2006 IEEE Congress on Evolutionary Computation
Sheraton Vancouver Wall Centre Hotel, Vancouver, BC, Canada
July 16-21, 2006

Dynamically Defined Functions In Grammatical Evolution

Robin Harper and Alan Blair, *Member, IEEE*

Abstract - **Grammatical Evolution is an extension of Genetic Programming, in that it is an algorithm for evolving complete programs in an arbitrary language. By utilising a Backus Naur Form grammar the advantages of typing are achieved as well as a separation of genotype and phenotype. This paper introduces a meta-grammar into Grammatical Evolution allowing the grammar to dynamically define functions, self-adaptively at the individual level without the need for special purpose operators or constraints. The user need not determine the architecture of the dynamically defined functions. As the search proceeds through genotype/phenotype space the number and use of the functions can vary. The ability of the grammar to dynamically define such functions allows regularities in the problem space to be exploited even where such regularities were not apparent when the problem was set up.**

I. Introduction

Genetic Programming [1] and other evolutionary algorithms are a proven successful form of *weak* problem solving in AI. Rather than relying on a significant amount of task specific knowledge (as required by a *strong* method) evolutionary methods only require a credit assignment mechanism that is representation specific rather than task specific. A representation specific method of assigning credit avoids the need for explicit task knowledge.

Angeline [2] argues that, unlike traditional weak methods, the adaptive nature of evolutionary methods is such that an empirical form of credit assignment is built into the dynamics of the system. This he believes supports the description of evolutionary computation as a “strong weak method”. There is a body of thought that given the lack of knowledge of the task domain it is important that the representation be sufficiently flexible to take advantage of any regularity in that domain, even though the existence of such regularity may not be known at the time the algorithm is run.

In Grammatical Evolution [3] a Backus Naur Form (BNF) grammar is used to map the genotype to the phenotype. A separation of genotype and phenotype allows the implementation of various operators that manipulate (for instance by crossover and mutation) the genotype (in Grammatical Evolution - a sequence of bits) irrespective of the genotype to phenotype mapping (in Grammatical Evolution - an arbitrary grammar). In this paper we introduce an extension of the grammar used. This extended grammar allows functions to be dynamically defined and exploited, enabling the automatic discovery of any such regularity in the problem domain. A sample problem domain, known to have an implicit regularity, is examined utilising grammars with (a) no functions, (b) automatically defined functions (i.e. where the parameters and architecture are user pre-defined) and (c) dynamically defined functions (DDFs). The DDFs are shown to exploit the regularity in a manner competitive with the automatically defined functions, whilst eliminating the need for the user to explicitly define the architecture of the phenotype.

The structure of this paper is set out as follows: The next section (Section II) discusses related work and the motivation for the introduction of DDFs. Section III discusses the principles behind Grammatical Evolution. Section IV introduces the methodology behind the DDFs and provides details on some of the challenges created by their implementation. Section V introduces the problem domain explored and Section VI presents the results. Section VII is the conclusion and presents our ideas for future work.

II. Related Work

The ability to evolve functions, as a method of exploiting regularity, has been the subject of much research e.g. [4] [5] [6]. We briefly review the most relevant work relating to the use of functions as well as related work involving the alteration of grammar as a means of biasing the search.

A. Automatically Defined Functions

Genetic Programming (as described by Koza [1]) involves the evolution of programs that can be represented as lisp s-expressions. These are typically presented as parse trees. Koza’s Automatically Defined Functions (ADFs) methodology separates the function definition branch of the evolving s-expression from the main (or productive) branch of the program. If there are multiple functions defined then each of their branches are separate, ensuring that a function is only able to refer to one that has been previously defined. Using the representation proposed by Koza the user must decide how many functions there will be and how many parameters each function will have. Presumably some knowledge of the task domain is needed to guide this choice; in more complex problems this knowledge may not be available. Two methods were proposed by Koza to overcome this limitation. The first [7] involved seeding the initial population with a large number of different functions and allowing the evolutionary process to weed out the non-advantageous representations. The second [8] involved the use of architectural altering operators - additional operators that (instead of applying crossover) delete, create or

Robin Harper and Alan Blair are with the School of Computer Science and Engineering, University of New South Wales, Sydney, 2052, Australia and the ARC Centre of Excellence for Autonomous Systems. Email: {robinh,blair}@cse.unsw.edu.au

duplicate functions and parameters. In both these cases crossover must be constrained to ensure syntactically valid children. The method used to constrain the crossover employed by Koza was point-typing crossover which involves flattening the structure of the point chosen on the second parent (called the female by Koza) and accepting the proposed substitute sub-tree only if the function and terminal sets used in that sub-tree match the function and terminal sets in the first parent (the father). No second child is produced. A fuller description can be found in [8]. As will be seen, unlike ADFs, the use of DDFs in GE does not constrain the type of crossover that can be used nor does it require the use of additional operators with the attendant further design decisions that accompany their use.

B. *Functions in Grammatical Evolution*

O'Neill and Ryan [9] discuss how to write a grammar to utilise functions defined in a similar way to Koza's ADFs. ([10] discuses a similar, but more general, grammar driven approach.) Whilst this adapts ADFs to GE the adaptation still suffers many of the same limitations of ADFs in GP; importantly the structure has to be determined prior to the run(s) commencing. In addition the method used to ensure that the function cannot call itself (or in the case of a hierarchy of functions, that the hierarchy is observed) means that each function body consists of unique non-terminals. Apart from the repetition in the grammar that this entails (O'Neill and Ryan suggest that this might be overcome through the use of an attribute grammar) this methodology does not allow productions defined in the main body to be incorporated into a function (and vice-versa). In other words, like Koza's ADFs the development of the function(s) and the body are separate (although contemporaneous).[1] In their proposed further work O'Neill and Ryan suggest the further investigation into dynamic grammar based function definition and the use of attribute grammars - although there does not appear to be any further papers published by them in this regard. The methodology presented in the current paper is a form of dynamically defined functions that does not require attribute grammars and differs from their static definition methodology in that it allows the exchange of productions between functions and the main body (and vice versa).

C. *Altering the bias by altering grammar productions*

Whigham [12] introduces a method of biasing the search by modifying the grammar. Rather than seeking to encapsulate higher-level functions, his method was one of biasing the search in the grammar by identifying a production that appears to be useful and encapsulating that completed production as an expansion in the new grammar. Effectively this re-writing of the grammar is designed to bias any future searches (by newly created individuals) making it more likely that proven successful expansions would be selected by those newly created individuals.

There are two points to note. The first is that the biasing of the grammar occurs whenever the expansion could occur. Certainly in problem domains such as the multiplexer domain examined by Whigham it is likely that such an expansion will be useful wherever it appears, but one can imagine more complex domains where in a later part of the program a different expansion is required than that required in the earlier part of the program. The second point to note is that the biasing is global; the grammar is altered for all new individuals. The whole population being created after the bias will be altered by it. This is in contrast to, say, Koza's architecturally altering operations where the change is made for that specific individual (and its children). This move towards a more global change is not in harmony with our desire to have changes made only on an individual, albeit inheritable, basis.

Rosca and Ballard [5] introduced another method of biasing the grammar, called Adaptive Representation (AR). The idea behind AR is that the levels in the hierarchy are discovered by using either heuristic information as conveyed by the environment or statistical information extracted from the population. This information is used to introduce functions composed of small highly fit building blocks as additional terminals in the grammar. One more point to note is that once the "building block" has been incorporated into the grammar it is fixed and can no longer change (either through crossover or mutation). As with Whigham's methods this method requires generational change whereby new individuals are generated to take advantage of the changed grammar.

Unlike the DDFs proposed in this paper both previous methods of altering grammar production require a global analysis of the search space and a global change to the grammar. This runs the risk of requiring a task type specific heuristic. Altering the grammar on a global basis contrasts with nature's individual alterations and, as previously discussed, is not a route we wish to go down. [13] introduces a method of evolving the grammar (as well as the genetic code) on an individual basis, but this methodology still requires work to overcome some problems with the crossover as identified in that paper.

D. *Evolutionary induction of subroutines*

Angeline and Pollack [4] introduced the idea of a Genetic Library Builder (GLiB). The concept here is to allow a separate operator to compress randomly selected subtrees. These compressed subtrees are placed in a library and are assigned a unique name. The compressed subtree is replaced by its unique name. Subsequently, through crossover, that subroutine could be passed throughout the whole population, each individual potentially being able to use the unique name to access the globally held subtree. To complement the compression operator an expansion operator is also introduced which would take a named subroutine and

[1] It should be noted their use of the ripple crossover complicates this analysis as one of the effects of that crossover is that codons used in one part of the grammar can (although with a different interpretation) be used in a different part of the grammar. This is termed ripple crossover and the interested reader is referred to [11].

expand it *in situ* in the parse tree in question. Subroutines in the library would be removed if they were no longer being used by any of the population. This differs from Rosca's approach as the decision as to whether a particular subroutine is useful or not is left entirely to the genetic algorithm which, as the authors note, is more in line with the "enlightening" guidelines provided for genetic algorithm design in [14]. Two points to note are that with GLiB when a subtree with a function call is transferred (via crossover) to a new parent, the function call continues to refer to the function it has always referred to (i.e. it uses the same unique name and refers to the same code in the library) whereas with ADFs the function call would now refer to the function in the new parent. It is not clear whether this is an advantage or not. The second point is that whilst the subroutine is in the library it cannot further evolve nor can any subsection of its gentic material be utilised. Once expanded it can again evolve, but that will only be for the individual (and its children) that happen to have had the expand operator applied to them and not those that still refer to the code held in the library.

E. *Summary*

As discussed above it is clear that there has been a large amount of work on different methods of introducing functions into genetic algorithms and Genetic Programming in particular. Although each method has several of the features listed below none of the methods of which we are aware has all of them:

- Does not require the architecture to be specified in advance.
- Allows the transfer of productions between the main body and each of the functions.
- Requires no high-level heuristics which attempt to modify or bias the genetic algorithm.
- Requires no alteration to the crossover operator.
- Allows the functions to continue to evolve contemporaneously with the main body.
- Makes no global change to the evolutionary process.
- Requires no additional operators to be applied (i.e. is completely within the control of the crossover and mutation operators).
- Requires minimal changes to the representation (or grammar).

DDFs have all of the features listed above. This paper seeks to show that DDFs are competitive with ADFs in an environment with known regularity where the optimal structure of ADFs can (from this domain knowledge) be identified in advance. Given that DDFs are competitive with optimally structured ADFs and that they require no domain knowledge, no special operators and minimal change to the grammar then we believe that they enhance the ability of GE and GP to solve problems. The next two sections serve to describe Grammatical Evolution and the implementation of DDFs.

III. Grammatical Evolution

A. *Introduction*

Grammatical Evolution (GE) is a form of genetic programming which utilises the evolutionary algorithm to evolve code written in any language, provided the grammar for the language can be expressed in a Backus Naur Form (BNF) style of notation [3]. Traditional genetic programming, as described by Koza [1] has the requirement of "Closure". The term closure is used by Koza to indicate that a particular function set should be well defined for any combination of arguments. Previous work by Montana [15] suggests that superior results can be achieved if the limitation of "closure" required in traditional genetic programming can be eliminated, for example through typing. Whigham [16] demonstrates the use of context free grammars to define the structure of the programming language and thereby overcome the requirement of closure (the typing being a natural consequence of evolving programs in accordance with the grammar). GE utilises the advantages of grammatical programming, but, unlike the method proposed by Whigham, separates the grammar from the underlying representation (or as this is commonly referred to; the genotype from the phenotype).

It has been argued that the separation of genotype from phenotype allows unconstrained genotypes (and unconstrained operations on genotypes) to map to syntactically correct phenotypes. Keller [17] presents empirical results demonstrating the benefit of this type of mapping. One of the interesting aspects of having such a simple underlying genotype as GE (a bit string) is that it is possible to design a number of operators that act on this simple bit string. For instance crossovers can be designed which mirror uniform crossover, single bit crossover and crossovers which swap complete expansions in the underlying grammar, each of which represent (after the genotype to phenotype mapping has been performed) radically different methods of searching the phenotype space. The way that GE maps the genotype to the phenotype is discussed in the next subsection. The style of crossover used in this paper is discussed in subsection III.C

B. *Grammatical Evolution – the mapping*

Rather than representing programs as parse trees GE utilises a linear genome representation to drive the derivation of the encoded program from the rules of an arbitrary BNF grammar. Typically the genome (being a variable length bit string) is split up into 8 bit codons (groups of 8 bits) and these codons are used sequentially to drive the choices of which branch of the grammar to follow. The maximum value of the codon is typically many times larger than the number of possible branches for any particular non-terminal in the grammar and a mod operator is utilised to constrain it to the required number. For instance if a simple program grammar were as follows:

<Program> :: = <Lines>
<Lines> :: = <Action> | <Action > <Lines>
<Action> ::= North | South | East | West

Assume an individual had the following DNA and codon pattern:

```
DNA: 00100001 00010100 00100000 00000011 00010000
Codons:   33       20       32       3        16
```

No codon would be used for the first expansion (since there is only one choice). The initial codon of the genome would be used to determine whether to expand <Lines> to <Action> or to <Action><Lines>. The value of 33 would be MOD'd with two (since there are two choices) to give a value of 1. The second choice (<Action><Lines>) would then be chosen. The expression now is "<Action><Lines>". The first non-terminal <Action> is expanded by using the next codon (20). <Action> has four choices, so 20 would be MOD'd with four, to give zero and North would be chosen. The expression is now "North <Lines>". The next non-terminal (<Lines>) is expanded using the codon 32. 32 Mod 2 = 0, so <Action> is chosen, leaving us with "North <Action>". <Action> is then expanded using the next codon (3), to give us West. The expression is now "North West". There being no further non-terminals in the expression the expansion is complete. The remaining codon is not used.

If the expression can be fully expanded by the available codons (i.e. the expansion reaches a stage where there are no non-terminals) the individual is valid; if the codons run out before the expression is fully expanded the individual is invalid and removed from the population.

C. *Crossover Operators*

GE typically utilises a simple one-point operator termed the "Ripple Crossover" [11]. Recently a different type of crossover has been proposed for GE [18], referred to as the LHS Crossover Operator. The LHS crossover acts as a two-point crossover whereby the expansion of a rule of the grammar (or the Left Hand Side of a grammatical rule) is replaced by an expansion of the same rule found in the other parent. Since the LHS Crossover is the closest type of crossover operator to the operator used in GP, for the purposes of the tests conducted in this paper we limited the crossover to the LHS Crossover discussed in that paper, although it should be noted DDFs can be used with any scheme which operates on a sequence of bits.

```
code:       i_value
i_value:    xval
            | number
            | ( i_value op i_value )
            | - i_value
number:     0 . digit digit digit
digit:      0 | 1 | … | 9
op:         + | - | * | %
```

Listing 1 – a simple example grammar

IV. Dynamically Defined Functions

A. *Details of Implementation*

In order to implement DDFs certain additional rules are introduced into the grammar to allow the grammar itself to define functions and determine the number of parameters. Listing 1 contains a simple BNF Grammar that, for instance, could be used for symbolic regression (xval represents the value passed to the program for evaluation).

```
code:       i_value
            | DEFUN code
DEFUN:      PARAMS FUNCBODY
i_value:    xval
            | number
            | ( i_value op i_value )
            | - i_value
            | FUNC
number:     0. digit digit digit
            | PUSE
digit:      0 | 1 | … | 9
op:         + | - | * | %
FUNCBODY:   i_value
FPAR:       i_value
PARAMS:     NOP | PARAM | PARAM PARAMS
```

Listing 2 – grammar in Listing 1 adapted to use DDFs

If one were to adapt this grammar to utilise DDFs the following are required:

(i) The keyword DEFUN must be inserted into the grammar, specifying where a function definition may appear. This would normally be before any of the "main body" code. DEFUN expands into the non-terminals PARAMS and FUNCBODY.

(ii) The expansion of the keyword PARAMS will determine the number of parameters the function has. Each PARAM it expands into equates to one parameter. For instance an expansion of PARAM would equate to one paramater, an expansion of PARAM PARAM equates to two parameters etc. A NOP expansion means no parameter. The PARAMS production rule contained in Listing 2 is a typical example definition of PARAMS allowing functions to to have none, one or more than one parameter.

(iii) FUNCBODY needs to be defined. This specifies the form of the function body. In Listing 2, the function body is an <i_value>.

(iv) The terminal PUSE needs to be inserted into the grammar. This terminal represents the use of a parameter in a function body. Where there is more than one parameter, the codon value is used to select between them.

(v) The non-terminal FPAR needs to be defined. Each time a function call is to be made the non-terminal FPAR is used to represent each parameter to be passed to the function. The codons in the genotype will then be used to expand these FPARs leading to the construction of the parameters passed to the function.

(vi) Finally the keyword FUNC needs to be placed in the grammar, to indicate where a previously defined function can be called. In Listing 2 a function can be called whenever an <i_value> is to be expanded (in the example this is logical as the body of the function will also be an <i_value>).

The actual process carried out in the mapping of genotype to phenotype (originally described in section III.B) would now be as follows. The process would start as normal. When a DEFUN is encountered the mapping process would start creating a new function with a unique name (say, F0 for the first one, F1 for the second etc). In order to create the function it would use the next available codons to fully expand the PARAMS non-terminal, the number of "PARAM" that this expands to determines the number of parameters the function will have (see (ii) above). It then uses codons to decode the FUNCBODY non-terminal (see (iii) above). During this process the PUSE non-terminal (if encountered) expands to refer to any of the parameters declared for that function (see (iv) above). So, for instance, if a function had two parameters, PUSE would (for that function) expand as follows:

```
PUSE: Parameter0 | Parameter1
```

Where PUSE is embedded as a possible right hand expansion of a different non-terminal (as in Listing 2), the potential range of expansions of that non-terminal is increased. So, for instance, if FUNCBODY were being expanded for a function with two parameters, then (using the grammar in Listing 2) the expansion of <number> would now be (after the appropriate substitution of PUSE):

```
number:   0. digit digit digit
          | Parameter0 | Parameter1
```

Once FUNCBODY has been fully expanded the function definition is complete. The non-terminal FUNC (see (vi) above) is amended so that it includes the newly defined function as a potential expansion. If the non-terminal FUNC is encountered then it will expand to a call to this function populated with the correct number of FPARS (see (v) above). So for instance assuming two functions (F0 and F1) had been defined previously, respectively with one and three parameters, then, at that point and for that individual, if the FUNC non-terminal is encountered it would expand as follows:

```
FUNC: F0 (FPAR) | F1 (FPAR, FPAR, FPAR)
```

Listing 3 contains the grammar for the problem domain discussed in this paper – the Minesweeper domain. Section V.C contains a full discussion of the problem, but the grammar is presented here as an example of a slightly more complex grammar and to aid discussion of some of the intricacies that should be borne in mind when adapting a grammar to utilise DDFs.

```
code:          defineFunc lines
               | defineFunc code
defineFunc:    NOP | DEFUN

lines:         line
               | line lines
line:          action
               |if ( obstacle_ahead )
                    { lines } else { lines }
               | complex_line
               | doFunc
doFunc:        FUNC
complex_line:  frog ( line )
               | v8A ( line , line )
action:        move
               | left
               | v8
               | PUSE
v8:            ( digit , digit )
digit:         0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
FUNCBODY:      lines
PARAMS:        NOP | PARAM | PARAM PARAMS
FPAR:          v8
```

Listing 3 – the MineSweeper DDF Grammar

B. Number of functions and parameters determined dynamically

As can be seen from the grammars in Listing 2 and Listing 3, implementing DDFs can be designed so that the number of DEFUN keywords in any particular decoding depends on the specific genotype of each individual. The number of functions therefore is under the control of the evolutionary mechanism. Similarly with the parameters for each function – with an appropriately designed grammar the number of parameters are under genotype (and hence evolutionary) control. Listing 2 and 3 are examples of grammars that can contain zero or more parameters.

C. Implicit Architecture Altering Operations

Using, for illustrative purposes, the grammar in Listing 3 and the functionality of the LHS Crossover operator it is easy to see how the crossover operator can bring in or delete functions in a child or change the number of parameters used by a function.

Assume Parent 1 has a parse tree which commences as follows (using the grammar in listing 3):

```
<code> -> **<defineFunc>** <code> -> NOP <code>
       -> NOP <defineFunc> <lines>
       -> NOP DEFUN <lines>
```

And Parent 2 has a parse tree that commences as follows:

```
<code> -> **<defineFunc>** <lines>
       -> DEFUN <lines>
```

It can be seen that both Parent 1 and Parent 2 each define one function. Now if the two <defineFunc> in bold are swapped by the crossover operator (note although not shown here, the whole expansion of DEFUN, including expansion of PARAMS and the expansion of FUNCBODY are swapped), child1 will become:

```
<code> -> <defineFunc> <code>
       -> DEFUN <code>
       -> DEFUN DEFUN <lines>
```

and child 2 will be

```
<code> -> <defineFunc> <lines>
       -> NOP <lines>
```

representing an insertion of a function (child1) and the deletion of a function (child2).

As can be seen from the above example the use of the extra non-terminal <defineFunc> with a potential expansion to a NOP dummy function or a productive DEFUN serves as a "reserve" spot on the list of codons to facilitate the swapping of complete expansions of function definitions.

D. Ability to swap genotypes between functions and the main body

Using the LHS crossover, an expansion of, say, <line> in the main body of the program could be swapped for a similar expansion in a function (or vice-versa). Unless the grammar specifically defines different expansions within a function body the LHS crossover will swap matching non-terminals regardless of where they appear in the phenotype. Other operators that act on the bit stream equally ignore function/main body boundaries.

The transfer of codons from one location to another (for instance from the main body to a function) does raise with it the question as to whether these codons will have the same effect (or meaning) in their new location. A thoughtfully designed grammar can increase the chances of some of the structure being preserved; for example, by "isolating" keywords which are likely to change in meaning depending on their location (such as FUNC) – see listing 3 for an example.

V. THE TEST ENVIRONMENT

A. The GE environment

All the problem domains were tested in a constant GE environment. The following strategies were used:

- Selection of individuals was based on a probability selector, i.e. the chance of any particular individual being chosen to breed was directly proportional to its fitness.
- Although there was no random reproduction or copying of individuals, a strategy of retaining the fittest 5% of individuals was adopted.
- A constant mutation rate of 1 in 2,000 (independent of the length of the DNA string) was applied to each child generated.
- Two children were generated for each crossover operator and the mutation operator was applied to each bit of the genotype of each child.
- Invalid individuals were given zero fitness and were not eligible for selection or breeding.
- Individuals were started with a random bit string; in particular no attempt was made to ensure that the first random individuals were a minimum size, although if an individual was invalid it was regenerated.
- Given that GE seems to perform best with small populations (say, 200) over multiple generations (say, 2,500), these were the parameters used for the reported test results. Similar results (although generally with lower overall averages) were obtained with a more typical GP population size and generation count (5,000 and 50), but space prevents the reporting of these runs here.

B. Bloat control

Like all GP, GE has the potential to suffer from what has been termed "bloat". In order to avoid this issue, the system utilised a mild fitness pressure on DNA size. The rule used was: if DNA size exceeds 3000 bits, then if two entities had the same fitness then the entity with the smaller DNA is ranked ahead. This only impacts on the decision as to the top 5% of individuals to copy across to the next generation (it does not impact on the probabilities of selection for crossover). This strategy appears to be sufficient to contain bloat in GE. During a typical long run, with each increase in fitness, DNA sizes of the top individuals increase then slowly shrink again – until the next fitness improvement is encountered. Finally DNA sizes were capped at 7000 bits; if they exceeded this then the entity was deemed illegal.

C. Problem Domain - The Minesweeper Problem

In this problem, used by Koza in his analysis of ADFs [6], an agent (the minesweeper) needs to traverse a toroidal 8x8 grid. There are six squares in the grid that will terminate the agent (mines). The agent needs to visit each of the 58 other squares without being terminated (blown up). Following Koza's design each agent is tested on two different maps leading to a maximum score of 116. Since this was originally a GP problem the grammar is closed. The operators used by Koza are MOVE, which moves the agent one forward, LEFT which turns it left. To ensure closure both MOVE and LEFT return a vector in the form of (0,0). FROG, which takes one vector argument and moves the agent to the location specified by that vector and returns its argument (i.e. it acts as an identity operator with a side effect). IF-OBSTACLE, which acts on the first or second branch dependent on whether a mine lies ahead of the agent. V8A, which takes two vectors and returns their addition (moded by 8), RV8, which is a constant vector and PROGN, which serves to concatenate the other functions. Koza decided on an ADF hierarchy with two functions, neither of which had parameters. On examining the preliminary results from the MineSweeper problem we noticed that most runs of the DDFs seemed to prefer a single function definition, so we included an additional run of ADFs defined using one ADF rather than the two utilised by Koza.

In order to implement the above in GE, separate grammars were written which replicated both the Non-ADF grammar and the ADF grammars in the manner illustrated by [9]. Unfortunately space precludes their replication in this paper.

VI. Results

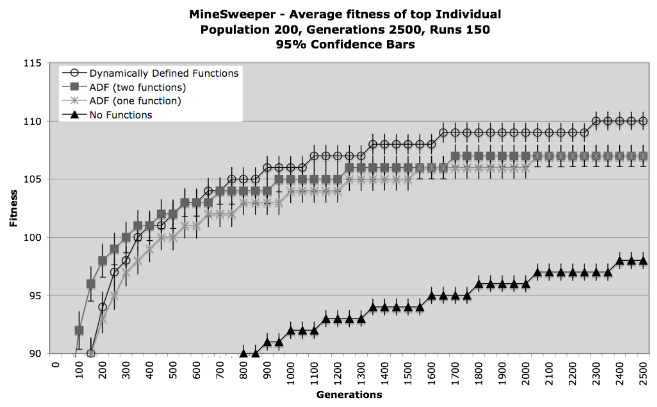

Chart 1- Average fitness over 250 runs

Chart 1 shows the fitness averaged over each of the 250 independent runs (note that the Fitness axis (the y-axis) does not go down to zero so as to enable the difference between the runs to be more easily seen). As can be seen from Chart 1 DDFs performed as well as (and towards the end of the runs better than) ADFs with one or two functions. The poor performance of the run with no functions confirms Koza's findings with respect to the improvement engendered by the use of functions in solving this problem domain.

Similar results are obtained if one looks at the percentage of independent runs generating 100% successful solutions – see chart 2 (note that this chart uses 116 fitness as a criteria of success– not the 112 success level used by Koza)

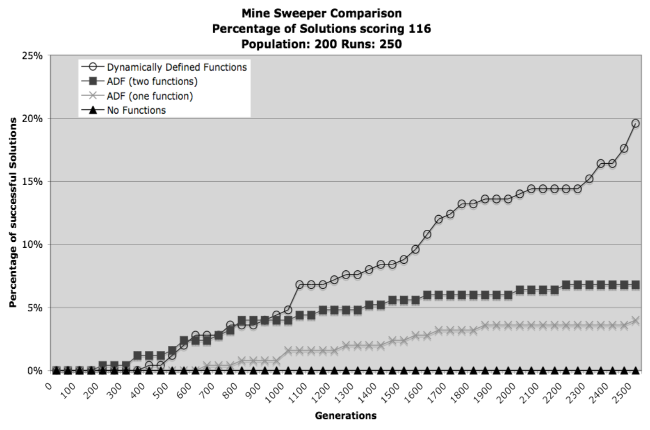

Chart 2 – Percentage of successful solutions

Again, as with Koza's results, the improved ability to solve the problem by utilising functions is demonstrated. The success rate with DDFs also appears to continue to rise with the number of generations at a higher rate than experienced with the ADFs. Each of the runs is independent of the others and this indicates that a grammar using DDFs, for this problem domain, appears to be more capable of continued evolution. An analysis of the number of functions contained within the genotypes (and expressed in the phenotype) of the individuals indicates that the runs tend to converge quickly (within 200 generations) to using one (approx. 64% of the runs) or two functions (approx 28% of the runs). However, within the population at any time there tends to be a few individuals with a different number of functions. Occasionally in certain runs the dominant number of functions shifts, from one to two functions or vice-versa. Sometimes this shift is permanent, sometimes it lasts for several hundred generations. Chart 3 shows a function distribution of a successful run which, whilst more volatile than most, illustrates some of the shifts that may occur. As can be seen individuals with no functions are quickly eliminated. Initially the dominant genome contains two functions. Although not very clear in the graph, the odd individual or small group of individuals with three or four functions are tried (for instance at generation 700 8 of the 200 individuals have three functions). At about 1050 generations a shift occurs to using one function. The system is in a state of flux between 1550 generations and 1650 generations - where there are a large number of individuals with one and two functions and some with three functions, before things settle down and the one function individuals then dominate the population until the end of the run.

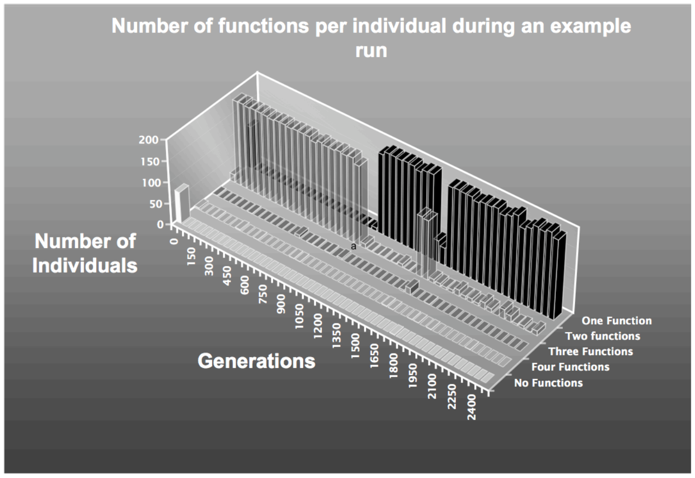

Chart 3 - Analysis of function distribution in a sample run

One of the questions raised by the above results is why, in this problem domain, DDFs appear to be more successful than ADFs in the continued evolution of solutions. The solutions found by the DDFs do not materially differ from those found by the ADFs, given that the DDFs appear to utilise one or two functions. Consequently, we do not believe that it's a matter of a difference in the expressive power between the two methods (since a one function ADF has the same expressive power as a DDF which happens to have one function) nor is it a matter simply of DDFs replicating successful solutions as we are looking at the number of independent runs which succeed. Instead, we believe the answer to this question might lie with the ability of DDFs to increase or decrease the number of functions during the course of a run. This ability to dynamically change the number of functions may allow not only the optimal architecture to be evolved (rather than created by the user) but also may allow the run to escape from local maxima that would otherwise trap it.

VII. Conclusions and Future Work

This paper serves as a proof of concept for DDFs. We have shown how a BNF grammar can be modified to allow the genome to dictate the number and use of functions freeing the user from the need to try and predict an optimal architecture beforehand. Our further goal was to allow such functions to be used without changing any other part of GE, i.e. without changing the mutation operator and without requiring changes to the crossover operator or the inclusion of different (architecture altering) operators.

We have shown that in the domain discussed in this paper, which does have a regularity to it (but not an obvious one function/two function regularity) DDFs are able to exploit that regularity. The runs show that even a small population (200) is capable of supporting a shift in the number of functions as the run explores the search space. In the problem domain examined this proved to be an advantage over a fixed architecture. We acknowledge it is likely that where there is clearly one optimal architecture which the user can define then DDFs might not be as efficient in finding a solution as the user pre-defined (and therefore smaller) search space. However, as the problems tackled by Genetic Programming and GE increase in complexity the difficulty in designing such hierarchies will also increase.

We see the main advantage of DDFs as freeing the user from guessing the hierarchy. We also hope that they prove conducive in problem sets where the fitness landscape varies during the search process.

One concern we still have with the functions is that a change in the number of parameters a function uses causes a "ripple" in the designation of which codons interpret which part of the grammar and will cause a large change in the phenotype (although as noted in [11] such a "ripple" is not *per se* destructive and can be beneficial). On way round this would be to employ some of the ideas of StackGP [19] whereby the number of parameters pushed onto the stack are chosen by the genome when calling the function (and not fixed by reference to the number of parameters the function expects). If too many parameters are pushed the excess ones are ignored. If too few (e.g. two parameters are pushed but the function expects three) an access of the excess parameters by the function returns a neutral or nil value. Whether this trade-of between expanding the search space and preventing the "ripple" effect currently experienced is beneficial will be interesting to explore.

References

[1] J.R. Koza Genetic Programming MIT Press/Bradford Books, Cambridge M.A., 1992

[2] Angeline, P. J. 1994. Genetic programming and emergent intelligence. In Advances in Genetic Programming, K. E. Kinnear, Ed. Mit Press In Series In Complex Adaptive Systems. MIT Press, Cambridge, MA, 75-97.

[3] Ryan C., Collins J.J., O'Neill M. Grammatical Evolution: Evolving Programs for an Arbitrary Language. Lecture Notes in Computer Science 1391. First European Workshop on Genetic Programming 1998.

[4] Angeline PJ and Pollack JB (1992) "The Evolutionary Induction of Subroutines," The Proceedings of the 14th Annual Conference of the Cognitive Science Society. Hillsdale, NJ: Lawrence Erlbaum, pp 236-241

[5] J. P. Rosca and D. H. Ballard. Genetic programming with adaptive representations. Technical Report TR 489, University of Rochester, Computer Science Department, Rochester, NY, USA, Feb. 1994.

[6] J.R. Koza Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, Mass., 1994.

[7] J.R. Koza Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Cambridge, Mass., 1994. Chapter 21.

[8] Koza, John R. 'Architecture-Altering Operations for Evolving the Architecture of a Multi-Part Program in Genetic Programming', Technical Report STAN-TR-CS-94-1528, Computer Science Department, Stanford University, 21 October.

[9] Michael O'Neill, Conor Ryan: Grammar based function definition in Grammatical Evolution. GECCO 2000: 485-490

[10] Ernesto Rodrigues and Aurora Pozo Grammar-Guided Genetic Programming and Automatically Defined Functions, Advances in Artificial Intelligence: 16th BrazilianSymposium on Artificial Intelligence, SBIA 2002

[11] Keijzer, M., Ryan, C., O'Neill, M., Cattolico, M., and Babovic, V. 2001. Ripple Crossover in Genetic Programming. In Proceedings of the 4th European Conference on Genetic Programming (April 18 - 20, 2001). J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. Tettamanzi, and W. B. Langdon, Eds. Lecture Notes In Computer Science, vol. 2038. Springer-Verlag, London, 74-86

[12] P.A. Whigham, Inductive Bias and Genetic Programming (1995) First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications, GALESIA

[13] O'Neill, Michael and Ryan, Connor (2004). Grammatical evolution by grammatical evolution: The evolution of grammar and genetic code. In Keijzer, Maarten, O'Reilly, Una-May, Lucas, Simon M., Costa, Ernesto, and Soule, Terence, editors, Genetic Programming 7th Euorpean Conference, EuorGP 2004, Proceedings, volume 3003 of LNCS, pages 138-149, Coimbra, Portugal. Springer-Verlag.

[14] Goldberg, D., 1989, "Zen and the Art of Genetic Algorithms", In Proceedings of the Third International Conference on Genetic Algorithms, J. Schaffer (ed), Los Altos, CA: Morgan Kaufmann Publishers, Inc.

[15] Montana D, (1994) Strongly Typed Genetic Prgramming. Technical Report 7866, Bolt Beranek and Newman Inc.

[16] Whigham P.A, Grammatically-based Genetic Programming (1995) Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pages 33-41. Morgan Kaufmann Pub.

[17] Robert E. Keller, Wolfgang Banzhaf 1996. Genetic Programming using Genotype-Phenotype Mapping from Linear Genomes into Linear Phenotypes (1996) Genetic Programming 1996: Proceedings of the First Annual Conference

[18] Robin Harper and Alan Blair, A Structure Preserving Crossover in Grammatical Evolution, 2005 IEEE Congress on Evolutionary Computation, 2537

[19] Perkis, T. 1994. Stack-Based Genetic Programming. In Proceedings of the 1994 IEEE World Congress on Computational Intelligence. pp. 148--153.