Crossover in Grammatical Evolution: The Search Continues

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

[1] Dept. Of Computer Science And Information Systems,
University of Limerick, Ireland.
{Michael.ONeill|Conor.Ryan}@ul.ie
[2] Danish Hydraulic Institute, Denmark. mak@dhi.dk
[3] Tiger Mountain Scientific. mike@tigerscience.com

Abstract. Grammatical Evolution is an evolutionary automatic programming algorithm that can produce code in any language, requiring as inputs a BNF grammar definition describing the output language, and the fitness function. The utility of crossover in GP systems has been hotly debated for some time, and this debate has also arisen with respect to Grammatical Evolution. This paper serves to continue an analysis of the crossover operator in Grammatical Evolution by looking at the result of turning off crossover, and by exchanging randomly generated blocks in a headless chicken-like crossover. Results show that crossover in Grammatical Evolution is essential on the problem domains examined. The mechanism of one-point crossover in Grammatical Evolution is discussed, resulting in the discovery of some interesting properties that could yield an insight into the operator's success.

1 Introduction

While crossover is generally accepted as an explorative operator in string based G.A.s [4] the benefit or otherwise of employing crossover in tree based Genetic Programming hasn't been fully established. Work such as [2] went as far as to dismiss GP as a evolutionary search method due to its use of trees, while [1] presented results which suggested that crossover in GP provides little benefit over randomly generating subtrees. Langdon and Francone et. al. have also addressed this issue, the former on tree based GP and the latter on linear structures, and have both introduced new crossover operators in an attempt to improve exploration [3] [7].

These exploit the idea of a homologous crossover that draws inspiration from the molecular biological crossover process. The principal exploited being the fact that in nature the entities swapping genetic material only swap fragments which belong to the same position and are of similar size, but which do not necessarily have the same functionality. This, it is proposed, will result in more productive crossover events, and results from both Langdon and Francone et. al. provide evidence to support this claim. A consequence arising from these conservative crossover operators is the reduction of the bloat phenomenon, occurring due to

the fact that these new operators are less destructive [8]. As with many non-binary representations, it is often not clear how much useful genetic material is being exchanged during crossover, and thus not clear how much exploration is actually taking place.

Grammatical Evolution (GE) an evolutionary algorithm that can evolve code in any language, utilises linear genomes [9] [17]. As with GP systems, GE has come under fire for its seemingly destructive crossover operator, a simple one-point crossover inspired by GAs. Previously we have sought answers to questions such as how destructive one-point crossover operator is, and could the system benefit from a homologous-like crossover operator as proposed for tree-based GP, and the linear AIM GP [11]. Initial results suggested that this destructive behaviour did not transpire, at the beginning of runs results show that the number of crossover events that produce individuals which are better than those in the current population is very high. On average, this ratio remains relatively consistent throughout runs, which tells us that crossover is in fact a useful operator in recombining individuals effectively, rather than causing mass destruction. The idea of an homologous crossover has also been explored in the context of GE [11], but results demonstrated the superiority of the simple one-point crossover operator currently adopted.

We now continue the analysis of crossover in GE by turning off crossover, and by exchanging random blocks during crossover in a headless chicken-type crossover [1]. Before a description of our findings we firstly give an overview of GE and introduce the one-point crossover operator adopted.

2 Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language. Rather than representing the programs as parse trees, as in standard GP [6], we use a linear genome representation. Each individual, a variable length binary string, contains in its codons (groups of 8 bits) the information to select production rules from a Backus Naur Form (BNF) grammar. BNF is a notation which represents a language in the form of production rules. It is comprised of a set of non-terminals which can be mapped to elements of the set of terminals, according to the production rules. An example excerpt from a BNF grammar is given below. These productions state that S can be replaced with any one of the non-terminals `expr`, `if-stmt`, or `loop`.

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

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

$$Rule = Codon\ Integer\ Value$$

$$MOD$$

$$Number\ of\ Rules\ for\ this\ nonterminal$$

If the next 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 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:

1. 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.
2. 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 individual's mapping process. This threshold is currently set at ten events.
3. 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.

GE uses a steady state replacement mechanism [16], such that, two parents produce two children, the best of which replaces the worst individual in the current population if the child has a greater fitness. The standard genetic operators of point mutation (applied at a probability of *pmut* (see Table 2) to each bit of the chromosome), and crossover (one point as outlined in Section 3) are adopted. It also employs a duplication operator that duplicates a random number of codons and inserts these into the penultimate codon position on the genome. A full description of GE can be found in [9].

3 The GE Crossover Operator

By default, GE employs a standard one point crossover operator as follows: (i) Two crossover points are selected at random, one on each individual (ii) The segments on the right hand side of each individual are then swapped. As noted in Section 1, it has been suggested that the cost of exploration via crossover is the possible destruction of building blocks. Indeed, there has been work which shows that a reason for the increase in bloat in many GP runs is to protect individuals from destructive crossover.

It is argued here that bloat occurs as a mechanism to prevent the disruption of functional parts of the individual arising from crossover events. To counteract the potentially destructive property of the crossover operator, and to indirectly reduce the occurrence of bloat, novel crossover operators have been developed [7][3]. Dubbed *homologous* crossover, these operators draw inspiration from the molecular biological process of crossover in which the chromosomes to crossover align, and common crossover points are selected on each individual (i.e. at the same locus), and typically a two point crossover occurs.

Results reported in [11] suggest that GE's one point crossover is not as damaging as has been suggested and appears to act as a global search operator throughout the duration of a run.

4 Experimental Approach

We wish to examine the strength of crossover in GE with a two pronged approach. Firstly, the probability of crossover will be set to zero, which effectively switches off the operator. Secondly, we will replace the standard one point crossover adopted with a headless chicken-type crossover. The headless chicken crossover operates by selecting crossover points as normal, but, instead of exchanging the blocks as is, new blocks are generated with a random bit generator.

For each experiment 50 runs were carried out on the Santa Fe ant trail and a symbolic regression problem [6]. Performance was ascertained by using a cumulative frequency of success measure. Tableau's describing the parameters and terminals are given in Tables 2 and 1, and the grammars for both problems are given in Appendix A and B.

5 Results

Results for the experiments can be seen in Figures 1 and 2. These graphs clearly demonstrate the damaging effects of the headless chicken crossover and switching crossover off completely. On the symbolic regression problem GE fails to find solutions in both of these cases, while on the Santa Fe ant trail the system's success rate falls off dramatically.

These results clearly demonstrate that the one point crossover operator is important to the effective operation of the system for the problems examined. Notice that mutation also makes a small difference to the success rate.

Table 1. Grammatical Evolution Tableau for Symbolic Regression

Objective :	Find a function of one independent variable and one dependent variable, in symbolic form that fits a given sample of 20 (x_i, y_i) data points, where the target function is the quartic polynomial $X^4 + X^3 + X^2 + X$
Terminal Operands:	X (the independent variable)
Terminal Operators	The binary operators $+, *, -$, and / (protected division used) The unary operators Sin, Cos, Exp and Log
Fitness cases	The given sample of 20 data points in the interval $[-1, +1]$
Raw Fitness	The sum, taken over the 20 fitness cases, of the absolute error
Standardised Fitness	Same as raw fitness
Hits	The number of fitness cases for which the error is less than 0.01
Wrapper	Standard productions to generate C functions
Parameters	$Population = 500$, $Generations = 50$ $pmut = 0.01$, $pcross = 0.9$

Table 2. Grammatical Evolution Tableau for the Santa Fe Trail

Objective :	Find a computer program to control an artificial ant so that it can find all 89 pieces of food located on the Santa Fe Trail.
Terminal Operators:	left(), right(), move(), food_ahead()
Fitness cases	One fitness case
Raw Fitness	Number of pieces of food before the ant times out with 615 operations.
Standardised Fitness	Total number of pieces of food less the raw fitness.
Hits	Same as raw fitness.
Wrapper	Standard productions to generate C functions
Parameters	$Population = 500$, $Generations = 50$ $pmut = 0.01$, $pcross = 0.9$

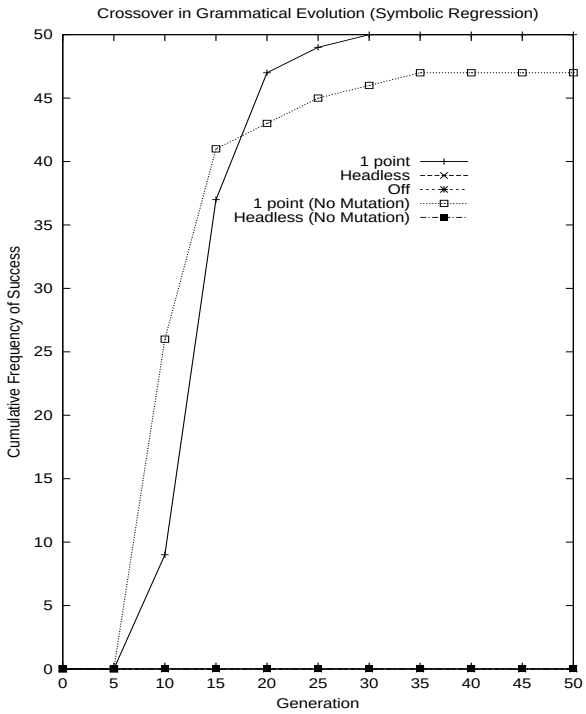

Fig. 1. A comparison of GE's performance on the symbolic regression problem is illustrated. When the headless chicken crossover is used the system fails to find solutions to this problem. This is also the case when crossover is switched off.

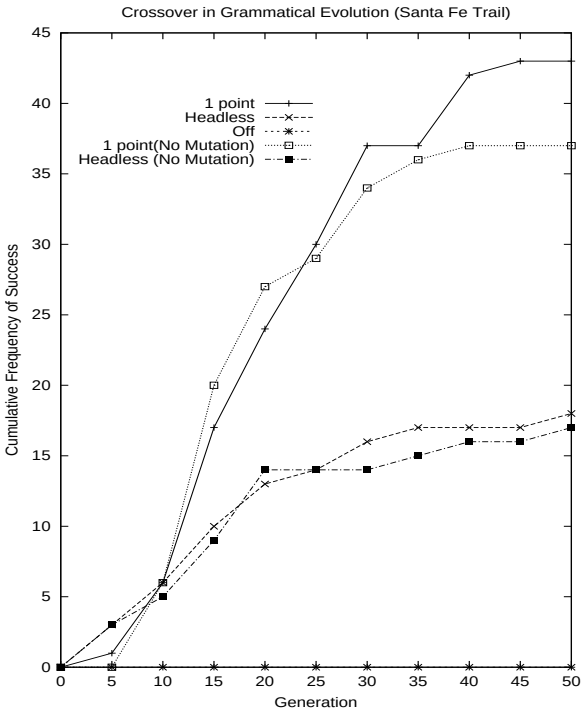

Fig. 2. A comparison of GE's performance on the Santa Fe ant trail can be seen. The graph clearly demonstrates the damaging effects of the headless chicken crossover and the case when crossover is switched off.

6 Discussion

The question arises then, as to why GE's one point crossover operator is so productive. If we look at the effect the operator plays on a parse tree representation of the programs undergoing crossover we begin to see more clearly the mechanism of this operator and it's search properties.

When mapping a string to an individual, GE always works with the left most non-terminal. Thus, if one were to look at the individual's corresponding parse tree, one would see that the tree is constructed in a pre-order fashion. Furthermore, if the individual is over-specified, that is, has codons left over, they form a *tail*, which is, effectively, a stack of codons, as illustrated in Fig. 3.

If, during a crossover event, one tried to map the first half of the remaining string, the result not surprisingly, would usually be an incomplete tree. However, the tree would not be incomplete in the same manner as one taken from the middle of a GP crossover event. The pre-order nature of the mapping is such that the result is similar to that of Fig. 3 and Fig. 4. That is, the tree is left with a *spine* and several *ripple sites* from which one or more sub-trees, dubbed *ripple trees* are removed. This crossover behaviour, which is an inherent property of GE, was first noticed by [5] where they termed it *ripple crossover*.

Each of the ripple trees is effectively dismantled and returned to the stack of codons in the individual's tail. Crossover then involves individuals swapping tails so that, when evaluating the offspring, the ripple sites on the spine will be filled using codons from the other parent.

There is no guarantee that the tail from the other parent will be of the same length, or even that it was used in the same place on the other spine. This means that a codon that represented which choice to make could suddenly be expected to make a choice from a completely different non-terminal, possibly even with a different number of choices. Fortunately, GE evaluates codons *in context*, that is, the exact meaning of a codon is determined by those codons that immediately precede it. Thus, we can say that GE codons have *intrinsic polymorphism*[5], as they can be used in any part of the grammar. Furthermore, if the meaning of one codon changes, the change "ripples" through all the rest of the codons. This means that a group of codons that coded a particular sub-tree on one spine can code an entirely different sub-tree when employed by another spine. The power of intrinsic polymorphism can even reach between the ripple trees, in that if one no longer needs all its codons, they are passed to the next ripple tree and, conversely, if it now requires more codons, it can obtain them from its neighbouring ripple tree.

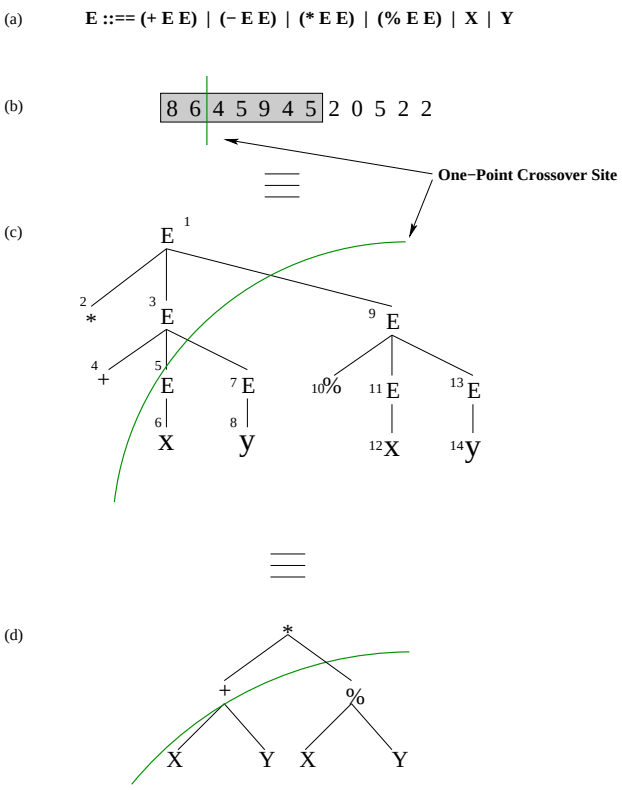

Fig. 3. The ripple effect of one-point crossover illustrated using an example GE individual represented as a string of codon integer values (b) and its equivalent derivation (c) and parse trees (d). The codon integer values in (b) represent the rule number to be selected from the grammar outlined in (a), with the part shaded gray corresponding to the values used to produce the trees in (c) and (d), the remaining integers are an intron. Fig. 4 shows the resulting spine with ripple sites and tails.

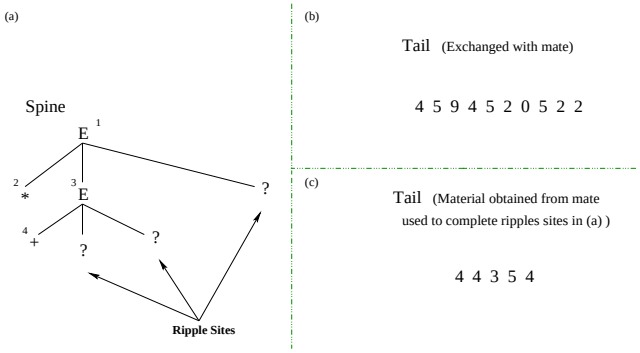

Fig. 4. Illustrated are the spine and the resulting ripple sites (a) and tails (b)(c) produced as a consequence of the one-point crossover in Fig. 3

7 Conclusions & Future Work

We have previously demonstrated the ability of GE's one point crossover as an operator that exploits an exchange of blocks in a productive manner on the problem domains examined. Results presented here also show the detrimental effects of switching off crossover for GE on these problems, in one case crossover being essential to the generation of a correct solution.

A discussion on the mechanism of GE's one point crossover reveals an interesting *ripple* property when it's effect on parse trees is examined, providing a possible explanation as to why this operator may be so profitable for GE. Further investigations will be required in order to ascertain the usefulness of this *ripple* crossover mechanism. It is proposed that the key to the system is that it exchanges on average, half of the genetic material of the parents during each crossover, regardless of the size of the individuals. This is made possible by a combination of the linear representation of the individuals and the property of intrinsic polymorphism, which permits any part of an individual's genome to legally be applied to any part of the grammar [5].

This paper also discuss the notion of spines, that is, the part of the tree that remains after crossover occurs. This could be the first step on the road to identifying a schema theorem for GE, as the system is clearly growing these rooted structures, in a manner similar to [10].

References

1. Angeline, P.J. 1997. Subtree Crossover: Building block engine or macromutation? In *Proceedings of GP'97*, pages 9-17.
2. Collins, R. 1992. Studies in Artificial Life. PhD thesis. University of California, Los Angeles.
3. Francone F. D., Banzhaf W., Conrads M, Nordin P. 1999. Homologous Crossover in Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 99*, pages 1021-1038.
4. Goldberg, David E. 1989. Genetic Algorithms in Search, Optimization and Machine Learning. Addison Wesley.
5. Keijzer M., Ryan C., O'Neill M., Cattolico M., Babovic V. 2001. Ripple Crossover in Genetic Programming. In *Proceedings of EuroGP 2001.*
6. Koza, J. 1992. *Genetic Programming.* MIT Press.
7. Langdon W.B. 1999. Size Fair and Homologous Tree Genetic Programming Crossovers. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 99*, pages 1092-1097.
8. Langdon W.B., Soule T., Poli R., and Foster J.A. 1999. The Evolution of Size and Shape. In *Advances in Genetic Programming Volume 3*, MIT Press 1999, pp 162-190.
9. O'Neill M. and Ryan C. 2001. Grammatical Evolution. *IEEE Trans. Evolutionary Computation*, 2001.
10. Justinian P. Rosca and Dana H. Ballard. 1999. Rooted-Tree Schemata in Genetic Programming, in *Advances in Genetic Programming 3*, Chapter 11, pp 243-271, 1999, MIT Press.

11. O'Neill M. and Ryan C. 2000. Crossover in Grammatical Evolution: A Smooth Operator? *Lecture Notes in Computer Science 1802, Proceedings of the European Conference on Genetic Programming*, pages 149-162. Springer-Verlag.
12. O'Neill M. and Ryan C. 1999. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. In *Proceedings of the Fifth European Conference on Artificial Life*.
13. O'Neill M. and Ryan C. 1999. Under the Hood of Grammatical Evolution. In *Proceedings of the Genetic & Evolutionary Computation Conference 1999*.
14. O'Neill M. and Ryan C. 1999. Evolving Multi-line Compilable C Programs. *Lecture Notes in Computer Science 1598, Proceedings of the Second European Workshop on Genetic Programming*, pages 83-92. Springer-Verlag.
15. Poli Riccardo, Langdon W.B. 1998. On the Search Properties of Different Crossover Operators in Genetic Programming. In *Proceedings of the Third annual Genetic Programming conference 1998*, pages 293-301.
16. Ryan C. and O'Neill M. 1998. Grammatical Evolution: A Steady State Approach. In *Late Breaking Papers, Genetic Programming 1998*, pages 180-185.
17. Ryan C., Collins J.J., and 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*, pages 83-95. Springer-Verlag.

A Symbolic Regression Grammar

$$N = \{expr, op, pre\_op\}$$

$$T = \{Sin, Cos, Exp, Log, +, -, /, *, X, ()\}$$

$$S = < expr >$$

And P can be represented as:

```
(1) <expr> ::= <expr> <op> <expr>      (A)
             | ( <expr> <op> <expr> )  (B)
             | <pre-op> ( <expr> )     (C)
             | <var>                   (D)


(2) <op> ::= + (A)
           | - (B)
           | / (C)
           | * (D)

(3) <pre-op> ::= Sin (A)
               | Cos (B)
               | Exp (C)
               | Log (D)

(4) <var> ::= X
```

B Santa Fe Trail Grammar

$$N = \{code, line, if - statement, op\}$$

$$T = \{left(), right(), move(), food\_ahead(), else, if, \{, \}, (,)\}$$

$$S =< code >$$

And P can be represented as:

```
(1) <code> :: =  <line>              (A)
                |<code><line>        (B)

(2) <line> :: =  <if-statement>      (A)
                |<op>                (B)

(3) <if-statement> :: = if(food_ahead()){<line>} else{<line>}

(5) <op> :: =    left()              (A)
               | right()             (B)
               | move()              (C)
```