πGrammatical Evolution

Michael O'Neill[1], Anthony Brabazon[2], Miguel Nicolau[1], Sean Mc Garraghy[2],
and Peter Keenan[2]

[1] Biocomputing and Developmental Systems Group
University of Limerick, Ireland
{Michael.ONeill, Miguel.Nicolau}@ul.ie
[2] University College Dublin, Ireland
{Anthony.Brabazon, john.mcgarraghy, Peter.Keenan}@ucd.ie

Abstract. πGrammatical Evolution is presented and its performance
on four benchmark problems is reported. πGrammatical Evolution is a
position-independent variation on Grammatical Evolution's genotype-
phenotype mapping process where the order of derivation sequence steps
are no longer applied to nonterminals in a predefined fashion from left to
right on the developing program. Instead the genome is used to specify
which nonterminal will be developed next, in addition to specifying the
rule that will be applied to that nonterminal. Results suggest that the
adoption of a more flexible mapping process where the order of non-
terminal expansion is not determined a-priori, but instead itself evolved,
is beneficial for Grammatical Evolution.

1 Introduction

In standard Grammatical Evolution (GE) [1], the practical effect of the appli-
cation of each rule to the non-terminals of the developing program is dependent
upon all previous rule applications. As we successfully choose rules to apply to
the developing program, given the tight coupling of the dependency from left to
right, the greater the probability of choosing an inappropriate rule at some point
during the creation of the program. Therefore, as programs become bigger the
harder it will be to find the target solution due to the increased likelihood of pick-
ing sub-optimal rules. In other words, in GE, it is hard to identify clear building
blocks apart from the extremely fine-grained individual production rules, al-
though analysis of GE's ripple crossover has provided some evidence to support
the useful exchange of derivation subsequences during crossover events [2]. In
terms of schema theory this has serious implications for the faithful propaga-
tion of these coarse-grained building blocks, and the propagation of the more
fine-grained production rule building blocks does not facilitate the creation of
hierarchical structures to any great degree. It must be stressed, however, that
the GE representation has been shown to be extremely robust and productive
at producing solutions on a broad range of problems, seemingly exploiting the
tight dependencies that exist within the representation. A representation that
exploited these dependencies on a smaller scale, allowing derivation subsequences
to act as building blocks, may provide more productive evolutionary search by

providing better building blocks to achieve a hierarchical solution construction. It is through GE's feature of intrinsic polymorphism that derivation subsequences can be exchanged to different locations with sometimes the same or indeed different contexts.

The main idea behind position independent GE, πGE[1], is to break the dependency chain into smaller fragments that can be exchanged between appropriate contexts through a specific position independent mechanism open to evolutionary search. This should facilitate the creation of smaller, functional, building blocks, similar to sub-trees in GP, which may be easier to preserve and thus should enhance the scalability of GE to harder problem domains. Given the position independent nature of the representation it means that as long as a rule, whether structural or terminal, is present, its context can then be adapted to solve the problem at hand. In addition, it is unclear what effect the depth first non-terminal expansion adopted in the standard GE mapping process has on performance. To this end, this study compares the performance of the standard mapping ordering to πGE's mapping where the order of non-terminal expansion is explicitly evolved.

The remainder of the paper is structured as follows. The next section provides a brief background on Grammatical Evolution, with Section 3 introducing πGE. The problem domains tackled, experimental setup, and results are provided in Section 4, followed by conclusions and future work (Section 5).

2   Grammatical Evolution

Grammatical Evolution (GE) is an evolutionary algorithm that can evolve computer programs in any language [1,2,3,4,5], and can be considered a form of grammar-based genetic programming. Rather than representing the programs as parse trees, as in GP [6,7,8,9,10], a linear genome representation is used. A genotype-phenotype mapping is employed such that each individual's 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. The grammar allows the generation of programs in an arbitrary language that are guaranteed to be syntactically correct, and as such it is used as a generative grammar, as opposed to the classical use of grammars in compilers to check syntactic correctness of sentences. The user can tailor the grammar to produce solutions that are purely syntactically constrained, or they may incorporate domain knowledge by biasing the grammar to produce very specific forms of sentences.

BNF is a notation that represents a language in the form of production rules. It is comprised of a set of non-terminals that can be mapped to elements of the set of terminals (the primitive symbols that can be used to construct the output program or sentence(s)), according to 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

---

[1] The name πGE is inspired by *Life of Pi* by Yann Martel in which a young boy Piscine Patel embarrassed by his name decides to rename himself π.

either `<expr><op><expr>` or `<var>`. An `<op>` can become either `+`, `-`, or `*`, and a `<var>` can become either `x`, or `y`.

```
<expr> ::= <expr><op><expr> (0)
         | <var>            (1)
  <op> ::= +                (0)
         | -                (1)
         | *                (2)
 <var> ::= x                (0)
         | y                (1)
```

The grammar is used in a developmental process to construct a program by applying production rules, selected by the genome, beginning from the start symbol of the grammar. In order to select a production rule in GE, the next codon value on the genome is read, interpreted, and placed in the following formula:

$$Rule = Codon\ Value\ \%\ Num.\ Rules$$
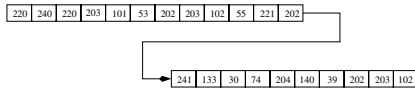
where % represents the modulus operator.

Fig. 1. An example GE individual's genome represented as integers for ease of reading.

Given the example individual's genome (where each 8-bit codon has been represented as an integer for ease of reading) in Fig.1, the first codon integer value is 220, and given that we have 2 rules to select from for `<expr>` as in the above example, we get 220 % 2 = 0. `<expr>` will therefore be replaced with `<expr><op><expr>`.
Beginning from the left hand side of the genome codon integer values are generated and used to select appropriate rules for the left-most non-terminal in the developing program from the BNF grammar, until one of the following situations arise: (a) 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. (b) 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. (c) 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. Returning to the example individual, the left-most `<expr>` in `<expr><op><expr>` is mapped by reading the next codon integer value 240 and used in 240 % 2 = 0 to become another `<expr><op><expr>`. The developing program now looks like

`<expr><op><expr><op><expr>`. Continuing to read subsequent codons and always mapping the left-most non-terminal the individual finally generates the expression `y*x-x-x+x`, leaving a number of unused codons at the end of the individual, which are deemed to be introns and simply ignored. A full description of GE can be found in [1].

3 πGrammatical Evolution

In the first derivation step of the example mapping presented earlier, `<expr>` is replaced with `<expr><op><expr>`. Then in the standard GE genotype-phenotype mapping process, the left-most non-terminal (the first `<expr>`) in the developing program is always expanded first. The πGE mapping process differs in an individual's ability to determine and adapt the order in which non-terminals will be expanded. To this end, a πGE codon corresponds to the pair $(nont, rule)$, where $nont$ and $rule$ are represented by N bits each (N=8 in this study), and a chromosome, then, consists of a vector of these pairs.

In πGE, we analyse the state of the developing program before each derivation step, counting the number of non-terminals present. If there is more than one non-terminal present in the developing program the next codons $nont$ value is read to pick which non-terminal will be mapped next according to the following mapping function, where NT means non-terminal:

$$NT = Codon \; nont \; Value \; \% \; Num. \; NT's.$$

In the above example, there are 3 non-terminals ($\texttt{<expr>}_0\texttt{<op>}_1\texttt{<expr>}_2$) after application of the first production rule. To decide which non-terminal will be expanded next we use $NT = 9 \; \% \; 3 = 0$, i.e., $\texttt{<expr>}_0$ is expanded (see second codon $(9, 102)$ left-hand side of Fig. 2). The mapping rule for selecting the appropriate rule to apply to the current non-terminal is given in the normal GE fashion:

$$Rule = Codon \; rule \; Value \; \% \; Num. \; Rules.$$

In this approach, evolution can result in a derivation subsequence being moved to a different context as when counting the number of non-terminals present we do not pay attention to the type of non-terminals (e.g. `<expr>` versus `<op>`). An alternative approach to πGE is to respect non-terminal types and only allow choices to be made between non-terminals of the same type, thus preserving the semantics of the following derivation subsequence, and simply changing the position in which it appears in the developing program. An example of this occurring can be seen in Fig. 2.

One could consider πGE to be similar in approach to the position independent GAuGE system [11] and related through its positional independence nature to Chorus [12]. However, aside from the motivational differences (i.e, to facilitate the evolution of derivation sub-sequence building blocks), there are a number of distinguishing characteristics arising from the type of position independence
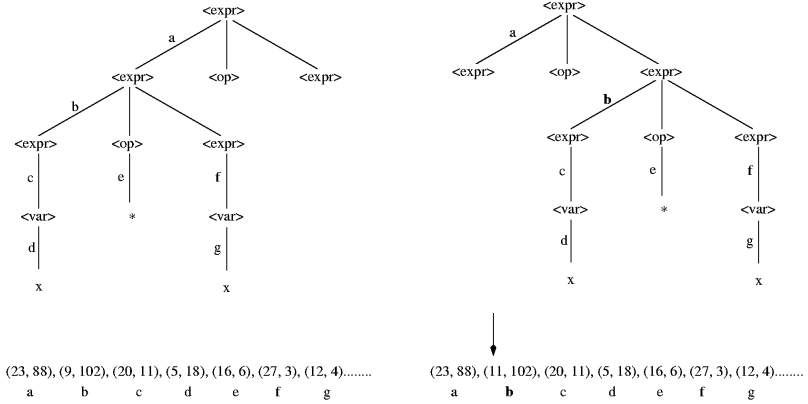
Fig. 2. On the bottom left an example πGE individuals' genome, with the 2 part 8-bit codons represented as integers for ease of reading, and its corresponding derivation tree (top left) can be seen. The right side of this figure illustrates the effect of an example mutation to the *nont* value of the second codon on the resulting derivation tree, where the subtree from the left-most `<expr>` has moved to the right-most non-terminal, also an `<expr>` in this case.
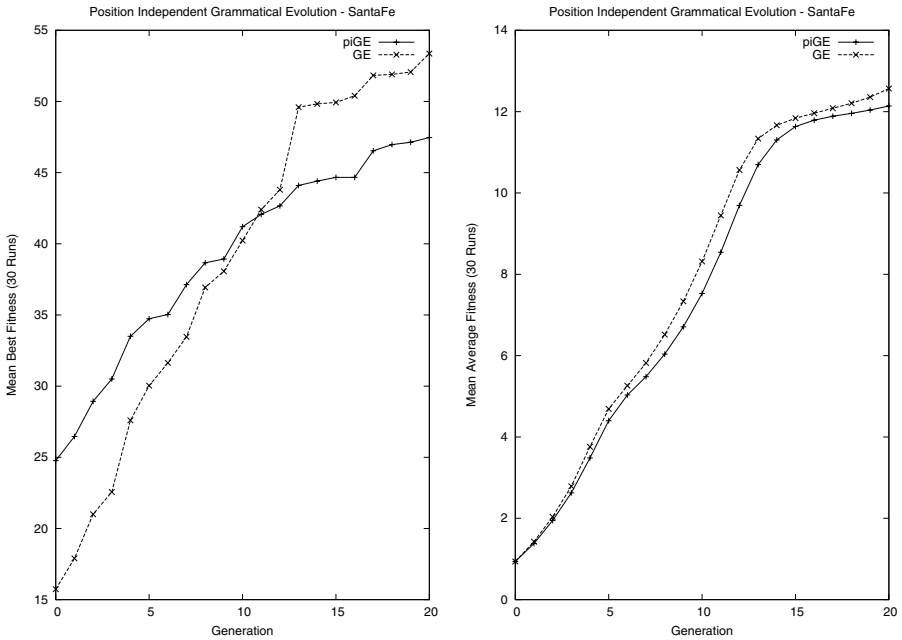

Fig. 3. Plot of the mean best and mean average fitness on the Santa Fe Ant Trail problem instance.
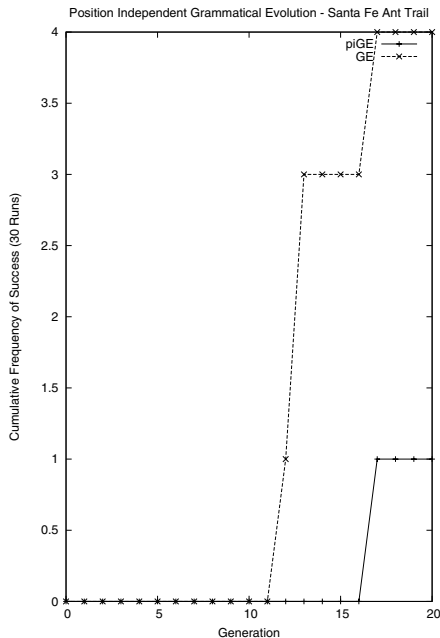
Fig. 4. Plot of cumulative frequency of success on the Santa Fe Ant Trail problem instance.

introduced. For example, the application of position independence to the order in which GE's mapping process occurs, as opposed to the order of bits in a phenotype; in the variable-length chromosomes due to the nature of the mapping process; and, in the constantly fluctuating ordering choices over the lifetime of the mapping process. Instead of making choices between a fixed number of homologous positions on a linear solution vector as in GAuGE, πGE makes choices between the variable-number of heterogeneous non-terminals that may exist in a developing program at any one point in the derivation sequence. The number of position choices available is no longer constant, instead depending on the state of the derivation sequence, with the number of available positions increasing as well as decreasing over time. Unlike in Chorus, where each gene's (referred to as codons in GE) meaning is fixed (i.e. the same gene corresponds to a specific grammar production rule irrespective of its location on the chromosomes), πGE maintains the intrinsic polymorphism property characteristic of the GE genotype-phenotype mapping. That is, a codon's meaning adapts to its context.

4   Experiments and Results

A variety of benchmark problems are tackled to demonstrate the feasibility of πGE, namely, the Santa Fe ant trail, a symbolic regression instance, mastermind, and a multiplexer instance. By feasibility of πGE, we mean that we wish to test

Position Independent Grammatical Evolution - Quartic Symbolic Regression

Position Independent Grammatical Evolution - Quartic Symbolic Regression
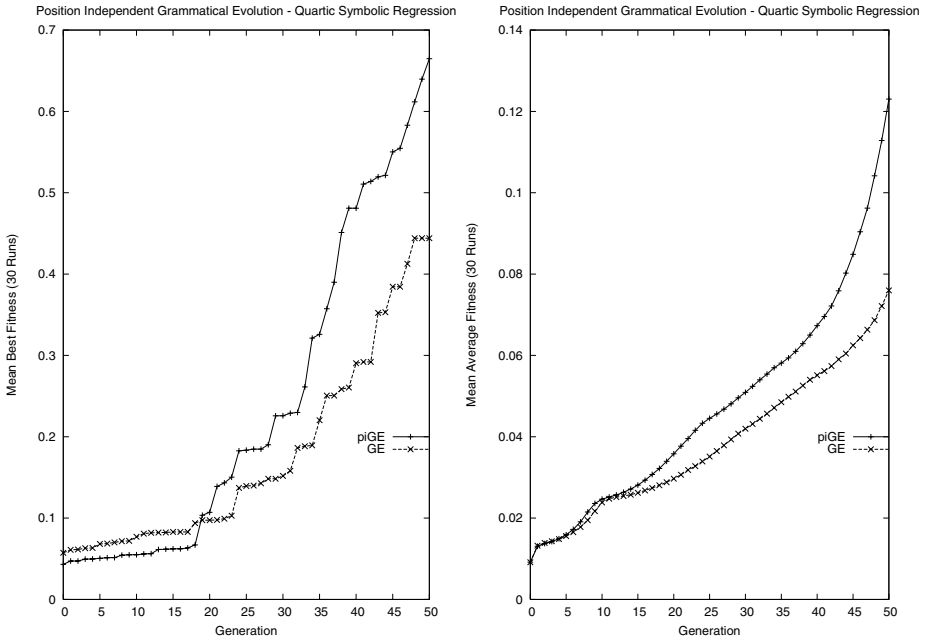
Fig. 5. Plot of the mean best and mean average fitness on the quartic symbolic regression problem instance.

if the standard depth first expansion of non-terminal symbols in a developing program can be improved upon by allowing the order of non-terminal expansion to be evolved. Performance is compared to GE on the same problem set. A random initialisation procedure is adopted that selects the number of codons in each individual to be in the range 1 to 20 codons. Wrapping is allowed with an upper limit of 10 events, and only mutation (probability of 0.01 per bit) and one-point crossover (probability of 0.9) are adopted, along with roulette selection, and a steady state replacement.

4.1 Santa Fe Ant Trail

The Santa Fe ant trail is a standard problem in the area of GP and can be considered a deceptive planning problem with many local and global optima [13]. The objective is to find a computer program to control an artificial ant so that it can find all 89 pieces of food located on a non-continuous trail within a specified number of time steps, the trail being located on a 32 by 32 toroidal grid. The ant can only turn left, right, move one square forward, and may also look ahead one square in the direction it is facing to determine if that square contains a piece of food. All actions, with the exception of looking ahead for food, take one time step to execute. The ant starts in the top left-hand corner of the grid facing the first piece of food on the trail. The grammar used in this problem
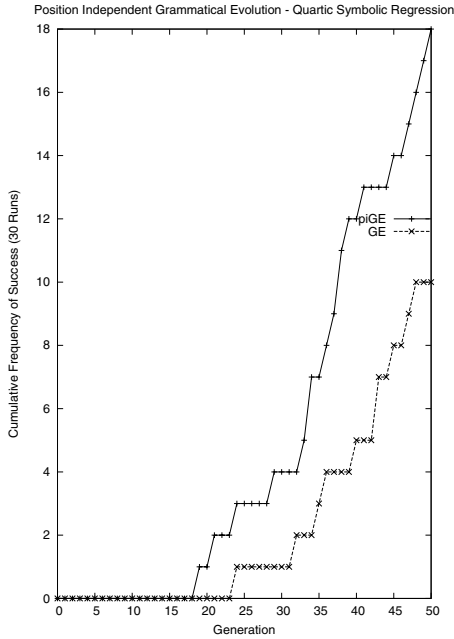
Fig. 6. Plot of cumulative frequency of success on the quartic symbolic regression problem instance.

is different to the ones used later for symbolic regression, mastermind, and the multiplexer problems in that we wish to produce a multi-line function in this case, as opposed to a single line expression. The grammar for the Santa Fe ant trail problem is given below.

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

Figures 3 and 4 shows the performance of πGE and GE on the Santa Fe ant trail problem using population sizes of 500 for 20 generations, with GE finding the greater number of correct solutions. Interestingly, πGE appears to achieve higher fitness levels earlier on in the run compared to GE.

4.2   Quartic Symbolic Regression

An instance of a benchmark symbolic regression problem is tackled in order to further verify that it is possible to generate programs using πGE. The target function is $f(a) = a + a^2 + a^3 + a^4$, and 100 randomly generated input vectors are created for each call to the target function, with values for the input variable drawn from the range [0,1]. The fitness for this problem is given by the reciprocal of the sum, taken over the 100 fitness cases, of the absolute value of the
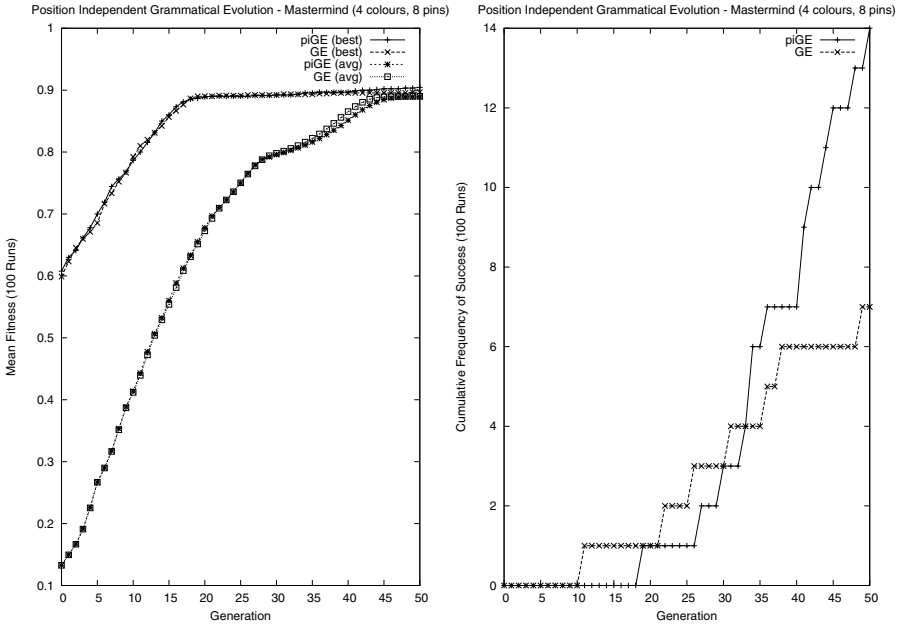
Fig. 7. Plot of the mean best and mean average fitness (left) and the cumulative frequency of success (right) on the Mastermind problem instance using 8 pins and 4 colours.

error between the evolved and target functions. The grammar adopted for this problems is as follows:

```
<expr> ::= <expr> <op> <expr> | <var>
<op> ::=  + | - | * | /
<var> ::= a
```

Results are presented for population sizes of 500 running for 50 generations in Fig's. 5 and 6, where it can be seen clearly that πGE outperforms GE.

4.3 Mastermind

In this problem the code breaker attempts to guess the correct combination of coloured pins in a solution. When an evolved solution to this problem (i.e. a combination of pins) is to be evaluated, it receives one point for each pin that has the correct colour, regardless of its position. If all pins are in the correct order than an additional point is awarded to that solution. This means that ordering information is only presented when the correct order has been found for the whole string of pins. A solution, therefore, is in a local optimum if it has all the correct colours, but in the wrong positions. The difficulty of this problem is controlled by the number of pins and the number of colours in the target combination. The instance tackled here uses 4 colours and 8 pins with the following values 3 2 1 3 1 3 2 0. The grammar adopted is as follows.
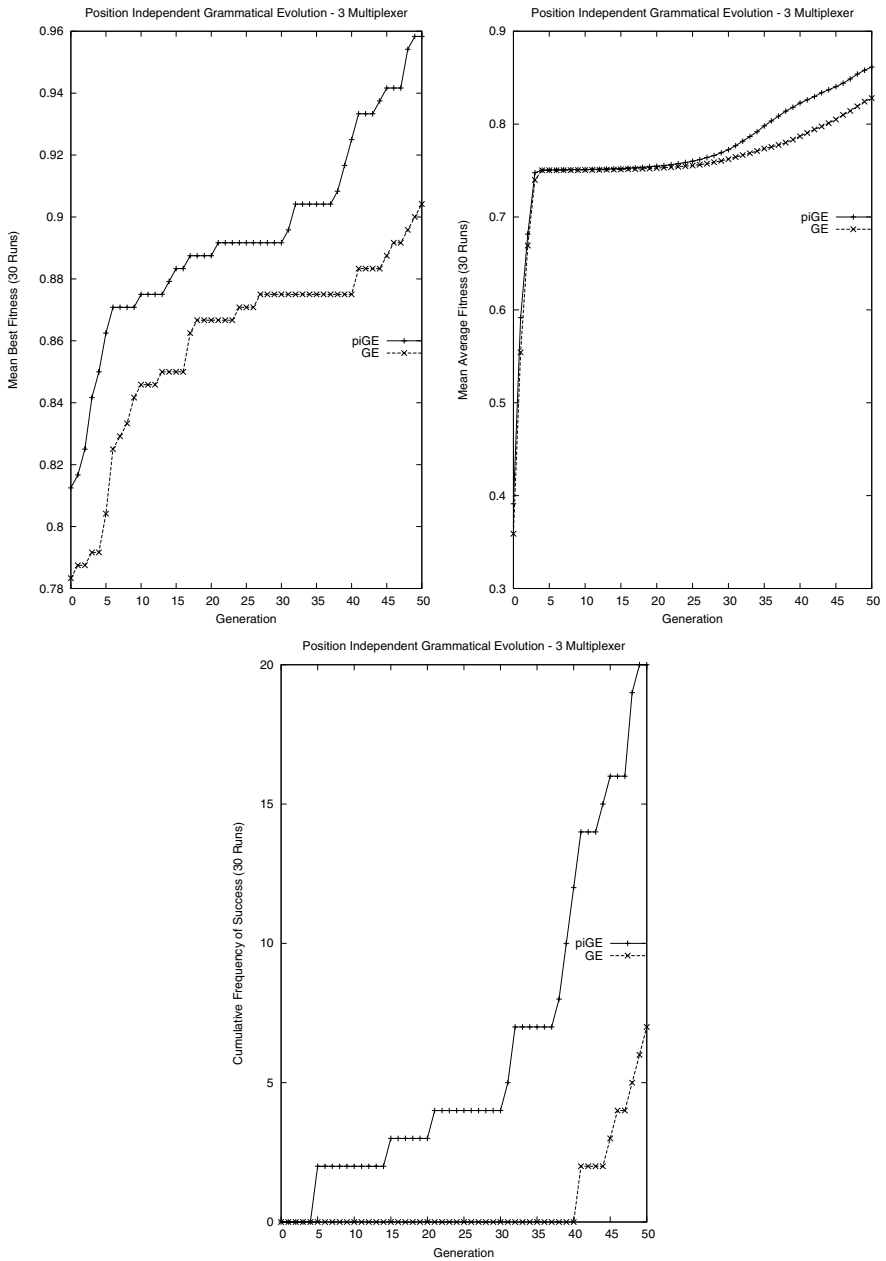
Fig. 8. Plot of the mean best and mean average fitness (top) and cumulative frequency of success (bottom) on the 3 Multiplexer problem instance.

```
<pin> ::= <pin> <pin> | 0 | 1 | 2 | 3
```

Runs are conducted for 50 generations with population sizes of 500, and the results are provided in Fig. 7.

4.4   3 Multiplexer

The aim with this problem is to discover a boolean expression that behaves as a 3 Multiplexer. There are 8 fitness cases for this instance, representing all possible input-output pairs. Fitness is the number of input cases for which the evolved expression returns the correct output. The grammar adopted is given below, and results are presented in Fig. 8 using a population size of 1000 running for 50 generations, where it can be seen clearly that πGE outperforms GE.

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

Table 1. A comparison of the results obtained for GE and πGE across all the problems analysed.

	Mean Best Fitness (Std.Dev.)	Mean Average Fitness (Std.Dev.)	Successful Runs
Santa Fe ant			
πGE	47.47 (10.98)	12.14 (0.44)	1
GE	**53.37** (20.68)	**12.57** (0.68)	**4**
Symbolic Regression			
πGE	**.665** (0.42)	**.123** (0.07)	**18**
GE	.444 (0.4)	.076 (0.05)	10
Mastermind			
πGE	**.905** (0.04)	.89 (0.001)	**14**
GE	.897 (0.02)	.89 (0.003)	7
Multiplexer			
πGE	**.958** (0.06)	**.861** (0.04)	**20**
GE	.904 (0.05)	.828 (0.05)	7

5   Conclusions and Future Work

This study demonstrates the feasibility of the generation of computer programs using πGE, and provides evidence to support positive effects on performance using a position independent derivation sequence over the standard left to right GE mapping. A comparison of the performance of πGE with GE across all the

problems presented here can be seen in Table 1. With the exception of the Santa Fe ant trail problem, πGE outperforms GE.

The additional degree of freedom provided in evolving the choice of non-terminal to which a production rule is applied has been demonstrated to have a positive influence across the majority of problems analysed here.

Looking towards future research on πGE, we wish to analyse derivation subsequence propagation, looking at mutation and crossover events to codon *nont* values, and to undertake an analysis of the parallels between πGE, GAuGE, and Chorus. We wish to test the hypothesis that the position-independent nature of the representation might allow the formation and propagation of more useful derivation subtree building blocks as derivation subsequences can easily move between non-terminal positions in the derivation sequence.

It would also be interesting to investigate variations on the position independent mapping theme introduced by πGE, for example, to determine the effects of restricting position choices to non-terminals of the same type. It is also our intention to test the generality of the results across a number of additional problems domains.

References

1. O'Neill, M., Ryan, C. (2003). *Grammatical Evolution: Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers.
2. O'Neill, M., Ryan, C., Keijzer M., Cattolico M. (2003). Crossover in Grammatical Evolution. *Genetic Programming and Evolvable Machines*, Vol. 4 No. 1. Kluwer Academic Publishers, 2003.
3. O'Neill, M. (2001). *Automatic Programming in an Arbitrary Language: Evolving Programs in Grammatical Evolution.* PhD thesis, University of Limerick, 2001.
4. O'Neill, M., Ryan, C. (2001). Grammatical Evolution, *IEEE Trans. Evolutionary Computation.* Vol. 5, No. 4, 2001.
5. Ryan, C., Collins, J.J., O'Neill, M. (1998). Grammatical Evolution: Evolving Programs for an Arbitrary Language. *Proc. of the First European Workshop on GP*, LNCS 1391, Paris, France, pp. 83-95, Springer-Verlag.
6. Koza, J.R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press.
7. Koza, J.R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.
8. 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.
9. Koza, J.R., Andre, D., Bennett III, F.H., Keane, M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving.* Morgan Kaufmann.
10. Koza, J.R., Keane, M., Streeter, M.J., Mydlowec, W., Yu, J., Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence.* Kluwer Academic Publishers.

11. Ryan, C., Nicolau, M., O'Neill, M. (2002). Genetic Algorithms Using Grammatical Evolution. *Proc. of the 4th European Conference on Genetic Programming, EuroGP 2002*, LNCS 2278, pp. 279-288. Springer-Verlag.
12. Ryan, C., Azad, A., Sheahan, A., O'Neill, M. (2002). No Coercion and No Prohibition, A Position Independent Encoding Scheme for Evolutionary Algorithms—The Chorus System. *Proc. of the 4th European Conference on Genetic Programming, EuroGP 2002*, LNCS 2278, pp. 132-142. Springer-Verlag.
13. Langdon, W.B., and Poli, R. (1998). Why Ants are Hard. In *Genetic Programming 1998: Proc. of the Third Annual Conference*, University of Wisconsin, Madison, Wisconsin, USA, pp. 193-201, Morgan Kaufmann.