Attributed Grammatical Evolution using Shared Memory Spaces and Dynamically Typed Semantic Function Specification

James Vincent Patten and Conor Ryan

Biocomputing and Developmental Systems Group,
University of Limerick, Ireland
{*james.patten*, *conor.ryan*}@ul.ie

Abstract. In this paper we introduce a new Grammatical Evolution (GE) system designed to support the specification of problem semantics in the form of attribute grammars (AG). We discuss the motivations behind our system design, from its use of shared memory spaces for attribute storage to the use of a dynamically type programming language, Python, to specify grammar semantics.
After a brief analysis of some of the existing GE AG system we outline two sets of experiments carried out on four symbolic regression type (SR) problems. The first set using a context free grammar (CFG) and second using an AG. After presenting the results of our experiments we highlight some of the potential areas for future performance improvements, using the new functionality that access to Python interpreter and storage of attributes in shared memory space provides.

Keywords: Grammatical Evolution, Symbolic Regression, Attribute Grammars

1 Introduction

Since it was first introduced [6], Grammatical Evolution (GE) has been successfully applied to solve a wide range of problems across a diverse set of domains. GE operates by producing potential solutions (usually in the form of programs), to a predefined problem, by combining symbols specified in Backus-Naur Form (BNF), a convenient way of describing a Context Free Grammar (CFG).

A CFG provides a means of specifying the syntax of programs, by outlining a set of rules which control the sequences of symbols allowed to appear in each program. While a CFG provides a means of specifying program syntax, it does not support specification of semantics, information which could guide the generation of more meaningful programs.

A GE system uses the rules of a CFG specification in combination with an individuals genotype to produce the individuals phenotype. After a phenotype is successfully produced we can extract the parse tree from it and then use this parse tree to evaluate the fitness of the individual across a set of training data

points. Usually in GE it is not until the assignment of fitness that issues of semantic correctness become apparent. A common practice is to include some means of detecting semantically invalid programs when running fitness evaluation, e.g. protected division, or assignment of worst fitness score to individuals whose fitness evaluation "throws" an error.

As fitness scores are used to decide which individuals get to act as parents during evolution and to decide which individuals to replace in a steady state population, the score assigned to an individual is very important. While semantically invalid individuals do "die out" due to the evolutionary process the effects of their initial introduction into a population is something that needs to be considered [4]. Also as training data sets become much larger and fitness evaluation time increases we need to more carefully consider the effects of evaluation time spent on individuals that eventually get assigned a worst fitness score. One method that has the potential to reduce these effects is the addition of semantic information to help guide the genotype to phenotype mapping process, ensuring individuals produced are not only syntactically but semantically correct.

Knuth [5] proposed a means of annotating a CFG with semantic information in the form of attributes and semantic functions, commonly referred to as Attribute Grammar (AG). Unlike a CFG, when used with GE in the creation of a derivation tree, an AG in addition to providing a set of production rules, will also provide an associated semantic function which specify attributes to annotate the nodes of the derivation tree with. The inclusion of attributes provides a means of giving context to the nodes of the derivation tree, with choices of terminal or non-terminal nodes at one point in the tree being able to influence choices of nodes at others.

An AG uses two distinct types of attributes, *inherited* and *synthesised*. The names are used to indicate the direction the attributes passes information in the derivation tree. Inherited being used to identify attributes which pass information down the tree and synthesised for attribute which pass information up or across tree nodes. Semantic functions are used to interpret attribute information, using it to make decisions at one point in the tree based on values of attributes set in another. Semantic functions may also include "helper" type functions that perform more subtle analysis of attributes and help semantic function make decision on values to assign to attributes.

One of the most powerful features of GE comes from its decoupling of an individuals underlying representation from that of the derivation tree it produces. All grammar information need merely be outlined in a BNF file and GE can begin generating derivation trees. We strongly feel that any extension to GE to support attribute grammars needs to strive to maintain this decoupling and with this in mind we propose a new GE system which supports AG in addition to CFG BNF specifications.

The main core of our system was designed using C++, with the attribute information, needed to be added to derivation tree nodes being stored in shared memory space using C++ pointers. Our system includes an embedded Python

interpreter used to run the semantic function and a C++ / Python interface which allows semantic functions interact with attributes in shared memory.

Storing attributes in shared memory allows them to be assigned to any number of nodes in the derivation tree and also facilitates the passing of information in any direction between the nodes. Changes to an attribute at one node are immediately seen at all other nodes that share the same attribute. As Python is dynamically typed it reduces the complexity of the semantic function specifications and allows the loading of semantics at runtime rather than having to compile them separately before running the GE system. This was a carefully chosen design to help maintain in as much as possible the containment of the entire AG specification on the single BNF file, like that of a CFG.

The rest of this paper is organised as follows: section 2 discusses some of the existing attribute grammar capable GE systems, discusses their use of AG and highlighting the difference of our proposed new system; section 3 outlines a set of experiments carried out using our new system, using first a CFG and then an annotated version of the CFG (an AG); finally section 4 concludes the paper, highlighting again the main motivations of our new GE system design and suggesting some of the areas we can extend our system into in the future.

2 Background

We are not the first to present results of experiments carried out using a GE system with added support for attribute grammars. As far back as 2005 de la Cruz et al. [1] presented results of experiments carried out on symbolic regression type problems using GE with CFG and AG specifications. More recently Karim and Ryan carried out a number of experiments using GE with AG on a variety of problem types including, but not limited to, their work on the artificial ant trail problem [3].

The results presented by both clearly demonstrate the performance gains a GE system can achieve by supporting AG problem specification. This is something which will become more important when dealing with problems with increasingly large train and test sets and ever more time consuming fitness evaluation cycles.

Both de la Cruz and Karim provided very little by way of description of their underlying GE systems design, choosing instead to only focus on the performance gains fitness gains seen in the solutions produced. Neither discusses attribute storage strategies or their effect on the information passing between the nodes of the derivation tree, or the means of specification of semantic functions and their interaction with the attributes in the derivation tree. Our system utilizes a number of features in an effort to keep the newly added AG specification as concise and clear as possible, something we feel merits highlighting a paper outlining an extension to GE to support AG.

The semantics outlined in an AG, as used by a GE system, act as a form of logic which, along with the grammar production rules, guides the generation of the derivation tree during the mapping process. Attributes can be used to

pass information between tree nodes giving them a context, something that is not possible with a CFG. From a design point of view when adding support for AG it makes sense to abstract out the logic (semantics) from the underlying representation (derivation tree) in the same way that a CFG does with the production rule specification in a BNF file. This is something we have done in our system. This will make the expression of semantics less troublesome, allowing them to be included in, and read directly from, a BNF at the same time as the production rules.

3 Experiments

We chose four symbolic regression (SR) type problems on which to test our new system. Problems 1 and 2 have a single independent input, X, while problems 3 and 4 have an additional independent input Y. Details of the problem equations, along with the range of data points used for train and test are outlined in Table 1.

Table 1. Problem Sets and Train and Test Data Point Ranges

	Problem	Training set $[min : step]$ 50 points Test set $[min : step]$ 200 points
1	$arcsinh(x)$	$[0.0 : 1.0]$ $[0.1 : 0.25]$
2	$x^3e^{-x}cos(x)sin(x)(sin^2(x)cos(x) - 1)$	$[0.0 : 0.2]$ $[0.05 : 0.05]$
3	$y^3e^{-x}cos(y)sin(x)(sin^2(y)cos(x) - 1)$	$x[0.0 : 0.2], y = x + 0.03$ $x[0.05 : 0.05], y = x + 0.03$
4	$y^2x^6 - 2.13y^4x^4 + y^6x^2$	$x[1.9 : 0.075], y = x + 0.015$ $x[1.91 : 0.019], y = x + 0.015$

3.1 Setup

An initial CFG specification was created which includes a set of basic mathematical operators $(+, -, *, /)$ and a set of 50 persistent random constants [2], PRC, generated in the range $PRC = \{c|c \in \Re \wedge -5 \leq c < 5\}$. The CFG was designed so there is a 50/50 chance of either an independent variable (X or Y) or a PRC getting added to the derivation tree. When a choice is made to add a PRC, i.e. <prc> ::= PRC, the codon value and mod operation are used to select which of the 50 available prc values to choose. For the sake of conciseness we use PRC in the grammar specification in Table 2, in the grammar used by our system this is replaced with the 50 prc values.

For our AG a set of attributes and semantic function were designed with two main goals:

1. Provide a globally accessible shared memory space, called “globalCache”, which all nodes in the derivation tree have access to. When a node gets expanded to a terminal its value is added to the globalCache so any node can access the current evaluation state of the derivation tree as sub-trees become fully formed (i.e. expanded to terminals)
2. To track when an <op> Symbol node get expanded to, ’/’, and pass the information back up the tree so the semantic function can use it to ensure that a / is not followed by value that could be zero

Along with providing access to add, read and update attribute information on the nodes of a derivation tree our systems C++ / Python interface also provides a means for the semantic function to directly access the Symbol information stored at node. As can be seen in the grammar specification in Table 2, using a masked property, “.Data”, the semantic function can change the terminal value from “X” to “X + PRC”, or “Y” to “Y + PRC” or whatever other value desired.

When each production rule is being read initially from a BNF our system tests to see if the production includes a set of additional terms, enclosed in a set of curly brackets. If the set of curly brackets is found then its contents are formatted into a Python function which is made available to the Python interpreter so it can be called during the creation of the derivation tree. We can also very easily include any other Python library, available on the system, in a semantic function, or create our own semantic helper functions designed specifically for use with a particular set of problems. In our AG we have defined a simple helper function which is included in all semantics functions. It was designed to perform a particular simple function, which is defined as follows:

appendSymbol(nodeOne.A, B)
If the derivation tree node, nodeOne has an attribute called ’A’, then its shared memory space is accessed and its contents is updated, appending the value ’B’ to whatever already exists in it. If nodeOne does not have an attribute called ’A’ then nothing is done

50 runs were carried out for each problem, using each type of grammar, and the results presented are averaged over those runs. Normalised linear scaled mean squared error (NLSMSE) [4] was used as a fitness measure in both the sets of experiments. Details of the GE system parameters used for each run are outlined in Table 3.

We had initially hoped to include the use of the Python Abstract Syntax Tree (AST) library in our semantic functions but it was unfortunately not fully operation in this version of our system. Using the AST library we could potentially evaluate expressions as they appear in sub-tree segments of the derivation tree. This could be a very powerful feature and among other things be used to help prevent the generation of more difficult to detect invalids. It is something we hope to have implemented in the next revision of our system.

Table 2. CFG and AG Specifications

			Semantics (AG only)
S	::=	<expr>	<expr>.globalCache = ' ';
<expr$_1$>	::=	<expr$_2$> <op> <expr$_3$>	<expr$_2$>.globalCache ← <expr$_1$>.globalCache <op>.globalCache ← <expr$_1$>.globalCache <expr$_3$>.globalCache ← <expr$_1$>.globalCache <op>.op = ' ' <expr$_3$>.lastOp ← <op>.op
	|	(<expr$_2$> <op> <expr$_3$>)	<expr$_2$>.globalCache ← <expr$_1$>.globalCache <op>.globalCache ← <expr$_1$>.globalCache <expr$_3$>.globalCache ← <expr$_1$>.globalCache <op>.op = ' ' <expr$_3$>.lastOp ← <op>.op
	|	<var>	<var>.globalCache ← <expr$_1$>.globalCache <var>.lastOp ← <expr$_1$>.lastOp
<op>	::=	+	<op>.op = '+' appendSymbol(<op>.globalCache, '+')
	|	−	<op>.op = '-' appendSymbol(<op>.globalCache, '-')
	|	*	<op>.op = '*' appendSymbol(<op>.globalCache, '*')
	|	/	<op>.op = '/' appendSymbol(<op>.globalCache, '/')
<var>	::=	<ind>	<ind>.globalCache ← <var>.globalCache <ind>.lastOp ← <var>.lastOp
	|	<prc>	<prc>.globalCache ← <var>.globalCache <prc>.lastOp ← <var>.lastOp

<ind>	::=	X	if(<ind>.lastOp == '/'):
			....prc = getPRC()
			....X.Data = 'X + ' + prc
			....appendSymbol(<ind>.globalCache, X.Data)
			else:
			....appendSymbol(<ind>.globalCache, 'X')

	|	Y	if(<ind>.lastOp == '/'):
			....prc = getPRC()
			....Y.Data = 'Y +' + prc
			....appendSymbol(<ind>.globalCache, Y.Data)
			else:
			....appendSymbol(<ind>.globalCache, 'Y')

<prc>	::=	PRC	appendSymbol(<prc>.globalCache, PRC)

Table 3. Run Configuration Parameters

Population Size	500
Run Terminates at	150 generations
Operator probabilities	Crossover: 0.9, mutation: 0.1
Tournament size	2
Replacement	Steady state, inverse tournament
PRC	$PRC = \{c \vert c \in \Re \wedge -5 \leq c < 5\}$ $\vert PRC \vert = 50$
Normalised Fitness	$\frac{1}{1+LSMSE}$
Initialisation	Ramped half and half (max. initial depth = 8)
Max wraps	5

3.2 Results

For a given problem the same set of training and testing data points were used for both the CFG and AG runs. Table 4 outlines the experimental results which include the mean and best fitness score achieved on both the train and test data sets for each problem.

As can be seen for the results, for each problem type the run using the AG achieved better fitness scores on both the train and test data sets. While the semantics included in our AG are relatively simple they do prevent the creation of certain invalids and this can be seen to influence the resulting scores

Table 4. Results

	Problem:	1	2	3	4
CFG	Mean Train	0.8784	0.9126	0.9154	0.8174
	Mean Test	0.9926	0.9089	0.9103	0.8973
	Best Train	0.8866	0.9205	0.9319	0.9541
	Best Test	0.9940	0.9111	0.9318	0.9418
AG	Mean Train	0.8895	0.9139	0.9184	0.8596
	Mean Test	0.9927	0.9098	0.9131	0.8410
	Best Train	0.9012	0.9282	0.9534	0.9999
	Best Test	0.9943	0.9282	0.9534	0.9999

3.3 Discussion

While the results presented do show an improvement in overall fitness by using the AG there is room for further improvement. The semantics we used are not very sophisticated and fail to take full advantage of the storage of attributes in shared memory and semantic functions access to a number of useful Python libraries.

There are a number of areas where we feel we can improve our systems performance further, evaluating sub-tree expressions in derivation and using the information to prevent the formation of more complex and difficult to detect invalids in population, dynamic pre-processing of train inputs at run time and automatic generation of semantics based on the grammar symbols and inputs, to name a few.

Each of the problems presented had a relatively small cost associated with fitness evaluation so the effects of evaluating invalids is not very pronounced. We can however see that in situations where this is not the case the increased precision of specification provided by AG will become even more important.

4 Conclusions

In this paper we introduced a new GE system designed to support the specification of problem semantics in the form of attribute grammars. We provided a description of the underlying motivations for our system design, with a core built using C++, storage of attribute information in shared pointers and support for semantic function specification in Python scripts.

We followed this by briefly discussing some of the existing GE AG systems comparing, in as much as possible, the main goals of our system design to them and emphasising why we feel our systems design could help make the specification and use of AG with GE much more straight forward and concise.

We then outlined a set of experiments carried out using the new GE system, one using a traditional CFG and another using a relatively simple AG. We discussed the results, highlighting the performance improvements seen by using an AG and finally we finish by suggesting some of the areas we feel we can extend our system in future.

References

1. de la Cruz Echeandía, M., de la Puente, A.O., Alfonseca, M.: Attribute grammar evolution. In: Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, pp. 182–191. Springer (2005)
2. Dempsey, I., O'Neill, M., Brabazon, A.: Constant creation in grammatical evolution. International Journal of Innovative Computing and Applications 1(1), 23–38 (2007)
3. Karim, M.R., Ryan, C.: Sensitive ants are sensible ants. In: Proceedings of the fourteenth international conference on Genetic and evolutionary computation conference. pp. 775–782. ACM (2012)
4. Keijzer, M.: Improving symbolic regression with interval arithmetic and linear scaling. In: Genetic programming, pp. 70–82. Springer (2003)
5. Knuth, D.E.: Semantics of context-free languages. Mathematical systems theory 2(2), 127–145 (1968)
6. O'Neil, M., Ryan, C.: Grammatical evolution. In: Grammatical Evolution, pp. 33–47. Springer (2003)