Automatic Generation of Caching Algorithms

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

Abstract

Grammatical Evolution (GE) is a grammar based GA to generate computer programs which has been shown to be comparable with GP when applied to a diverse array of problems. GE has the distinction that its input is a BNF, which permits it to generate programs in any language, of arbitrary complexity, including loops, multiple line functions etc. Part of the power of GE is that it is closer to natural DNA than GP, and thus can benefit from natural phenomena such as a separation of search and solution spaces through a genotype to phenotype mapping, and a genetic code degeneracy which can give rise to silent mutations (Mutations that have no effect on the phenotype). In this paper we describe how GE was applied to the *real world* problem of evolving a Caching Algorithm, at which GP has been found to generate algorithms that did not perform as well as those designed by humans.

1 Introduction

Grammatical Evolution (GE) is a grammar based, linear genome system which is capable of generating programs or expressions in any language. Rather than the functions and terminals associated with GP [4], GE takes a BNF specification of a language, or subset thereof, from which it can subsequently generate compilable code. The BNF is used to build a program by applying production rules to elements of the non-terminal set of the BNF definition, in a mapping process to generate the output code from a simple binary string. The evolutionary process is thus applied to the binary strings that are the individuals in the population of the evolutionary algorithm. During the development of GE we have attempted to harness some of the features of the genetic machinery of living organisms which are theorised to have an impact on the phenomenon of evolution [1].

GE has been successfully applied to a number of diverse problem domains such as symbolic regression [9], finding trigonometric identities [10], symbolic integration [11], and the Santa Fe trail [5]. The results compared favorably with systems such as GP, and have been shown to outperform GP [5]. A description of GE follows.

2 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 [5], where

$$N = \{stmts, stmt, expr, term, num, mant, zeros, fun\}$$

$$T = \{+, -, *, div, rem, victim, write\_x, read\_x, random\_x, small\_x, large\_x,$$
$$counter, else, if, 0, 1, 2, 5, =, \{, \}, (,), ;, ,\}$$
$$S = < stmts >$$

And P is represented as:

```
(1)  <stmts> :: = <stmt>
                | <stmt>;<stmts>

(2)  <stmt> :: = if(<expr>){<stmts>;}else{<stmts>;}
               | write_x(<expr>,<expr>);
               | victim=<expr>;

(3)  <expr> :: = <term>
               | <term>+<term>
               | <term>-<term>
               | <term>*<term>
               | div(<term>,<term>)
               | rem(<term>,<term>)

(4)  <term> :: = CACHESIZE
               | <num>
               | <fun>
               | (<expr>)

(5)  <num> :: = <mant>
              | <mant><zeros>

(6)  <mant> :: = 0
               | 1
               | 2
               | 5

(7)  <zeros> :: = 0
                | 0<zeros>

(8)  <fun>  :: = counter()
               | read_x(<expr>)
               | small_x()
               | large_x()
               | random_x()
```

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.

3 The Biological Approach

The GE system is inspired largely by the biological process of generating a protein from the genetic material of an organism. Proteins are fundamental in the proper development and operation of living organisms, and are responsible for traits such as eye colour, and height [1].

The genetic material, usually called DNA, contains the information required to produce specific proteins at different points along the molecule. 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.

In order to generate a protein from the sequence of nucleotides in the DNA, the nucleotide sequence is firstly transcribed into a slightly different format, that being a sequence of elements on a molecule known as RNA. The RNA molecule contains elements that are a slightly modified form of the nucleotides that are contained within the DNA molecule. Codons within the RNA molecule are then translated to determine the sequence of amino acids that are contained within the protein molecule.

The result of the expression of the genetic material as proteins in conjunction with environmental factors is termed it's phenotype. In GE the phenotype is a running computer program. The process of generating the phenotype from the genetic material (the genotype) is termed a genotype-phenotype mapping process, which is unlike the standard method of generating a solution (a program in the case of GE) directly from an individual in an evolutionary algorithm by explicitly encoding the solution within the genetic material. Instead, a many-to-one mapping process is employed within which the real power of the GE system lies.

Figure 1 outlines the mapping process employed in both GE, and biological organisms.

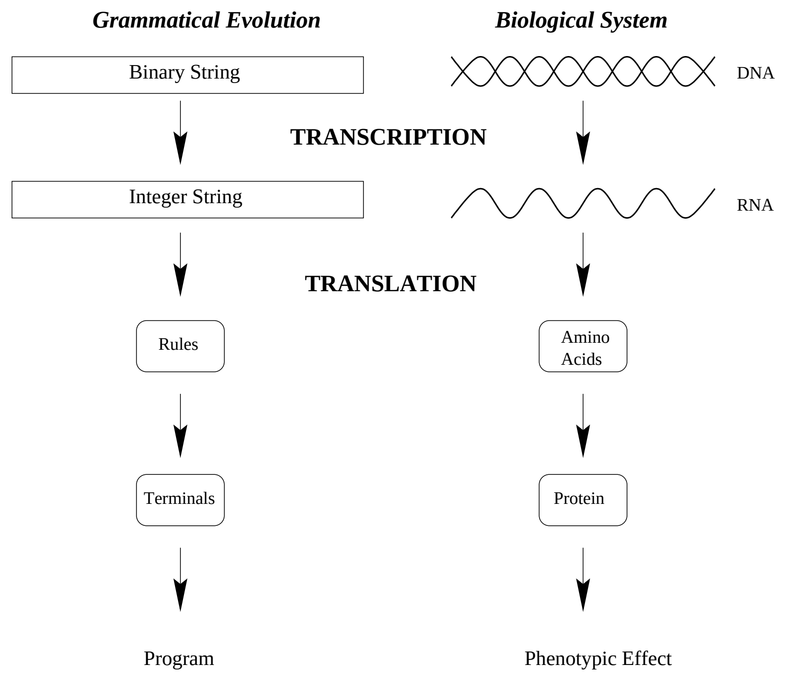

Figure 1: The Grammatical Evolution System and a Biological System

4 Grammatical Evolution

When tackling any problem with GE, a suitable BNF definition must first be decided upon. The BNF can be either the specification of an entire language, or perhaps more usefully, a subset of a language geared towards the problem at hand.

4.1 The Mapping Process

The genotype is then used to map the start symbol onto terminals by reading codons of 8 bits to generate a corresponding integer value, from which an appropriate production rule is selected. A rule is selected by using the following,

$$(Integer\ Codon\ Value)\ \ MOD$$

$$(Number\ of\ Production\ Rules\ for\ the\ current\ non-terminal)$$

Considering the following rule,

```
(2)  <stmt> :: = if(<expr>){<stmts>;}else{<stmts>;} (A)
               | write_x(<expr>,<expr>);            (B)
               | victim=<expr>;                     (C)
```

i.e. given the non-terminal *stmt* there are three production rules to select from. If we assume the codon being read produces the integer 6, then $6\ MOD\ 3\ =\ 0$ would select rule

$$(A)\ if(<expr>)\{<stmts>;\}\ else\{<stmts>;\}$$

Each time a production rule has to be selected to map from a non-terminal, another codon is read, and in this way, the system traverses the genome.

During the genotype to phenotype mapping process it is possible for individuals to run out of codons, and in this case we wrap the individual, and reuse the codons. This is quite an unusual approach in EA's, as it is entirely possible for certain codons 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 same codon is expressed it will always generate the same integer value, but depending on the current non-terminal to which it is being applied, it may result in the selection of 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. It is possible that an incomplete mapping could occur, even after wrapping, and in this event the individual in question is given the lowest fitness value possible, then the selection and replacement mechanisms should operate accordingly to increase the likelihood that this individual is removed from the population.

An incomplete mapping could arise if the integer values expressed by the genotype were applying the same production rules over and over. For example, given an individual with three codons, if the first codon specified rule B from below,

```
(1)  <stmts> :: = <stmt>           (A)
                | <stmt>;<stmts>   (B)
```

and the second, and third also specified this same rule, even after wrapping the mapping process would be incomplete and would carry on indefinitely unless stopped. To reduce the number of invalid individuals being passed from generation to generation, a Steady State replacement mechanism is employed, which in itself is a more biologically plausible than the standard generational replacement mechanisms employed by many other evolutionary algorithms. A result of the Steady State method is its tendency to maintain fit individuals at the expense of less fit, and in particular, invalid individuals.

4.2 Genetic Code Degeneracy

Given an 8 bit binary number, each codon in GE can represent 256 distinct integer values. However, many of these integer values can represent the same production rule, taking production rule 2 as an example, if the current codon value was 6, then $6\ MOD\ 3 = 0$ would select rule (A) as shown above. The same rule would be chosen if the codon 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, and it has been observed that the first two nucleotides in the codon are often sufficient to specify a particular amino acid, and so the value of the nucleotide at the third position is often irrelevant. 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, due to the flexibility at the third codon position. With respect to GE this means that subtle changes in the search space (genotype) may have no effect on the solution space (phenotype), which could result in the maintenance of genotypic diversity throughout a run of the system, and the preservation of valid individuals.

Kimura's neutral theory of evolution [3] states that it is these silent mutations that are responsible for the genetic diversity which has been observed in natural populations, a phenomenon which has been exhibited within GE.

As the population being evolved comprises simple binary strings, we do not have to employ any special crossover or mutation operators, and as such an unconstrained search is performed on these strings due to the genotype to phenotype mapping process which will generate syntactically correct individuals.

5 Problem Domain

This paper serves to describe the application of GE to the problem of the automatic generation of a caching algorithm, which has been previously tackled in [8].

A cache memory is a small, fast memory, ideally into which the *address space* of a running program is held in order to improve efficiency of the programs execution. Caches operate on the notion of *locality of reference*; that is, if a word in memory has recently been accessed, it is likely to be required again soon. The executing program generates a stream of *requests* of main memory locations to access. A recording of this request stream is referred to as a *trace*. The cache has a fixed number of *lines* each of which can hold a word (an address). When a *request* is met from the cache memory this is called a *hit*, otherwise a *miss* occurs. In the event of a miss, the requested memory location is copied into the cache to the first empty *line* available, otherwise a caching algorithm determines which line of the cache will be replaced (the *victim*). In order that the caching algorithm can make an educated decision as to which line will become the victim, typically some management information is available, e.g. an ordered list of lines.

For the purposes of these experiments, we adhered as closely as possible to the procedures outlined in [8], in order to facilitate a comparison of the systems. Thus, the same trace files, a similar grammar (no default or multiple rules were used) with the same terminal set, an integer management array $info[\ ]$ (one element per line), and the same fitness function were used. The raw fitness of a caching algorithm is therefore given as

$$number\ of\ runs - number\ of\ misses$$

where a run is a consecutive series of identical requests. The population size used in our case was 50 as opposed to the 500 used in [8]. The trace file ken2.00200 [2] was used for the fitness evaluations, and comprises approximately 400 000 requests, 250 000 of which are unique.

The terminal operators and their respective functions are given in Table 1.

Terminal Operator	Function
$write\_x(i, v)$:	sets info[i] to v
$read\_x(i)$:	returns info[i]
$small\_x(i, v)$:	index of smallest element of info[]
$large\_x(i, v)$:	index of largest element of info[]
$random\_x(i, v)$:	index of random element of info[]
$counter()$:	successive values 0, 1, 2 etc
$div(x, y)$:	if y==0 then 1 else x / y
$rem(x, y)$:	if y==0 then 1 else x % y

Table 1: Available terminal operators.

5.1 Results

Similar to Paterson[8], we employ three trace files from Flanagan[2]. One being used for training GE, while the resulting algorithms were tested using the other two. The training file `ken2.00200` contains around 400,000 requests with about 250,000 distinct addresses. Paterson reduced memory requirements by eliminating the lower order 10 bits of each addresses, thus reducing the alphabet size to just 500. We use the files in their original state, and show comparisons with Paterson's algorithm below. Paterson found that the speed of the cache simulator frustrated his attempts to run many experiments, and we suffered the same experience. To alleviate this, we use a population of just 50 individuals, compared to his 500.

5.1.1 Small Cachesize

The first experiment consists of using a cache size of just 20, clearly, it is very difficult to exploit any locality of reference with such a small size. The best individual generated by GE was

```
GE 1 :        victim = counter() - CACHESIZE;
```

This curious looking individual simply cycles through the cache when searching for a victim, selecting it without consideration for how recently it was used. There is, however, an implicit temporal quality about the manner in which victims are selected. A word is guaranteed to remain in the cache for at least $CACHESIZE$ requests, longer if it (or other cache members) is requested. This outperforms the human designed Least Recently Used algorithm (LRU) by 15.7%, and, by virtue of its simplicity, is far less expensive to use.

5.1.2 Large Cachesize

A cachesize of 200 lines was used in this instance, and one of the best individuals produced was again GE 1. However, in this case the number of misses was far fewer, outperforming LRU by 46.2%.

5.1.3 Using info [ ]

Given the level of disinterest exhibited by GE to the use of the info[] array, we ran an experiment that forced it to write into some element before choosing its victim. The best of run individual generated by this system was:

```
GE 2 :        write_x( CACHESIZE + counter(), CACHESIZE + counter())
              victim = CACHESIZE + counter();
```

Rather predictably, this individual simply ignores what it writes into the array, and promptly chooses its victim using the same method as in the other experiments. In fact, the ubiquity of the beautifully simple $CACHESIZE + counter()$ algorithm is demonstrated by virtue of its appearance in no less than three parts of this algorithm.

5.2 Constants

Paterson reported on an experiment where the use of constants was prohibited. He was pleased to note that imposing this restriction actually improved the performance of their system. In no case did GE produce an individual that required constants, and it appears that the system discovered that constants were unnecessary in this case.

5.3 Comparison of Algorithms

A comparison between the LRU algorithm and those which had been evolved was carried out by rating their performance on the other two trace files, and the number of misses obtained in each case can be seen in Table 2.

Algorithm (Cachesize)	**ken2.00100 (Misses)**	**ken2.00200 (Misses)**	**Average of % Improvement over LRU**
LRU (20)	374,596	380,041	-
LRU (200)	367,104	373,935	-
GE1 (20)	300,569	318,444	17.97
GE1 (200)	106,067	82,856	74.51
GE2 (20)	300,569	318,445	17.97
GE2 (200)	106,068	82,855	74.51

Table 2: Algorithm performance comparison.

Paterson was dismayed to report that the algorithms they produced were very poor at generalising to unseen data, and that they suffered greatly from overfitting. The algorithms produced by GE don't suffer from this problem, and perform extremely well on the new data. One factor that might contribute to the large difference in performance was Paterson's decision to simplify the problem - the problem as posed to GE was possibly too large to spot any patterns in.

The solution generated by GE was so general that when applied to a larger cache size, a huge increase in performance was observed. When the cache size was increased by an order of magnitude for LRU, the accompanying increase in performance shows that the associated cost of such an increase would probably not pay off.

6 Conclusions

We have successfully used GE to evolve caching algorithms in C. Given the difficulty experienced by previous researchers, we believe that the simple algorithm we generated is more a comment on the power of GE to extract elegant solutions than on the relative ease or difficulty of the problem.

The representation scheme employed by GE is such that any BNF can be used, and enjoy the benefits of the underlying process, e.g. genetic code degeneracy, wrapping, duplication, and crossover.

7 Acknowledgement

Thanks to Norman Paterson for contributing his BNF's and trace files, and to Laur Ivan for invaluable input with the caching simulator.

References

[1] Elseth Gerald D., Baumgardner Kandy D. 1995.
Principles of Modern Genetics. *West Publishing Company.*

[2] Flanagan K., et al. 1992. BACH: BYU Address Collection Hardware. Technical Report TR-A150-92.1. Electrical and Computer Engineering Department, Brigham Young University.

[3] Kimura, M. 1983. The Neutral Theory of Molecular Evolution. Cambridge University Press.

[4] Koza, J. 1992. *Genetic Programming.* MIT Press.

[5] O'Neill M., Ryan C. 1999. Evolving Multi-line Compilable C Programs. In *Proceedings of the Second European Workshop on Genetic Programming 1999.*

[6] O'Neill M., Ryan C. 1999. Under the Hood of Grammatical Evolution. In *Proceedings of the Genetic & Evolutionary Computation Conference 1999.*

[7] O'Neill M., Ryan C. 1999. Genetic Code Degeneracy: Implications for Grammatical Evolution and Beyond. Submitted to the 5th European Conference on Artificial Life 1999.

[8] Paterson, N & Livesey, M. 1997. Evolving caching algorithms in C by Genetic Programming. In *Genetic Programming 1997*, pages 262-267. MIT Press.

[9] 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.

[10] 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.

[11] Ryan C., O'Neill M. Grammatical Evolution: A Steady State Approach. In *Late Breaking Papers, Genetic Programming 1998*, pages 180-185.