Grammatically-based Genetic Programming

P.A.Whigham
Department of Computer Science,
University College, University of New South Wales
Australian Defence Force Academy
Canberra ACT 2600 AUSTRALIA

Abstract

The genetic programming (GP) paradigm is a functional approach to inductively forming programs. The use of *natural selection* based on a *fitness function* for reproduction of the program population has allowed many problems to be solved that require a non-fixed representation. Attempts to extend GP have focussed on typing the language to restrict crossover and to ensure legal programs are always created. We describe the use of a context free grammar to define the structure of the initial language and to direct crossover and mutation operators. The use of a grammar to specify structure in the hypothesis language allows a clear statement of inductive bias and control over typing. Modifying the grammar as the evolution proceeds is used as an example of learnt bias. This technique leads to declarative approaches to evolutionary learning, and allows fields such as incremental learning to be incorporated under the same paradigm.

1 Introduction

The Genetic Programming paradigm (GP) has received some attention lately as a form of adaptive learning [Koza, 1992b]. The technique is based upon the genetic algorithm (GA), [Holland, 1992], which exploits the process of natural selection based on a fitness measure to breed a population of trial solutions that improves over time. The ability of GA's to efficiently search large conceptual spaces makes them suitable for the discovery and induction of generalisations from a data set. A summary of the genetic programming paradigm may be found in [Koza, 1992a].

1.1 Closure and Genetic Programming

The requirement of *closure* made many program structures difficult to express. Closure, as defined by Koza [Koza, 1992b], was used to indicate that the function set should be well defined for any combination of arguments. This allowed any two points in a program to be *crossed over* by swapping their program structures at these points in the program tree.

Many problems require functions with typed arguments. Generally these are handled by constraining the syntactic structure of the resultant programs [Koza, 1992c]. In particular, when a crossover point is selected, the second point for crossover must match the syntactic type of the first point. This ensures that only *syntactically legal programs* are created when 2 programs swap components [Koza, 1992b]. These typing issues have been further addressed by Montana [Montana, 1994], who sets the type of each variable, constant, argument and return value before the initial population is created. This constrains the initialisation process and the subsequent genetic operators so that only legal programs are created.

1.2 Bias in Genetic Programming

Explicit typing may be considered as a language bias. Bias has been used with other machine learning methods, such as Inductive Logic Programming [Cohen, 1993], to restrict the hypothesis language. Although syntactic typing restricts the form of the resultant program, it *does not allow* higher level structure to be represented.

This paper introduces the use of context free grammars (CFG) to overcome the closure requirements for GP. In particular, the grammar allows the user to bias the initial GP structures, and automatically ensure typing and syntax are maintained by manipulating the explicit derivation tree from the grammar. We also describe extensions to this form that allow bias to be learnt as the evolution of a solution proceeds. The system will be referred to as *context-free grammar genetic programming*, or CFG-GP.

1.3 Context Free Grammars

An introduction to grammars may be found in [W.A. Barrett and J.D.Couch, 1986].

A context free grammar is a four-tuple $(N, \sum, P, S)$, where N is the nonterminal [1] alphabet, $\sum$ is the terminal alphabet, P is the set of productions and S is the designated start symbol. The productions are of the form $x \rightarrow y$, where $x \in N$, $y \in \{\sum \cup N\}^*$. Productions of the form

$x \rightarrow y$

$x \rightarrow z$

may be expressed using the disjunctive symbol $|$, as

$x \rightarrow y \mid z$

1.3.1 Derivation Step

A derivation step represents the application of a production from P to some nonterminal $A \in N$. We use the symbol $\Rightarrow$ to represent the derivation step. For example, given the nonterminal A, we represent the derivation step from A, by applying production $A \rightarrow \theta$, as:

$$\alpha \ A \ \beta \overset{A \rightarrow \theta}{\Rightarrow} \alpha \ \theta \ \beta$$

where $\alpha, \beta, \theta \in \{N \cup \sum\}^*$ and $A \in N$.

A derivation *rooted* in A, where $A \in N$, is defined as

$$A \overset{*}{\Rightarrow} \alpha \text{ where } \alpha \in \{N \cup \sum\}^*$$

Here $\overset{*}{\Rightarrow}$ represents zero or more derivation steps. A series of derivation steps may be represented as a tree, such as figure 2.

2 A Grammatically-based Learning System

We will use the 6-multiplexer as an example to initially describe the differences between traditional GP and CFG-GP. A full description of the 6-multiplexer may be found in [Koza, 1992b].

[1] Grammars traditionally use the definitions *terminal* and *nonterminal* to represent the atomic tokens and symbols to be replaced,respectively. GP have used these terms to distinguish functions with > 0 arguments, and $0 - arity$ functions or atomic values. To ensure there is no confusion when discussing GP constructs we will use the words GPterminals and GPnonterminals.

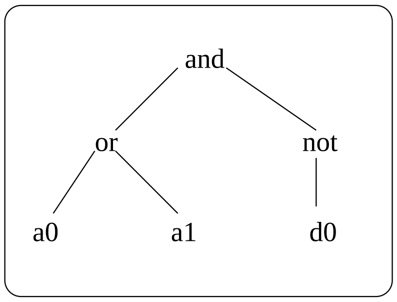

Figure 1: A Program for the 6-Multiplexer

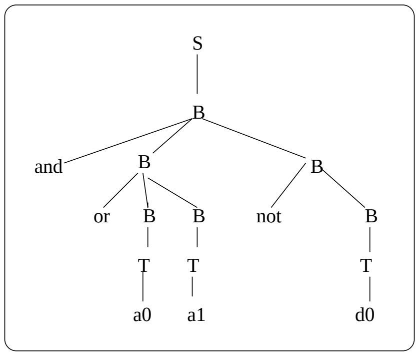

Figure 2: A Program created from a CFG

Table 1: 6-Multiplexor GP Representation	
GPTerminals	a0 a1 d0 d1 d2 d3
GPNonterminals	and(2) or(2) not(1) if(3)

The above table shows the GP definitions for representing the 6-multiplexer problem. A possible tree created using these definitions is shown in figure 1.

A grammar (one of many possibilities) that allows the creation of the same functional structures is:

$S \rightarrow B$

$B \rightarrow and \ B \ B \mid or \ B \ B \mid not \ B \mid if \ B \ B \ B \mid T$

$T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

A derivation tree using this grammar is shown in figure 2. The trees of figures 1 and 2 represent the function $and(or(a0, a1), not(d0))$.

2.1 Creating the Initial Population

The initial population of GP programs are created using the technique of *half-ramping*. Typically programs are created from a depth of 2 upwards, some forced to be the full depth, others randomly generated up to the (current) maximum depth.

When using CFG-GP the initial population is defined by a series of parameters *DEPTH* **depth number-of-programs**, which creates **number-of-programs** with a parse-tree depth *not exceeding* **depth**. The following steps are done to create the initial program population using the CFG $\{N, \sum, P, S\}$.

1. Label each production $A \rightarrow \alpha$,where $A \in N$ and $\alpha \in \{N \cup \sum\}^*$, with the minimum number of derivation steps to create only terminals. ie. the minimum number of derivation steps to create $A \overset{A\to\alpha}{\Rightarrow} \alpha \overset{*}{\Rightarrow} \beta$ where $\beta \in \sum^*$.

   We define $A \rightarrow \beta$, where $\beta \in \sum^*$, as having a depth of 1.

2. For the range of depths and number for each depth $D = i..j$ do
   - (a) Select the start symbol S and label it as the current nonterminal A
   - (b) Randomly select a production $P_1 \in P$ of the form $A \rightarrow \alpha$ with minimum derivation steps to $\sum^* < D$
   - (c) For each nonterminal $B \in \alpha$, label B as the current nonterminal, and repeat steps (b) and (c).

To ensure a measure of diversity all programs in the initial population are required to have different parse (derivation) trees.

2.2 Selection of Individual Programs

The selection of programs uses the same process as GP - the programs are selected with some probability related to their fitness measure. We use *proportional fitness selection* for the problems described in this paper.

Programs are executed using a *pre-order* traversal of their derivation trees.

2.3 Reproduction

Programs are copied to the next generation as a *REPRODUCTION* operation based on their proportional fitness, in a similar manner to GP.

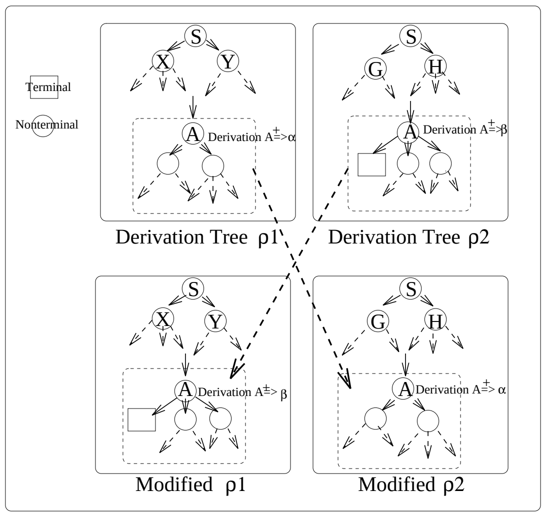

Figure 3: Crossover using Derivation Trees

2.4 Crossover using a CFG

Examining figure 2 shows that the executable programs are constructed from the terminals of the grammar. All terminals have at least one nonterminal above them in the program tree (at the very least S), so without loss of generality we may constrain crossover points to be located *only on nonterminals*. The crossover operation maintains legal programs of the language (as defined by the grammar) by ensuring that the same non-terminals are selected at each crossover site. The parameter *MAX-TREE-DEPTH* is used to indicate the deepest parse tree that may exist in the population.

The crossover algorithm (see figure 3) is:

1. Select two programs, with derivation trees ρ_1 and ρ_2, from the population based on fitness
2. Randomly select a non-terminal $A \in \rho_1$
3. If no non-terminal matches in ρ_2, goto *step 1*
4. Randomly select $A \in \rho_2$
5. Swap the subtrees below these non-terminals

We note that the parameter *MAX-TREE-DEPTH* may exclude some crossover operations from being performed. In the current system, if following crossover either new program exceeds *MAX-TREE-DEPTH* the entire operation is aborted, and the crossover procedure recommenced from step 1.

2.5 Mutation

Mutation applies to a single program. A program is selected for mutation, and one non-terminal is randomly selected as the site for mutation. The tree below this non-terminal is deleted, and a new tree randomly generated from the grammar using this non-terminal as a starting point. The tree is limited in total depth by the current maximum allowable program depth (*MAX-TREE-DEPTH*), in an operation similar to creating the initial population.

3 Applying CFG-GP to the 6-Multiplexer Problem

The grammar productions used to define the structures for solving this problem were:

$S \rightarrow B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T$

$T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

The start symbol is S, nonterminals $\{B, T\}$ and terminals $\{if, and, or, not, a0, a1, d0, d1, d2, d3\}$.

Table 2: CFG 6-Multiplexer Setup

GENERATIONS	50
POPULATION SIZE	500
DEPTH 4	100
DEPTH 5	100
DEPTH 6	100
DEPTH 7	100
DEPTH 8	100
CROSSOVER	450
REPRODUCTION	50
MAX-TREE-DEPTH	8
FITNESS MEASURE	64 possible cases
PROGRAM SELECTION	Proportionate

The grammar was applied for 100 different runs based on table 2, with the resulting *probability of success* determined as 29%. We note that traditional GP applied to the multiplexer with *similar* population parameters has a *probability of success* of approximately 67% [Koza, 1992b].

3.1 Discussion of Initial Results

There is some difficulty with directly comparing GP applied to the 6-multiplexer, and the technique that we have just described. When applying GP, 90% of the crossover occurred at internal points in the tree, and only 10% at the tips. With the CFG, we merely selected a random site from one of the non-terminals. This would normally bias more towards sub-trees close to a terminal, as the number of nodes generally increases at each level in the tree. The differing results of CFG-GP may be partially accounted for by this distinction. Other factors, such as the differing tree structures and initial population seeding may also influence the result. Further work is indicated to elucidate these differences.

4 Applying Bias in the 6-Multiplexer Grammar

To examine the effect of applying bias in the grammar, we created several, more specific, versions of the grammar previously presented.

Each new grammar was more specific (to the solution) than the last, and should therefore have given a higher probability of success. The first grammar (below) biased the solution to using the if functions as the first function in the program.

$S \rightarrow IF$

$IF \rightarrow if\ B\ B\ B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T$

$T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

The probability of success was found to be approximately 41%. Extending the bias further, we used the knowledge that the address line $a1$ partially selected the resulting data line to create the initial function based on the address line, as follows:

$S \rightarrow IF$

$IF \rightarrow if\ a1\ B\ B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T$

$T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

This grammar achieved a probability of success of approximately 74%. The final bias was to only allow programs that used both $a0$ and $a1$ as the initial selection mechanisms to be created:

$S \rightarrow IF$

$IF \rightarrow if\ a1\ IFAPART\ B$

$IFAPART \rightarrow if\ a0\ B\ B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T$

$T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

This grammar achieved a probability of success of approximately 98%.

4.1 Discussion of Bias with the 6-Multiplexer

The previous results show that using a bias in the grammar supports the probability of finding a solution. This leads to the argument that as the problem space becomes large we should use a language bias to restrict the possible program structures, and therefore hopefully have greater success in finding solutions. The grammar has allowed the user to impose a search bias and language bias to the forms of program that are created, and the structure of programs during evolution. Although we may augment GP using specialised functions to improve performance, there are differences between the 2 approaches. Using CFG-GP the bias may be declared without changing the underlying functions used for the solution, whereas GP must define new functions explicitly. Also, it is difficult to express combinations of functions as a bias in GP.

5 Using a CFG to Control Typing

As an example of a program that requires typing, we examine the grammar to define the *wetness index*, a classification used to describe landscape moisture. This problem has been described by the author [Whigham, 1994], where the GP approach required the spatial operations to be propositionalised to avoid typing conflicts with the boolean operators. We attempted to solve the boolean case for one of the wetness values that required a spatial description. The typing constraints are necessary as the spatial operators do not return boolean values, and therefore may not be used as arguments to the boolean functions.

$S \rightarrow B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid OBJECT$

$OBJECT \rightarrow landunit\ REL\ LUVAL\ SPAEXP$

$OBJECT \rightarrow slope\ REL\ SLVAL\ SPAEXP$

$REL \rightarrow < \mid > \mid \leq \mid \geq \mid =$

$LUVAL \rightarrow Floodplain\ Inundated$

$LUVAL \rightarrow Present\ Floodplain$

$LUVAL \rightarrow Tributary\ Stream$

$LUVAL \rightarrow Major\ Stream \mid Terrace \mid Sand\ Dunes$

$LUVAL \rightarrow Valley$

$LUVAL \rightarrow Lower\ Footslopes \mid Upper\ Footslopes$

$LUVAL \rightarrow Hillslopes\ and\ Crests$

$LUVAL \rightarrow Footslopes \mid Low\ Hillslopes$

$LUVAL \rightarrow Gentle\ Slopes$

$LUVAL \rightarrow Dam \mid Quarry$

$SLVAL \rightarrow 0-5\% \mid 5-10\% \mid 10-15\% \mid 15-20\%$

$SLVAL \rightarrow 20-25\% \mid > 25\%$

$SPAEXP \rightarrow all\ SPAOP \mid any\ SPAOP \mid current$

$SPAOP \rightarrow adjacent$

Note that the $LUVAL$ and $SLVAL$ have ordered terminals, allowing meaningful statements using the relations from REL. Using this grammar the program created partial solutions to the wetness problem, with the following settings:

Table 3: Wetness Index Setup

GENERATIONS	50
POPULATION SIZE	500
DEPTH 5	50
DEPTH 6	150
DEPTH 7	150
DEPTH 8	150
CROSSOVER	450
REPRODUCTION	50
MAX-TREE-DEPTH	11
FITNESS MEASURE	3192 Wetness Values
PROGRAM SELECTION	Proportionate

Although we did not discover a complete solution using the above settings, the program demonstrated that typing constraints could be maintained during crossover via the grammatical definition. An example of a program generated using the settings of Table 3 is shown below:

```
and(or(not(landunit(<=,Upper Footslope,
                         all(adjacent))),
      and(or(landunit(<=,Footslope,current),
             slope(<=,5%,current)),
          and(slope(>=,10-15%,current),
              landunit(<,Present Floodplain,
                         current)))),

    landunit(=,Low Hill Slopes,current))
```

6 Generating New Productions

There has been some interest in the GP community to adapting the representation language as the evolution of a solution proceeds [Rosca and Ballard, 1994]. These approaches tend to take a group of functions and turn them into a single function, with the GPterminal values turned into variables (of the new function). The interest is in creating higher level functions that appear to be useful, by identifying and encapsulating useful building blocks. Other work has studied automatically defined functions (ADF) to increase the descriptive power (the language) of the initial constructs [K. E. Kinnear, 1994]. These ADF's are true functions, in the sense of having a list of arguments, a function body and a return value. We will not consider this extended version of changing the language, but will focus on modifying the grammar as a *weak form* of changing representation.

Using the language presented as a CFG, we suggest that new productions may be discovered from an analysis of the current fit individuals in a population. The productions representing these individuals may be used to modify the original grammar. The grammar may then be used to introduce new programs, or parts of programs, to the population. We do not consider creating new functions, however we will demonstrate a form of *encapsulation* that builds new terms into the initial language.

6.1 Modifying a CFG to constrain a search space

Our goal is to create a set of operations that modify the CFG that is initially used as the bias for the learning system. Any system that attempts to modify the grammar must:

- recognize the need for the change of representation
- identify the program individuals to be used to direct changes to the grammar
- identify the program productions that are to be refined
- incorporate the changes to the grammar
- incorporate the grammar back into the population of programs

We will now examine each of these points in more detail.

6.2 The Need for a Change of Representation

Assumption 1 *The Population, after each generation, will contain useful productions.*

While there is some variation in the fitness of the population this will be true.

6.3 Identifying Individuals

Identifying which programs (or parts of programs) should be identified as useful has been studied by Rosca and Ballard [Rosca and Ballard, 1994], where two criterion were studied: *fit blocks* and *frequent blocks*. They concluded that *fit blocks* were the most useful measure to determine a building block.

Assumption 2 *A fit program will contain (some) relatively fit productions, which may be exploited in changes to the grammar.*

We will chose as our candidate program the fittest individual in the population. When more than one program is equally fit we will chose the program with the least depth of parse tree.

6.4 Identifying Useful Productions

Given we have selected a program to extend our grammar, we have a range of options concerning which productions (or parts of productions) are to be used. Our first goal will be to allow a refinement of the grammar to proceed, as the initial grammar may be viewed as the *most general* description of the program search space.

The most easily incorporated refinement is to propagate a terminal up the program tree to the next level of non-terminals. As shown in figure 4, a selected terminal is propagated up the parse tree to the next production which has other terminals or non-terminals *at the same level*. This will describe a new production which places the terminal in a previous non-terminal spot in the grammar. Our problem now becomes one of determining which terminal should be selected.

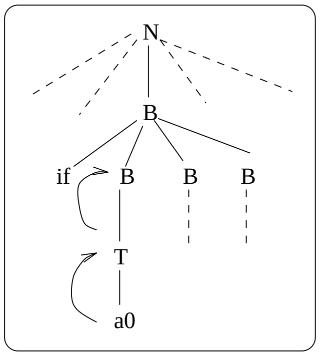

Figure 4: Propagating a terminal up the tree

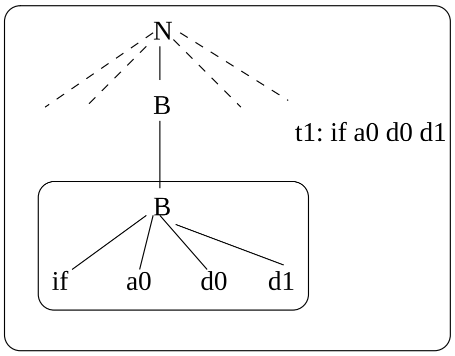

Figure 5: Encapsulating a production

Assumption 3 *The deepest terminals in the parse tree will have the least influence on the overall structure of the program* [2]*.*

We chose the terminal at this most general point as the location to refine. This will mean that we attempt to initially refine the arguments to functions, rather than the functions themselves. When more then one terminal is at the deepest point in the derivation tree of the program, the left-most terminal is selected for refinement.

6.5 Incorporating Productions into the CFG

A grammar represents a generalisation of the space of solutions for a problem. Hence some productions will be selected more often when composing a (correct) solution. We can represent this concept explicitly by giving each production a merit value, which represents its probability of selection from some non-terminal. The initial grammar may now be biased by changing the *merit* value of some production, thereby changing its likelyhood of being applied when a program is generated using the grammar. A grammar that begins without any *production bias* will set all productions to have a *merit* of 1. The probability of a production being selected during an *epoch-replacement* is now *proportional to its merit.*

As an example, consider the production $T \rightarrow a0$ which has been used in the derivation steps:

$$if\ T\ x_1\ x_2 \overset{T \rightarrow a0}{\Rightarrow} if\ a0\ x_1\ x_2 \text{ where } x_1, x_2, a0, if \in \sum$$

[2]This may be true syntactically, however any terminal may have an unpredicatable effect on the semantics of program. Our study is focussed on the syntactic manipulation of programs - the semantics are only revealed by the evaluation of the program as a complete unit.

We extend the productions from B to include:

$$B \rightarrow if\ a0\ B\ B$$

If this production does not already exist, it is given a *merit* of 1. If the production does exist, its *merit* is incremented, thereby increasing its probability of selection when using the grammar to create programs.

A second situation of propagating the terminal symbols occurs when all siblings of a propagated terminal are also terminal symbols. Here, for example, a production $B \rightarrow if\ a0\ d0\ d1 \mid$ from the fittest program has been found to contain the deepest terminals, as shown in figure 5. We wish to encapsulate the terminals as one functional symbol. This is performed by creating a new terminal that binds all of these terminal symbols together. For example, the production $B \rightarrow if\ a0\ d0\ d1 \mid$ could be covered by creating the new encapsulated terminal $t1$ with definition $if\ a0\ d0\ d1$. The terminal $t1$ is then included in the set of possible terminals, and in the production from B (with an initial *merit* of 1):

$$B \rightarrow t1$$

This new symbol $t1$ of the grammar may now be involved in future refinements and encapsulations, thus allowing useful components to be combined and propagated.

6.6 Incorporating the Grammar into the Population

Modifying the grammar has not changed the current population of programs that have been created. We use an *epoch replacement* [Rosca and Ballard, 1994] approach to introducing new programs during the evolution. The term REPLACEMENT is used to represent the number of new programs to be created each generation.

6.6.1 Does Merit Selection Work ?

We wish to first demonstrate that adapting the *merit* values of productions during the evolution of the solution is an effective mechanism for improving the refinement of grammars. The settings of the CFG-GP used for the 6-multiplexer problem are shown in table 4.

Table 4: CFG-GP 6-Multiplexer Parameters

GENERATIONS	50
POPULATION SIZE	500
DEPTH 4	50
DEPTH 5	150
DEPTH 6	100
DEPTH 7	100
DEPTH 8	100
CROSSOVER	400
REPRODUCTION	50
REPLACEMENT	50
MAX-TREE-DEPTH	8
FITNESS MEASURE	64 possible cases
PROGRAM SELECTION	Proportionate

The system used $REPLACEMENT = 50$ so that 10% of the population was recreated each generation. Using the initial CFG-GP system (without refinement) the REPLACEMENT operator merely used the initial (static) grammar to replace individuals for each generation. The *probability of success* for this arrangement was found to be 22%, based on 100 runs of the system.

The refinement operator was then introduced using the same settings, and the *probability of success* was found to increase to 52%. The refinement was performed each generation, using the fittest program and selecting the deepest terminal for inclusion in the grammar, as described earlier.

The introduction of a *merit proportionate selection* for production selection further increased the success probability to 66%[3]. Each of these increases are statistically significant [Whigham, 1995].

The following grammar was created by a *successful* run of the CFG-GP, after 41 generations (<num> indicates the *merit* value for the production).

$S \rightarrow < 1 > B$

$B \rightarrow < 7 > if\ a1\ d3\ B\ |< 4 > or\ d1\ B$

$B \rightarrow < 3 > if\ a0\ d1\ d3\ |< 3 > if\ a0\ d1\ B$

$B \rightarrow < 3 > if\ a0\ B\ B < 2 > or\ a1\ B$

$B \rightarrow < 1 > if\ a0\ d1\ d2\ |< 1 > if\ a1\ d1\ B$

$B \rightarrow < 1 > if\ a1\ B\ B < 1 > and\ d1\ B$

$B \rightarrow < 1 > if\ a0\ d1\ d0$

$B \rightarrow < 1 > or\ d0\ B\ |< 1 > and\ d2\ B\ |< 1 > and\ a1\ B$

$B \rightarrow < 1 > or\ B\ a1 < 1 > and\ B\ B\ |< 1 > or\ B\ B$

$B \rightarrow < 1 > if\ B\ B\ B\ |< 1 > not\ B\ |< 1 > T$

$T \rightarrow < 1 > a0\ |< 1 > a1$

$T \rightarrow d0\ |< 1 > d1\ |< 1 > d2\ |< 1 > d3$

This grammar clearly shows a bias towards using the *if* function with the address lines $a0$ and $a1$ as the first test values. The grammar is beginning to reflect (in a general sense) the form of a preferred solution.

6.7 The Effect of Encapsulation

We studied the effect of encapsulation using a $REPLACEMENT$ of 50 individuals per generation. The setup was that of Table 4, and encapsulation was performed whenever it was detected. For an encapsulation to be detected required previous grammatical changes to have forced a refinement to the point where we tried to refine a production that was composed *only of terminals*. Hence an encapsulation (given the previous initial grammar) may only occur several times (if at all) during the 50 generations. We performed 100 runs of the genetic program using encapsulation and $REPLACEMENT = 50$. The *probability of success* was found to be 63%, which implied that there is little benefit in using encapsulation for this problem.

This (partially) negative result may be a reflection of the limited number of generations that were performed (50). In a typical (failed) run of the CFG-GP only 2 or 3 encapsulations would be performed, thus limiting the possible benefits of discovering (and incorporating) these new useful terms into the population. Certainly for some runs of the genetic program useful encapsulations were discovered, such as:

$$t1 \equiv if\ a1\ d2\ d0$$

This shows that the address line $a1$ has been found

[3]The initial system (without refinement) did not change when merit proportionate selection of productions was introduced, since the grammar was not modified during the evolution of the system.

to be a useful selection point for determining which of the data lines is chosen.

To investigate whether the limited generations were a contributing factor to the lack of effect using encapsulation, the previous experiments were redone with the maximum number of generations increased to 100. We found that refinement without encapsulation improved to 85%, and when encapsulation was included the probability of success was 81%, thereby showing a similar lack of improvement to the 50 generation results.

7 Discussion of Previous Results

The previous work has shown that by refining a grammar using productions selected from fit programs, and reintroducing programs into the population via the modified grammar, there are measurable benefits in terms of finding a solution for the 6-multiplexer. We have also shown that the merit based selection of the grammatical productions aids in creating more useful programs during the evolution. The lack of positive results for the encapsulation operation may be due to the small probability that they had time to propagate within the population before the number of generations was completed. Even with the 100 generation tests at most 3 encapsulations were created (out of the 100 runs). Encapsulation, given our current grammar, is not easily created, and does not appear to affect the results.

8 Incremental Learning using a Biased Grammar

At the completion of a CFG-GP run, using the previous grammatical operators, we would expect the evolved grammar to represent a bias towards the solution. If this is the case, using this modified grammar as the initial grammar should increase the performance of the genetic program. We tested this hypothesis by using the grammar described in section 6.6 as the initial grammar for the CFG-GP [4]. The *merit* of each production was loaded as part of the initial grammar information, as well as all productions that formed the complete grammar. Otherwise, the parameter settings were those of Table 4. This technique represents a form of incremental learning, in that information from a previous run of the program was passed onto a subsequent run, with an expected improvement in performance.

The *probability of success* using this learnt grammar was found to be 88%. This confirmed that the grammar had been modified in such a way as to bias the search space of the program for the 6-multiplexer.

[4]We ignored the encapsulation operator as its benefit was minimal for the 6-multiplexer

9 Future Work

This paper has described the starting point for a new genetically-based system for learning. Work is required in the following areas:

- Studying the structure of initial populations, and their influence on the *probability of success*.
- Applying the inductive bias techniques to other problems to find out how well these operators generalise.
- Examining whether we can learn a grammar that may be applied to a similar (but different) problem with an expected improvement. For example, will the grammar learnt from the 6-multiplexer be useful for the 11-multiplexer? This approaches the problem of defining classes of problems that are suitable for learning inductive bias.
- Describing the class of problems amenable to description using a context-free language.

10 Conclusion

This paper has demonstrated the use of a context-free grammar to define the structure of the language manipulated by a genetic program. The grammar was used both as the generation mechanism and to allow crossover and mutation to occur without violating the requirements of closure. Learning new productions represents an attempt to shape the search space (as represented by the grammar) while the evolution proceeds. The positive results achieved by using a refinement operator to gradually change the underlying grammatical definition has much promise in applying automatic bias to genetic learning.

The encapsulation operator did not appear to have a significant influence on the probability of success for the problems that were examined. Further work is required to determine whether this was due to the form of the problems, or that the operator does not represent any significant information. Certainly new symbols were created that encompassed relevant constructions, however these discoveries were not normally exploited back into the population with enough frequency to have an impact on the learnt productions.

We have shown that the changes to the grammar do represent the underlying form of the problem. This was demonstrated by using the modified grammar as the starting point for learning, which led to an improved *probability of success*. This is a form of incremental learning.

11 Acknowledgements

The author would like to express his thanks to Bob McKay for discussions, and to William Cohen for seeding the idea of applying language grammars in an evolutionary context. Thanks must also go to Justinian Rosca for comments and suggestions.

References

[Cohen, 1993] William W. Cohen. Grammatically biased learning: Learning logic programs using an explicit antecedent description language. Technical report, AT and T Bell Laboratories, Murray Hill, NJ, 1993.

[Holland, 1992] John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, second edition, 1992.

[K. E. Kinnear, 1994] Jr. K. E. Kinnear, editor. *Advances in Genetic Programming.* cambridge, MA: The MIT Press, 1994.

[Koza, 1992a] John R. Koza. *Dynamic, Genetic and Chaotic Programming*, chapter 10, pages 203–323. John Wiley and Sons, Inc., 1992.

[Koza, 1992b] John R. Koza. *Genetic Programming:on the programming of computers by means of natural selection.* A Bradford Book, The MIT Press, 1992.

[Koza, 1992c] John R. Koza. *Genetic Programming:on the programming of computers by means of natural selection*, chapter 19, pages 480–526. A Bradford Book, The MIT Press, 1992.

[Montana, 1994] David J. Montana. Strongly typed genetic programming. Technical Report BBN 7866, Bolt Beranek and Newman, Inc.,Cambridge, MA 02138, 1994.

[Rosca and Ballard, 1994] Justinian Rosca and Dana Ballard. Learning by adapting representations in genetic programming. In *The IEEE Conference on Evolutionary Computation*, pages 407–412. Morgan Kaufmann Pub., June 1994.

[W.A. Barrett and J.D.Couch, 1986] D.A.Gustafson W.A. Barrett, R.M. Bates and J.D.Couch. *Compiler Construction: Theory and Practice.* Science Research Assoc, Inc., 1986.

[Whigham, 1994] P.A. Whigham. An empirical investigation of the genetic programming paradigm. Technical Report CS10/94, University College, University of New South Wales, 1994.

[Whigham, 1995] P.A. Whigham. Operations that modify a grammar to direct the search space of a genetic program. Technical Report CS2/95, University College, University of New South Wales, 1995.