Automatic Grammar Complexity Reduction in Grammatical Evolution

Miguel Nicolau

Biocomputing and Developmental Systems Group
Computer Science and Information Systems Department
University of Limerick, Ireland
Miguel.Nicolau@ul.ie

Abstract. Grammatical Evolution is an automatic programming system, where a population of binary strings is evolved, from which phenotype strings are generated through a mapping process, that employs a grammar to define the syntax of such output strings. This paper presents a study of the effect of grammar size and complexity on the performance of the system. A simple method to reduce the number of non-terminal symbols in a grammar is presented, along with the reasoning behind it. Results obtained on a series of problems suggest that performance can be increased with the approach presented.

1 Introduction

Grammatical Evolution (GE) [4, 5, 8] is a Genetic Programming [3] approach, that uses an evolutionary algorithm to evolve binary strings, and, through a mapping process, transforms those strings onto programs, which can then be evaluated. The language for the output programs is specified by a context-free grammar, usually specified by a Backus-Naur Form (BNF) grammar.

One of the strengths of GE is its flexibility and adaptability to a variety of problem domains: virtually any context-free grammar can be used with GE, which means that any problem whose solutions can be described by such a grammar can be tackled with GE.

Although GE has been applied to a variety of problems [8, 9], little research has been conducted on the effect of grammar design, on the performance of the system. More specifically, what is the effect of the complexity of grammars on the performance of GE?

The current paper presents preliminary research into this matter. It introduces a simple method to reduce the number of non-terminal symbols in a grammar, along with a theoretical explanation as to why this procedure should prove useful. Experiments conducted on a series of problems, although not conclusive, show promising results for the approach presented.

This paper is structured as follows. The next section briefly presents GE, while Section 3 addresses the topic of crossover in GE, and the implications of using reduced grammars. Section 4 presents the experiments conducted and their results, while Section 5 analyses those results. Finally, Section 6 draws some conclusions on this work, and highlights possible future work directions.

2 Grammatical Evolution

GE is an evolutionary approach to automatic program generation, which evolves strings of binary values, and uses a BNF grammar to map those strings into programs. This mapping process involves transforming each binary individual into a string of integer values, and using those values to choose transformations from the given grammar. A given start symbol will then be mapped onto a syntactically correct program, by applying the chosen transformations.

The mapping process employed in GE is based on the idea of a genotype to phenotype mapping [1]: an individual comprised of binary values (genotype) is evolved, which, before being evaluated, has to undergo a mapping process that will create a functional program (phenotype), which is then evaluated by the given fitness function. This process separates two distinct spaces, a search space and a solution space.

Finally, the production rule chosen by each of the values in the integer string is dependent on the values preceding it, as those values determine which non-terminal symbols remain to be mapped in the current individual. This creates a functional dependency between each gene and those preceding it, which in turn guides each individual to be built from the leftmost genes to the rightmost ones, and helps the individual in preserving a good structure in its left-hand side during the evolution process, when it is submitted to the harsh effects of operators like crossover. This has been termed the "Ripple Effect" [7].

3 Crossover in Grammatical Evolution

3.1 Ripple Crossover

The crossover operator in GE has been previously studied in detail [7, 8]. When mapping an individual, GE transforms the left most non-terminal symbols first, and gradually works its way through its generated expression, until only terminal symbols are left (the phenotype string). This means, if one were to look at each individual's parse tree, that this tree is built in a pre-order fashion.

For example, take the following grammar:

```
<expr>  ::= <expr> <op> <expr>
        |   (<expr> <op> <expr>)
        |   <pre-op>(<expr>)
        |   <var>
<op>    ::= + | - | / | *
<op>    ::= sin | cos | exp | log
<var>   ::= X | 1.0
```

In Figure 1, using this grammar, the derivation tree of each example individual is given. If the given crossover points were chosen, then each individual is left with a *spine* and several *ripple sites*, from which one of more sub-trees, called *ripple trees*, are removed [8]. The portions of each individual after their crossover points are then exchanged. This means that the new genetic material, received from the second parent, is used to map all ripple points in each individual.

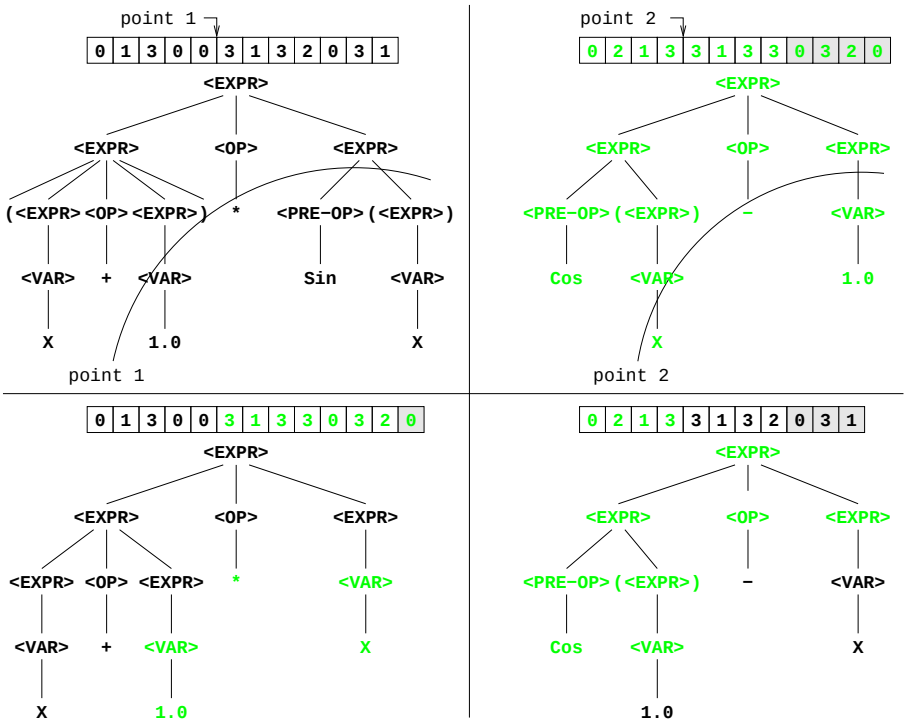

Fig. 1. Two individuals expressed both as integer strings and derivation trees. With the chosen crossover points, the sub-trees below the crossover curve, called *ripple trees*, are removed. These are then mapped using new genetic material from the second parent.

Note that if the context of interpretation for the new genetic material changes, so will the functionality of that material [8]; indeed, the higher the number of non-terminal symbols in the grammar, the more likely this is to happen.

3.2 Reducing Grammars

An interesting approach would therefore be to reduce the number of non-terminal symbols in a grammar, to increase the likelihood that new genetic material, when placed after the chosen crossover point, keeps its original functionality. In theory, this will increase the exchange of meaningful information, as the new material keeps the functionality it had in its original location (an individual fit enough to be selected for crossover).

The process of reducing the number of non-terminal symbols is a simple one, and can be easily automated. For example, taking the previous grammar, the first step is to go through the first set of transformations, and find a symbol that can be replaced by all of its associated productions. Looking at the transformations associated with the `<expr>` symbol, one can see that this symbol cannot be

removed, as it is recursively defined. The symbol <op>, however, is not, and can therefore be replaced by all of its four transformations, wherever it appears throughout the grammar. This means that the grammar becomes:

```
<expr>  ::= <expr> + <expr> | <expr> - <expr> | <expr> / <expr> | <expr> * <expr>
        |   (<expr> + <expr>) | (<expr> - <expr>) | (<expr> / <expr>) | (<expr> * <expr>)
        |   <pre-op>(<expr>)
        |   <var>
```

This allows one to remove the <op> symbol from the grammar, reducing the number of non-terminal symbols. However, this replacement affects the bias of choices associated with the <expr> symbol, as before there was a 1/4 probability of choosing the transformation <expr> -> <var>, whereas now there is only a probability of 1/10[1]. To keep the original bias of transformation choices, each transformation not involving the <op> symbol must appear in the same number as there were productions associated with the <op> symbol (in this case, four):

```
<expr>  ::= <expr> + <expr> | <expr> - <expr> | <expr> / <expr> | <expr> * <expr>
        |   (<expr> + <expr>) | (<expr> - <expr>) | (<expr> / <expr>) | (<expr> * <expr>)
        |   <pre-op>(<expr>) | <pre-op>(<expr>) | <pre-op>(<expr>) | <pre-op>(<expr>)
        |   <var> | <var> | <var> | <var>
```

In a similar fashion, the <pre-op> and <var> symbols can also be replaced, leaving the grammar with just one non-terminal symbol, <expr>, defined as:

```
<expr>  ::= <expr> + <expr> | <expr> - <expr> | <expr> / <expr> | <expr> * <expr>
        |   (<expr> + <expr>) | (<expr> - <expr>) | (<expr> / <expr>) | (<expr> * <expr>)
        |   sin(<expr>) | cos(<expr>) | exp(<expr>) | log(<expr>)
        |   X | 1.0 | X | 1.0
```

Note that it is not always possible to reduce a grammar to just one non-terminal symbol: recursively defined symbols cannot be automatically removed.

How does this affect the crossover operator in GE? Figure 2 shows an example of two individuals generated using this grammar, and how the exchanged material between them keeps its functionality. It can be seen that each ripple point is filled with information that existed in the second parent.

4 Experiments and Results

A series of experiments were setup, to compare the performance of GE using normal grammars, and their reduced equivalent. The problems chosen were typical genetic programming benchmarks, to which GE has been applied before.

The experimental setup used is shown in Table 1. Selection was roulette-wheel, with steady-state replacement. The wrapping operator was not used.

[1] This does not take into account the bias of choices caused by the *mod* operator [6].

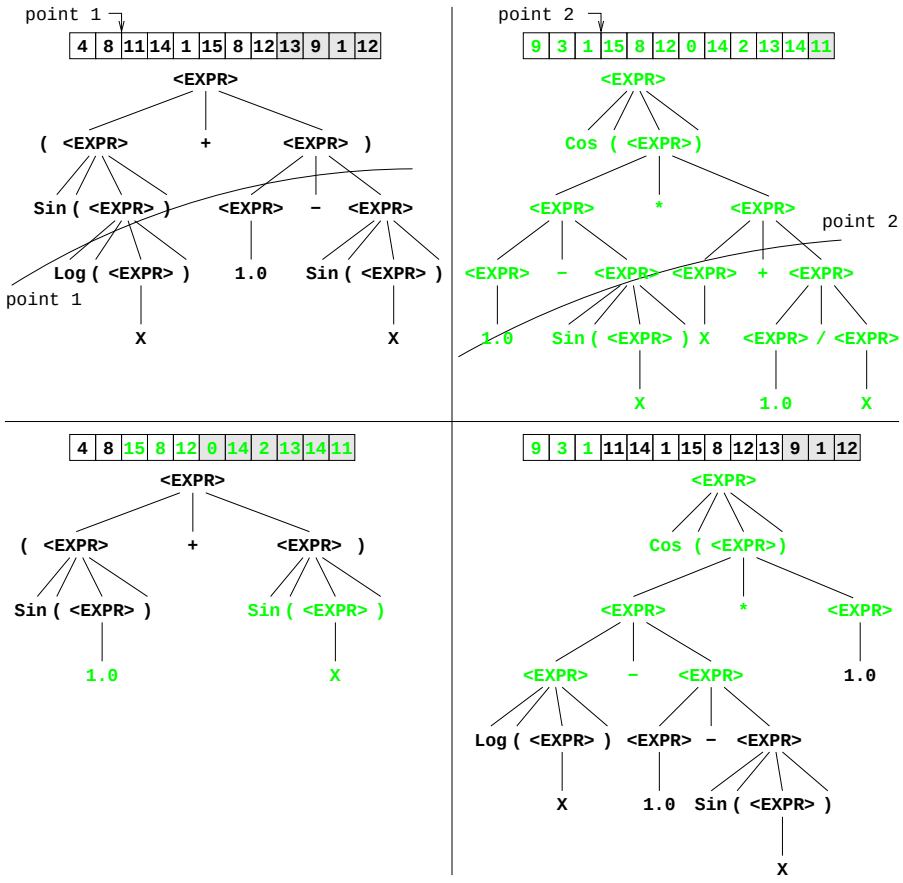

Fig. 2. Two individuals expressed as integer strings, and their corresponding derivation trees. With the chosen crossover points, the ripple sub-trees are removed. These are then mapped using new genetic material from the second parent, which keeps its meaning.

4.1 Santa Fe Ant Trail

The Santa Fe Ant Trail was one of the original problems used as a proof of concept for GE. The objective is to evolve a computer program, that controls the movement of an artificial ant, on a toroidal grid of size 32 by 32. The fitness of an evolved program is the number of food pieces that it can collect, out of the 89 scattered pieces on the grid. Each evolved program is run in a loop, until all pieces of food have been found, or the ant has executed 615 movements.

The grammar used with GE on this problem is as follows:

```
<code> ::= <line> | <code> <line>
<line> ::= <condition> | <op>
<condition> ::= if(food_ahead()) { <op> } else { <op> }
<op> ::= left(); | right(); | move();
```

Table 1. Experimental setup, used on all experiments

Population size:	500
Number of generations:	100
Codon size:	8 bits
Crossover probability:	0.9
Point mutation probability:	0.01
Number of wrapping events:	0
Range of initial individual size:	10-30

To generate the equivalent reduced grammar, the first step is to replace all references to the `<line>` symbol by both `<condition>` and `<op>`:

```
<code> ::= <condition> | <op> | <code> <condition> | <code> <op>
<condition> ::= if(food_ahead()) { <op> } else { <op> }
<op> ::= left(); | right(); | move();
```

The next step consists of replacing all references to the `<condition>` symbol:

```
<code> ::= if(food_ahead()) { <op> } else { <op> } | <op>
        | <code> if(food_ahead()) { <op> } else { <op> } | <code> <op>
<op> ::= left(); | right(); | move();
```

Finally, the last step consists in replacing all references to the `<op>` symbol. Note that, in order to keep same bias between transformation choices for the `<code>` symbol, all possible combinations of the `<op>` symbol are expanded, leading to an increase in the number of all the other productions:

```
<code> ::= if(food_ahead()) { left(); } else { left(); }
        | if(food_ahead()) { left(); } else { right(); }
        | if(food_ahead()) { left(); } else { move(); }
        | if(food_ahead()) { right(); } else { left(); }
        | if(food_ahead()) { right(); } else { right(); }
        | if(food_ahead()) { right(); } else { move(); }
        | if(food_ahead()) { move(); } else { left(); }
        | if(food_ahead()) { move(); } else { right(); }
        | if(food_ahead()) { move(); } else { move(); }
        | left(); | left(); | left();
        | right(); | right(); | right();
        | move(); | move(); | move();
        | <code> if(food_ahead()) { left(); } else { left(); }
        | <code> if(food_ahead()) { left(); } else { right(); }
        | <code> if(food_ahead()) { left(); } else { move(); }
        | <code> if(food_ahead()) { right(); } else { left(); }
        | <code> if(food_ahead()) { right(); } else { right(); }
        | <code> if(food_ahead()) { right(); } else { move(); }
        | <code> if(food_ahead()) { move(); } else { left(); }
        | <code> if(food_ahead()) { move(); } else { right(); }
        | <code> if(food_ahead()) { move(); } else { move(); }
        | <code> left(); | <code> left(); | <code> left();
        | <code> right(); | <code> right(); | <code> right();
        | <code> move(); | <code> move(); | <code> move();
```

The results for the Santa Fe problem, using the original (grammar 1) and reduced (grammar 2) approaches, are shown in Figure 3.

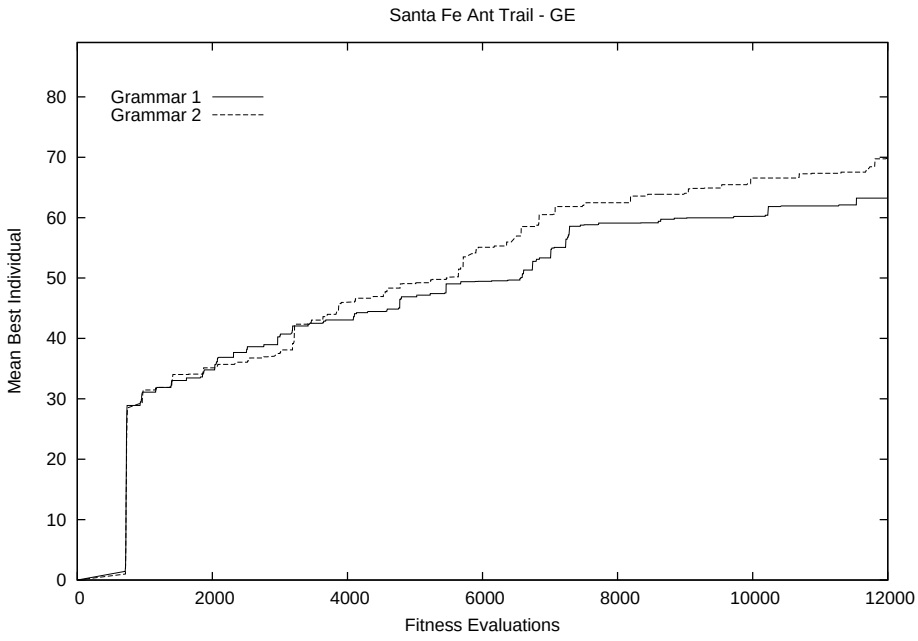

Fig. 3. Results obtained for the Santa Fe Ant Trail problem, using the original and reduced grammars. The *x-axis* shows the number of fitness evaluations, and the *y-axis* the mean best individual (from 30 independent runs).

4.2 Symbolic Regression 1

The symbolic regression problem is a well known problem domain in Genetic Programming, and has also been tackled by GE in several occasions [8, 9]. It involves finding a mathematical expression in symbolic form, that maps each value of a given set of input values to their corresponding output values.

The function used to generate the input and output pairs is:

$$f(x) = x^4 + x^3 + x^2 + x \qquad x \in \{-1, 1\}$$

where x is an input value, and $f(x)$ its corresponding output value.

The grammar used in this experiment was:

```
<expr> ::= <op>(<expr>,<expr>) | <var>
<op>   ::= add | sub | mult | pdiv
<var>  ::= X | 1.0
```

After applying the reduction process, the equivalent reduced grammar is:

```
<expr> ::=     add(<expr>,<expr>)
       |       sub(<expr>,<expr>)
       |       mult(<expr>,<expr>)
       |       pdiv(<expr>,<expr>)
       |       X | X | 1.0 | 1.0
```

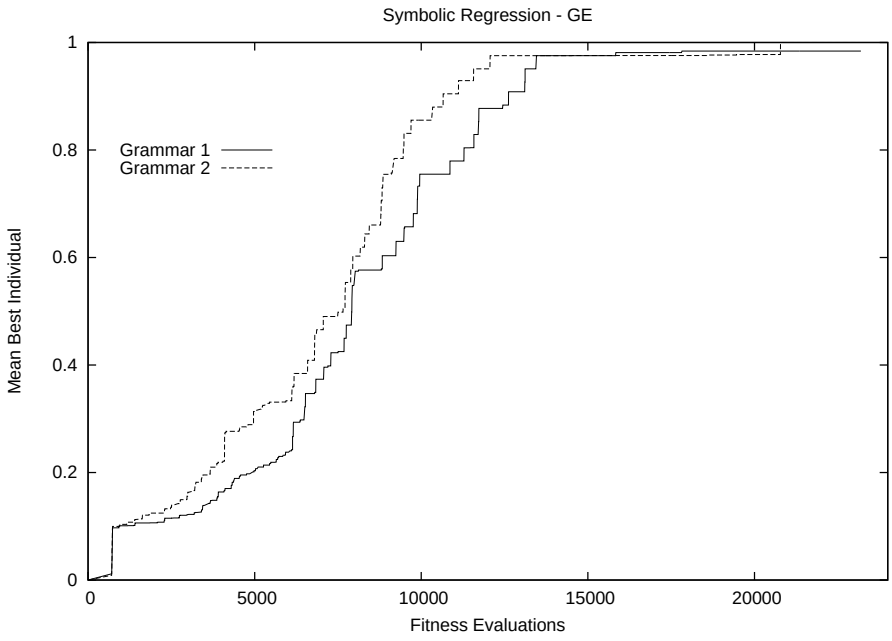

Fig. 4. Results obtained for the first instance of the Symbolic Regression problem, using the original and reduced grammars. The *x-axis* shows the number of fitness evaluations, and the *y-axis* the mean best individual (from 30 independent runs).

The results obtained with both grammars are shown in Figure 4.

4.3 Symbolic Regression 2

This is another instance of the Symbolic Regression problem, using the following function to generate the input and output values:

$$f(x) = x^2 + x + 1 + cos(x) \qquad x \in \{-1, 1\}$$

where x is an input value, and $f(x)$ its corresponding output value.

The grammar used in this experiment was:

```
<expr> ::= <op1>(<expr>) | <op2>(<expr>,<expr>) | <var>
<op1>  ::= sin | cos | tan
<op2>  ::= add | sub | mult | pdiv
<var>  ::= X | 1.0
```

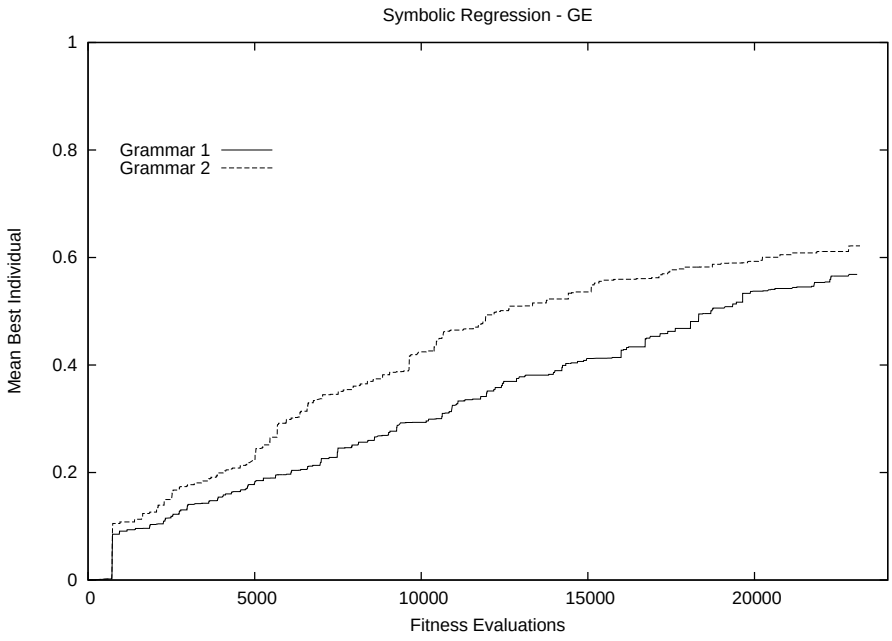

Fig. 5. Results obtained for the second instance of the Symbolic Regression problem, using the original and reduced grammars. The *x-axis* shows the number of fitness evaluations, and the *y-axis* the mean best individual (from 30 independent runs).

After applying the reduction process, the equivalent reduced grammar is:

```
<expr> ::= sin(<expr>) | cos(<expr>) | tan(<expr>)
       |   sin(<expr>) | cos(<expr>) | tan(<expr>)
       |   sin(<expr>) | cos(<expr>) | tan(<expr>)
       |   sin(<expr>) | cos(<expr>) | tan(<expr>)
       |   add(<expr>,<expr>) | sub(<expr>,<expr>) | mult(<expr>,<expr>) | pdiv(<expr>,<expr>)
       |   add(<expr>,<expr>) | sub(<expr>,<expr>) | mult(<expr>,<expr>) | pdiv(<expr>,<expr>)
       |   add(<expr>,<expr>) | sub(<expr>,<expr>) | mult(<expr>,<expr>) | pdiv(<expr>,<expr>)
       |   X | 1.0 | X | 1.0 | X | 1.0 | X | 1.0 | X | 1.0 | X | 1.0
```

The results obtained for this experiment are shown in Figure 5.

4.4 Multiplexer

This is the last problem analysed. It consists in evolving a boolean expression, that matches a given set of boolean inputs to a boolean output. For this problem, three input variables were used; the truth table defining the problem is given in Table 2.

Table 2. Truth table for the Multiplexer problem

Input values			Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

GE has previously been applied to this problem [9], and the standard grammar used in this experiment is the same:

```
<mult> ::= output = <bexpr> ;
<bexpr> ::= ( <bexpr> <bilop> <bexpr> ) | <ulop> ( <bexpr> ) | <input>
<bilop> ::= and | or
<ulop> ::= not
<input> ::= input0 | input1 | input2
```

After applying the reduction process, the generated reduced grammar is:

```
<mult> ::= output = <bexpr> ;
<bexpr> ::= ( <bexpr> and <bexpr> ) | ( <bexpr> and <bexpr> ) | ( <bexpr> and <bexpr> )
        | ( <bexpr> or <bexpr> ) | ( <bexpr> or <bexpr> ) | ( <bexpr> or <bexpr> )
        | not ( <bexpr> ) | not ( <bexpr> ) | not ( <bexpr> )
| not ( <bexpr> ) | not ( <bexpr> ) | not ( <bexpr> )
        | input0 | input0 | input1 | input1 | input2 | input2
```

The results obtained with both grammars are shown in Figure 6.

5 Analysis

Across all experiments, a small increase in performance can be seen when using a reduced grammar, rather than the original grammar. This increase can be explained by the fact that, when applying the crossover operator between two individuals, the second halves of both parents have a higher probability of being interpreted within the same context. That is, the codons on each second half, when exchanged, are likely to still be used to choose a production for the same non-terminal symbol as they were on the original parent strings.

The reported increase in performance is, however, non-statistically significant: mean best individual plots with error bars showed an overlap of standard deviation between the two grammars used, across all experiments. A possible justification for this fact is related to a hotly debated subject in the GP field: is there an equivalent in GP to the notion of building-blocks in the GA literature? Several studies have addressed this subject [10], but no general consensus has been reached as of yet. One of the reasons seems to be the variety of problem

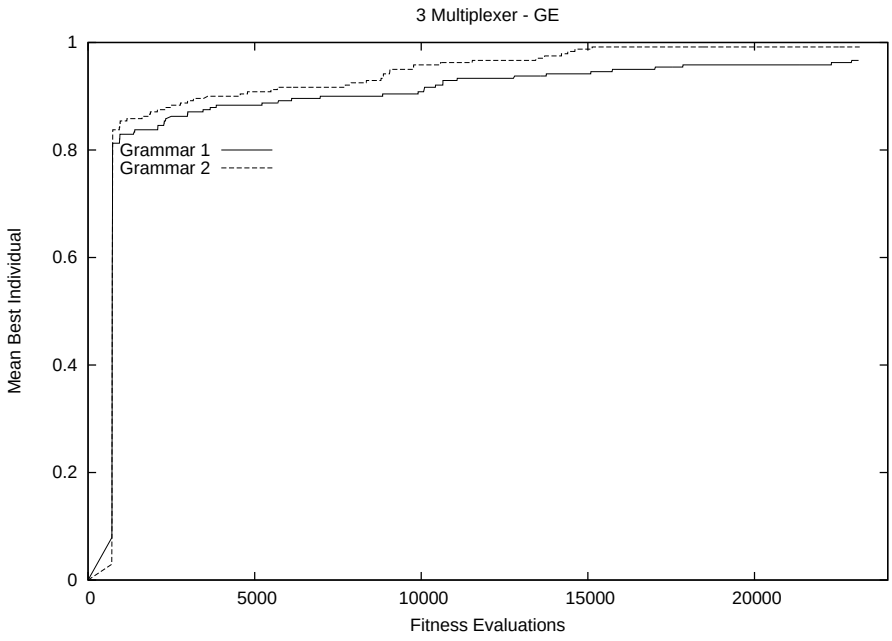

Fig. 6. Results obtained for the Multiplexer problem, using the original and reduced grammars. The *x-axis* shows the number of fitness evaluations, and the *y-axis* the mean best individual (from 30 independent runs).

domains that GP can be applied to, and to what defines a building-block in each of these problem domains.

Another possible explanation is closely related. If there are no building-blocks on the problems tackled, or if they exist but are not properly exchanged between individuals, then it is normal to expect a wide variance in the results obtained with different runs, on the problems tackled in this paper.

6 Conclusions and Future Work

When using complex grammars with GE, the higher the number of non-terminal symbols in the grammar, the higher the chance of a change of context for the genetic material exchanged through crossover. Based on this reasoning, this paper has presented a simple method to reduce the number of non-terminal symbols in any context-free grammar written in BNF format.

The results obtained on a set of experiments show an increase in performance, when compared with the standard GE approach. The significance or otherwise of the results obtained has its roots on the question of whether it is useful to exchange sub-portions of individuals (building-blocks) in genetic programming.

A possible continuation of this work might involve identify useful blocks for exchange between individuals, choosing appropriate crossover points. The less likelihood of context changes by using reduced grammars could prove beneficial in this scenario.

References

1. Banzhaf, W.: Genotype-Phenotype Mapping and Neutral Variation - A case study in Genetic Programming. In: Davidor et al., (Eds.): Proceedings of the third conference on Parallel Problem Solving from Nature. Lecture Notes in Computer Science, Vol. 866. Springer-Verlag. (1994) 322–332
2. Kimura, M.: The Neutral Theory of Molecular Evolution. Cambridge University Press. (1983)
3. Koza, J.: Genetic Programming. MIT Press. (1992)
4. O'Neill, M.: Automatic Programming in an Arbitrary Language: Evolving Programs with Grammatical Evolution. PhD Thesis, University of Limerick (2001)
5. O'Neill, M., and Ryan, C.: Grammatical Evolution. IEEE Transactions on Evolutionary Computation, Vol. **5**, No. 4. (2001) 349–358
6. O'Neill, M., Ryan, C., and Nicolau, M.: Grammar Defined Introns: An Investigation Into Grammars, Introns, and Bias in Grammatical Evolution. In Spector et al., (Eds.): Proceedings of the Genetic and Evolutionary Computation Conference GECCO-2001. Morgan Kaufmann Publishers, San Francisco. (2001) 97–103
7. O'Neill, M., Ryan, C., Keijzer, M., and Cattolico, M.: Crossover in Grammatical Evolution. Genetic Programming and Evolvable Machines, Vol. **4**, No. 1. (2003) 67–93
8. O'Neill, M. and Ryan, C.: Grammatical Evolution - Evolving programs in an arbitrary language. Kluwer Academic Publishers. (2003)
9. O'Neill, M., Brabazon, A., Nicolau, M., Mc Garraghy, S., and Keenan, P.: πGrammatical Evolution. In: Deb et al, (Eds.): Proceedings of the Genetic and Evolutionary Computation Conference - GECCO 2004. Lecture Notes in Computer Science, to be published. Springer-Verlag. (2004)
10. O'Reilly, U., and Oppacher, F.: The Troubling Aspects of a Building Block Hypothesis for Genetic Programming. In: Whitley and Vose, (Eds.): Foundations of Genetic Algorithms 3. Morgan Kaufmann Publishers, San Francisco. (1995) 73–88