How To Do Anything With Grammars

Conor Ryan
Computer Science and Information Systems
University of Limerick
Ireland
Conor.Ryan@ul.ie

Michael O'Neill
Computer Science and Information Systems
University of Limerick
Ireland
Michael.ONeill@ul.ie

Abstract

This paper describes Grammatical Evolution in terms of a family of algorithms, and demonstrates how it can easily be changed to emulate any of the common Evolutionary Algorithms.

1 Introduction

The beauty of grammars is that they provide a single mechanism that can be used to describe all manner of complex structures. Grammars can describe languages, graphs, neural networks, mathematical expressions, the manner in which molecules arrange themselves in compounds, etc. The beauty of using grammars as part of an evolutionary computation tool is that one is effectively using the weapon of the enemy against them. That is, instead of evolving a structure directly, which can attract problems such as invalid crossovers and infeasible individuals, the search is instead performed on the derivation sequence required to produce the desired structure. The more one knows about a particular problem, the more one can enrich the grammar with all manner of insider information. Importantly, the grammar simply states what *can* be done, while the search dictates what *should* be done.

This paper gives an overview of current work in Grammatical Evolution being carried out at Limerick, and demonstrates that Grammatical Evolution is more of a family of algorithms, than a single method. The following section, Section 2 summarises the current state of the art of Grammatical Evolution, and explains why the method of representation it employs is particularly plastic. That section further explains why the choice of "Grammatical Evolution" was a rather unfortunate one, as that term would be better used to describe the family of algorithms that operate in the same manner.

Sections 3 - 7 give simple grammars showing how the Grammatical Evolution approach can emulate some of the more common Evolutionary Algorithms, and describes what, if anything, else has to be done to make the systems equivalent. Section 7 gives a flavour of just how powerful the Grammatical Evolution approach is by demonstrating how one can replace the Genetic Algorithm part of Grammatical Evolution with Grammatical Evolution.

It is this plasticity that makes grammars so attractive as part of an Evolutionary Computation tool. Not only are they convenient methods of describing legal solutions, they can also be tuned during the course of a run in response to the system output. Furthermore, such is the flexibility of grammars that employing a completely different grammar can make a system appear to work very differently, yet the fundamentals remain the same. This means that a tool that uses a grammatical component can be applied to an enormous set of problems, simply by changing the grammar to the appropriate type.

2 Grammatical Evolution

Grammatical Evolution evolves simple linear genomes of binary strings, which are used to dictate choices in the derivation sequence. This gives a complete separation of search and solution space, where the genetic operators are applied to the genome, after which it is mapped onto a legal phenotype. As in nature, there are no genetic operations applied to the phenotype, which means that it can be as complex as necessary, and problems such as over- and under-specification, and infeasible crossovers simply do not occur.

The binary string is first decoded into a set of n bit codons, each of which is used to make a choice at a particular step in the derivation sequence. Typically, eight bit codons are used, so each can encode 256 different choices (2^8), the number is **mod**ed by the number of legal choices. This results in a situation where *every* codon can legally be applied to *every* choice.

2.1 Grammars

A grammar is simply a convenient way to represent legal instances of a structure. Traditionally, GE has used a context free grammar described in BNF. There is no reason why other grammars could not be used, and, recently, versions of GE have appeared that use attribute grammars.

A BNF grammar is 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 productions rules that map the elements in N to those in T, and S is a start symbol which is a member of N.

Consider the simple grammar below :

```
S = <expr>
<expr>::=  <expr> <op> <expr>        (0)
        | ( <expr> <op> <expr>)      (1)
        | <var>                      (2)
<op>  ::= + (4) | - (5) | % (6) | * (7)
<var> ::= 1.0 (8) | X (9)
```

When mapping an individual, one starts with the start symbol S, which, in this case is the non-terminal $< expr >$. The first value is read from the genome, and **mod**ed by three, as there are three choices to be made. This process continues with the left-most non-terminal until there are either no non-terminals left (i.e., a successful mapping) or the genome is exhausted. In the latter case, the individual is wrapped, that is, one starts reading from the start once more. There is an upper limit on the number of wrapping events, which, when exceeded an individual is aborted. This upper limit is usually set to ten, although newer algorithms in the Gramamtical Evolution family don't employ wrapping, and some experiments have suggested that wrapping isn't a prerequisite for success.

This very simple approach has a number of beneficial features. The coding is inherently degenerate, which permits neutral evolution; the seperation of search and solution spaces permit one to apply the system to radically different problems simply by changing the grammar; re-use and re-interpretation of genetic material; very strongly type system etc.

The flexibility of the system is such that, by simple manipulation of the grammars and/or mapping, it can emulate the behaviour of many other algorithms, but also has scope to implement newer systems.

3 How to do Genetic Programming with Grammars

The term *closed grammar* was introduced to denote the type of grammar normally employed by GP practitioners. Although many GP users may be surprised to have it claimed that they have actually being using grammars, rather than simple sets of functions and terminals, this is indeed the case. A function and terminal set implicitly describes a grammar; one indicates the arity of the functions and, as every function can take every terminal as well as the output of every function, it is simply a matter adhering to the arity demands of every function to produce legal programs.

The arity of the functions could easily be described by a CFG. Consider the GP function and terminal set

$$F = \{+, *, -, \%\}, T = \{x, y\}$$

Where each of the four functions have an arity of two. This could also be described by the Context Free Grammar (CFG):

```
E ::== x | y | (+ E E)|(* E E)|(- E E)|(% E E)
```

where E is the start symbol.

This kind of CFG differs from the standard type only in that there is a single non-terminal. The use of a single non-terminal implies that the grammar has the closure property, so desirous of the GP crossover operator. Notice that use of more than one non-terminal does not preclude a grammar from being closed. The question of its closure can only be resolved by determining if it can be rewritten as an equivalent grammar with a single non-terminal.

Crossover is considered to be the driving force behind GP, and the crossover used by GE has been identified as *ripple* crossover. To convert GE to the sub-tree crossover used by GP, one simply needs to perform a restricted form of two point crossover.

4 How to do Genetic Algorithms with Grammars

The recently described GAUGE uses GE with an attribute grammar to produce a position independent GA.

The common view of Genetic Programming is that, given a particular problem statement, a program that satisfies the fitness function is to be generated. In other words, *given a set of terminals, how should they be arranged so that the fitness function is satisfied?*

The approach GAUGE takes is that *all* problems, even those traditionally considered the topic of Genetic Algorithms, can be looked upon as automatic programming ones. In particular, if one considers a problem to have two parts, that is, not only *what values should the genes have?* but also *where should the genes reside on the chromosome?*, it is reasonable to compare this class of problems with those above.

When applying GAUGE to a Genetic Algorithm problem, one uses a set of codon pairs, one for each gene position in the original problem. An individual is processed in a similar manner to GE, by `moding` each gene by an appropriate value to produce a useful value. The codons are 8-bit values, giving a degenerate (redundant) encoding. For GAUGE, the values required are the position that the pair will code for, and the value for that bit position. To produce these values, a list of *unspecified* positions is maintained, that is, those positions that have not been given a value yet. Thus, even when evolving individuals that are binary strings, GAUGE always has a distinct genotype and phenotype. The mapping happens in two stages; first a set of (position, value) pairs are generated using the grammar below, while th e second step simply puts them in the correct order.

GAUGE operates using a tuple $< G, A, q, l >$ where G is the genome, A the attribute grammar below, q a list of free *locations* in the phenotype and l is the length of the phenotype. The maximum value that can appear in the phenotype is governed by the variable max, while the function $rember(n, q)$ returns the nth member from list q and removes it from the list.

1. $<\text{start}_n >$ ::= if $(n < l)$
   $\{ <\text{P}_n ><\text{V}_n ><\text{start}_{n-1} >\}$
2. $<\text{P}_n ><\text{V}_n >$::= $<\text{Q}_{(G[2n]\%(l-n))} >$ G[2n+1]%max.
3. $<\text{Q}_n >$ ::= rember (n,q)

5 How to do Linkage Learning with Grammars

The family of competent GAs developed at IlliGAL generally take the view that, as problem difficulty scales up, the amount of linkage between loci increases. A set of problems, known as *linkage* problems have been posed, and standard GAs are unable to solve them.

The idea is to extend the tight linkage between the gene positions, as seen in GAUGE, to the gene values themselves. This is achieved by extending GAUGE's mapping process: every time a value is to be placed on the phenotype string, it is calculated by adding all the previous *value* fields in each (*position, value*) pair and then normalizing the result over the range of accepted values. The value each gene will provide can therefore be calculated by the formula

$$(\sum_{i=0}^{n} x_i)\%v$$

where
n = order of the gene (i.e. gene 0, gene 1, etc)
x_i = number in *value* field for gene i
v = value to normalize (for binary strings, 2 is used)

It should be noted that, theoretically, any function could be used to introduce dependency between the values; the suitability of other functions will be the subject of further research.

6 How to do Gene Expression Programming with Grammars

The recently released Gene Expression Programming (GEP) has been the subject of much debate, not least of which is its similarity to GE. In GEP, similar to GP, a user specifies the function and terminal set using a GEP-specific language, Karva. As with standard GP, above, Karva can simply be recast into a closed grammar, as it is a single typed system.

A further difference between GEP and GE is the manner in which the expression tree is filled from a linear genome. GE uses a depth first approach (i.e. the left-most non-terminal) while GEP uses a breadth first approach (i.e. the *oldest* non-terminal.)

Consider the following closed grammar, with each production numbered for clarity :

```
S = (Q S) | (+ S S) | (- S S) | (/ S S)
      0        1         2         3
  | (* S S) | a | b | c | d | e
       4      5   6   7   8   9
```

To get GE to produce individuals in the same manner, one simply employs a breadth first approach, which is easily managed if one numbers the non-terminals as they are produced, in ascending order. The first transition would be : $S \rightarrow (+\ S_{x+1}\ S_{x+2})$

where x+1 is the number of the last terminal created. Then, when looking for a non-terminal to transform, we just pick the one with the lowest value. For the first transition we would get : $S \rightarrow (+\ S_1\ S_2)$

The next item to replace is S_1, followed by S_2, *regardless* of the presence or otherwise of non-terminals produced from S_1. Consider the following individual, taken from the GEP tutorial: `+/Q*c-abde\*.` or, in terms of the above grammar : `1304725689`

- $Gene1 : (+\ S_1\ S_2)$
- $3 : (+\ (/\ S_3\ S_4)\ S_2)$

The mapping continues in this manner[1] with the right hand child of + (the oldest non-terminal) untiel eventually the expression $(+\ (/\ (*\ a\ b)\ c)\ (Q\ (-\ d\ e)))$ is produced, which is the identical individual created by GEP (as reported in the WSC 6 tutorial). Furthermore, as both systems employ a simple linear genome, crossover is identical except for the fact that GE individuals can grow and contract.

7 How to do Grammatical Evolution with Grammars

The derivation of an individual Grammatical Evolution is driven by two key components, the binary string and the grammar. How the binary string is produced is completely transparent to the system, in fact, methods such as Simulated Annealing and Hill Climbing have successfully been used in place of a Genetic Algorithm.

Furthermore, the GAUGE system as described in section 7 is a position independent Genetic Algorithm, powered by a grammar, which means that this can be used to generate the binary strings.

8 How to do Grammars with Grammars

The key to employing GE is the identification of a grammar that governs legal instances of the structure being involved. Clearly, this isn't always a trivial task, particularly when the interactions of the various constituent parts of a structure are unknown. This can be overcome by adding a pre-processing classification step to the system.

This involves the evolution of regular expressions that correctly classify negative and positive examples of the structure under consideration. Once a satisfactory regular expression has been derived, one can then produce the corresponding grammar to drive the system.

[1]The full derivation sequence, as well as other examples are available on http://www.grammatical-evolution.org/how-to/

8.1 Dynamic Grammars

Currently, every individual in the population employs the same, static, grammar. However, there is no reason why the grammar must remain so. Individuals could use their own translation table which can be used instead of a simple mod rule. This can be subject to evolution, so one would effectively co-evolve the grammar with the individuals.

9 How to do Anything with Grammars

Any system with regular behaviour, or object with regular, deterministic behaviour can be modelled with a grammar. Nature, in particular, abounds with regular structures, some of which have identified grammars, while others, such as the manner in which proteins fold, clearly have some kind of rule set, even though it has not been discovered yet.

With the exception of GAUGE, all the above grammars are context free, the simplest type of grammar. By employing other types of grammars, such as context sensitive or attribute grammars, one can describe far more complex interactions.

Crucially, this approach only requires a user to identify a grammar specific to their problem - all other major aspects of the system remain unchanged. It is for this reason that we believe the term "Grammatical Evolution" should refer to any system that evolves structures that obey rules of a specified (but possibly evolvable) grammar.

10 Conclusions and Future Work

The Grammatical Evolution family of algorithms is still extremely new. Early research concentrated on the canonical GE algorithm, in particular on issues such as degeneracy, mapping functions, crossover etc., while current research is focusing on grammar development.

This paper has given a flavour of how plastic the system is when the grammars are varied. Current work is focusing on developing new algorithms, ideally those that are suited to particular problem domains as dictated by evolved grammars.

Work is also being carried out on a position independent version of Grammatical Evolution, called Chorus. Chorus appears to have the same flexibility as GE, but with an almost completely position independent representation.