The Best Things Don't Always Come in Small Packages: Constant Creation in Grammatical Evolution

R. Muhammad Atif Azad and Conor Ryan

CSIS Department, University of Limerick, Ireland
{atif.azad,conor.ryan}@ul.ie
http://bds.ul.ie

Abstract. This paper evaluates the performance of various methods to constant creation in Grammatical Evolution (GE), and validates the results against those from Genetic Programming (GP). Constant creation in GE is an important issue due to the disruptive nature of *ripple crossover*, which can radically remap multiple terminals in an individual, and we investigate if more compact methods, which are more similar to the GP style of constant creation (*Ephemeral Random Constants* (ERCs), perform better.

The results are surprising. The GE methods all perform significantly better than GP on unseen test data, and we demonstrate that the standard GE approach of *digit concatenation* does not produce individuals that are any larger than those from methods which are designed to use less genetic material.

Keywords: Grammatical Evolution, Constants, Symbolic Regression, Genetic Programming, Digit Concatenation.

1 Introduction

Typically, symbolic regression finds a function to explain a given data set. In traditional Machine Learning [1] this involves optimising the parameters of a predefined *objective* function using an Artificial Neural Network (ANN), Support Vector Machine (SVM) or some other numerical method. These methods work efficiently partly because, with a known target function, they only explore the parameter space to minimise the error between expected and predicted outputs.

Genetic Programming (GP) takes symbolic regression to another level: it explores both the space of functions and the associated parameters (constants) simultaneously. Therefore, finding suitable numeric constants is essential to how GP performs. However, GP typically does not involve specialised mechanisms for optimising numeric constants. Instead, GP uses *ephemeral random constants (ERCs)* [2], that randomly initialise numeric terminal nodes in a GP population. Thereafter, genetic operators recombine and filter out (possibly erroneously) these ERCs. The combined tasks of optimising structure and constants can be very difficult: for example Keijzer [3] noted that given a target function of $100+x^2$

M. Nicolau et al. (Eds.): EuroGP 2014, LNCS 8599, pp. 186–197, 2014.

such that $x \in [-1, 1]$, GP approximated the numeric constant 100 but lost the genetic material to encode x^2. To combat this, Keijzer proposed *linear scaling*, a form of linear regression to optimise the slope and intercept of evolving GP functions to assist GP. Other proposals include numerical methods [4][5] as well as specialised mutation operators [6][7].

This paper investigates methods for evolving constants in Grammatical Evolution (GE) [8] on a number of problems from the symbolic regression domain. GE is a genetic programming system that maps a genotype, a linear string of 8 bit integers termed *codons*, to a functional expression from a language of choice, which is defined by a context free grammar (CFG). Usually, GE uses *digit concatenation* [9] to evolve constants. In this method, a string of GE codons select the constant defining rules from a grammar to yield the desired constant.

Since digit concatenation uses several codons to produce a number, that number can change when passed onto offspring, unlike a number encoded in a more compact way, i.e. as in GP. This is due to the so-called *ripple effect* of GE crossover, [10] which propagates changes to genetic material from left to right. We compare digit concatenation to two other *compact* methods that do not require several codons to encode a constant: these are, a GE version of ERCs called *persistent random constants (PRCs)* [9] and a *codon injection* method that directly converts a GE codon into a floating point value.

This work goes further than previous studies which focused on evolving solutions which were a single constant [11–13], because, as [6] notes, optimising constants alongside mathematical functions is a different challenge and, we believe, more relevant to the GP community. One early related work [9] showed the utility of digit concatenation to a few instances of *Binomial-3* problem [14]; here we *also* consider other problems. Moreover, we compare different methods both with and without linear scaling and also compare against the benchmark results from GP because GP is commonly used for symbolic regression. Finally, previous work solely compared **training** results; instead, we also consider unseen **test** data as well as genome lengths of the individuals to ascertain if the compact methods breed relatively more predictive and compact genomes.

The results show that GP consistently outperforms GE on training data; however, on the test data, GE, regardless of the constant creating method, does better. However, among themselves, the various GE methods perform equally well on all the criteria. Notably, the genome lengths with digit concatenation are no greater than those with the compact methods. Moreover, using GP-like PRCs does not bridge the gap in training results of GP and GE, which suggests that the key difference between GP and GE is how the respective genetic operators behave. We also conclude that the compact methods are not *effectively* compact, give our reasons for that and give directions for further work.

The rest of the paper is organised as follows: section 2 gives a background to constant creating methods in GE and builds a motivation to this study; section 3 describes the experimental setup, presents the results and discusses the lessons we can learn from these results; finally, section 4 concludes the paper.

2 Background

Digit concatenation with GE [11] [9] requires a CFG with appropriate rules for generating numeric constants. For example, with the grammars below and a rule `<expr> ::= <const> | -<const>`, `cat-UnLtd` can, in theory, encode any real constant, whereas `cat-0-to-5` limits the values to the domain $(-5, 5)$.

```
cat-UnLtd:                          cat-0-to-5:

<const> ::= <cat>.<cat>             <const>::= <fdig>.<cat>
<cat>  ::= <cat><digit>|<digit>     <cat>  ::= <cat><digit> | <digit>
<digit>::= 0 | 1 | 2 | 3 | 4       <fdig> ::= 0 | 1 | 2 | 3 | 4
        | 5 | 6 | 7 | 8 | 9        <digit> ::= 0 | 1 | 2 | ... | 9
```

This approach has some side effects. First, the number of codons GE takes to encode a constant is equal to the number of digits in it. Later, crossover can break the constant so that it does not pass on to the offspring intact. This is unlike as in GP, where an ERC is atomic. Thus, a stronger *causality* exists in GP, where offspring are likelier to resemble their parents. In fact, as noted in [6], a small number of ERCs quickly dominate the population, with many appearing multiple times in later generations. This is what initially motivated us to ask if GE can benefit from a more GP-like approach, as it appears as though GP first settles on the constants and then builds structure (functions) around them. Second, GE is free to encode a greater number of digits than that allowed by the underlying machine architecture, and as the machine ignores these additional digits, they provide a bloating opportunity. Thus, the next question is: does digit concatenation produce longer genomes than those with an ERC type approach?

To answer these questions we consider two *compact* representations for GE constants. The first, termed *persistent random constants* (PRCs) [11] embeds randomly generated constants (from a given range) inside the grammar as alternative choices. A single codon can pick a constant by selecting the corresponding rule. Previously digit concatenation outperformed PRCs when the objective was to evolve a single constant [11]. As the second method, we consider a *codon injection* method [15], whereby when the non-terminal `<const>` is read, the following 8 bit codon value is converted into a floating point value in a given range. As in [12, 13], only a single codon produces a numeric constant.

While previous work investigated evolving a fixed constant, this paper concerns the more traditional symbolic regression. We check if compact representations are *effectively* more compact: that is, whether these methods produce higher fitness *and* smaller genomes. We also note results on unseen data to see if any method produces better predictive models.

3 Experiments

For the *best fit* individual we note: score on training data (best fitness); score on unseen (test) data; and genome length. We record genome lengths to compare which method requires more genetic material. Digit concatenation takes multiple

codons to create a single constant (unless the constant has just a single digit); likewise, multiple PRCs may combine to create a constant. We record these statistics every generation and present their mean values over 100 independent runs. Moreover, we also record all the statistics with linear scaling.

Also, we use results for GP as a benchmark. Clearly, GP differs from GE in many ways: the genetic representation and genetic operators differ; consequently, we expect some difference in performance. However, since GP is more widely used for symbolic regression, we consider its results to validate the performance of GE. We want to see if the difference in performance is consistent (GP is always better or worse than GE), and whether using a relatively more GP-like approach with PRCs bridges the gap in performances of GP and GE.

We consider five different constant creating methods for GE. These are (legends in brackets): digit concatenation with constants from an infinite real domain (`cat-UnLtd`); digit concatenation with *absolute values* of constants limited to $(0, 5)$ (`cat-0-to-5`); 50 and 25 persistent random constants embedded in the grammar (`50-PRC-0-to-5` and `25-PRC-0-to-5`) also derived from $(0, 5)$; and the codon injection method that directly decodes a GE codon into a numeric value (`codon-0-to-5`). All these methods can also generate negative numbers.

The respective grammars incorporate problem specific input variables and arithmetic operators in a prefix notation.

3.1 Problem Suite and Evolutionary Parameters

All experiments use a population size of 500, roulette wheel selection, steady state replacement and crossover with a probability of 0.9. For GE, we use the conventional [8][16] bit wise mutation with a probability of 0.01, while for GP, we use point mutation with a standard probability value of 0.1 [2]. We use ramped half and half initialisation for GP with an initial maximum tree depth of 4; for GE we use the grammatical counterpart of this initialisation termed *sensible initialisation* [16]. Sensible initialisation uses a context free grammar to generate derivation trees for GE using a ramped half and half approach. We use a maximum initial depth of derivation trees of 10 (which is larger than 4 for GP) since a big derivation tree can still yield a small abstract syntax tree and GE grows trees at a slower rate than with standard GP [9].

Although we do not constrain tree sizes or maximum depth for GP (and GE), in the experiments reported here the average tree size for GP never exceeds 250; this is well below the maximum size allowed by a commonly used maximum tree depth of 17 for binary trees. Another side effect of a maximum tree depth is that it can prohibit extremely deep *skinny* trees. Deep skinny trees can encode a particularly non-linear behaviour which may promote overfitting the training data if the functions set contains unary transcendental functions [17]; however, we only use binary arithmetic functions in this study.

We use six different problems from the symbolic regression domain here. As Keijzer [3] notes, choosing a good set of problems for testing symbolic regression is difficult in the absence of an established set of benchmarks. Like Keijzer, we use the following problems from previous work on symbolic regression.

$$f(x) = 0.3xsin(2\pi x) \tag{1}$$
$$f(x) = 1 + 3x + 3x^2 + x^3 \tag{2}$$
$$f(x, y) = 8/(2 + x^2 + y^2) \tag{3}$$
$$f(x, y) = x^4 - x^3 + y^2/2 - y \tag{4}$$
$$f(x, y) = x^3/5 + y^3/2 - y - x \tag{5}$$
$$f(x_1, \cdots, x_{10}) = 10.59x_1x_2 + 100.5967x_3x_4 - 50.59x_5x_6 + 20x_1x_7x_9 + 5x_3x_6x_{10} \tag{6}$$

(1) comes from [18]; (2), also termed as *Binomial-3*, is a scalably difficult problem for GP [14] and has been investigated with GE [9]; (3), (4) and (5) come from [4]; and (6), referred to as *Poly-10* in the figures in this paper, is a version of a difficult problem described in [19]. The dimensionality of these problems varies between 1 and 10 and their difficulty to GP type approaches also varies as is visible from the scales of the best fitness plotted in Fig. 1.

We use a variant of the standard one point crossover for GE termed *effective crossover* [15]. Since the entire lengths of GE chromosomes may not be used for mapping, the non-mapping regions in GE chromosomes can grow larger and larger; this transforms crossover into a duplication operator as crossing over in the non-mapping regions does not innovate in the phenotype space. Therefore, the effective crossover restricts the crossover point to within the mapping regions.

As noted in [3], protected division (and protected operators in general) can lead GP to producing models that do not generalise well to unseen data; therefore, we do not use protected division. Instead, in the case of a division by zero, we penalise the offending individual by assigning it the worst fitness value of 0.0.

All the GE experiments use libGE [15], while the GP experiments use TinyGP[1]. Evolutionary runs terminate after completing 50 generations. GP uses 50 constants from the domain $(-5, 5)$ and like GE, only uses arithmetic operators.

For each problem, we randomly initialise input variables between -1.5 and 1.5 and generate 100 data points. We randomly choose 50 data points for training and an equal number of data points for testing on unseen data (test data).

3.2 Results

Figures 1-4 plot the results of the experiments. The x-axis consistently corresponds to 50 generations. The training and test scores are sums of squared errors (SSE) normalised between 0.0 and 1.0 (1.0 being the ideal score) as follows: $score = \frac{1}{1+SSE}$. Each sampled point in the plots depicts an average over 100 independent runs. As in [20], the 95% confidence limits of the error bars at each point are computed as follows: $\overline{X} \pm 1.96\frac{\sigma}{\sqrt{n}}$, where $\overline{X}$ and σ are the mean and standard deviation of n observations; $n = 100$ represents the number of runs in this case. We can be 95% confident that the statistical population lies within these limits, and that a lack of overlap with another error bar means that the corresponding populations are different.

[1] `http://cswww.essex.ac.uk/staff/rpoli/TinyGP/`

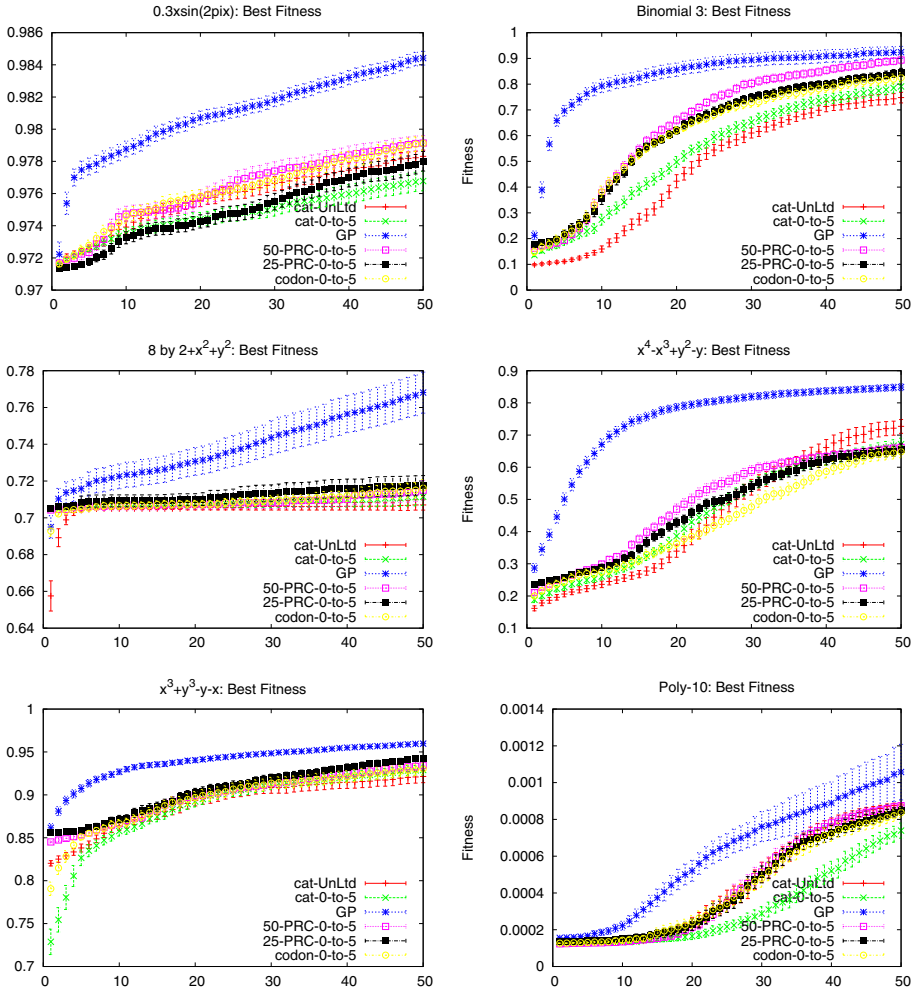

Fig. 1. This figure plots mean of best fitness achieved per generation for all the problems. No GE setup wins or loses consistently. On four problems, GP is significantly better than any GE method.

Figures 1-3 plot the results for experiments without using linear scaling. Fig. 1 plots the best fitness on *training* data and shows that none of the GE constant creating setups stands out consistently. In fact, various GE methods do quite similarly. Moreover, GP does at least as well as GE (and usually better). Also, using the PRCs does not bring GE any closer to GP.

Of particular interest is `cat-UnLtd`: unlike all other methods, GE chooses from an infinite domain of constants. Except for problem (6), a domain of [-5,5] is suitable and even advantageous. However, `cat_UnLtd` does no worse than the

other GE methods, suggesting that the larger range of constants available (and the correspondingly larger search space) poses no extra difficulty.

For problem (6) we also tried a domain of [-49,49] to assist methods other than `cat-UnLtd` in approximating important constants of 100 and 50 but even that did not change the relative performances; owing to space constraints we can not reproduce those results in this paper. Results also do not show that the brittle nature of constants with digit concatenation when facing crossover is a disadvantage any more than that with compact methods: both `cat_UnLtd` and `cat-0-to-5` perform competitively with respect to the compact methods.

Fig. 2 plots the results for the same individuals as in Fig. 1 on the unseen data. Again, no single GE method stands out. GP, however, changes behaviour on the unseen data: unlike on the training data where GP performed at *least* as well as GE methods, it now performs only at *most* as well as GE methods, and some times significantly worse. Again, using PRCs does not affect GE significantly.

Next, we check if digit concatenation costs more by requiring longer genomes. Fig. 3 plots the genome lengths for the best fit individuals and shows that again digit concatenation is no worse than the compact methods. Moreover, while GP genomes clearly grow towards the end of the runs, the lengths of GE genomes remain relatively stable after an initial growth or drop. Note that GE genomes encode derivation trees instead of abstract syntax trees (ASTs) in GP. However, the set of leaves of a GE derivation tree can be interpreted as an AST and this AST can be much smaller than the corresponding derivation tree; hence, at the end of the runs the ASTs encoded by GE derivation trees are smaller than those produced by GP even when the genome lengths are similar.

Next, we consider results with linear scaling. Due to space restrictions, we present results only on test data; we only summarise the results on training data and omit those on genome lengths because their relative trends are quite similar to those without linear scaling.

While, as expected, linear scaling helps improve best fitness for all the setups during training, the relative performances of various GE methods remain mutually competitive. Also, with linear scaling, the gap in the performance of GP and GE narrows towards the end of the run; however, again, none of the compact methods performs consistently better or worse than digit concatenation.

The scores on test data in Fig. 4 are also similar to those without linear scaling: again, all of the various GE setups perform competitively; similarly, GP performs at most as well as GE on the unseen data.

3.3 Discussion

The results from section 3.2 show that with the given evolutionary parameters and data sets, GE performs equally well with a variety of constant creating methods; however, GE differs significantly from GP. We only compared with GP to check if there is enough reason to improve GE so it can match the more widely prevalent method, that is, GP. The best fitness results, particularly without linear scaling, show that GP *trains* better than GE; however, it does so at the cost of degrading test set results. This is not altogether surprising given the growing

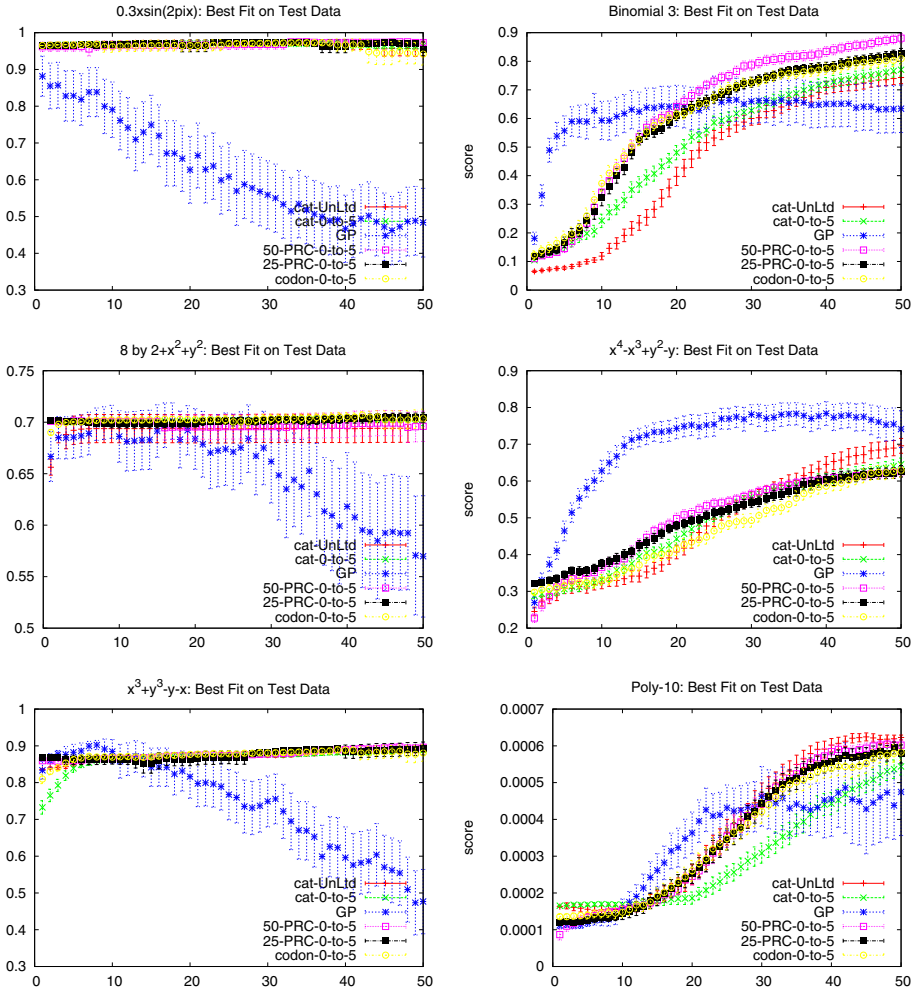

Fig. 2. This figure plots mean of Test Score per generation corresponding to Best-Fit individuals reported in Figure 1. No GE setup wins or loses consistently. On three problems, performance consistently degrades for GP.

GP literature which aims to improve performance on unseen data [17] [18]. What is surprising, however, is that GE does so much better, at least on these problems.

The real focus of this work, however, is on comparing various constant creating methods with GE. Digit concatenation is natural and easy to implement with GE; however, it can take many codons to encode a single constant. As a result, GE has to find a right sequence of codons and then ensure that crossover does not break that sequence. Moreover, with the ripple crossover [10] in GE, constants can not always transfer intact from the parent to offspring. However, the results

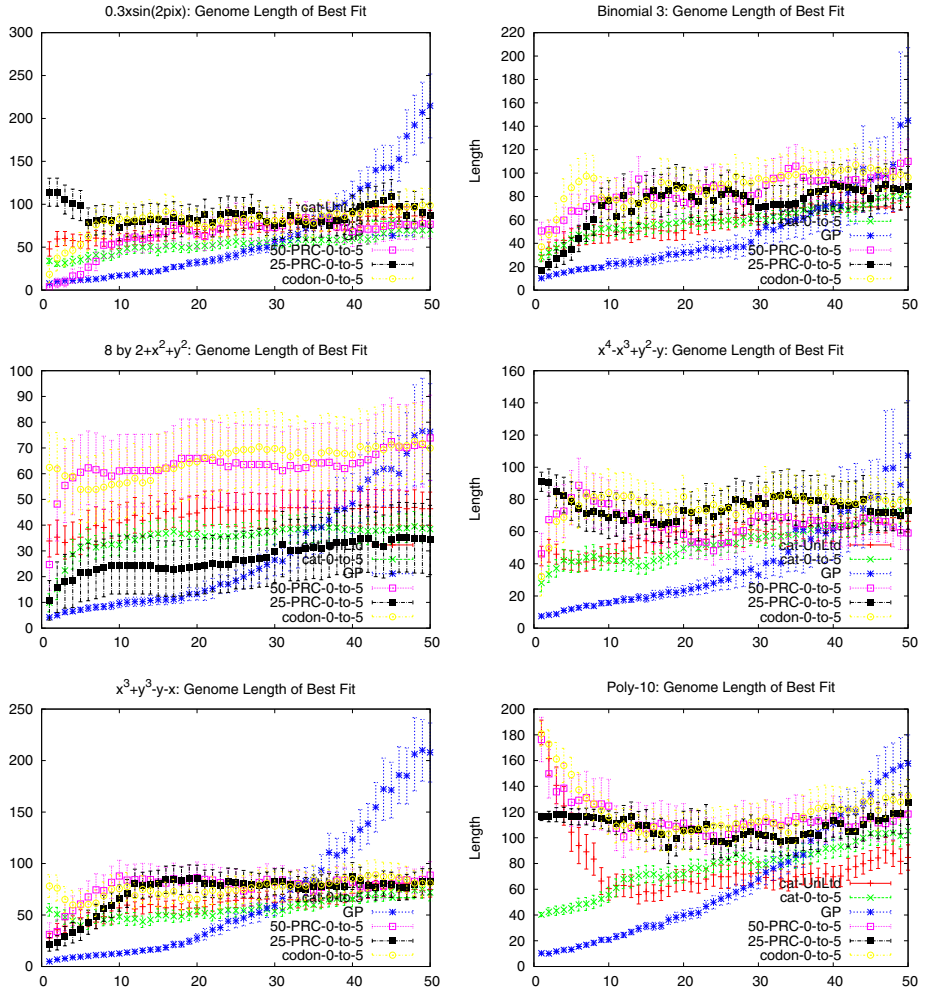

Fig. 3. This figure plots the mean genome lengths of the best fit individuals reported in Figure 1. No GE setup maintains significantly different lengths.

here show that the compact methods (PRCs and codon injection) do not *train* better than digit concatenation; this agrees with results in [9] [11]. However, on a greater set of problems, we additionally find that compact methods produce neither smaller genomes (surprisingly) nor better test set results.

The question then is: why does digit concatenation work as well as the other methods? There can be two reasons. First, even with the compact methods if the desired constant is not available, evolution combines various constants to get the right one. Thus, PRCs, or ERCs in GP, are not always less *breakable* with crossover. Secondly, [21, pp151-153] showed for a symbolic regression problem

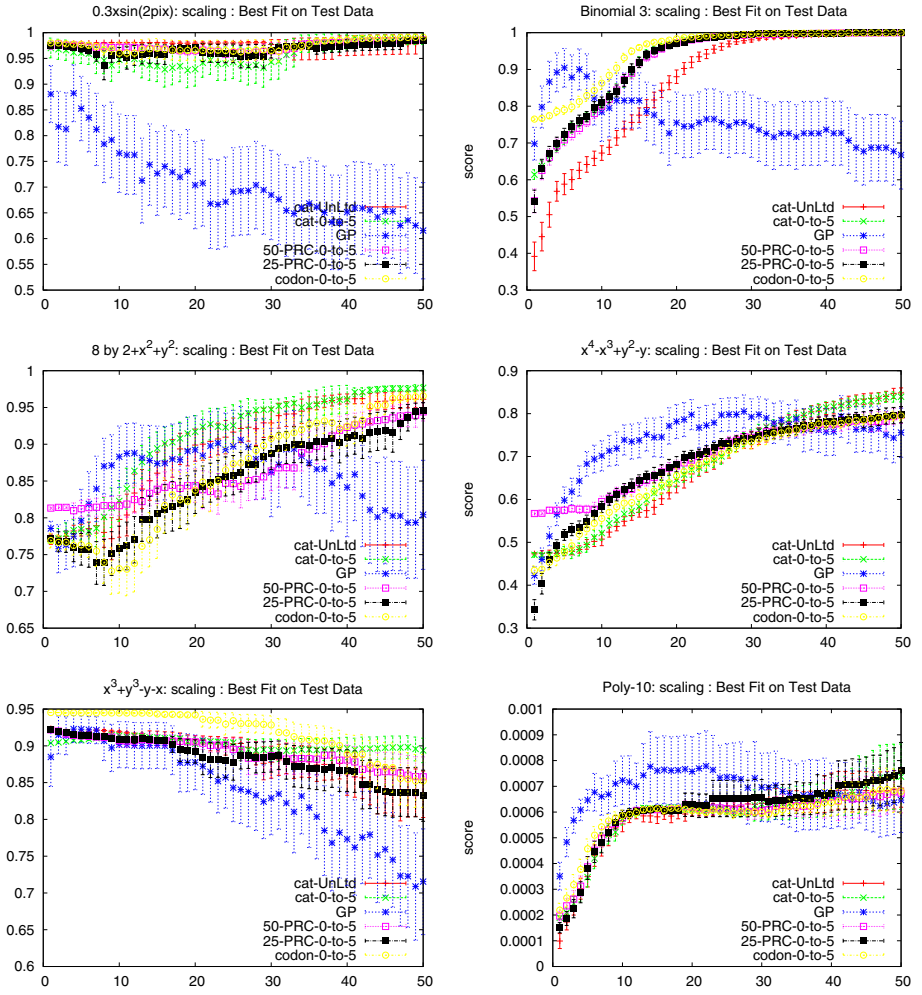

Fig. 4. (With Linear Scaling): this figure plots mean of Test Score per generation for the Best-Fit individuals. No GE setup wins or loses consistently.

that crossover mostly produces offspring with significantly worse fitness values. Also, [6] showed that even with a careful numeric mutation that only slightly changes the constants in a GP tree, crossover does no better than with random mutation that uniformly replaces a constant from within a given range. Despite that GP trains well in this paper. This suggests that passing constants from the parents to offspring is not crucial to GP: after all, even a constant ideal for a parent may be totally unsuitable for the offspring.

Therefore, to improve the performance of GE on symbolic regression there are two ways forward. First, find a crossover operator that recombines individuals in

a more favourable way, although this issue is not unique to just GE. The second is to assist the genetic operators with numerical methods such as in [3] [4] [5].

4 Conclusions

This paper compares the so-called digit concatenation method of creating constants in Grammatical Evolution with what this paper calls the compact methods to creating constants. The paper raises two questions: first, whether the constants with digit concatenation are so brittle against crossover that taking a more GP-like approach to constants with compact methods improves the performance of GE; and second, whether digit concatenation breeds longer genomes than those with compact methods. The results from the problems considered in this paper suggest that the answer to both the aforementioned questions is a resounding no. Because compact representations may also have to *synthesise* a constant when a suitable one is not available, we hypothesise that these constants are also not robust enough to outperform digit concatenation.

A fascinating result is that, although GP outperforms GE on training data, GE actually does substantially better on unseen test data.

The next steps in this research will be to do further critical evaluation of the performance of GE on test data, as well as its ability to generalise. In particular, work such as [17] [18] should be added to GE to ascertain if GE enjoys the same benefits that GP does from them. Finally, given that the desruptive nature of GE's crossover appears to be extremely valuable for generalisation, we propose creating a GP equivalent, *GPRipple*, which will have the same operation.

References

1. Mitchell, T.M.: Machine learning. McGraw Hill, New York (1996)
2. Koza, J.R.: Genetic Programming: On the Programming of Computers by Means of Natural Selection. MIT Press, Cambridge (1992)
3. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 70–82. Springer, Heidelberg (2003)
4. Topchy, A., Punch, W.F.: Faster genetic programming based on local gradient search of numeric leaf values. In: Spector, et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001), July 7-11, pp. 155–162. Morgan Kaufmann, San Francisco (2001)
5. McKay, B., Willis, M., Searson, D., Montague, G.: Non-linear continuum regression using genetic programming. In: Banzhaf, et al. (eds.) Proceedings of GECCO 1999, Orlando, Florida, USA, July 13-17, vol. 2, pp. 1106–1111. Morgan Kaufmann (1999)
6. Ryan, C., Keijzer, M.: An analysis of diversity of constants of genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 404–413. Springer, Heidelberg (2003)
7. Evett, M., Fernandez, T.: Numeric mutation improves the discovery of numeric constants in genetic programming. In: Koza, et al. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference, University of Wisconsin, Madison, Wisconsin, July 22-25, pp. 66–71. Morgan Kaufmann (1998)

8. O'Neill, M., Ryan, C.: Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language. Genetic programming, vol. 4. Kluwer Academic Publishers (2003)
9. Byrne, J., O'Neill, M., Hemberg, E., Brabazon, A.: Analysis of constant creation techniques on the binomial-3 problem with grammatical evolution. In: Tyrrell, et al. (eds.) 2009 IEEE Congress on Evolutionary Computation, Trondheim, Norway, May 18-21, pp. 568–573. IEEE Computational Intelligence Society, IEEE Press (2009)
10. O'Neill, M., Ryan, C., Keijzer, M., Cattolico, M.: Crossover in grammatical evolution. Genetic Programming and Evolvable Machines 4(1), 67–93 (2003)
11. Dempsey, I., O'Neill, M., Brabazon, A.: Constant creation in grammatical evolution. International Journal of Innovative Comput. and Applic. 1(1), 23–38 (2007)
12. Augusto, D.A., Barbosa, H.J.C., Barreto, A.M.S., Bernardino, H.S.: Evolving numerical constants in grammatical evolution with the ephemeral constant method. In: Antunes, L., Pinto, H.S. (eds.) EPIA 2011. LNCS, vol. 7026, pp. 110–124. Springer, Heidelberg (2011)
13. Augusto, D.A., Barbosa, H.J.C., Barreto, A.M.S., Bernardino, H.S.: A new approach for generating numerical constants in grammatical evolution. In: Krasnogor, et al. (eds.) GECCO 2011: Proceedings of the 13th Annual Conference Companion on GECCO, Dublin, Ireland, July 12-16, pp. 193–194. ACM (2011)
14. Daida, J.M., Bertram, R.R., Stanhope, S.A., Khoo, J.C., Chaudhary, S.A., Chaudhri, O.A., Polito II, J.A.: What makes a problem GP-hard? Analysis of a tunably difficult problem in genetic programming. Genetic Programming and Evolvable Machines 2(2), 165–191 (2001)
15. Nicolau, M., Slattery, D.: libGE - Grammatical Evolution Library (2006)
16. Ryan, C., Azad, R.M.A.: Sensible initialisation in grammatical evolution. In: Barry, A.M. (ed.) GECCO 2003: Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference, Chigaco, pp. 142–145. AAAI (July 2003)
17. Vladislavleva, E.J., Smits, G.F., den Hertog, D.: Order of nonlinearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. IEEE Trans. on Evolutionary Computation 13(2), 333–349 (2009)
18. Keijzer, M., Babovic, V.: Genetic programming, ensemble methods and the bias/variance tradeoff - introductory investigations. In: Poli, R., Banzhaf, W., Langdon, W.B., Miller, J., Nordin, P., Fogarty, T.C. (eds.) EuroGP 2000. LNCS, vol. 1802, pp. 76–90. Springer, Heidelberg (2000)
19. Poli, R.: A simple but theoretically-motivated method to control bloat in genetic programming. In: Ryan, C., Soule, T., Keijzer, M., Tsang, E.P.K., Poli, R., Costa, E. (eds.) EuroGP 2003. LNCS, vol. 2610, pp. 204–217. Springer, Heidelberg (2003)
20. Costelloe, D., Ryan, C.: On improving generalisation in genetic programming. In: Vanneschi, L., Gustafson, S., Moraglio, A., De Falco, I., Ebner, M. (eds.) EuroGP 2009. LNCS, vol. 5481, pp. 61–72. Springer, Heidelberg (2009)
21. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, San Francisco (1998)