Grammatical Evolution Rules: The Mod and the Bucket Rule

Maarten Keijzer[1], Michael O'Neill[2], Conor Ryan[3], and Mike Cattolico[4]

[1] Free University, Amsterdam, mkeijzer@cs.vu.nl
[2] University of Limerick, Michael.ONeill@ul.ie
[3] University of Limerick, Conor.Ryan@ul.ie
[4] Tiger Mountain Scientific Inc., mike@tigerscience.com

Abstract. We present an alternative mapping function called the *Bucket Rule*, for Grammatical Evolution, that improves upon the standard modulo rule. Grammatical Evolution is applied to a set of standard Genetic Algorithm problem domains using two alternative grammars. Applying GE to GA problems allows us to focus on a simple grammar whose effects are easily analysable. It will be shown that by using the bucket rule a greater degree of grammar independence is achieved.

1 Introduction

Grammatical Evolution [9], [8], [10] is an evolutionary automatic programming system [5],[6], [7], [1] that can generate programs in an arbitrary language. Grammatical Evolution adopts a genotype-phenotype mapping process, such that a syntactically correct program is generated, according to the codons of the genotype selecting production rules from a BNF grammar, which describes the output language. The production rules are selected by calling the mapping rule, which is a simple modulo function in the current incarnation of Grammatical Evolution.

We present an alternative mapping function, called the Bucket Rule, for Grammatical Evolution that improves upon the standard modulo rule.
Consider a simple context free grammar that can be used to generate variable length bitstrings:

```
<bitstring>     ::= <bit> | <bit> <bitstring>.

<bit>               ::= 0 | 1.
```

This context free grammar has two non-terminal symbols, each of which have two production rules. In the Grammatical Evolution system a string of so-called codons are maintained, each consisting of 8 bits of information. The modulo rule defines a degenerate mapping from those 8 bits of information to a choice for a production rule in the following way. Given a set of n non-terminals with a corresponding number of production rules $[c_1, \ldots, c_n]$ and given the current symbol r, the mapping rule used is:

$$\text{choice}(r) = \text{codon} \mod c_r \tag{1}$$

J.A. Foster et al. (Eds.): EuroGP 2002, LNCS 2278, pp. 123–130, 2002.

This modulo rule ensures that the codon value is mapped into the interval $[0, c_r]$ and thus represents a valid choice for a production rule. As the codons themselves are drawn from the interval $[0, 255]$, the mapping rule is degenerate: many codon values map to the same choice of rules. Unfortunately, in the case of the context-free grammar given above, the modulo rule will map all even codon values to the first rule and all odd values to the second rule, regardless of the non-terminal symbol that is active. In effect, when using this context-free grammar in combination with the modulo mapping rule, it is the least significant bit in the codon value that fully determines the mapping: all other 7 bits are redundant.

In the context of the untyped crossover and associated intrinsic polymorphism that is usually used in GE [4] — strings of codon values taken from two independently drawn points from the genotypes can be swapped. Here it is possible that a codon value that was used to encode for the `<bitstring>` non-terminal can be used to encode a choice for the `<bit>` non-terminal, due to intrinsic polymorphism, that is a codon can specify a rule for every non-terminal in the grammar and therefore, has meaning in every context.

As the codon values only make a distinction between *first* and *second* choice through their least significant bits, in the above bitstring grammar, a linkage between production rules belonging to different non-terminals is introduced. In the bitstring context-free grammar studied here, this linkage leads to

Codon Value	**non-terminal Symbol**	
	`<bit>`	`<bitstring>`
0	0	`<bit>`
1	1	`<bit> <bitstring>`

A codon value of 0, for example, always selects the first production rule for every non-terminal (i.e. in above, `<bit>` would become 0, and `<bitstring>` would become `<bit>`). Thus regardless of the context (the non-terminal) in which the codon is used, it will have a fixed choice of production rules. It would be better for GE's intrinsic polymorphism if this linkage did not exist, thus each non-terminal's production rule choice is independent of all the other non-terminals when intrinsic polymorphism comes into play.

2 The Bucket Rule

The linkage between production rules belonging to different non-terminal symbols, in combination with untyped variation operators introduces a bias in the search process. This bias is undesirable because it depends on the layout of the program and its impact on the search is not clear. In effect this means that the order in which the rules are defined are expected to make a difference to the search efficiency.

To remove this bias the mapping rule is changed. Given again our set of clauses for n non-terminal symbols $[c_1, \ldots, c_n]$, the codon values are now taken from the interval $[0, \prod_{i=1}^{n} c_i]$. The mapping rule is subsequently changed to:

$$\text{choice}(r) = \frac{\text{codon}}{\prod_{i=1}^{r-1} c_i} \mod c_r \tag{2}$$

This rule is simply the standard method for mapping multi-dimensional matrices into a contiguous array of values. With this rule, every legal codon value encodes a unique set of production rules, one from each non-terminal. In the grammar discussed here, the codons are drawn from $[0, 3]$. The codon values encode for the production rules given the non-terminals in the following way:

Codon Value	**non-terminal Symbol**	
	`<bit>`	`<bitstring>`
0	0	`<bit>`
1	1	`<bit> <bitstring>`
2	0	`<bit> <bitstring>`
3	1	`<bit>`

We choose the name *buckets* because we believe the manner in which a single codon value can code for a number of different choices across different rules is similar to the manner in which keys can hash to identical locations in certain hashing algorithms.

3 Experimental Setup

Grammatical Evolution is applied to a set of standard Genetic Algorithm [3], [2] problem domains using two alternative grammars. Applying GE to GA problems allows us to focus on a simple grammar whose effects are easily analysable.

We perform experiments that demonstrate the existence of these problems, and consequently we introduce the Bucket Rule as an alternative mapping function that overcomes the limitations of the standard modulo rule.

Four experimental setups exist using two separate grammars, Grammar 0 and Grammar 1 are given below, where the only difference between these two grammars is the ordering of the productions rules for the non-terminal `<bit>` .

Grammar 0

```
(A) <bitstring> :=
          <bit><bitstring> (0)
        | <bit>            (1)

(B) <bit> := 0   (0)
           | 1   (1)
```

Grammar 1

```
(A) <bitstring> :=
          <bit><bitstring> (0)
        | <bit>            (1)

(B) <bit> := 1   (0)
           | 0   (1)
```

The first two experimental setups (Setup 0 and Setup 1) use the two grammars with the standard modulo mapping rule, the third and fourth experimental setups use grammar 0 (Setup 2) and grammar 1 (Setup 3) respectively, but in both cases the Bucket rule is adopted.

Three seperate problems are used in the analysis of these grammars, One Max, a variant of One Max, which we call Half One's, and a deceptive trap problem (comprising ten four-bit subfunctions). The problem specific experimental parameters can be seen in Tables 1 and 2.

In the case of One Max, the goal, as normal, is to set all the bits of the phenotype string to 1. For the Half One's problem, the goal is to set the first half of the string to 1's and the second half to 0's. The deceptive trap problem involves finding ten subfunctions (a subfunction is a group of four-bits), where the optimum fitness is achieved when all four-bits of each subfunction are 0. The trap arises due to the fact that when a subfunction has all four-bits set to 1, the next best fitness is achieved, e.g. looking at one subfunction (the fitness for each subfunction is summed to produce the overall fitness):

$$0\ 0\ 1\ 0\ (\ Fitness\ =\ 1\ )$$

$$1\ 0\ 1\ 0\ (\ Fitness\ =\ 2\ )$$

$$1\ 1\ 1\ 1\ (\ Fitness\ =\ 4\ )$$

$$0\ 0\ 0\ 0\ (\ Fitness\ =\ 5\ )$$

Table 1. Experimental parameters for One Max and Half One's

Parameter	**Value**
Popsize	50
Generations	20
Maximum genome length	401
Evaluation length	101
Genome Initialisation length	20
Tournament Size	3
Replacement	Steady State
Crossover Probability	1
Mutation Probability	0
Wrapping	Off

Given the grammars being analysed in this paper we can identify the form of the global optimum's genotype for each of the problems addressed. For example, in the case of One Max, with Grammar 0 this would take the form 0101010101..., whereas with Grammar 1 this would be 1111111111.... Global genotype forms for all three problems are given in Table 3.

Table 2. Experimental parameters for the Deceptive Trap problem.

Parameter	Value
Popsize	1000
Generations	100
Maximum genome length	160
Evaluation length	40
Genome Initialisation length	20
Tournament Size	3
Replacement	Steady State
Crossover Probability	1
Mutation Probability	0
Wrapping	Off

4 Results

A plot of the average over 100 runs of the mean fitness values at each generation of a run over the three problems can be seen in Fig. 1.

Table 3. Global optimum genotype forms for both Grammar 0 and Grammar 1 on each problem examined.

Problem	Global Optimum Form	
	Grammar 0	**Grammar 1**
One Max	01010101...	11111111...
Half One's	0101...0000...	0101...0101...
Deceptive Trap	00000000...	01010101...

The mean fitness values for the final generation on each of the three problems investigated can be seen in Table 4.

Table 4. Average over 100 runs of the mean fitness values for the final generation on each of the three problems investigated. Setup 0 uses Grammar 0 with the mod rule, Setup 1 uses Grammar 1 with the mod rule, Setup 2 uses Grammar 0 with the Bucket Rule, and Setup 3 uses Grammar 1 with the Bucket Rule.

	Average Mean Fitness			
Problem	**Setup 0**	**Setup 1**	**Setup 2**	**Setup 3**
One Max	42.476	39.627	39.627	39.181
Half One's	29.026	29.2	30.094	30.335
Deceptive Trap	21.904	23.913	22.077	22.5

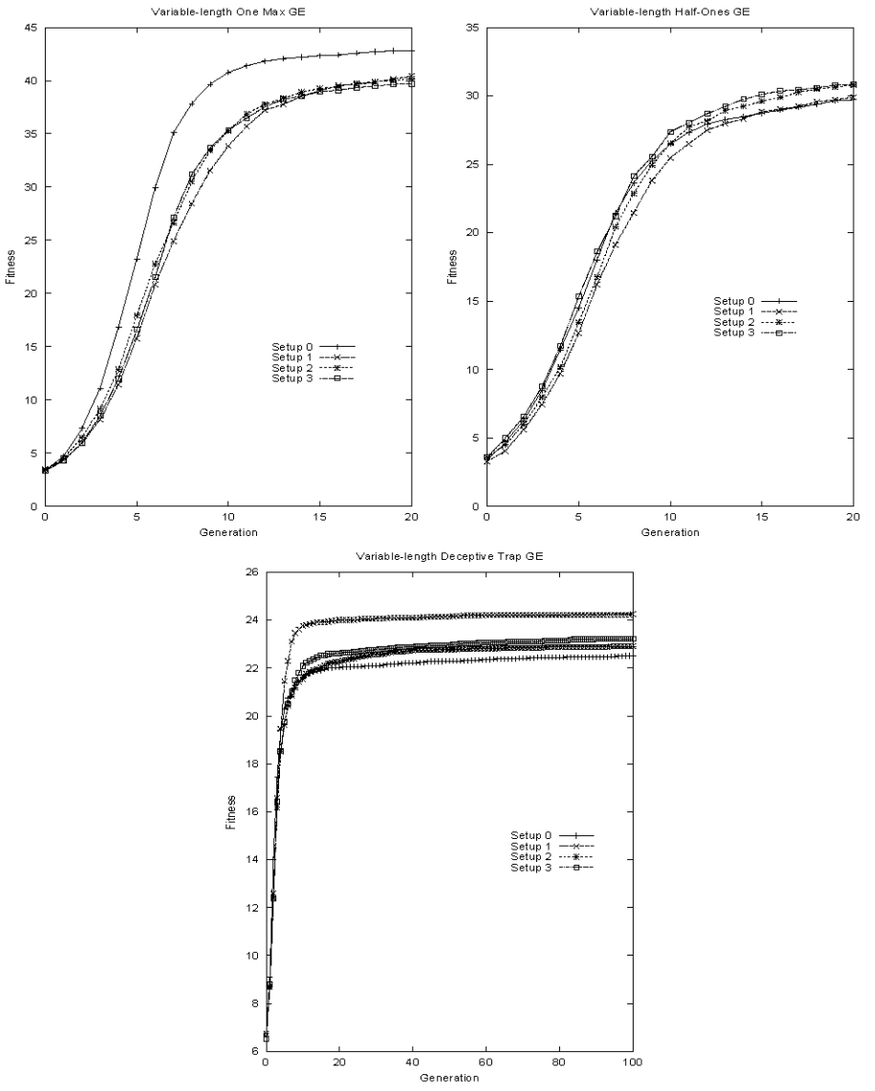

Fig. 1. A comparison of the average over the 100 runs of the best fitness on the four experimental setups for One Max, Half One's, and the Deceptive Trap problems.

A statistical analysis of the mean fitness values contained in Table 4 is conducted using a ttest and, as confirmation of these results, a bootstrap ttest is also applied. In our analysis we wish to investigate if there is a statistically significant difference (at the 95% confidence level) between the means for the different grammars.

Table 5. Results of the statistical analysis using both a ttest and bootstrap ttest. Setup 0 uses Grammar 0 with the mod rule, Setup 1 uses Grammar 1 with the mod rule, Setup 2 uses Grammar 0 with the Bucket Rule, and Setup 3 uses Grammar 1 with the Bucket Rule.

Problem	**Significance of Comparison**	
	(Setup 0 vs. Setup 1)	**(Setup 2 vs. Setup 3)**
One Max	Yes	No
Half One's	No	No
Deceptive Trap	Yes	No

On both One Max and Deceptive Trap there is a statistically significant difference in the means for the two grammars when adopting the standard modulo mapping rule (Setup 0 and Setup 1) as can be seen in Table 5. As such, we find that different grammars can make significant changes to the performance of GE, due to the improper exploitation of intrinsic polymorphism. In the case of the Half One's problem, this is acting like a control experiment in that half of the optimum solution string is suited to Grammar 0 and half is suited to Grammar 1, thus any effects due to bias and intrinsic polymorphism should be non-existent.

When comparing Setup 2 and Setup 3 in which the Bucket Rule was adopted with the two grammars (also see Table 5), there is no statistical difference between the means. Any difference that existed between the two grammars while using the modulo rule has been removed as a result of adopting the Bucket rule. Thus the Bucket rule clearly aids the exploitation of intrinsic polymorphism. The results of statistical analysis with the remaining combinations of comparisons between experimental setups is given in Table 6.

Table 6. Results of the statistical analysis using both a ttest and bootstrap ttest. Setup 0 uses Grammar 0 with the mod rule, Setup 1 uses Grammar 1 with the mod rule, Setup 2 uses Grammar 0 with the Bucket Rule, and Setup 3 uses Grammar 1 with the Bucket Rule.

Problem	**Significance of Comparison**			
	(0 vs. 2)	**(0 vs. 3)**	**(1 vs. 3)**	**(1 vs. 2)**
One Max	Yes	Yes	No	No
Half One's	Yes	Yes	Yes	No
Deceptive Trap	No	Yes	Yes	Yes

On these problems the performance without the Bucket rule tends to be better, see Table 6, however, this performance is very sensitive to the setup of the grammar adopted. Whereas, with the Bucket rule performance is insensitive to the grammar used.

5 Conclusions & Future Work

We have presented the Bucket rule, a new mapping function for Grammatical Evolution that improves upon the standard modulo mapping rule. An analysis of the Bucket rule compared to the modulo rule has been conducted on a number of GA problems of varying difficulty in conjunction with two simple grammars. The grammars and problems were selected to allow our analysis to elucidate the effects that the mapping rules play on the system's performance. Results of the analysis clearly show the benefits of adopting the Bucket in place of the modulo rule.

Future work will investigate the possiblity of alternative mapping rules for Grammatical Evolution, and a more thorough analysis of the effects of the Bucket rule on evolutionary dynamics, particularly with respect to the degenerate code and neutral evolution. It is hypothesised that the Bucket rule will faciliate the exploitation of neutral evolution to a greater extent than the modulo rule allowed.

References

1. 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.
2. Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*, Boston, USA, Addison Wesley, Longman.
3. Holland, J. (1975). *Adaptation in Natural and Artificial Systems*. Ann Arbor, USA, University of Michigan Press.
4. Keijzer M., Ryan C., O'Neill M., Cattolico M, Babovic V. (2001) Ripple Crossover in Genetic Programming. *LNCS 2038, Proc. of the Fourth European Conference on Genetic Programming*, Lake Como, Italy, April 2001, pp.74-86. Springer.
5. Koza, J.R. (1992). *Genetic Programming*. MIT Press.
6. Koza J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press.
7. Koza J.R., Andre D., Bennett III F.H., and Keane M. (1999) *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufman.
8. O'Neill M. (2001) Automatic Programming in an Arbitrary language: Evolving Programs with Grammatical Evolution. Ph.D. thesis, University of Limerick, 2001.
9. O'Neill M., Ryan C. (2001) Grammatical Evolution. *IEEE Trans. Evolutionary Computation*, Vol. 5 No. 4, August 2001.
10. Ryan C., Collins J.J., O'Neill M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *LNCS 1391, Proc. of the First European Workshop on Genetic Programming*, pp. 83-95. Springer-Verlag.