Grammatical Differential Evolution

Michael O'Neill
Natural Computing Research and Applications Group,
University College Dublin
Ireland
Email: M.ONeill@ucd.ie

Anthony Brabazon
Natural Computing Research and Applications Group,
University College Dublin
Ireland
Email: Anthony.Brabazon@ucd.ie

Abstract— **This proof of concept study examines the possibility of specifying the construction of programs using Differential Evolution, and represents a new form of grammar-based genetic programming, Grammatical Differential Evolution (GDE). In GDE each individual member of the population represents a specific choice of program construction rules, where these rules are specified using a Backus-Naur Form grammar. The results demonstrate that it is possible to generate programs using the Grammatical Differential technique.**

Keywords: Grammatical evolution, differential evolution, automatic program generation.

I. Introduction

This paper details an investigation examining the possibility of specifying the automated construction of a computer program using a Differential Evolution learning model. In the Grammatical Differential Evolution (GDE) approach developed in this study, each vector represents choices of program construction rules specified as production rules of a Backus-Naur Form grammar.

GDE is grounded in the linear program representation adopted in Grammatical Evolution (GE) [1], which uses grammars to guide the construction of syntactically correct programs, specified by variable-length genotypic binary or integer strings. The search heuristic adopted with the canonical GE methodology is a variable-length Genetic Algorithm. In the GDE technique presented here, a particle's real-valued vector is used in the same manner as the genotypic binary string in canonical GE.

The remainder of the paper is structured as follows. Before describing the mechanism of GDE in section 4, introductions to the salient features of Differential Evolution (DE) and Grammatical Evolution (GE) are provided in sections 2 and 3 respectively. Section 5 details the experimental approach adopted and results, section 6 provides some discussion of the results, and finally section 7 details conclusions and future work.

II. Differential Evolution

Differential evolution (DE) (Storn and Price, 1995 and 1997; Price 1999; Storn 1999) is a population-based search algorithm. The algorithm draws inspiration from the field of Evolutionary Computation, as it embeds implicit concepts of mutation, recombination and fitness-based selection, to evolve from an initial randomly generated population to a solution to a problem of interest. It also borrows principles from Social Algorithms through the manner in which new individuals are generated. Unlike the binary chromosomes typical of GAs, an individual in DE is generally comprised of a real-valued chromosome.

Although several DE algorithms exist we only describe one version of the algorithm based on the *DE/rand/1/bin* scheme (Storn and Price, 1995). The different variants of the DE algorithm are described using the shorthand DE/x/y/z, where x specifies how the base vector to be perturbed is chosen (*rand* if it is randomly selected or *best* if the best individual is selected), y is the number of difference vectors used, and z denotes the crossover scheme used (*bin* for crossover based on independent binominal experiments, and *exp* for exponential crossover).

At the start of this algorithm, a population of N, d-dimensional vectors $X_j = (x_{i1}, x_{i2}, \ldots, x_{id})$, $j = 1, \ldots, n$, is randomly initialised and evaluated using a fitness function f. During the search process, each individual (j) is iteratively refined. The modification process has three steps:

1) Create a variant solution, using randomly selected members of the population.
2) Create a trial solution, by combining the variant solution with j (crossover step).
3) Perform a selection process to determine whether the trial solution replaces j in the population.

Under the mutation operator, for each vector $X_j(t)$, a variant solution $V_j(t+1)$ is obtained using equation 1:

$$V_j(t+1) = X_m(t) + F(X_k(t) - X_l(t)) \qquad (1)$$

where $k, l, m \in 1, \ldots, N$ are mutually different, randomly selected indices, and all the indices $\neq j$ (X_m is referred to as the base vector, and $X_k(t) - X_l(t)$ is referred to as a difference vector). Variants on this step include the use of more than three individuals from the population, and/or the inclusion of the highest-fitness point in the population as one of these individuals (Storn and Price, 1995). The difference between vectors X_k and X_l is multiplied by a scaling parameter F (typically, $F \in (0, 2]$). The scaling factor controls the amplification of the difference between X_k and X_l, and is used to avoid stagnation of the search process.

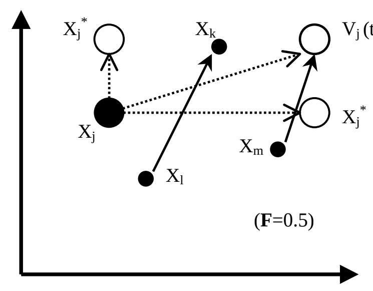

Fig. 1. A representation of the Differential Evolution variety-generation process. The value of F is set at 0.50. In a simple 2-d case, the child of particle X_j can end up in any of three positions. It may end up at either of the two positions X_j^*, or at the position of particle $V_j(t+1)$.

Following the creation of the variant solution, a trial solution $U_j(t+1) = (u_{j1}, u_{j2}, \ldots, u_{jd})$ is obtained from equation 2.

$$U_{jn}(t+1) = \begin{cases} V_{jn}, \text{if } (rand \leq CR) \text{ or } (j = rnbr(i)) \ ; \\ X_{jn}, \text{if } (rand > CR) \text{ and } (j \neq rnbr(i)). \end{cases} \tag{2}$$

where $n = 1, 2, \ldots, d$, $rand$ is drawn from a uniform random number generator in the range (0,1), CR is the user-specified crossover constant from the range (0,1), and $rnbr(i)$ is a randomly chosen index chosen from the range $(1, 2, \ldots, n)$. The random index is used to ensure that the trial solution differs by at least one component from $X_i(t)$. The resulting trial solution replaces its predecessor, if it has higher fitness (a form of selection), otherwise the predecessor survives unchanged into the next iteration of the algorithm (equation 3).

$$X_i(t+1) = \begin{cases} U_i(t+1), & \text{if } f(U_i(t+1)) < f(X_i(t)); \\ X_i(t), & \text{otherwise.} \end{cases} \tag{3}$$

Fig. 1 provides a graphic of the adaptive process described above.

The DE algorithm has three parameters, the population size (N), the crossover rate (CR), and the scaling factor (F). Higher values of CR tend to produce faster convergence of the population of solutions.

III. Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language [1], [2], [3], [4], [5], and can be considered a form of grammar-based genetic programming. Rather than representing the programs as parse trees, as in GP [6], [7], [8], [9], [10], a linear genome representation is used. A genotype-phenotype mapping is employed such that each individual's variable length binary string, contains in its codons (groups of 8 bits) the information to select production rules from a Backus Naur Form (BNF) grammar. The grammar allows the generation of programs in an arbitrary language that are guaranteed to be syntactically correct, and as such it is used as a generative grammar, as opposed to the classical use of grammars in compilers to check syntactic correctness of sentences. The user can tailor the grammar to produce solutions that are purely syntactically constrained, or they may incorporate domain knowledge by biasing the grammar to produce very specific forms of sentences.

BNF is a notation that represents a language in the form of production rules. It is comprised of a set of non-terminals that can be mapped to elements of the set of terminals (the primitive symbols that can be used to construct the output program or sentence(s)), according to the production rules. A simple example BNF grammar is given below, where `<expr>` is the start symbol from which all programs are generated. These productions state that `<expr>` can be replaced with either one of `<expr><op><expr>` or `<var>`. An `<op>` can become either `+`, `-`, or `*`, and a `<var>` can become either `x`, or `y`.

```
<expr> ::= <expr><op><expr> (0)
         | <var>            (1)
  <op> ::= +                (0)
         | -                (1)
         | *                (2)
 <var> ::= x                (0)
         | y                (1)
```

The grammar is used in a developmental process to construct a program by applying production rules, selected by the genome, beginning from the start symbol of the grammar. In order to select a production rule in GE, the next codon value on the genome is read, interpreted, and placed in the following formula:

$$Rule = Codon\ Value\ \%\ Num.\ Rules$$

where % represents the modulus operator.

Given the example individuals' genome (where each 8-bit codon is represented as an integer for ease of reading) in Fig.2, the first codon integer value is 220, and given that we have 2 rules to select from for `<expr>` as in the above example, we get $220\ \%\ 2\ =\ 0$. `<expr>` will therefore be replaced with `<expr><op><expr>`.

Beginning from the the left hand side of the genome, codon integer values are generated and used to select appropriate rules for the left-most non-terminal in the developing program from the BNF grammar, until one of the following situations arise: (a) A complete program is generated. This occurs when all the non-terminals in the expression being mapped are transformed into elements from the terminal set of the BNF grammar. (b) The end of the genome is reached, in which case the *wrapping* operator is invoked. This results in the return of the genome reading frame to the left hand side of the genome once again. The reading of codons will then continue unless an upper threshold representing the maximum number of wrapping events has occurred during this individuals mapping process. (c) In the event that a threshold on the

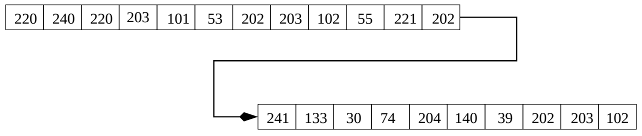

Fig. 2. An example GE individuals' genome represented as integers for ease of reading.

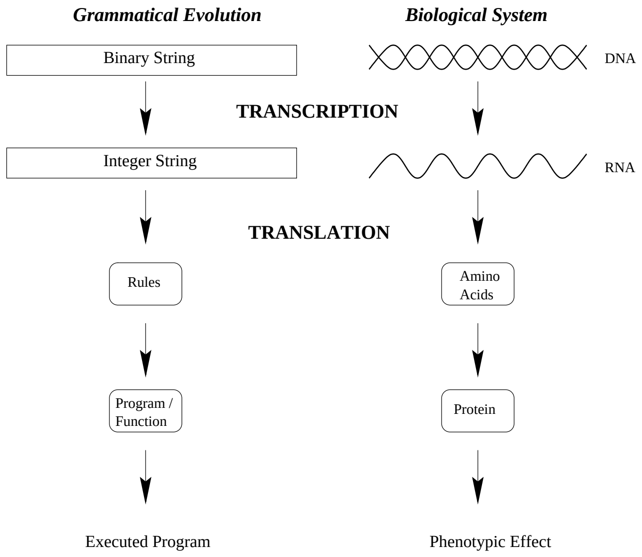

Fig. 3. A comparison between Grammatical Evolution and the molecular biological processes of transcription and translation. The binary string of GE is analogous to the double helix of DNA, each guiding the formation of the phenotype. In the case of GE, this occurs via the application of production rules to generate the terminals of the compilable program. In the biological case by directing the formation of the phenotypic protein by determining the order and type of protein subcomponents (amino acids) that are joined together.

number of wrapping events has occurred and the individual is still incompletely mapped, the mapping process is halted, and the individual assigned the lowest possible fitness value. Returning to the example individual, the left-most `<expr>` in `<expr><op><expr>` is mapped by reading the next codon integer value 240 and used in $240 \ \% \ 2 \ = \ 0$ to become another `<expr><op><expr>`. The developing program now looks like `<expr><op><expr><op><expr>`. Continuing to read subsequent codons and always mapping the left-most non-terminal the individual finally generates the expression `y*x-x-x+x`, leaving a number of unused codons at the end of the individual, which are deemed to be introns and simply ignored. Fig.3 draws an analogy between GE's mapping process and the molecular biological processes of transcription and translation. A full description of GE can be found in [1].

IV. Grammatical Differential Evolution

Grammatical Differential Evolution (GDE) adopts a Differential Evolution learning algorithm coupled to a Grammatical Evolution (GE) genotype-phenotype mapping to generate programs in an arbitrary language.

The standard GE mapping function is adopted with the real-values in the vectors being rounded up or down to the nearest integer value, for the mapping process. In the current implementation of GDE, fixed-length vectors are adopted within which it is possible for a variable number of elements to be required during the program construction genotype-phenotype mapping process. A vector's values may be used more than once if the wrapping operator is used, and in the opposite case it is possible that not all elements will be used during the mapping process if a complete program comprised only of terminal symbols is generated before reaching the end of the vector. In this latter case, the extra element values are simply ignored and considered introns that may be switched on in subsequent iterations.

V. Experiments & Results

A diverse selection of benchmark programs from the literature on genetic programming are tackled using GDE to demonstrate proof of concept for the method. The parameters adopted across the following experiments are a popsize of 500, the algorithm is run for 60 iterations, strlen=100, F=0.8, CR=0.8, gene values bound to the range $[0 \rightarrow 255]$. Four versions of DE are used, and results are reported for all four GDE variants.

The same problems are also tackled with Grammatical Evolution in order to get some indication of how well GDE is performing at program generation in relation to the more traditional variable-length Genetic Algorithm-driven search engine of standard GE. A standard population size of 500 running for 60 generations is adopted for Grammatical Evolution. The remaining parameters for Grammatical Evolution are roulette selection, steady state replacement, one-point crossover with probability of 0.9, and a bit mutation with probability of 0.01.

A. *Santa Fe Ant trail*

The Santa Fe ant trail is a standard problem in the area of GP and can be considered a deceptive planning problem with many local and global optima [11]. The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a non-continuous trail within a specified number of time steps, the trail being located on a 32 by 32 toroidal grid. The ant can only turn left, right, move one square forward, and may also look ahead one square in the direction it is facing to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid facing the first piece of food on the trail. The grammar used in this problem is different to the ones used later for symbolic regression and the multiplexer problem in that we wish to produce a multi-line function in this case, as opposed to a single line expression. The grammar for the Santa Fe ant trail problem is given below.

```
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::= if(food_ahead()) { <line> }
                else { <line> }
<op> ::=  left(); | right(); | move();
```

B. *Quartic Symbolic Regression*

The target function is $f(a) = a + a^2 + a^3 + a^4$, and 100 randomly generated input-output vectors are created for each call to the target function, with values for the input variable drawn from the range [0,1]. The fitness for this problem is given by the reciprocal of the sum, taken over the 100 fitness cases, of the absolute error between the evolved and target functions. The grammar adopted for this problem is as follows:

```
<expr> ::= <expr> <op> <expr> | <var>
<op> ::=  + | - | * | /
<var> ::= a
```

C. *3 Multiplexer*

An instance of a multiplexer problem is tackled in order to further verify that it is possible to generate programs using Grammatical Swarm. The aim with this problem is to discover a boolean expression that behaves as a 3 Multiplexer. There are 8 fitness cases for this instance, representing all possible input-output pairs. Fitness is the number of input cases for which the evolved expression returns the correct output. The grammar adopted for this problem is as follows:

```
<mult> ::= guess = <bexpr> ;
<bexpr> ::= ( <bexpr> <bilop> <bexpr> )
          | <ulop> ( <bexpr> )
          | <input>
<bilop> ::= and | or
<ulop> ::= not
<input> ::= input0 | input1 | input2
```

D. *Mastermind*

In this problem the code breaker attempts to guess the correct combination of colored pins in a solution. When an evolved solution to this problem (i.e. a combination of pins) is to be evaluated, it receives one point for each pin that has the correct color, regardless of its position. If all pins are in the correct order than an additional point is awarded to that solution. This means that ordering information is only presented when the correct order has been found for the whole string of pins.
A solution, therefore, is in a local optimum if it has all the correct color, but in the wrong positions. The difficulty of this problem is controlled by the number of pins and the number of colors in the target combination. The instance tackled here uses 4 colors and 8 pins with the following values 3 2 1 3 1 3 2 0. The grammar adopted is as follows.

```
<pin> ::= <pin> <pin> | 0 | 1 | 2 | 3
```

Table I provides a summary and comparison of the performance of Grammatical Differential Evolution, Grammatical Evolution and Grammatical Swarm on each of the problem domains tackled. In three out of the four problems Grammatical Evolution outperforms GDE, Grammatical Swarm outperforms Grammatical Evolution on one problem instance, and there is a tie between the methods on the Mastermind problem. The key finding is that the results demonstrate proof of concept that GDE can successfully generate solutions to problems of interest.

VI. Conclusions & Future Work

This study demonstrates the feasibility of generating computer programs using Grammatical Differential Evolution over four different problem domains. While a performance comparison to Grammatical Evolution has shown that GDE is outperformed on three of the problems analyzed, the ability of GDE to generate solutions without optimization of the algorithm's parameters is very encouraging for future development of the GDE. Future work will involve further investigation of the performance of GDE variants, and further parameter optimization for each GDE variant. Another interesting avenue is the development of a variable-length Differential Evolution algorithm to remove GDE's fixed length structure, and the investigation of the impact using a continuous encoding over the discrete encoding variant applied in this study.

References

[1] O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language*. Kluwer Academic Publishers.

[2] O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution*. PhD thesis, University of Limerick, 2001.

[3] O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation*. Vol. 5, No.4, 2001.

[4] O'Neill, M., Ryan, C., Keijzer M., Cattolico M. (2003). Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, Vol. 4 No. 1. Kluwer Academic Publishers, 2003.

[5] Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, 83-95, Springer-Verlag.

[6] Koza, J.R. (1992). *Genetic Programming*. MIT Press.

[7] Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.

[8] Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann.

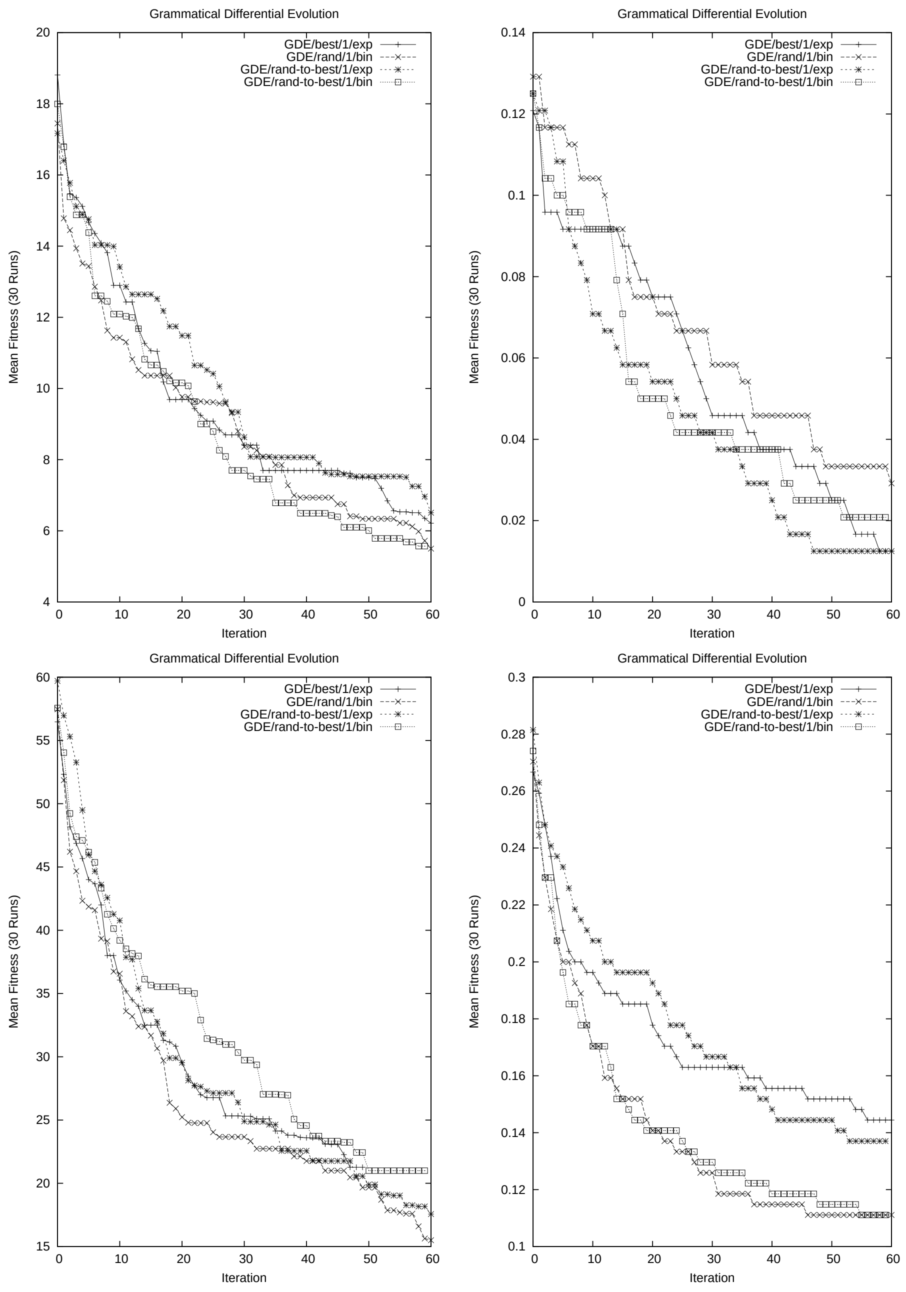

Fig. 4. Plot of the mean best fitness on the quartic (top left), 3 multiplexer (top right), Santa Fe ant (bottom left) and Mastermind (bottom right) problems.

TABLE I

A comparison of the results obtained for Grammatical Differential Evolution to Grammatical Swarm and Grammatical Evolution across all the problems analyzed.

	Santa Fe ant	**Multiplexer**	**Symb.Regression**	**Mastermind**
GDE/rand/1/bin	10	23	6	0
GDE/best/1/exp	7	**27**	4	0
GDE/rand-to-best/1/exp	9	**27**	4	0
GDE/rand-to-best/1/bin	7	25	5	0
GS	11	23	5	**3**
GE	**17**	15	**24**	**3**

[9] Koza, J.R., Andre, D., Bennett III, F.H., Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufmann.

[10] Koza, J.R., Keane, M., Streeter, M.J., Mydlowec, W., Yu, J., Lanza,

G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

[11] Langdon, W.B., and Poli, R. (1998). Why Ants are Hard. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, pp. 193-201, Morgan Kaufmann.