A New Approach for Generating Numerical Constants in Grammatical Evolution

[Extended Abstract]

Douglas A. Augusto
LNCC/MCT
Petrópolis, RJ, Brazil
douglas@lncc.br

Helio J.C. Barbosa
LNCC/MCT & DCC/UFJF
Petrópolis, RJ, Brazil
hcbm@lncc.br

André M.S. Barreto
LNCC/MCT
Petrópolis, RJ, Brazil
amsb@lncc.br

Heder S. Bernardino
LNCC/MCT
Petrópolis, RJ, Brazil
hedersb@lncc.br

ABSTRACT

A new approach for numerical-constant generation in Grammatical Evolution is presented. Experiments comparing our method with the three most popular methods for constant creation are performed. By varying the number of bits to represent a constant, we can increase our method's precision to the desired level of accuracy, overcoming by a large margin the other approaches.

Categories and Subject Descriptors

I.2.0 [**Computing Methodologies**]: [Artificial Intelligence, General]; I.2.2 [**Automatic Programming**]: [Program synthesis]

General Terms

Algorithms, Experimentation

Keywords

Constant Creation, Grammatical Evolution

1. INTRODUCTION

Grammatical Evolution (GE) is an elegant way of evolving programs where candidate solutions are binary strings which encode programs through a user-specified grammar [3].

Regardless of the particular metaheuristic used as the search engine, the success of any GE method depends crucially on its capability of generating numeric constants within the programs [3]. In this paper we propose a new approach for constant handling in GE which we call *ephemeral constant*. Our method can be seen as a translation of the classical genetic programming's ephemeral random constant to the GE framework. Its most distinctive feature is that it decouples the number of bits used to encode the grammar's production rules from the number of bits used to represent a constant. This makes it possible to increase the method's representational power without incurring in an overly redundant encoding scheme.

2. EVOLVING CONSTANTS IN GE

GE's binary string encodes an integer array using b bits (usually $b = 8$) for each integer i which is then used to select a rule from the grammar via $rule = (i \mod m)$, where m is the number of rules for the current nonterminal. Some of the most important constant-handling approaches found in the GE literature are as follows. The original method for constant handling in GE, here referred to as the *traditional approach* and denoted by "trad", operates by defining grammatical rules to allow the mathematical manipulation of single-digit integers (usually between 0 and 9). As a result, the generation of real values might require the evolution of complex mathematical expressions. The *digit concatenation* approach, denoted here by "dc", was proposed in [2] as a way to circumvent such difficulty. In this method a numerical constant is generated by the concatenation of decimal digits (including the dot separator for real values). In the *persistent random constants* method, denoted here by "prc", real numbers are randomly generated and included in the production rules at the outset of the search process. This method is essentially a variation of the traditional approach, since constants are created through the arithmetic manipulation of a finite number of predefined values.

3. EPHEMERAL CONSTANT

Our method resembles the canonical implementation of GP's ephemeral random constant, in which numerical constants are randomly generated and assigned to a program during its creation [1]. As in GP, the proposed method stores the constants directly in the program's genotype but, following GE's representation scheme, they are encoded as strings of bits rather than as real values. The implementation is very simple: it only requires the introduction of the production rule <constant> ::= *ephemeral*. The terminal symbol *ephemeral* carries a special meaning: whenever it is selected during the program's decoding process, the next n bits are decoded into a real number. Afterwards, the decoding process resumes normally past those n decoded bits. There are two user-defined parameters: the interval $[c_{\min}, c_{\max}]$ into which the numerical constants will be mapped, and the number of bits n used to represent each ephemeral constant.

The ephemeral constant method can be seen as a modification of the persistent random constant technique. The main improvement of the former with respect to the latter is the separation between the representation of *constants* and *production rules*. This makes it possible to tune the precision with which the constants are represented without affecting the dynamics of the grammar's decoding process. In principle one could argue that the same effect could be obtained with the persistent random constant method. However, in

GECCO'11, July 12–16, 2011, Dublin, Ireland.
ACM 978-1-4503-0690-4/11/07.

that case the number of bits used to encode the production rules would also increase, which could harm the evolutionary process. Besides, since in the persistent random constant approach the constants must be explicitly enumerated in the grammar, increasing the precision may lead to practical difficulties. For example, a 32-bit representation can hold up to 2^{32} constants; using standard double precision to represent such constants would require 32 gigabytes of physical memory. Another difference between the ephemeral constant method and its precursor is the fact that in our method the constants are (implicitly) sorted. This may help the evolutionary search if an appropriate representation scheme, such as a Gray code, is adopted. Besides, the constants used by our approach are evenly distributed over the interval $[c_{\min}, c_{\max}]$, what makes the underlying GE algorithm less susceptible to the contingencies inherent to the persistent random constant's sampling process.

4. COMPUTATIONAL EXPERIMENTS

Computational experiments were performed to evaluate the approach proposed above. Four variants of our method (denoted by "ec") were analyzed, corresponding to the use of 8, 16, 24, and 32 bits for constant encoding.

The standard search mechanism adopted for all approaches was a simple Gray coded genetic algorithm. A hundred independent runs were performed with the GE parameters set to: population size 500, chromosome length 800, mutation rate 1/(chromosome length), one-point crossover with probability 0.9, tournament selection with two individuals, 500 generations, and each integer encoded in 8 bits.

The objective of the test-problems was to evolve the following real constants: 1.23, 123000, 0.000123, 45.60099, 45600.99, and 0.0004560099. The fitness, to be maximized, was defined as $f(x) = 1/(1+|target-x|)$. The results, listed according to increasing mean relative error, are summarized in Table 1, which presents the relative error ($|target - x|/target$) and the mean expression size—which was defined here as the number of arithmetic operators and numerical constants appearing in the final expression.

From Table 1, it is clear that "dc" presents the best overall behavior among the proposals from the literature. However, when considering the mean relative error, it is outperformed by the proposed "ec-16bits" variant. Also the mean relative error of the "ec-24bits" and "ec-32bits" variants are even smaller, as expected. Finally, the results indicate that, as the number of bits used to encode numerical values in the "ec" variants increase, the expression size decreases. However, the "dc" technique produced the shortest expressions.

5. CONCLUSIONS

By dissociating the number of bits used to represent a constant from the number of bits used to index a production rule, our new approach provides extra flexibility for constant handling in GE. The experiments indicate that an increase in the number of bits used in the constants' representation increases accuracy, as expected, and reduces the size of the final expressions evolved. By increasing our method's numeric precision, the alternative approaches are outperformed with respect to approximation accuracy. As for the complexity of the final solutions, digit concatenation generates expressions that are slightly simpler than those returned by our approach.

Method	Relative error ($\times 10^2$) Mean	Min.	Max.	Mean size
Constant 1.23				
ec-32bits	0.000 ± 0.00	0.000	0.000	4.4 ± 2.5
ec-24bits	0.000 ± 0.00	0.000	0.000	4.2 ± 2.8
dc	0.001 ± 0.01	0.000	0.121	2.0 ± 1.2
ec-16bits	0.003 ± 0.01	0.000	0.024	4.3 ± 2.6
ec-8bits	0.048 ± 0.09	0.000	0.693	8.5 ± 6.3
trad	0.089 ± 0.21	0.000	1.626	17.2 ± 8.9
prc	0.150 ± 0.26	0.000	1.293	10.9 ± 8.1
Constant 123000				
ec-32bits	0.000 ± 0.00	0.000	0.000	11.0 ± 2.8
ec-16bits	0.000 ± 0.00	0.000	0.001	12.7 ± 4.8
ec-24bits	0.000 ± 0.00	0.000	0.000	11.6 ± 2.9
ec-8bits	0.011 ± 0.04	0.000	0.356	15.2 ± 5.7
prc	0.061 ± 0.15	0.000	1.036	16.4 ± 7.3
trad	0.075 ± 0.11	0.000	0.429	23.1 ± 7.1
dc	0.637 ± 3.23	0.000	18.699	2.2 ± 2.1
Constant 0.000123				
ec-32bits	0.000 ± 0.00	0.000	0.000	10.8 ± 3.1
ec-16bits	0.000 ± 0.00	0.000	0.001	11.5 ± 3.9
ec-24bits	0.000 ± 0.00	0.000	0.000	11.4 ± 3.3
dc	0.001 ± 0.01	0.000	0.116	4.0 ± 1.5
prc	0.048 ± 0.14	0.000	1.278	17.3 ± 6.8
ec-8bits	20.012 ± 40.20	0.000	100.000	12.6 ± 8.1
trad	98.029 ± 13.86	0.046	100.000	4.5 ± 7.2
Constant 45.60099				
ec-32bits	0.000 ± 0.00	0.000	0.000	1.7 ± 1.9
ec-24bits	0.000 ± 0.00	0.000	0.000	1.8 ± 2.4
dc	0.001 ± 0.00	0.000	0.013	1.4 ± 0.9
ec-16bits	0.001 ± 0.00	0.000	0.001	2.1 ± 3.0
ec-8bits	0.131 ± 0.12	0.000	0.243	5.6 ± 7.3
prc	0.161 ± 0.26	0.000	1.215	10.9 ± 10.5
trad	0.259 ± 0.43	0.000	1.318	17.0 ± 8.7
Constant 45600.99				
ec-32bits	0.000 ± 0.00	0.000	0.000	8.9 ± 2.8
ec-16bits	0.000 ± 0.00	0.000	0.001	10.3 ± 3.7
ec-24bits	0.000 ± 0.00	0.000	0.000	10.2 ± 3.8
dc	0.001 ± 0.00	0.000	0.002	1.3 ± 0.9
ec-8bits	0.017 ± 0.03	0.000	0.250	12.6 ± 5.5
prc	0.068 ± 0.17	0.000	1.345	15.9 ± 7.6
trad	0.109 ± 0.17	0.000	0.715	21.3 ± 7.1
Constant 0.0004560099				
ec-32bits	0.000 ± 0.00	0.000	0.002	10.5 ± 3.0
ec-16bits	0.000 ± 0.00	0.000	0.001	11.5 ± 3.6
ec-24bits	0.000 ± 0.00	0.000	0.000	10.8 ± 3.0
dc	0.002 ± 0.01	0.000	0.093	3.9 ± 1.5
prc	0.078 ± 0.27	0.000	2.572	18.2 ± 8.2
ec-8bits	11.018 ± 31.44	0.000	100.000	13.4 ± 7.4
trad	92.027 ± 27.17	0.000	100.000	5.9 ± 9.0

Table 1: Results

Of course, more experimentation is needed to verify whether our method will outperform its counterparts when more complex solutions are to be evolved.

6. ACKNOWLEDGMENTS

The authors thanks the support from CAPES, CNPq (308317/2009-2) and FAPERJ (grants E-26/102.825/2008, E-26/102.025/2009 and E-26/100.308/2010).

7. REFERENCES

[1] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[2] M. O'Neill, I. Dempsey, A. Brabazon, and C. Ryan. Analysis of a digit concatenation approach to constant creation. In *Proc. of the European Conference on Genetic Programming*, pages 173–182. Springer, 2003.

[3] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Trans. Evol. Comput.*, 5(4):349–358, 2001.