next column gives the percentage of hits, then the average number of evaluations for a hit (if any).

	`generational LDEP`			`steady state LDEP`			`standard GP`			`CMA-ES`		
Problem	Fit.	%hits	Eval.	Fit.	%hits	Eval.	Fit.	%hits	Eval.	Fit.	%hits	Eval.
f_1	0.0	100%	7957	0.0	100%	7355	0.002	98%	3435	0.03	20%	6500
f_2	0.02	95%	16282	0.0	100%	14815	0.0	100%	4005	2.76	0%	NA
f_3	0.4	52.5%	24767	0.0	100%	10527	0.02	93%	7695	5.33	0%	NA
f_4	0.36	42.5%	21941	0.278	45%	26501	0.33	23%	24465	2.06	6%	10900
f_5	0.13	2.5%	34820	0.06	15%	29200	0.07	0%	NA	13.35	0%	NA
f_6	0.59	0%	NA	0.63	0%	NA	0.21	0%	NA	5.12	0%	NA

outside its original field. The LDEP mix of linear programs and constant management seems interesting enough, when compared to standard GP, to deserve further study.

B. Santa Fe Ant Trail

The Santa Fe ant trail is a quite 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 times. The trail is situated on a 32×32 toroidal grid. The problem is known to be rather hard, at least for standard GP (see [21]), with many local and global optima, which may explain why the size of the TreeDE population was increased to $N = 30$ in [13].

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 square in the direction it is facing, in order to determine if 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.

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 Equation 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`. 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 (thus it may be iterated if time steps are remaining).

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 being a program with fitness 0, i.e. able to pick up all food on the grid.

Results are summed-up in Table III. Contrary to the regression experiment, the generational variant of LDEP is now better than the steady state. We think this is ex-

Table III
Santa Fe Trail artificial ant problem.

The 1st columns is the number of allowed time steps, then for each heuristics, we give the average of the best fitness value over the 40 independent runs (taken from [13] for TreeDE), then the percentage of hits (solutions that found all 89 food pellets), then the average number of evaluations for a hit if applicable.

	generational LDEP			steady state LDEP			TreeDE			standard GP		
# steps	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.	Fit.	% hits	Eval.
400	11.55	12.5%	101008	14.65	7.5%	46320	17.3	3%	24450	8.87	37%	126100
600	0.3	82.5%	88483	1.275	70%	44260	1.14	66%	22530	1.175	87%	63300

	CMA-ES		
# steps	Fit.	% hits	Eval.
400	37.45	0%	NA
600	27.05	0%	NA

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

```
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 } }
```

plained by the hardness of the problem: more exploration is needed, and it pays no more to accelerate convergence.

GP provides 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. LDEP is also better than TreeDE on both steps limits. CMA-ES performed really poorly, and its first results were so bad it motivated us to try rather high initial variance levels ($\sigma >= 10$), 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).

Table IV shows an example of a perfect solution found by LDEP for 400 time steps.

V. Conclusion and future works

This paper is a further investigation into Differential Evolution engines applied to automatic programming. Unlike TreeDE [13], our scheme allows the use of constants in symbolic regression problems and translates the continuous DE representation to linear programs, thus avoiding the systematic search for the best tree depth that is required in TreeDE.

Comparisons with GP confirm that DE is a promising area of research 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 the need for fine-tuning the architecture of solutions. For both regression and artificial ant, CMA-ES 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 CMA-ES. Nonetheless it is also the case for DE, which still manages to produce interesting solutions, thus this points to a fundamental difference in behavior between both heuristics.

The results of LDEP on the artificial ant problem is also a great incentive to try other problems apart from symbolic regression, such as sorts or data structure manipulations, that were already tackled with GP. 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? Which parameters are crucial for DE-based automatic programming?

References

[1] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza, *Genetic Programming IV Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers, 2003.

[2] S. H. Al-Sakram, J. R. Koza, and L. W. Jones, “Automated re-invention of a previously patented optical lens system using genetic programming,” in *[22]*, 2005, pp. 25–37.

[3] K.-H. Liu and C.-G. Xu, “A genetic programming-based approach to the classification of multiclass microarray datasets,” *Bioinformatics*, vol. 25, no. 3, pp. 331–337, 2009.

[4] A. Gepp and P. Stocks, “A review of procedures to evolve quantum algorithms,” *Genetic Programming and Evolvable Machines*, vol. 10, no. 2, pp. 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 *[23]*, 2009, pp. 2780–2787.

[6] M. Brameir and W. Banzhaf, *Linear Genetic Programming*. Springer, 2007.

[7] H. Abbass, N. Hoai, and R. Mckay, "AntTAG: A new method to compose computer programs using colonies of ants," in *The IEEE Congress on Evolutionary Computation*. Citeseer, 2002, pp. 1654–1659.

[8] Y. Shan, H. Abbass, R. 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*, vol. 30. Citeseer, 2002.

[9] R. P. Salustowicz and J. Schmidhuber, "Probabilistic incremental program evolution," *Evolutionary Computation*, vol. 5, no. 2, pp. 123–141, 1997.

[10] E. N. Regolin and A. T. R. Pozo, "Bayesian automatic programming," in *[22]*, 2005, pp. 38–49.

[11] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[12] M. O'Neill and A. Brabazon, "Grammatical differential evolution," in *International Conference on Artificial Intelligence (ICAI'06)*, Las Vegas, Nevada, USA, 2006, pp. 231–236.

[13] C. B. Veenhuis, "Tree based differential evolution," in *[24]*, 2009, pp. 208–219.

[14] 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*, May 2008, pp. 000 033 –000 036.

[15] C. G. Moles, P. Mendes, and J. R. Banga, "Parameter Estimation in Biochemical Pathways: A Comparison of Global Optimization Methods," *Genome Research*, vol. 13, no. 11, pp. 2467–2474, 2003. [Online]. Available: http://genome.cshlp.org/content/13/11/2467.abstract

[16] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation," in *International Conference on Evolutionary Computation*, 1996, pp. 312–317.

[17] A. Auger and N. Hansen, "A restart cma evolution strategy with increasing population size," in *Evolutionary Computation, 2005. The 2005 IEEE Congress on*, vol. 2, 2005, pp. 1769 – 1776 Vol. 2.

[18] S. Baluja and R. Caruana, "Removing the genetics from the standard genetic algorithm." Morgan Kaufmann Publishers, 1995, pp. 38–46.

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

[20] S. Luke and L. Panait, "Is the perfect the enemy of the good?" in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds. New York: Morgan Kaufmann Publishers, 9-13 Jul. 2002, pp. 820–828.

[21] W. B. Langdon and R. Poli, "Why ants are hard," University of Birmingham, School of Computer Science, Tech. Rep. CSRP-98-4, Jan. 1998, presented at GP-98.

[22] M. Keijzer, A. Tettamanzi, P. Collet, J. van Hemert, and M. Tomassini, Eds., *8th European Conference, EuroGP 2005*, ser. LNCS, vol. 3447, Lausanne, Switzerland, mar 2005.

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

[24] L. Vanneschi, S. Gustafson, A. Moraglio, I. D. Falco, and M. Ebner, Eds., *12th European Conference, EuroGP 2009*, ser. LNCS, vol. 5481, Tubingen, Germany, apr 2009.