Evolving Multi-line Compilable C Programs

Michael O'Neill & Conor Ryan

Dept. Of Computer Science And Information Systems
University of Limerick
Ireland
{Michael.ONeill|Conor.Ryan}@ul.ie

Abstract. We describe a Genetic Algorithm called *Grammatical Evolution* (GE) that can evolve complete programs in an arbitrary language using a variable length linear genome. The binary genome determines which production rules in a Backus Naur Form grammar definition are used in a genotype to phenotype mapping process to a program. Expressions and programs of arbitrary complexity may be evolved using this system.
Since first describing this system, GE has been applied to other problem domains, and during this time GE has undergone some evolution. This paper serves to report these changes, and also describes how we evolved multi-line C-code to solve a version of the Santa Fe Ant Trail. The results obtained are then compared to results produced by Genetic Programming, and it is found that GE outperforms GP on this problem.

1 Introduction

Grammatical Evolution (GE) is a Genetic Algorithm based system which evolves programs in any language using a Backus Naur Form (BNF) definition. Unlike Genetic Programming (GP), GE does not perform the evolutionary process on the actual programs but rather on simple binary strings. Various attempts at using BNF grammars with Genetic Programming have been made [13] [14] [4] [9] [3] with some success. We describe here a different approach to using BNF grammars for automatic programming, using a biologically inspired genotype to phenotype mapping process, where the genes are used to select production rules in a BNF grammar definition.
The original paper on GE was concerned with tackling a Symbolic Regression problem [10], and has subsequently been applied to a Symbolic Integration problem [12], and to solving Trigonometric Identities [11]. During this time it has been found that using a Steady State replacement strategy has enhanced the performance of GE.
We begin by presenting GE in its evolved state, before describing how GE was successfully applied to evolving multi-line C-code when finding a solution to the Santa Fe Trail. The results obtained whilst tackling the Santa Fe Trail are then compared to results which were produced by a GP system, and it is shown that GE outperforms GP on this problem.

2 Grammatical Evolution

Our system codes a set of pseudo random numbers, which are used to decide which production rule to use when a non terminal of the BNF definition has one or more possible outcomes. A description of BNF definitions is given in Appendix A, along with the definition used for the Santa Fe Trail problem described later. A chromosome in GE consists of a variable number of 8 bit binary genes, each of which encodes an integer value.

Consider an example of a BNF production rule, #5 from the BNF definition in Appendix A:

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

In this case, the non-terminal *op* can produce one of three different results, our system takes the next available number (gene integer value) from the chromosome to decide which production to use and gets the modulus of it by the number of rules to choose from, in this case three. Each time a decision has to be made, another pseudo random number is read from the chromosome, and in this way, the system traverses the chromosome.

The genetic code of biological organisms is usually contained in a special molecule called DNA. For simplicity, consider DNA to be a string of building blocks called nucleotides, of which there are four, named A, T, G, and C, for Adenine, Tyrosine, Guanine, and Cytosine respectively. Groups of three nucleotides, called a codon, are used to specify the building blocks of proteins. These protein building blocks are known as Amino Acids, and the sequence of these Amino Acids in the protein is determined by the sequence of codons on the DNA. The sequence of Amino Acids is very important as it plays a large part in determining the final three dimensional structure of the protein, which in turn has a role to play in determining its functional properties. Proteins have a major role to play in biological organisms, such as, in metabolism, and as structural components of the body to name but a few. Figure 2 outlines the mapping process described here from Genetic Code to Phenotype in biological organisms.

In a fashion similar to natural biology the genes in GE are expressed as proteins, in our case terminals, Figure 2. These terminals can act either independently or in conjunction with other terminals [1], the physical results depending on the other terminals that are present immediately before and after a genes expression. During the genotype to phenotype mapping process it is possible for individuals to run out of genes, and in this case we wrap the individual, and reuse the genes. This is quite an unusual approach in EAs, as it is entirely possible for certain genes to be used two or more times. This technique of wrapping the individual draws inspiration from the gene overlapping phenomenon which has been observed in many organisms in nature [1]. In GE, each time the gene is expressed it will always generate the same integer, but depending on the other rules present, may have a different effect, that is, it may select a different production rule. What is crucial, however, is that each time a particular individual is mapped from its genotype to its phenotype, the same output is generated. This is because the same choices are made each time.

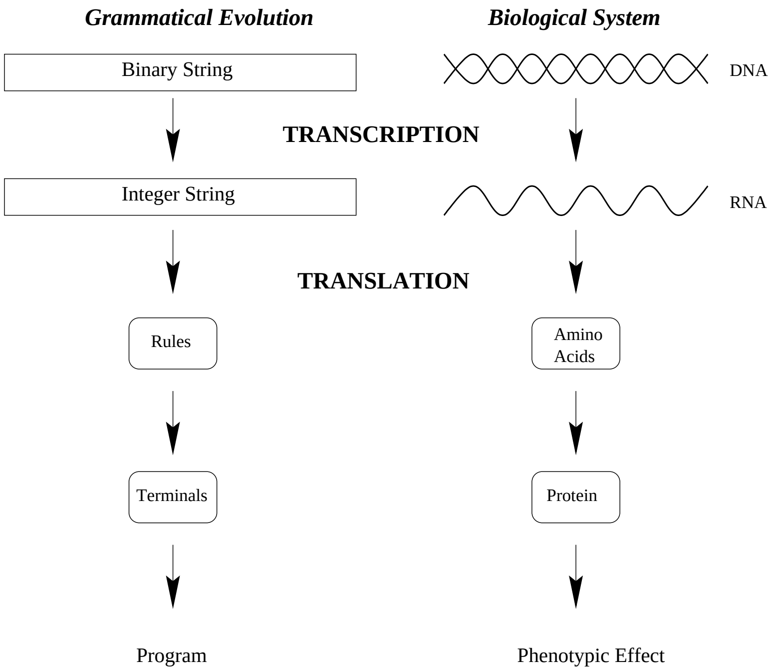

Fig. 1. The Grammatical Evolution System and a Biological System

An interesting point to note in the mapping process is that given an 8 bit binary number, each gene can represent 256 distinct integer values. However, many of these integer values can represent the same production rule, taking production rule 5 above as an example, if the current gene value was 6, then 6 MOD 3 = 0 would select rule (A) $left()$. The same rule would be chosen if the gene value was 3, 9, 12, etc. A similar phenomenon can be observed in the genetic code of biological organisms, referred to as *Degenerate Genetic Code* [1].
There are 4^3, i.e. 64, unique combinations of nucleotides in a codon, 61 of these coding for a specific Amino Acid, the other three are special codons which delimit the end of a gene on the DNA. On average, there are three codons for every amino acid, that is more than one codon can represent the same Amino Acid. This code degeneracy has interesting implications when it comes to mutation effects. A mutation at the third codon position

can often produce what is called a *silent mutation*, meaning that the Amino Acid specified will be the same as the one before the mutation event. With respect to GE this means that subtle changes in the search space (genotype) may have no effect on the solution space (phenotype). In Figure 2 we see how in the genetic code of biological organisms, the nucleotide at position three of the codon is independent of the amino acid produced (Valine) for this example. Similarly with GE it can be seen in the given example that a single bit mutation has no effect on the rule used in this case i.e. $2\ MOD\ 2 = 18\ MOD\ 2 = 34\ MOD\ 2 = 66\ MOD\ 2 = 0$, the 0th rule $< line >$.

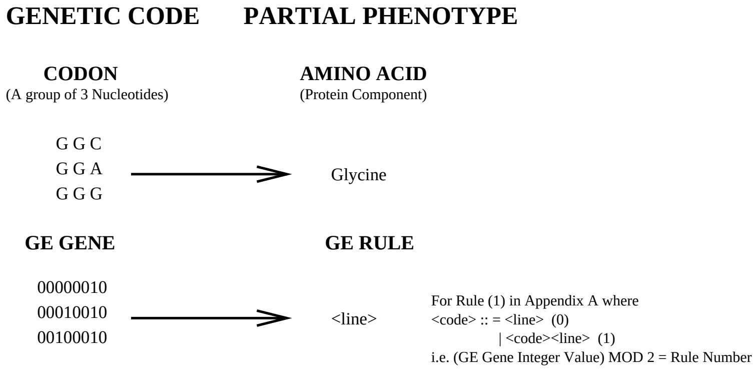

Fig. 2. Genetic Code Degeneracy

2.1 Genetic Operators

In [10] we described two distinctive genetic operators, Pruning, and Duplication. While Duplication is still used in our system in the hope that it can produce either more of a particular protein, or provide a means by which new functionality can be obtained independently of the current functionality, the Pruning operator is used to a lesser extent. As use is made of the wrapping feature, described earlier, Pruning would remove any of the introns at the end of a chromosome, thus removing potentially useful genetic material. For this reason, and the fact that introns have been shown to be beneficial in other Genetic Algorithms [8], Pruning is now applied with a much smaller probability typically 0.01.

2.2 Steady State

[12] described how a Steady State replacement mechanism was employed with GE as opposed to the previous Generational mechanism with great success. GE's performance over three problem domains was reported for both replacement mechanisms, and the results are summarised in Table 1. We believe that the Steady State replacement mechanism is better

able to preserve those solutions with higher fitness, propagating them from generation to generation, whereas previously it was common for these high performing individuals to be lost to a combination of destructive crossover and mutation events, and misfortune during selection.

Table 1. Comparison of successful runs for 3 problems.

Problem Domain	**% Successful**	
	Steady State	Generational
Symbolic Regression	95	2
Trigonometric Identities	87	4
Symbolic Integration	87	22

3 The Problem Space

In order to show GE can produce multi-line functions the Santa Fe Ant Trail problem is tackled. The Santa Fe Ant Trail has been used as a benchmark problem in the area of Genetic Programming, and can be considered a deceptive planning problem with many local and global optima [7]. 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 code evolved is then executed in a loop until the number of time steps allowed has elapsed. The ant can only turn left, right, move forward one square, and may also look ahead one square in the direction it's facing to determine if that square contains a piece of food. The actions left, right, and move each take one time step to execute.
While there are many possible fitness cases to the Santa Fe Trail only one case was taken for the purposes of this experiment. The ant started in the top left hand corner of the grid facing the first piece of food on the trail. The grammar used in this problem is as given in Appendix A, note that we do not use an equivalent for $PROGN$. A summary of the problem can be seen in Table 2.

4 Results

The problem was firstly tackled with GE using a population size of 500 individuals running for a maximum of 50 generations. The only fitness measure used was the number of pieces of food the ant picked up over 600 time steps. GE was successful at finding a solution to this case of the Santa Fe Trail, and indeed was capable of producing multi-line code. It was then decided to run this problem using GP to compare it with the performance of GE. The GP system, as described in [2], uses the solutions length as a measure of fitness, as well as the number of pieces of food picked up.

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	$M = 500$, $G = 51$

During the initial runs using GP, and also with GE, it was noted that a lot of solutions picked up all but one of the pieces of food within the 600 time steps. Given a few more time steps these solutions would achieve perfect fitness in terms of the number of pieces of food picked up, so it was decided to allow the number of time steps to be increased to 615. 100 runs using each system were carried out, allowing the GP system to use the solution length as a measure of fitness, and allowing GE no such measure. A cumulative frequency measure for both of these runs can be seen in Figure 3. As can be seen performance of the two systems is on a par with each other, with GP slightly outperforming GE in the initial half of the run.

In order to see how the GP system would perform without the solution length as a measure of fitness, we ran 100 more runs of the GP system removing the solution length fitness measure. We felt that using solution length as a measure of fitness required a prior knowledge of the solutions length and therefore gave an unfair advantage to the GP system as GE did not use such a measure. Figure 4 shows a comparison of the cumulative frequency measure for these results with the results produced by GE. As can be seen from the figure the performance of the GP system was compromised and as a result GE outperformed GP.

Sample solutions produced by both systems are given below.

GP example solution:

```
(PROG2 (PROG2 (IF-FOOD-AHEAD LEFT RIGHT) RIGHT) (PROG2
(IF-FOOD-AHEAD FORWARD RIGHT) (PROG2 FORWARD RIGHT)))
```

GE example solution:

```
if(food_ahead())
```

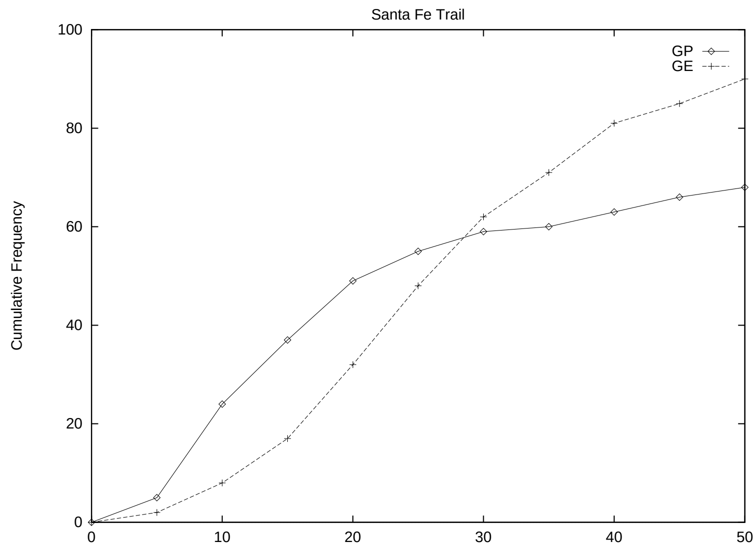

Fig. 3. GE in comparison to GP, with GP using solution length in fitness.

```
   { move(); }
else
   { move(); }
if(food_ahead())
   { left(); }
else
   { left(); }
if(food_ahead())
   { left(); }
else
   { right(); }
if(food_ahead())
   { right(); }
else
   { right(); }
if(food_ahead())
   { move(); }
else
   { left();}
```

5 Conclusions & Future Work

The results have demonstrated that GE successfully found a solution to the Santa Fe Trail for the case examined here, and so can produce multi-line code. This was achieved by a simple modification to the grammar namely,

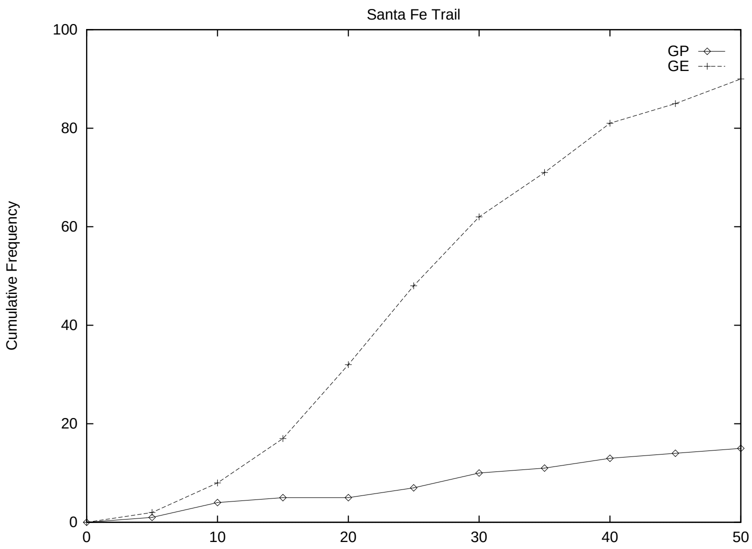

Fig. 4. GE in comparison to GP without solution length in fitness measure.

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

(2) <line> :: =  <expr>
```

where the code could become either a single line of code or multi-line code, where each line consisted of some expression.

On comparison with GP, it was found that GE outperformed GP for this problem. Performance of GP appeared to suffer as a result of solution bloat while GE did not. Instead of producing a single expression here, GE produces more legible multi-line code. Further analysis of solution length and its effects on the performance of GE is required.

It is now our intention to compare the performance of GE to other learning methods over some problem domains. The original purpose of GE was to produce multi-line, compilable code, in arbitrary programming languages and to this end GE has been successful to date.

A Backus Naur Form

Backus Naur Form (BNF) is a notation for expressing the grammar of a language in the form of production rules. BNF grammars consist of terminals, which are items that can appear in the language, and non-terminals, which can be expanded into one or more terminals and non-terminals. A grammar can be represented by the tuple, $\{N, T, P, S\}$, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. For example, below is the BNF used for this problem, where

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

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

$$S =< code >$$

And P can be represented as:

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

(2) <line> :: =  <expr>

(3) <expr> :: =  <if-statement>     (A)
                |<op>               (B)

(4) <if-statement> :: = if(food_ahead())<if-true><if-false>

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

(6) <if-true>:={<expr>}

(7) <if-false>:=else{<expr>}
```

Unlike a Koza-style approach, there is no distinction made at this stage between what he describes as functions (operators in this sense) and terminals, however, this distinction is more of an implementation detail than a design issue. In GE the BNF definition is used to describe the output language produced by the system, that is, the compilable code produced will consist of elements of the terminal set T. As the BNF is a plug-in component of the system it means that, GE can produce code in any language that can be specified in the form of a BNF definition, giving the system a unique flexibility.

References

1. Elseth Gerald D., Baumgardner Kandy D. Principles of Modern Genetics. *West Publishing Company*
2. Fraser Adam, Weinbrenner Thomas. 1997. The Genetic Programming Kernel Version 0.5.2.
3. Freeman, J. J. A Linear Representation for GP using Context Free Grammars. In *Proceeding of Genetic Programming 1998*, pages 72-77. MIT Press.

4. Horner, H *A C++ class library for GP*. Vienna University of Economics.
5. Keller, R. E. & Banzhaf, W. 1996. Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes. In *Genetic Programming 1996*, pages 116-122. MIT Press.
6. Koza, J. 1992. *Genetic Programming*. MIT Press.
7. Langdon, W. & Poli, R. Why Ant's Are Hard. In *Proceedings of Genetic Programming 1998*, pages 193-201 .
8. Levenick, James. R. Inserting Introns Improves Genetic Algorithm Success Rate: Taking a Cue from Biology. In *Proceedings of the Fourth International Conference on Genetic Algorithms 1991*, pages 123-127.
9. Paterson, N & Livesey, M. 1997. Evolving caching algorithms in C by GP. In *Genetic Programming 1997*, pages 262-267. MIT Press.
10. Ryan C., Collins J.J., 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.
11. Ryan C., O'Neill M., Collins J.J. 1998. Grammatical Evolution: Solving Trigonometric Identities. In *Proceedings of Mendel '98: 4th International Conference on Genetic Algorithms, Optimization Problems, Fuzzy Logic, Neural Networks and Rough Sets*, pages 111-119.
12. Ryan C., O'Neill M. Grammatical Evolution: A Steady State Approach. In *Late Breaking Papers, Genetic Programming 1998*, pages 180-185.
13. Whigham, P. 1995. Grammatically-based Genetic Programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 3-41. Morgan Kaufmann Pub.
14. Wong, M. and Leung, K. 1995. Applying logic grammars to induce subfunctions in genetic programming. In *Proceedings of the 1995 IEEE conference on Evolutionary Computation*, pages 737-740. USA:IEEE Press.