Analysis of a Digit Concatenation Approach to Constant Creation

Michael O'Neill[1], Ian Dempsey[1], Anthony Brabazon[2], and Conor Ryan[1]

[1] Dept. Of Computer Science & Information Systems
University of Limerick, Ireland
Michael.ONeill@ul.ie, Conor.Ryan@ul.ie
[2] Dept. Of Accountancy, University College Dublin, Ireland
Anthony.Brabazon@ucd.ie

Abstract. This study examines the utility of employing digit concatenation, as distinct from the traditional expression based approach, for the purpose of evolving constants in Grammatical Evolution. Digit concatenation involves creating constants (either whole or real numbers) by concatenating digits to form a single value. The two methods are compared using three different problems, which are finding a static real constant, finding dynamic real constants, and a quadratic map, which on iteration generates a chaotic time-series. The results indicate that the digit concatenation approach results in a significant improvement in the best fitness obtained across all problems analysed here.

1 Introduction

The objective of this study is to determine whether the adoption of a novel approach to constant creation by digit concatenation can outperform the more traditional expression based approach to constant creation in Grammatical Evolution, that relies on the recombination of constants using the functions and operators provided. Digit concatenation involves creating constants, which can be either whole or real numbers, by concatenating digits to form a single value.

Existing applications of digit concatenation in Grammatical Evolution adopted this approach to constant creation in the automatic generation of caching algorithms, and a financial prediction problem [10,3]. This paper extends these previous studies by conducting an analysis of the digit concatenation approach in comparison to the more traditional expression based approach using specific constant creation problem domains. The two problem domains tackled previously did not exploit constant creation to a substantial degree in successful solutions, hence the need to conduct our current investigation on different problems.

1.1 Background

Ephemeral random constants are the standard approach to constant creation in Genetic Programming (GP), having values created randomly within a prespecified range at a runs initialisation [6]. These values are then fixed throughout

C. Ryan et al. (Eds.): EuroGP 2003, LNCS 2610, pp. 173–182, 2003.
© Springer-Verlag Berlin Heidelberg 2003

a run, and new constants can only be created through combinations of these values and other items from the function and terminal set.

Since then there have been a number of variations on the ephemeral random constant idea in tree-based GP systems, all of which have the common aim of making small changes to the initial constant values created in an individual.

Constant perturbation [14] allows GP to fine-tune floating point constants by multiplying every constant within an individual by a random number between 0.9 and 1.1, having the effect of modifying a constants value by up to 10% of their original value.

Numerical terminals and a *numerical terminal mutation* were used in [1] instead of ephemeral random constants, the difference being that the numerical terminal mutation operator selects a real valued numerical terminal in an individual and adds to it Gaussian noise with a particular variance, such that small changes are made to the constant values.

A *numeric mutation* operator, that replaces all of the numeric constants in an individual with new ones drawn at random from a uniform distribution within a specified selection range, was introduced in [4]. The selection range for each constant is specified as the old value of that constant plus or minus a temperature factor. This method was shown to produce a statistically significant improvement in performance on a number of symbolic regression problems ranging in difficulty.

1.2 Structure of Paper

This contribution is organised as follows. Section 2 provides a short introduction to Grammatical Evolution. Section 3 describes the problem domains and the experimental approach adopted in this study. Section 4 provides the results under each of the grammars. Finally, conclusions and an outline of future work are provided in Section 5.

2 Grammatical Evolution

Grammatical Evolution (GE) [12,11,9] is an evolutionary algorithm that can evolve computer programs in any language. Rather than representing the programs as parse trees, as in GP [6], a linear genome representation is used. Each individual, a variable length binary string, contains in its codons (groups of 8 bits in these experiments) the information to select production rules from a Backus Naur Form (BNF) grammar. BNF is a notation that represents a language in the form of production rules. It is comprised of a set of non-terminal symbols that can be mapped to elements from the set of terminal symbols, according to the production rules. An example excerpt from a BNF grammar is given below.

These productions state that `S` can be replaced with either one of the non-terminals `expr`, `if-stmt`, or `loop`.

```
S ::= expr      (0)
    | if-stmt   (1)
    | loop      (2)
```

The grammar is used in a generative process to construct a program by applying production rules, selected by the genome, beginning from a given start symbol (S in this case).

In order to select a rule in GE, the next codon value on the genome is generated and placed in the following formula:

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

For example, if the next available codon integer value was 4, given that we have 3 rules to select from as in the above example, we get $4\ MOD\ 3\ =\ 1$. S will therefore be replaced with the non-terminal if-stmt.

Beginning from the the left hand side of the genome codon integer values are generated and used to select rules 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 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. A full description of GE can be found in [11].

3 Problem Domain and Experimental Approach

In this study, we compare the utility of different grammars for evolving constants by performance analysis on three different types of constant creation problems. The problems tackled are, Finding a Static Real Constant, Finding Dynamic Real Constants, and the Logistic Equation. A description of each problem follows.

3.1 Finding a Static Real Constant

The aim of this problem is to evolve a single real constant. Three target constants of increasing difficulty were selected arbitrarily, 5.67, 24.35, and 20021.11501. Fitness in this case is the absolute difference between the target and evolved values, the goal being to minimise this difference value.

3.2 Finding Dynamic Real Constants

This instance of finding dynamic real constants involves a dynamic fitness function that changes its target real constant value at regular intervals (every 10th generation). Two instances of this problem are tackled, the first sets the successive target values to be 24.35, 5.67, 5.68, 28.68, 24.35, and the second instance

oscillates between the two values 24.35 and 5.67. The aim with these problems is to analyse the different constant representations in terms of their ability to adapt to a changing environment, and to investigate that behaviour in the event of both small and large changes. As in the finding static real constant problem, fitness in this case is the absolute difference between the target and evolved values, with the goal being the minimisation of this difference value.

3.3 The Logistic Equation

In systems exhibiting chaos, long-term prediction is problematic as even a small error in estimating the current state of the system leads to divergent system paths over time. Short-term prediction however, may be feasible [5]. Because chaotic systems provide a challenging environment for prediction, they have regularly been used as a test-bed for comparative studies of different predictive methodologies [8,2,13]. In this study we use time-series information drawn from a simple quadratic equation, the logistic difference equation[1]. This equation has the form:

$$x_{t+1} = \alpha x_t(1 - x_t) \qquad x \in (0.0, 1.0)$$

The behaviour of this equation is crucially driven by the parameter α. The system has a single, stable fixed point (at $x = (\alpha - 1)/\alpha$)for $\alpha < 3.0$ [13]. For $\alpha \in (3.0, \approx 3.57)$ there is successive period doubling, leading to chaotic behaviour for $\alpha \in (\approx 3.57, 4.0)$. Within this region, the time-series generated by the equation displays a variety of periodicities, ranging from short to long [7]. In this study, three time-series are generated for differing values of α. The choice of these values is guided by [7], where it was shown that the behaviour of the logistic difference equation is qualitatively different in three regions of the range (3.57 to 4.0). To avoid any bias which could otherwise arise, parameter values drawn from each of these ranges are used to test the constant evolution grammars. The goal in this problem is to rediscover the original α value. As this equation exhibits chaotic behaviour, small errors in the predicted values for α will exhibit increasingly greater errors, from the target behaviour of this equation, with each subsequent time step. Fitness in this case is the mean squared error, which is to be minimised. 100 initial values for x_t were used in fitness evaluation, and for each x_t iterating 100 times (i.e. x_t to x_{t+100}).

3.4 Constant Creation Grammars

The grammars adopted are given below. The concatenation grammar (Cat) only allows the creation of constants through the concatenation of digits, this is in contrast to the Traditional grammar (Trad) that restricts constant creation to the generation of values from expressions. The third grammar analysed here is the Traditional & Concatenation Combination grammar (Cat+Trad), which

[1] This is a special case of the general quadratic equation $y = ax^2 + bx + c$ where $c = 0$ and $a = -b$.

allows the use of both the digit concatenation and expression based constant creation approaches. The fourth grammar (Trad+Real) provides real values to the Trad grammar, giving an explicit mechanism for creating real values without relying on the arithmetic operators.

Concatenation (Cat) Grammar

```
value : real

real: int dot int | int

int: int number | number

number: 0 | 1 | 2 | 3 | 4 | 5
      | 6 | 7 | 8 | 9

dot: .
```

Traditional (Trad) Grammar

```
value: value op value
     | ( value op value )
     | number

op: + | - | / | *

number: 0 | 1 | 2 | 3 | 4 | 5
      | 6 | 7 | 8 | 9
```

Traditional & Concatenation Combination (Cat+Trad) Grammar

```
value: value op value
     | ( value op value )
     | real

op: + | - | / | *

real: int dot int | int

int: int number | number

number: 0 | 1 | 2 | 3 | 4 | 5
      | 6 | 7 | 8 | 9

dot: .
```

Traditional & Real Combination (Trad+Real) Grammar

```
value: value op value
     | ( value op value )
     | number
     | real

op: + | - | / | *

number: 0 | 1 | 2 | 3 | 4 | 5
      | 6 | 7 | 8 | 9

real : .1 | .2 | .3 | .4 | .5
      | .6 | .7 | .8 | .9
```

4 Results

For each grammar on every problem instance, 30 runs were conducted using population sizes of 500, running for 50 generations on the static and dynamic constant problems, and 100 generations for the logistic equation, adopting one-point crossover at a probability of 0.9, and bit mutation at 0.1, along with roulette selection and a replacement strategy where 25% of the population is replaced each generation. The crossover operator was allowed to select crossover points within the 8-bit codons adopted here. The results are as follows.

4.1 Finding a Static Real Constant

On all three instances of this problem, a t-test and bootstrap t-test (5% level) on the best fitness values reveal that the digit concatenation grammars (Cat & Cat+Trad) significantly outperform the standard expression based approach (Trad & Trad+Real) to constant creation through expressions. Statistics of performance for each grammar are given in Table 1, and a plot of the mean best fitness at each generation for the three grammars analysed can be seen in Fig. 1.

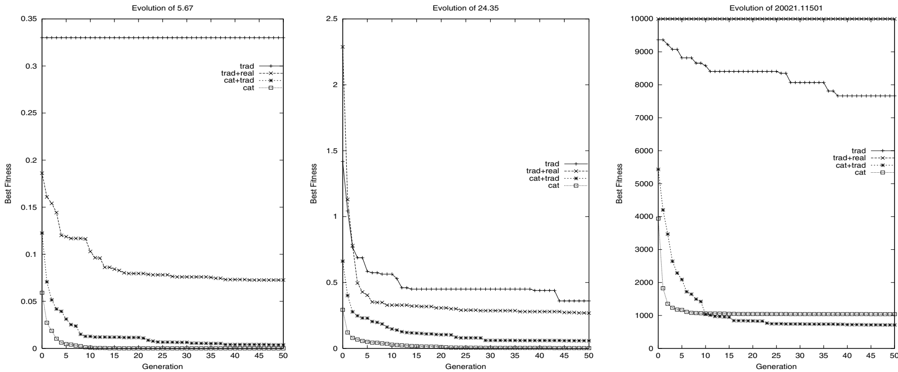

Fig. 1. Mean best fitness values (lower value is better) plotted against generations for each of the three grammars. Target values are 5.67 (left), 24.35 (center), and 20021.11501 (right).

Table 1. Statistics for best fitness values (lower value is better) at generation 50 on the Static Real Constant Problem.

Target Constant	Grammar	**Mean**	Median	*Std. Dev.*
5.67	Trad	**0.33**	0.33	*0.0*
	Trad+Real	**0.071**	0.03	*0.118*
	Cat+Trad	**0.004**	0.0	*0.017*
	Cat	**0.0**	0.0	*0.0*
24.35	Trad	**0.36**	0.35	*0.055*
	Trad+Real	**0.261**	0.35	*0.205*
	Cat+Trad	**0.057**	0.01	*0.081*
	Cat	**0.002**	0.0	*0.009*
20021.11501	Trad	**7741.35**	1.000e+04	*3828.9*
	Trad+Real	**1.000e+04**	1.000e+04	*0.0*
	Cat+Trad	**689.01**	2.117e+01	*2531.2*
	Cat	**1005.24**	9.100e-01	*3049.5*

Interestingly, the Trad+Real grammar did not perform as well as the Trad grammar on the hardest of these three problem instances, while the Cat and Cat+Trad grammars performance was statistically the same. This demonstrates that a grammar that has a concatenation approach to constant creation is significantly better at generating larger numbers[2].

4.2 Finding Dynamic Real Constants

For the first instance of this problem where the successive target constant values are 24.35, 5.67, 5.68, 28.68, 24.35 over the course of 50 generations, performance

[2] It is worth stressing that larger numbers could just as easily be large whole numbers or numbers with a high degree of precision (reals).

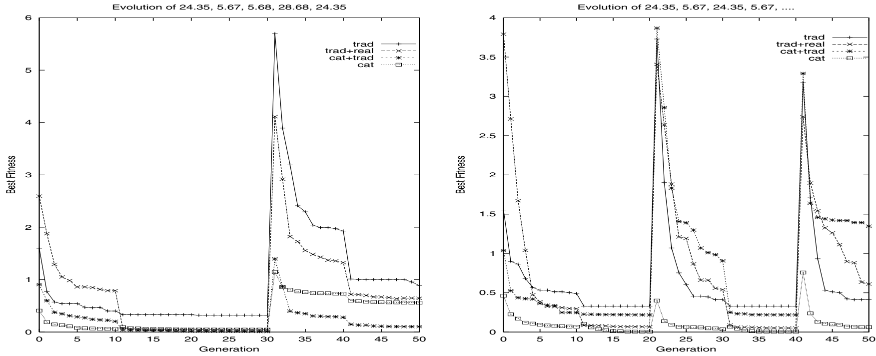

Fig. 2. Mean best fitness values (lower value is better) plotted against generations for each of the three grammars. Target values are 24.35, 5.67, 5.68, 28.68, 24.35 (left), and 24.35, 5.67,.. (right).

Table 2. Statistics for best fitness values (lower value is better) on the Dynamic Real Constant Problem (Target Constants: 24.35, 5.67, 5.68, 28.68, 24.35).

Generation	Target Constant	Grammar	**Mean**	Median	*Std. Dev.*
10	24.35	Trad	**0.4**	0.35	*0.114*
		Trad+Real	**0.766**	0.35	*1.539*
		Cat+Trad	**0.219**	0.11	*0.296*
		Cat	**0.061**	0.01	*0.133*
20	5.67	Trad	**0.33**	0.33	*0.0*
		Trad+Real	**0.05**	0.03	*0.078*
		Cat+Trad	**0.017**	0.006	*0.025*
		Cat	**0.047**	0.0	*0.17*
30	5.68	Trad	**0.32**	0.32	*1.129e-16*
		Trad+Real	**0.04**	0.02	*7.7499e-02*
		Cat+Trad	**0.009**	0.001	*2.283e-02*
		Cat	**0.046**	0.0	*1.724e-01*
40	28.68	Trad	**2.063**	1.5	*3.474*
		Trad+Real	**1.356**	0.68	*2.581*
		Cat+Trad	**0.283**	0.16	*0.347*
		Cat	**0.707**	0.007	*3.585*
50	24.35	Trad	**0.937**	0.35	*2.755*
		Trad+Real	**0.638**	0.3	*1.56*
		Cat+Trad	**0.101**	0.05	*0.244*
		Cat	**0.541**	0.002	*2.799*

statistics are given in Table 2, and a plot of mean best fitness values for each grammar can be seen in Fig. 2 (left).

Performing a t-test and bootstrap t-test on the best fitness values at generations 10, 20, 30, 40 and 50, it is shown that there is a significant (5% level) performance advantage in favour of the concatentation grammars (Cat & Cat+Trad) up to generation 30, beyond this point the advantages of one grammar over the

Table 3. Statistics for best fitness values (lower value is better) on the Oscillating Dynamic Real Constant Problem (Target Constants: 24.35, 5.67, 24.35, 5.67, 24.35).

Generation	Target Constant	Grammar	**Mean**	Median	*Std. Dev.*
10	24.35	Trad	**0.507**	0.35	*0.426*
		Trad+Real	**0.302**	0.35	*0.148*
		Cat+Trad	**0.252**	0.35	*0.143*
		Cat	**0.089**	0.011	*0.193*
20	5.67	Trad	**0.33**	0.33	*0.0*
		Trad+Real	**0.065**	0.03	*0.092*
		Cat+Trad	**0.222**	0.33	*0.156*
		Cat	**0.005**	0.0	*0.0167*
30	24.35	Trad	**0.487**	0.35	*0.426*
		Trad+Real	**0.55**	0.35	*1.113*
		Cat+Trad	**0.963**	0.35	*2.765*
		Cat	**0.046**	0.022	*0.07*
40	5.67	Trad	**0.33**	0.33	*0.0*
		Trad+Real	**0.050**	0.03	*0.077*
		Cat+Trad	**0.222**	0.33	*0.155*
		Cat	**0.004**	0.0	*0.010*
50	24.35	Trad	**0.487**	0.35	*0.426*
		Trad+Real	**0.625**	0.35	*1.53*
		Cat+Trad	**1.358**	0.35	*3.815*
		Cat	**0.061**	0.014	*0.131*

other are not as clear cut. Given the dynamic nature of this problem other issues such as loss of diversity may be coming into play, possibly obfuscating any effect of the different constant generation techniques.

In the second instance of this problem, where the target constant value oscillates, every 10 generations, between 24.35 and 5.67 over the 50 generations, again we see a similar trend. In this case, the concatenation grammar (Cat) is significantly better (based on best fitness analysis using t-tests and bootstrap t-tests at the 5% level) than all the other constant creation grammars at each of 10, 20, 30, 40 and 50 generations, however, this difference is decreasing over time. Again, loss of diversity over time is most likely playing a role here. A plot of the mean best fitness can be seen in Fig. 2 (right), and statistics are presented in Table 3.

From the results on both of these dynamic problem instances, there are clearly adaptive advantages to using the concatenation grammar over the traditional expression based approach.

4.3 The Logistic Equation

The results for all three instances of this problem can be seen in Table 4 and Fig. 3. Statistical analysis using a t-test and bootstrap t-test (5% level) reveal that the concatenation grammars (Cat & Cat+Trad) significantly outperform

Table 4. Statistics for best fitness values (lower value is better) at generation 100 on the Logistic Equation Problem.

Target Constant	Grammar	**Mean**	Median	*Std. Dev.*
3.59	Trad	**6.074e-03**	6.074e-03	*2.647e-18*
	Trad+Real	**2.203e-04**	3.613e-06	*1.108e-03*
	Cat+Trad	**1.109e-05**	8.256e-13	*5.321e-05*
	Cat	**4.818e-07**	3.902e-19	*1.249e-06*
3.80	Trad	**1.310e-03**	1.310e-03	*6.616e-19*
	Trad+Real	**5.715e-04**	1.485e-06	*0.001*
	Cat+Trad	**4.724e-19**	4.724e-19	*0.0*
	Cat	**4.724e-19**	4.724e-19	*0.0*
3.84	Trad	**7.113e-04**	7.113e-04	*2.206e-19*
	Trad+Real	**4.146e-04**	7.113e-04	*3.457e-04*
	Cat+Trad	**6.564e-05**	6.065e-19	*2.017e-04*
	Cat	**6.065e-19**	6.065e-19	*9.794e-35*

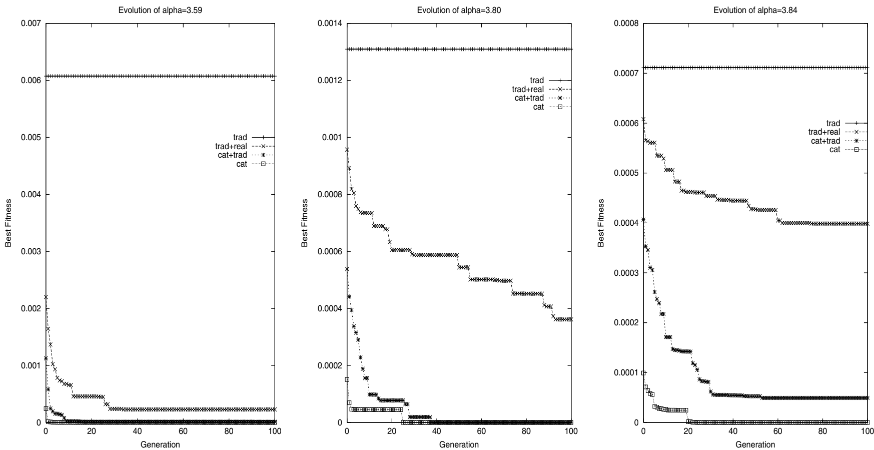

Fig. 3. Mean best fitness values (lower value is better) plotted against generations for each of the three grammars. Target α values are 3.59 (left), 3.80 (center), and 3.84 (right).

the traditional constant creation approach on each problem instance, successfully rediscovering the target α value in each case.

5 Conclusions and Future Work

An analysis of a digit concatenation approach to constant creation in Grammatical Evolution is presented. In general, the performance of concatenation grammars across the three problem domains investigated here, exhibits significantly improved fitness when compared to the more traditional expression based constant creation approach.

We now intend to extend this study by conducting a comparison of digit concatenation to an equivalent version of ephemeral random constants in Grammatical Evolution, and to look at a broader set of problem domains.

References

1. Angeline, Peter J. (1996). Two Self-Adaptive Crossover Operators for Genetic Programming. In Peter J. Angeline and K. E. Kinnear, Jr. (Eds.), Advances in Genetic Programming 2, Chapter 5, pp.89-110, MIT Press.
2. Castillo, E. and Gutierrez, J. (1998). Nonlinear time series modeling and prediction using functional networks. Extracting information masked by chaos, *Physics Letters A*, 244:71-84.
3. Dempsey, I., O'Neill, M. and Brabazon, T. (2002). Investigations into Market Index Trading Models Using Evolutionary Automatic Programming, In *Lecture Notes in Artificial Intelligence, 2464*,Proceedings of the 13th Irish Conference in Artificial Intelligence and Cognitive Science, pp. 165-170, edited by M. O'Neill, R. Sutcliffe, C. Ryan, M. Eaton and N. Griffith, Berlin: Springer-Verlag.
4. Evett, Matthew and Fernandez, Thomas. (1998). Numeric Mutation Improves the Discovery of Numeric Constants in Genetic Programming, Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, USA, pp.66-71, Morgan Kaufmann.
5. Holland, J. (1998). *Emergence from Chaos to Order*,Oxford: Oxford University Press.
6. Koza, J. (1992). *Genetic Programming*. MIT Press.
7. May, R. (1976). Simple mathematical models with very complicated dynamics, *Nature*, 261:459-467.
8. Nie, J. (1997). Nonlinear time-series forecasting: A fuzzy-neural approach, *Neurocomputing*, 16:63-76.
9. O'Neill, M. (2001). Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution. PhD thesis, University of Limerick, 2001.
10. O'Neill, M., Ryan, C. (1999). Automatic Generation of Caching Algorithms, In K. Miettinen and M.M. Mäkelä and J. Toivanen (Eds.) Proceedings of EUROGEN99, Jyväskylä, Finland, pp.127-134, University of Jyväskylä.
11. O'Neill, M., Ryan, C. (2001) Grammatical Evolution, *IEEE Trans. Evolutionary Computation*, 5(4):349-358, 2001.
12. Ryan C., Collins J.J., O'Neill M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Lecture Notes in Computer Science 1391, Proceedings of the First European Workshop on Genetic Programming*, 83-95, Springer-Verlag.
13. Saxen, H. (1996). On the approximation of a quadratic map by a small neural network, *Neurocomputing*, 12:313-326.
14. Spencer, G. (1994). Automatic Generation of Programs for Crawling and Walking. In Kenneth E. Kinnear, Jr. (Ed), Advances in Genetic Programming, Chapter 15, pp. 335-353, MIT Press.