

A Continuous Approach to Genetic Programming

Cyril Fonlupt and Denis Robilliard

LISIC — ULCO
Univ Lille Nord de France
BP 719, F-62228 Calais Cedex, France
{fonlupt,robilliard}@lisic.univ-littoral.fr

Abstract. Differential Evolution (DE) is an evolutionary heuristic for continuous optimization problems. In DE, solutions are coded as vectors of floats that evolve by crossover with a combination of best and random individuals from the current generation. Experiments to apply DE to automatic programming were made recently by Veenhuis, coding full program trees as vectors of floats (Tree Based Differential Evolution or TreeDE). In this paper, we use DE to evolve linear sequences of imperative instructions, which we call Linear Differential Evolutionary Programming (LDEP). Unlike TreeDE, our heuristic provides constant management for regression problems and lessens the tree-depth constraint on the architecture of solutions. Comparisons with TreeDE and GP show that LDEP is appropriate to automatic programming.

1 Introduction

In 1997, Storn and Price proposed a new evolutionary algorithm called Differential Evolution (DE) [7] for continuous optimization. DE is a stochastic search method akin to Evolution Strategies, that uses information from within the current vector population to determine the perturbation brought to solutions (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 [5]. 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 [9] 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,

as in Linear Genetic Programming (LGP) [1]. We can thus implement real-valued constants management inspired from 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.

This paper is organized in the following way. Section 2 presents the main DE concepts. In section 3, our scheme Linear Differential Evolutionary Programming (LDEP) is introduced while section 4 and 5 give experimental results and comparisons with TreeDE and standard GP. Future works and conclusions are discussed in section 6.

2 Differential Evolution

This section only introduces the main *DE* concepts. The interested reader might refer to [7] for a full presentation. DE is a search method working on a set of N d -dimensional vector solutions X_i , f being the fitness function.

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

DE can be roughly decomposed into an initialization phase, where the genes of the initial population are randomly initialized with a Gaussian law, and four vary steps that are iterated on: mutation, crossover, evaluation and selection.

Mutation. In the initial DE [7], 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:

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

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 [9,6], the mutation method that leads to the best results on most problems is the method DE/best/2/bin, with:

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

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

Crossover. As explained in [7], 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 crossovered.

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

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

where x_{ji} is the j th component of vector x_i ; v_{ji} is the j th component of the variant vector $V_j(t+1)$; $rand$ 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 evaluation-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).

3 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.

3.1 Representation

For the representation of programs, our work is based on LGP [1], with slight modifications.

For regression, we use mainly 3-register imperative instructions that 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: $r_i = (r_j|c_j)\text{op} (r_k|c_k)$; where r_i is the destination register, r_j and r_k are calculation registers and c_j, c_k are constants. Only one constant is allowed per instruction as in LGP. Of course, even if the programming language is basically a 3-register instruction language, we can drop the last register/constant to include 2-register instructions like $r_i = \sin(r_j|c_i)$ if needed.

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 [1].

All the constants available for computation are initialized at the beginning of the run with values in a range defined by the user, then stored once for all in read-only registers that will be accessed in mode from now on. 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 p_c , as explained below.

3.2 Implementation

LDEP splits the population vectors into subsets of consecutive 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. The real values will be cast into integers to index the set of operators or registers as needed, using the following equation :

- Let $n_{\text{operators}}$ be the number of operators, the operator index is:

$$\# \text{operator} = (\lfloor (v_j - \lfloor v_j \rfloor) \times n_{\text{operators}} \rfloor) \quad (5)$$

- If the decimal part of an operand value, returned by the expression $(v_j - \lfloor v_j \rfloor)$, is greater than the fixed constant probability parameter p_c , then a register must be used. Let $n_{\text{registers}}$ be the number of registers, the register index is:

$$\# \text{register} = (\lfloor (v_j - \lfloor v_j \rfloor) \times n_{\text{registers}} \rfloor) \quad (6)$$

- If $(v_j - \lfloor v_j \rfloor) \leq p_c$ then a constant must be used, provided no other constant was already used as first operand for this expression. Let C be the number of constants, the constant registers index is:

$$\# \text{ constant} = \lfloor ((v_j) \times C) \rfloor \bmod C \quad (7)$$

Example: Let us suppose LDEP works with the 4 following operators $\{0 : +, 1 : -, 2 : \times, 3 : \div\}$, that 6 calculation registers (r_0 to r_5) are available, 50 constant registers and $p_c = 0.1$. Our example solution vector 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, $\text{operator} = \lfloor 0.17 \times 4 \rfloor = 0$, meaning that the first operator will be $+$.

The second value $v_2 = 2.41$ is turned into the destination register. According to Equation 5, $\text{operand} = \lfloor 0.41 \times 6 = 2.4 \rfloor = 2$, meaning that r_2 will be used as the destination register.

The next value $v_3 = 1.84$ is a register since the decimal part is greater than the constant probability: $\lfloor 0.84 \times 6 \rfloor = 5$, r_5 will be the leftmost operand.

The decimal part of the next value is: $(v_4 - \lfloor v_4 \rfloor) = 0.07$, as it is less than the constant probability, v_4 will be used to index a constant register. 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:

$$\begin{aligned} r_2 &= r_5 + 1.87 \\ r_1 &= r_1 \times r_1 \end{aligned}$$

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 for regression experiments. 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: either generational replacement of individuals as in the original Storn and Price paper [7], or steady state replacement, which seems to be used by Veenhuis [9]. 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 4.

During the iteration loop, the DE vector solutions are decoded using Equations 5 and 6. The resulting linear programs are then evaluated on a set of fitness cases (training examples).

4 Symbolic Regression and Artificial Ant Experiments

In order to validate our scheme against TreeDE [9], 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/~ecjlab/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 [4], however we included them to allow a complete comparison with [9] 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 4.2.
- 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 [9]. Runs were done for every tree depth in the range $\{1, \dots, 10\}$, 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 [9]. 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 2).

4.1 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 [9]:

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

As TreeDE benchmarks were run without constants in [9], 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:

$$f_5 = \pi \quad (\text{constant function})$$

$$f_6 = \frac{x}{\pi} + \frac{x^2}{\pi^2} + 2x\pi$$

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 $b = 0$).

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 1, 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 [9] 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 2, 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.

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

Table 1. 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 [9] for TreeDE); then we have the percentage of hits, then the average number of evaluations for a hit.

Problem	generational LDEP			steady state LDEP			TreeDE			standard GP		
	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 2. 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		
Problem	Fit.	%hits	Eval.	Fit.	%hits	Eval.	Fit.	%hits	Eval.
f_1	0.0	100%	7957	0.0	100%	7355	0.002	98%	3435
f_2	0.02	95%	16282	0.0	100%	14815	0.0	100%	4005
f_3	0.4	52.5%	24767	0.0	100%	10527	0.02	93%	7695
f_4	0.36	42.5%	21941	0.278	45%	26501	0.33	23%	24465
f_5	0.13	2.5%	34820	0.06	15%	29200	0.07	0%	NA
f_6	0.59	0%	NA	0.63	0%	NA	0.21	0%	NA

management seems interesting enough, when compared to standard GP, to deserve further study.

4.2 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 time (either 400 or 600 time steps). The trail is situated on a 32×32 toroidal grid. The problem is known to be rather hard, at least for standard GP (see [2]), with many local and global optima, which may explain why the size of the TreeDE population was increased to $N = 30$ in [9].

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.

Table 3. 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 [9] 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.

# steps	generational LDEP			steady state LDEP			TreeDE			standard GP		
	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

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 5. 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 3. Contrary to the regression experiment, the generational variant of LDEP is now better than the steady state. We think this is explained 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.

5 Evolving a Stack with LDEP

As it seems that LDEP achieved quite interesting results on the previous benchmarks for genetic programming we decided to move forward and to test whether or not LDEP was able to evolve a more complex data structure: a stack. Langdon [3] successfully showed that GP was able to evolve five operations needed to manipulate the stack (push, pop, top, empty and makenull). Some of these operations are considered to be inessential (top, empty) but we chose to follow the settings introduced by Langdon's original work. Table 4 presents the five primitives that were used in our implementation with some comments.

Table 4. The five primitives to evolve (from [3])

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

This is in our opinion a more complex problem as the correctness of each trial solution is established using only the values returned by the stack primitives and only three (pop, top and empty) out of the five operations return values.

Choice of primitives. As explained in [3], the set of primitive 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 (increment or decrement the stack pointer) 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 arguments function arg1 and arg2. It evaluates the two arguments and sets the indexed memory pointed by arg1 to arg2 ($\text{stack}[\text{arg1}] = \text{arg2}$). 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).

5.1 Architecture and Fitness Function

We used a slightly modified version of LDEP as the stack problem requires the evolution of the five primitives (makenull, top, pop, push and empty) simultaneously. An individual is now composed of 5 vectors, one for each primitive. Mutation and crossover are only performed with vectors of the same type (*i.e.* vectors evolving the top primitive for example). Moreover the vector associated to a primitive is decoded as Polish Notation (or prefix notation), that means an operation like ($\text{arg1} + \text{MAX}$) is coded as + arg1 max. Each primitive vector has a maximum length of 100 values (this is several times more than sufficient to code any of the five primitives needed to manipulate the stack).

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

Results. In his original work, Langdon chose to use a population of 1,000 individuals with 101 generations. That means that roughly 100,000 fitness functions evaluations were used. In the LDEP case, based on our previous experience it appeared that using large population is usually inadequate and that LDEP performs better with small size population. In this case, a population of 10 individuals with 10,000 generations were used.

Langdon wrote that with his own parameters 4 runs (out of 60), produced successful individuals. It was interesting to see that similar rates of results were obtained: 6 runs out of 100 yielded perfect solutions. An example of successful run is given in table 5 with the evolved code and with the simplified code (redundant code removed).

Table 5. Example of an evolved push-down stack

Operation	Evolved stack	Simplified stack
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

6 Conclusion and Future Works

This paper is a further investigation into Differential Evolution engines applied to automatic programming. Unlike TreeDE [9], 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.

These results led us to try to evolve a set of stack management primitives. The experiment proved successful, with results similar to the GP literature. 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. Brameier, M., Banzhaf, W.: *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer, Heidelberg (2007)
2. Langdon, W.B., Poli, R.: Why ants are hard. Tech. Rep. CSRP-98-4, University of Birmingham, School of Computer Science (January 1998); presented at GP 1998
3. Langdon, W.B.: *Genetic Programming and Data Structures = Automatic Programming!* Kluwer Academic Publishers, Dordrecht (1998)
4. Luke, S., Panait, L.: Is the perfect the enemy of the good? In: Langdon, W.B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M.A., Schultz, A.C., Miller, J.F., Burke, E., Jonoska, N. (eds.) *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, July 9-13, pp. 820–828. Morgan Kaufmann Publishers, New York (2002)
5. O’Neill, M., Brabazon, A.: Grammatical differential evolution. In: *International Conference on Artificial Intelligence (ICAI 2006)*, Las Vegas, Nevada, USA, pp. 231–236 (2006)
6. Price, K.: Differential evolution: a fast and simple numerical optimizer. In: *Biennial Conference of the North American Fuzzy Information Processing Society*, pp. 524–527 (1996)
7. Storn, R., Price, K.: Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization* 11(4), 341–359 (1997)
8. Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.): *EuroGP 2009*. LNCS, vol. 5481. Springer, Heidelberg (2009)
9. Veenhuis, C.B.: Tree based differential evolution. In: [8], pp. 208–219 (2009)