Analysis of Constant Creation Techniques on the Binomial-3 Problem with Grammatical Evolution

Jonathan Byrne, Michael O'Neil, Erik Hemberg and Anthony Brabazon
Natural Computing Research & Applications Group, Complex and Adaptive Systems Lab, University College Dublin

***Abstract*— This paper studies the difference between Persistent Random Constants (PRC) and Digit Concatenation as methods for generating constants. It has been shown that certain problems have different fitness landscapes depending on how they are represented, independent of changes to the combinatorial search space, thus changing problem difficulty. In this case we show that the method for generating the constants can also influence how hard the problem is for Genetic Programming.**

I. Introduction

The metaphor of a fitness landscape has been used to explain how difficult a problem is for different types of evolutionary computation. This metaphor was originally conceived for use with Genetic Algorithms and has since been applied to the field of Genetic Programming (GP) [7]. The problem is that it has been demonstrated that the metaphor is not applicable to GP in every case. It has been shown that a number of factors such as how the problem is represented and the fitness evaluation used to solve the problem can also have an impact on the fitness landscape [1] [12]. The Binomial-3 problem is an example of a tunable problem where the range of constants have a dramatic impact on how hard the problem is for GP, even though the combinatorial space of the problem remains the same [1]. For this paper we decided to explore different grammatical representations for constants to determine if this would also impact the fitness landscape.

This paper is organised as follows. Section II describes previous work in the area of problems with tunable difficulty and the results that were obtained. It also gives a brief explanation of Grammatical Evolution (GE), a grammar based representation of GP. Section III explains the differences between Ephemeral Random Constants (ERC), Persistent Random Constants (PRC) and Digit Concatenation as constant creation techniques. Section IV explains the parameters that were used while carrying out the experiments. Section V describes how the experiment was implemented and the experimental approach that was used to carry out the experiments. Section VI examines the results we obtained from each of the constant generation methods in the experiment. Section VII discusses the possible conclusions can be drawn from the results and outlines what future work should be examined in this area.

II. Background and Previous work

The reasons for a problem being difficult for GP has only had limited theoretical analysis. Many of the approaches are themselves stochastic, involving a random sampling of the search space or approximations to calculate how many trails are required to get a certain probability of success. Koza suggested in his first book a measure to calculate the amount of processing to solve a problem. He accomplished this by measuring the number of individuals that must be processed in order to satisfy the success predicate within a certain probability [5]. Langdon and Poli [6] suggested a systematic exploration of the search space through either exhaustive or random sampling and Tomassini et al [13] provided a Distance Correlation Metric to statistically show how the difficulty of a problem is based on a fitness landscape, but all these methods have their shortcomings. One of the main obstacles with these approaches is that they assume that the problem and its respective fitness landscape are invariant to how the problem is represented or how the algorithm solving the problem is implemented.

One example that hints at a deeper complexity are problems where the difficulty can be varied. There are a number of problems however that the difficulty can be tuned. Koza [5] initially set out the the boolean multiplexer and the boolean parity functions as problems which can be varied from easy to hard. These have since been used as a benchmark for testing GP. Punch et al [12] devised the tunably difficult Royal Tree Problem for GP based on the the Royal road problem in GA. They then used this as a platform to test GP abilities and as platform for tuning GP parameters. There is also symbolic regression which is based on polynomial equations. These problems increase in difficulty by adding more variables or changing the coefficients. These tunable problems open up the possibility of discovering what really makes a difference to the difficulty by allowing us to alter certain settings while leaving the rest unchanged.

A. The Binomial-3 Problem

The Binomial-3 problem is a problem which can be cast as Symbolic Regression. The Binomial-3 polynomial is:

$$1 + 3x + 3x^2 + x^3 \tag{1}$$

The 3 refers to the order of the equation and it is Binomial because of the sequence of coefficients in it.

$$(1 + x)^3 \tag{2}$$

The reason this problem was chosen is because it only requires one constant to be generated, and that constant is 1, otherwise

it is a simple symbolic regression problem to solve. There are many paths to the correct solution of the problem. The coefficients for the solution can be generated by using a constant of approximate value, multiplying a number by its reciprocal eg:$(4 * 0.25)$ or by summing individual variables eg:(x+x) [1]. Some example solutions are given below.
$(1 - -x)^3$
$(1 + x)(1 + 2x + x^2)$
$(1 + x + x + x + x^2 + x^2 + x^2 + x^3)$
$(x + 1)/(1/(1 + (x/5) + (x/(1/x)))$
There is also the possibility that it could generate an approximate function that would match the Binomial-3 problem for the range we are testing the solution fitness [-1,1] In its most basic form it is easily solved by GP. We are basing our experiments on this polynomial because it was previously shown by Daida et al[1] that by increasing the range of the constants used the problem becomes exponentially more difficult. This is different from other tunably difficult problems because the coefficients and variables themselves do not change, only the range of possible values. This is particularly interesting because the only difference is how the algorithm represents the problem while the problem itself remains unchanged. It was also shown that this has no affect on the combinatorial solution space as the terminal set on the leaf nodes remains unchanged, only the values contained within those nodes. The Original Binomial-3 experiment was conducted using Ephemeral Random Constants. In our experiment we will use Grammatical Evolution with Persistent Random Constants and Digit Concatenation to examine if the grammatical representation for constant creation will affect the solutions generated.

B. Grammatical Evolution

Grammatical Evolution is an evolutionary algorithm that is a grammar based form of GP[8]. It differs from standard GP by representing the parse-tree based structure of GP as a linear genome. It accomplishes this by using a Genotype-Phenotype mapping of a chromosome represented by a variable length bit or integer string. The chromosome is made up of codons eg:(integer based blocks). Each codon in the string is used to select a production rule from a Backus Naur Form (BNF) grammar. The BNF represents a language in the form of production rules. Each rule is comprised of non-terminals that map to either terminals or other non-terminals depending on the production rules. A simple example BNF grammar is given below, where `<expr>` is the start symbol from which all programs are generated. The grammar states 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> | <var>
<op> ::= + | - | *
<var> ::= x | y
```

The codons decide on which rule is chosen by simply calculating the modulus of the codon value with the number of rules. This can be represented with the following formula:

$$\textit{Rule} = \textit{Codon Value} \ \% \ \textit{Num. Rules} \qquad (3)$$

By iterating through the codons the BNF rules are applied and a derivation tree is built. If you remove the non-terminal rules from the derivation trees we end up with tree structures that are identical to the tree structures of GP.

III. Constant Creation Techniques

We now present the different representations for constant generation examined in this study. These include the classic Ephemeral Random constants of GP, Persistent Random Constants, and Digit Concatenation.

A. Ephemeral Random Constants

Ephemeral Random constants were devised as a means for constant creation in GP [5]. It operates by adding a new terminal R to the terminal set. This terminal is used normally during the initialisation phase of the population but before the experiment starts each R is replaced with a randomly generated number of a specified data type within the set range. Once generated and inserted into the initial tree population these constants remain fixed. The constants themselves are ephemeral because if they are removed from the population by selection pressure then they cannot be recovered.

B. Persistent Random Constants

Persistent Random Constants take advantage of the BNF production rules for adding constants to the tree structure by having a non-terminal for constants. For example, at the initialisation of the population the production part of the rule is replaced with a set of approximately 100 constants of the specified data type within the set range. Once generated the constants remain fixed. They are persistent because even if they are taken out of a population there is the possibility they can return if a leaf node is mutated. An example of PRC is given below.

```
before initialisation:
<PRC> ::= "10 random real constants"
after initialisation:
<PRC> ::= 10.5|37.3|45.9|52|3.7|97.6|
          12.8|64.8|20.8|41.7
```

C. Digit Concatenation

Digit Concatenation [3] also utilises the BNF grammar without limiting the number of available constants to a limited set that is generated at initialisation. It does this by concatenating digits to form a single value. By availing of the recursive nature of the non-terminals it is possible to create any constant within the specified range. This method of constant creation has already been tested against traditional methods for constant creation in GE [9] [2] and it was shown to exhibit a significant improvement. An example of digit concatenation for real numbers is shown in the grammar below:

```
<realCat> ::= <cat> <dot> <cat> | <cat>
```

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

The grammar rules were adapted in our experiments to limit the range, Some examples of the digit rules are given below:

```
for the range [0,5]
<int> ::= <nzdigit>.<digit><digit>
<digit> ::= 1|2|3|4|5|6|7|8|9|0
<nzdigit> ::= 0|1|2|3|4

for the range [0,100]
<int> ::=<nzdigit><digit>.<digit><digit>|
         <digit>.<digit><digit>
<digit> ::= 1|2|3|4|5|6|7|8|9|0
<nzdigit> ::= 1|2|3|4|5|6|7|8|9

for the range [0,5000]
<int> ::=<anzdigit><digit><digit><digit>
         .<digit><digit> |
         <nzdigit><digit><digit>
         .<digit><digit>|
         <nzdigit><digit>.<digit><digit>|
         <digit>.<digit><digit>
<digit> ::= 1|2|3|4|5|6|7|8|9|0
<nzdigit> ::= 1|2|3|4|5|6|7|8|9
<anzdigit> ::= 1|2|3|4
```

IV. Experiment Procedure

This experiment was implemented using GEVA [10], this is an open source framework for Grammatical Evolution in Java designed by the NCRA group in UCD. As this experiment is based on the research already done on the Binomial-3 problem [1] we matched the experimental conditions as closely as possible. Population size = 500, Crossover rate = 0.9, replication rate= 0.1, maximum generations =200, the Mersenne Twister as the random number generator and fitness proportionate selection using the Roulette Wheel selection operator. It was decided to use a precision of two decimal places to simulate real values. The fitness was calculated using 50 randomly selected points between the range -1 and 1. The mutation rate was not specified so we used the same mutation rate that Koza used for Symbolic Regression [5].In our experiment the GE version of ramped half-and-half initialisation was used, A Ramped Full Grow initialiser with the tree depth set to 17. This tree depth is greater than the previous experiments set-up of 2-6 but this is because Grammatical Evolution grows the trees at a slower rate during crossover than standard GP. This also avoids the problem of oversampling smaller solutions that occurs when the solution length is limited which, in turn, leads to bloat [4] [11] .The replacement operator was not specified in the previous experiment so tests were run for both Generational and Steady-State replacement. Our experiment differed from the previous one in that we did not consider negative values. To remove the impact of the sign on the constant we kept the range positive ie; [-100,100] became [0,200]. This still has the same amount of elements in the set and so should still be considered of comparative difficulty.

V. Experiment Description

This experiment was carried out on the Binomial-3 problem for different ranges of Real constants to 2 decimal places. The ranges used for both PRC and Digit Concatenation were [0,5], [0,10], [0,50], [0,100],[0,500], [0,1000], [0,2000] and [0,5000]. For each range there were 60 trials run, 30 for Steady State and 30 for Generational. Another experiment was executed where the set size for the Persistent Random Constants was increased from 100 to 1000. For each range there were 30 trial runs for Generational replacement only. The grammar used for this experiment is shown below:

```
<expr> ::= (<op><expr><expr>)|
             <var>|<const>
<op>   ::= +|-|*
<var> ::= x0
<const>::= "constant generation method"
```

TABLE I: results for Persistent Random Constants (Steady-State)

Case	BestFitness	stdDev	Average Fitness	stdDev
PRC5	36.15	27.28	37.7	28.04
PRC10	59.58	36.74	60.71	38.01
PRC50	158.11	45.90	158.11	45.9
PRC100	186.96	54.92	187.12	54.72
PRC500	205.59	58.7	206.47	58.09
PRC1000	216.9	43.19	217.6	42.48
PRC2000	209.43	47.29	211.14	46.18
PRC5000	202.48	37.05	202.47	37.05

TABLE II: results for Digit Concatenation (Steady-State)

Case	BestFitness	stdDev	Average Fitness	stdDev
Concat5	1.14	1.65	727.36	2709.2
Concat10	28.05	21.43	29.20	20.89
Concat50	35.25	21.38	36.02	21.24
Concat100	36.6	29.73	37.28	29.83
Concat500	36.78	25.87	37.69	25.19
Concat1000	33.34	18.83	33.68	18.74
Concat2000	32.97	21.41	34.21	21.15
Concat5000	34.56	22.89	35.35	19.76

TABLE III: results for Persistent Random Constants (Generational)

Case	BestFitness	stdDev	Average Fitness	stdDev
PRC5	9.56	10.9	4596.1	15122.4
PRC10	16.6	20.4	2579.5	6858.4
PRC50	90.09	64.3	13976.9	43121.6
PRC100	104.7442	71.9	2624609	1437136
PRC500	140.6	57.7	1512553876	8276308040
PRC1000	164.3	65.04	2996614	13835151
PRC2000	164.1	50.8	586146.7	305680.9
PRC5000	160.9	62.2	8.716e+16	4.77e+17

TABLE IV: results for Digit Concatenation (Generational)

Case	BestFitness	stdDev	Average Fitness	stdDev
Concat5	1.14	1.65	727.3	2709.2
Concat10	3.19	5.16	10654.27	34445.8
Concat50	4.5	6.88	52486.8	124798.6
Concat100	5.2	6.97	1418010	6980128
Concat500	6.15	8.22	84063967	409599805
Concat1000	8.2	8.68	901673203	4857220418
Concat2000	5.44	7.25	2815668301	10691830451
Concat5000	3.43	4.75	240268558871	1.311852e+12

VI. Experimental results

The results for the Steady State Replacement are shown in tables I and II and in Figures 1 and 2. In figure 1 it is clear that this problem is indeed tunably difficult for Pesistent Random Constants as the error steadily increases for the larger ranges. The results plateau after the range [0,100], this is because the algorithm eschewed using constants and instead creating coefficients by adding several variables together. It is clear that Digit Concatenation accomplished the task more successfully as shown in Figure 2. The results for Generational Replacement are shown in tables III and IV and also in figures 3 and 4. Again it shows that PRCs are tunably difficult for the Binomial-3 problem(figure 3) and that Digit Concatenation performed much better on the same problem(figure 4). Although the range did have some effect on Digit Concatenation, it still regularly managed to find the optimal solution even with the range [0,5000].The results in Figures 5 and 6 show the difference to the fitness when the set size was increased to 1000. Figure 5 seems to show a better performance during the run with the increased set size in every range instance. a two way Analysis of Variance test was carried out against the set size and the varying ranges to see if it had an impact on the results. It failed to show any statistically significant improvement. Figure 6 shows that for ranges greater than [0,500] it ended up performing equivalent to the smaller set. Once the range exceeded 1000 however, the results turned out to be worse than the smaller set size.

A. Discussion

It is clear that Digit Concatenation performed better than PRC in both examples. It also showed that the range alone had less of an impact on the tunability of the problem than the method for creating the constants. Our initial reasoning for this improvement was that Digit Concatenation seemed to have the possibility of generating any one of the constants within the set range whereas PRC is partially limited by its set size. When examining the individuals in the population it became clear that the right combinations for getting the desired constants were more readily available for Digit Concatenation. To further test this hypothesis the set size was increased to 1000 and the experiments repeated. the results initially suggested that this improved the overall fitness up to a point. Once the set size matched the range the performance became worse. We performed an analysis of variance test on the results and it failed to show any statistical significance. This weakens the set size hypothesis and we must conclude that set size is not the primary factor for this difference. The issue of the difference between PRC and ERC must also be raised. In ERC there is the possibility that many more than 100 constants could be generated which would undermine the set size theory, however as the run progresses the number of constants can decrease due to losses from mutation so again the trees could be stuck with the same problem of not having enough of the right constants to make the desired coefficients. We have to conclude from this result that there is something else at play other than set

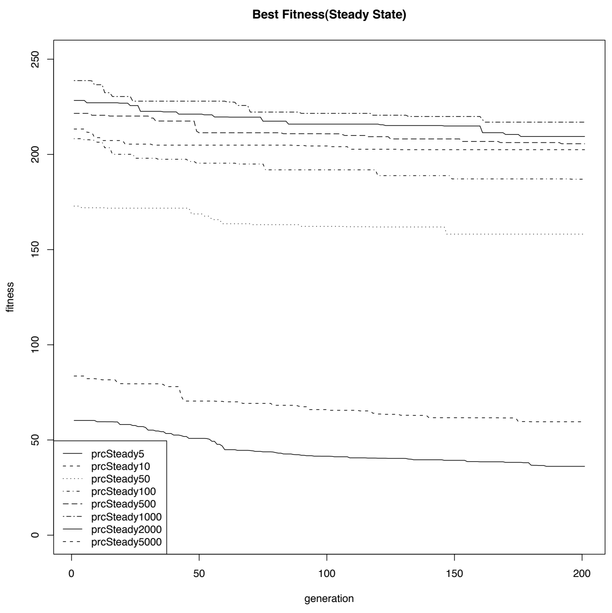

Fig. 1: PRC using Steady State Replacement.

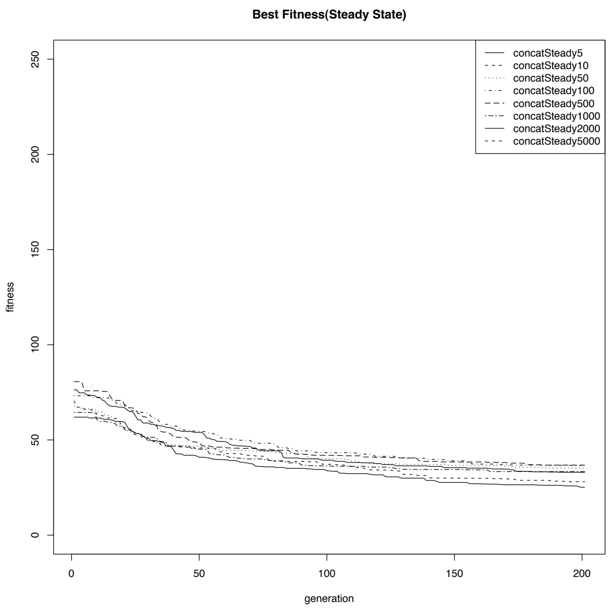

Fig. 2: Digit Concatenation using Steady State Replacement.

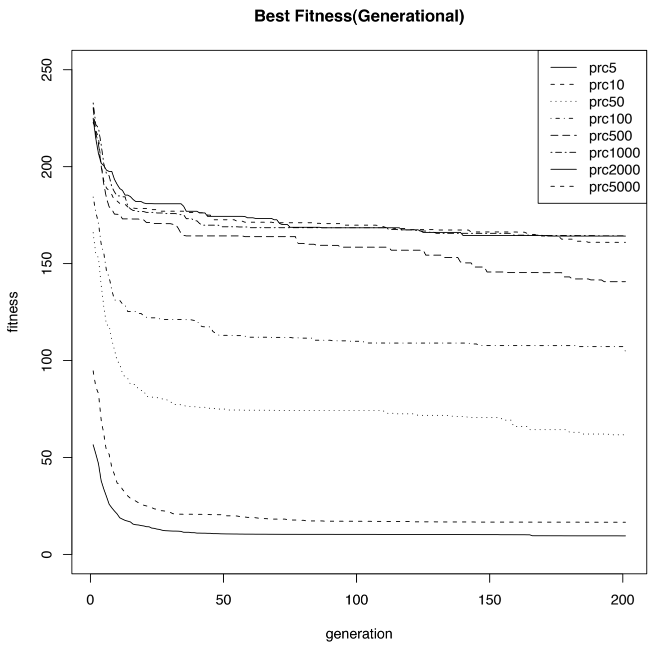

Fig. 3: PRC using Generational Replacement.

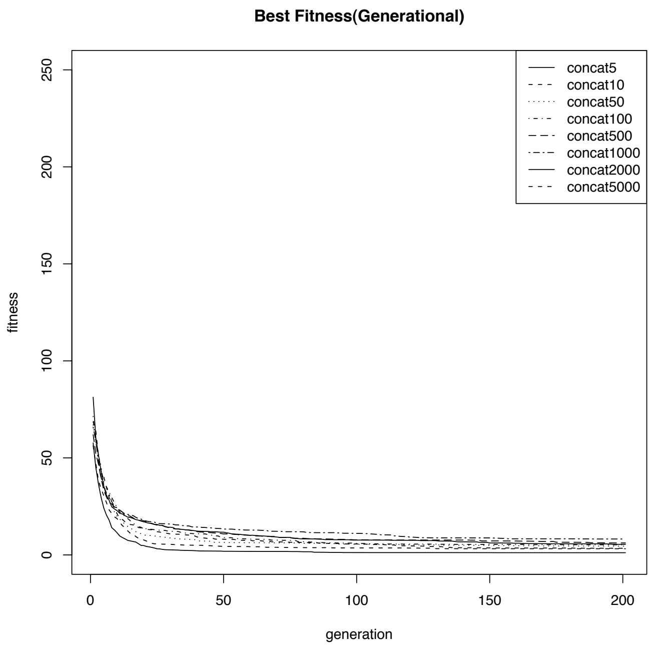

Fig. 4: Digit Concatenation using Generational Replacement.

size which makes this problem more difficult for PRC than Digit Concatenation.

VII. Conclusions and future work

In this paper we examined the impact of constant creation techniques on a tunably difficult problem. We showed that there are many factors contributing to the difficulty of a problem in GP and that the fitness landscape can be radically

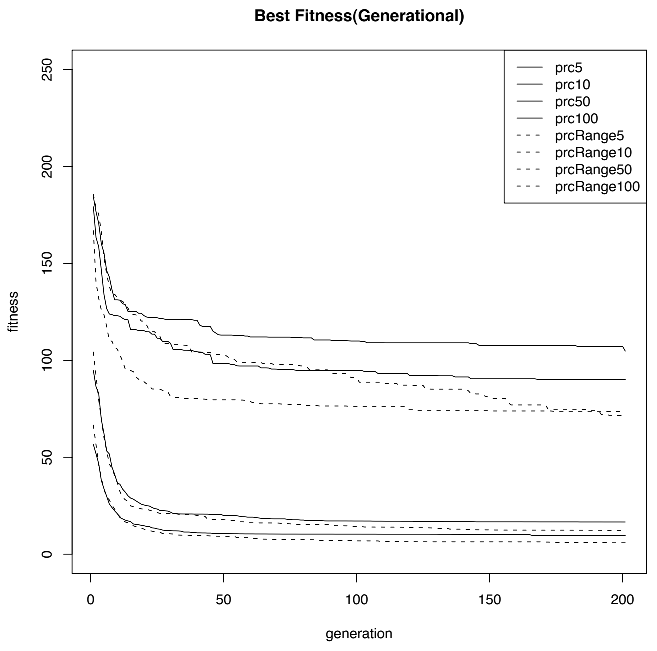

Fig. 5: Comparison between different set sizes for the ranges 5 to 100

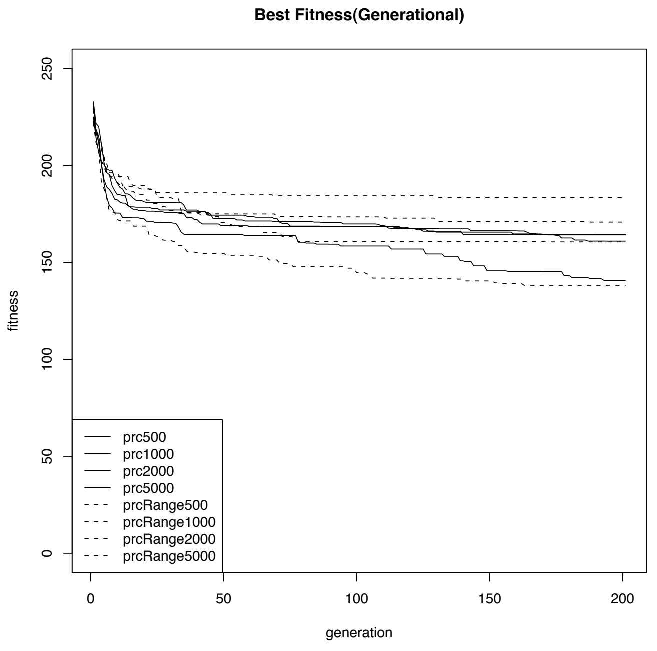

Fig. 6: Comparison between different set sizes for the ranges 500 to 5000.

altered by elements from outside the problem space. We showed that the Binomial-3 problem is tunably difficult for Persistent Random Constants but our results also showed that Digit Concatenation can circumnavigate the tunability of the same problem. This suggested that it might be something else other than the range that has an impact on the difficulty.

Considering that we used PRC in our experiment as opposed to ERC, future work in this area should examine if the PRC is an accurate representation of ERC for Grammatical Evolution.

VIII. Acknowledgments

We would like to thank James McDermott for his help with the statistical analysis of our results. This publication has emanated from research conducted with the financial support of Science Foundation Ireland

References

[1] Jason M. Daida, Robert R. Bertram, Stephen A. Stanhope, Jonathan C. Khoo, Shahbaz A. Chaudhary, Omer A. Chaudhri, and John A. Polito II. What makes a problem GP-hard? analysis of a tunably difficult problem in genetic programming. *Genetic Programming and Evolvable Machines*, 2(2):165–191, June 2001.

[2] Ian Dempsey. *Grammatical Evolution in Dynamic Environments*. PhD thesis, University College Dublin, Ireland, 2007.

[3] Ian Dempsey, Michael O'Neill, and Anthony Brabazon. Constant creation in grammatical evolution. *International Journal of Innovative Computing and Applications*, 1(1):23–38, 2007.

[4] Stephen Dignum and Riccardo Poli. Crossover, sampling, bloat and the harmful effects of size limits. In Michael O'Neill, Leonardo Vanneschi, Steven Gustafson, Anna Isabel Esparcia Alcazar, Ivanoe De Falco, Antonio Della Cioppa, and Ernesto Tarantino, editors, *Proceedings of the 11th European Conference on Genetic Programming, EuroGP 2008*, volume 4971 of *Lecture Notes in Computer Science*, pages 158–169, Naples, 26-28 March 2008. Springer.

[5] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[6] W. B. Langdon and R. Poli. Why ants are hard. Technical Report CSRP-98-4, University of Birmingham, School of Computer Science, January 1998. Presented at GP-98.

[7] W. B. Langdon and Riccardo Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[8] Michael O'Neill. *Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution*. PhD thesis, University Of Limerick, Ireland, August 2001.

[9] Michael O'Neill, Ian Dempsey, Anthony Brabazon, and Conor Ryan. Analysis of a digit concatenation approach to constant creation. In Conor Ryan, Terence Soule, Maarten Keijzer, Edward Tsang, Riccardo Poli, and Ernesto Costa, editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 173–182, Essex, 14-16 April 2003. Springer-Verlag.

[10] Michael O'Neill, Erik Hemberg, Eliott Bartley, Anthony Brabazon, and Conor Gilligan. Geva - grammatical evolution in java. `ncra.ucd.ie/GEVA`, 2008.

[11] R. Poli, W. B. Langdon, and Stephen Dignum. On the limiting distribution of program sizes in tree-based genetic programming. Technical Report CSM-464, Department of Computer Science, University of Essex, December 2006.

[12] William F. Punch, Douglas Zongker, and Erik D. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. MIT Press, Cambridge, MA, USA, 1996.

[13] Marco Tomassini, Leonardo Vanneschi, Philippe Collard, and Manuel Clergue. A study of fitness distance correlation as a difficulty measure in genetic programming. *Evolutionary Computation*, 13(2):213–239, Summer 2005.