Inductive Bias and Genetic Programming

P.A.Whigham

Department of Computer Science,
University College, University of New South Wales
Australian Defence Force Academy
Canberra ACT 2600 AUSTRALIA
Email: paw@csadfa.cs.adfa.oz.au

Abstract

Many engineering problems may be described as a search for one near optimal description amongst many possibilities, given certain constraints. Search techniques, such as genetic programming, seem appropriate to represent many problems. This paper describes a grammatically based learning technique, based upon the genetic programming paradigm, that allows declarative biasing and modifies the bias as the evolution proceeds. The use of bias allows complex problems to be represented and searched efficiently.

1 Introduction

The Genetic Programming paradigm (GP) is a form of adaptive learning [4]. The technique is based upon the genetic algorithm (GA), [2], which exploits the process of natural selection based on a fitness measure to breed a population 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 [6].

1.1 Closure and GP

GP defined two main constraints upon the genetic operators *crossover* and *mutation*; closure and sufficiency. The requirement of *closure* made many program structures difficult to express. Closure, as defined by Koza [4], 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.

Closure has been extended using strongly-typed genetic programs [5], ensuring that each function and argument is matched for type when performing crossover.

This paper describes a different approach to the closure problem, where a context-free grammar (CFG) is used to define the legal constructions from which a program is composed. [1] By storing the parse tree that has been used to generate each program, the initial structure defined by the grammar may be maintained during crossover and mutation [9].

Additionally, we describe operations that modify the initial grammar to allow further biasing of generated programs. This learnt bias represents a probabilistic approach to describing the search space of a program.

1.2 Relevance to Engineering

The engineering community has shown interest in genetic approaches to learning for some time, as they are robust mechanisms for describing non-linear and complex problems. Engineering problems are often described as a search for some near-optimal solution, given a set of possibilities, and some constraints over their form. For example, Roston [1] showed a genetic approach to designing structural configurations, using a formal grammar, improved the forms of solutions and allowed constraints to be represented as part of the language. The ability to constrain a search space through explicit and learnt bias will allow more complex problems to be represented and solved.

2 Language Bias and Learning

Given a language to express the solution to a problem, *bias* is the set of all factors that influence the form of each program (hypothesis) [7]. Bias may describe both the possible types of programs that may be generated,

[1] An introduction to context-free grammars may be found in [8].

and the way in which these structures can be modified. Two important components of bias are *strength* and *correctness*:

- Bias may be categorised as strong or weak. A strong bias focuses upon a small number of possible hypothesis, whereas a weak bias allows a relatively large number of possible solutions.
- Correctness describes how well a bias is suited to a problem. If a bias is *not correct* the solution to the problem *cannot be expressed*. Conversely, a *correct* bias will allow the program to express a solution to the problem. Hence there is competition between limiting the search space without discounting meaningful solutions.

We argue that bias in evolutionary learning systems, such as genetic programming, is an essential component if we are to achieve consistent and understandable results. In fact, for many complex problems, bias is necessary to achieve any useful progress.

Bias may be described in two forms:

- Declarative bias, which defines the possible forms of initial and subsequent program structures, and
- Learnt bias, which modifies the way in which possible forms are generated and introduced into the competing population of functional programs.

3 GP and Grammars

Each member of the GP population is represented by a tree which defines the parse tree created from a CFG [2]. A CFG $(N, \sum, P, S)$ will create programs of the form:

$$S \stackrel{+}{\Rightarrow} \alpha$$

where α is a composition of symbols from $\sum$. The grammar productions P represent the space of possible programs that may be created during the evolution.

3.1 Program Evaluation

The functions are evaluated in a left-most fashion, based on the derivation tree for each program. In general we define a function f in the grammar as: $A \Rightarrow f\ \alpha\ \beta \ldots \mid$ which defines f as $f(\alpha, \beta, \ldots)$. The position in the grammar of any function determines the arguments that are available to that function, with a binding to the right.

3.2 Creating the Initial Population

The following steps create the initial program population:

1. Label each production $A \Rightarrow \alpha$ based on the minimal depth of tree to a terminal symbol. We define $A \Rightarrow terminal$ as having a depth of 1.
2. for the range of depths $D = i..j$ do
   (a) Select the start symbol S and label it as the current nonterminal
   (b) Select (using a normal distribution) a production $B \Rightarrow \alpha$ from the current nonterminal that does not exceed D
   (c) For each nonterminal β in α, label β as the current nonterminal, and repeat steps (b) and (c).

To ensure a measure of diversity all initial programs are required to have different parse trees. This creates a good selection of differently shaped programs, all of which satisfy the structural constraints of the grammar.

3.3 Selection of Individual Programs

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

3.4 Crossover using a CFG

All terminals have at least one nonterminal above them in the program tree (at the very least the start symbol S), so without loss of generality we 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 nonterminals are selected at each crossover site. To limit the possible size of all programs, the parameter *MAX-TREE-DEPTH* is used is indicate the deepest tree that may exist in the population.

The crossover algorithm is as follows:

1. Select two programs P and Q from the population based on fitness
2. Randomly select a non-terminal from P
3. If no non-terminal matches in Q, fail, otherwise

[2] Grammars traditionally use the definitions *terminal* and *nonterminal* to represent the atomic tokens and left hand side of productions,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 discussion GP constructs we will use the words GPterminals and GPnonterminals.

4. Randomly select a matching non-terminal from Q
5. Swap the subtrees based on 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.

3.5 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.

4 The 6-Multiplexer

We have selected the Boolean 6-multiplexer as the example to demonstrate bias. The problem has been previously studied by Koza [4], who used the *probability of success* for a run of the genetic program as a measure of the problem's complexity and the efficiency of the genetic operators. The GP 6-Multiplexer has the following structure:

Table 1: 6-Multiplexer GP Representation

GPTerminals	a0 a1 d0 d1 d2 d3
GPNonterminals	and(2) or(2) not(1) if(3)

The 6-Multiplexer is a simple selection mechanism. The address lines *a0* and *a1* are used to select between the four data lines *d0* ... *d3*. The result of the multiplexer is the data line value corresponding to the selection of the address lines. The complete 64 combinations of the address and data lines were generated as the test data. A *raw fitness* represented the error count for each of the 64 cases, hence the *raw fitness* approached 0 for fit program individuals.

A grammar (one of many possibilities) that allows the functional structures of Table 1 is:

$S \Rightarrow B$
$B \Rightarrow and\ B\ B \mid or\ B\ B \mid not\ B$
$B \Rightarrow if\ B\ B\ B \mid T$
$T \Rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

We will refer to the grammatically-based genetic program as CFG-GP, to distinguish it from the normal genetic program. We initially use the previous grammar as the structural bias for the multiplexer.

Table 2: CFG-GP 6-Multiplexer Setup

GENERATIONS	50
POPULATION SIZE	500
CROSSOVER	90%
REPRODUCTION	10%
MAX-TREE-DEPTH	8
FITNESS MEASURE	64 possible cases
PROGRAM SELECTION	Proportionate

The CFG-GP was applied 100 times based on table 2, with the resulting *probability of success* determined as 29%. This represents the base level using the initial grammar.

5 Biasing the Multiplexer

To examine the effect of declarative bias 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 give a higher probability of success. The first grammar biased the solution to use if as the first function in the program.

$S \Rightarrow IF$
$IF \Rightarrow if\ B\ B\ B$
$B \Rightarrow and\ B\ B \mid or\ B\ B \mid not\ B$
$B \Rightarrow if\ B\ B\ B \mid T$
$T \Rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

The probability of success increased to 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:

$S \Rightarrow IF$
$IF \Rightarrow if\ a1\ B\ B$
$B \Rightarrow and\ B\ B \mid or\ B\ B \mid not\ B$
$B \Rightarrow if\ B\ B\ B \mid T$
$T \Rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

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

$S \Rightarrow IF$
$IF \Rightarrow if\ a1\ IFAPART\ B$
$IFAPART \Rightarrow if\ a0\ B\ B$
$B \Rightarrow and\ B\ B \mid or\ B\ B \mid not\ B$
$B \Rightarrow if\ B\ B\ B \mid T$
$T \Rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3$

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

5.1 Discussion of Bias

The previous results show that using a bias in the grammar of possible programs supports the probability of finding an acceptable solution. This leads to the argument that as the problem space becomes large we

should use a language bias to restrict the format of possible programs. The grammar has allowed the user to impose a search bias and language bias to the forms of program that are created.

6 Learning Bias

Adapting the representation language as the evolution of a solution proceeds [3] represents a form of learnt bias. 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. We will focus on modifying the grammar as a *weak form* of changing representation.

Using the language presented as a CFG, 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

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:

- recognise 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 Change

Our implementation of the grammar modifications used the assumption that after each generation our description of what is *potentially* a useful production in the language could be extended.

6.3 Identifying Individuals

Identifying which programs (or parts of programs) should be identified as useful has been studied by Rosca and Ballard [3], 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. We proceed with the assumption that a fit program will contain (some) relatively fit productions, which may be exploited in changes to the grammar.

We 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 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.

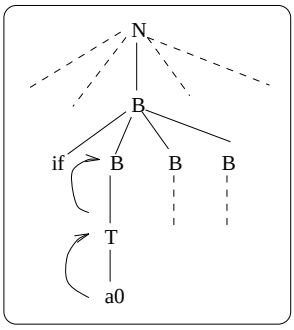

Fig. 1. Propagating a terminal up the tree

The simpliest refinement is to propagate a terminal up the program tree to the next level of non-terminals. As shown in figure 1, 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.

6.5 Production Selection

Given a program with the associated parse tree that created it, we have a tree structure

with tips as terminals, and internal nodes as nonterminals. The selection of a production to refine the grammar must use *at least* one terminal, so the problem may be rephrased as saying "which terminal should be chosen from the program to be used for refinement?".

If we consider the productions as a tree, we proceed down the tree by reapplication of nonterminals. The deepest terminals that are chosen will have the least influence on the overall structure of the program [3] . The terminal at the deepest point in the production tree may be considered at the *most general point*, and therefore influence the overall form of the program the least. Hence we chose the terminal at this most general point as the location to refine.

6.6 Incorporating Productions into the CFG

The previous grammar for the multiplexer may select a production from B using if and $a0$ to extend the grammar. This construction is incorporated as:

$$B \Rightarrow if\ a0\ B\ B$$

This represents a form of *weak bias*, as we do not constrain the possible forms of solution that may be generated. However, we begin to shape the probability of certain solutions being created, as represented by the grammar.

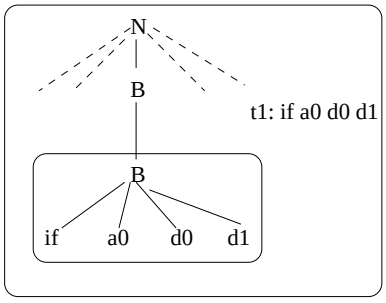

Fig. 2. Encapsulating a production

A second situation of propagating the terminal symbols occurs when all siblings of a propagated terminal are also terminal symbols. For example, a production $B \Rightarrow if\ a0\ d0\ d1$ may be found useful, as shown in figure 2. We wish to *encapsulate* the terminals as one functional symbol. This is performed by creating a new terminal, say $t1$, that binds all of these terminal symbols together. The terminal $t1$ is then included in the set of possible terminals, and in the production from B, as follows:

$$B \Rightarrow if\ B\ B\ B \mid t1 \mid or\ B\ B \mid \ldots$$

6.7 Changing the Population

The grammatical changes are introduced back into the population by a new operator, called REPLACEMENT. This represents the number of new programs to be created each generation from the modified grammar. Note that MUTATION may also introduce changes into the population.

6.8 Selection of Productions

The generation of program individuals, mutation and replacement all use the grammar to direct their operation. The initial grammar is defined as having all productions with equal probability of selection. We label each production $B \Rightarrow \alpha$ with an initial count of one, representing a *fitness* for each production. We now consider the case of a new production being created from a refinement, such as

$$B \Rightarrow if\ a1\ B\ B$$

This production will be labelled with an initial fitness of one. If the same refinement is found at some later time, the production fitness will be incremented. When we now select a production from B, whilst doing mutation or replacement, this production will be chosen as a proportion of the fitness of all other productions from B. This approach ensures that as a useful production is repeatedly identified the probability of using this production in the population increases proportionally based on its fitness within the grammar.

6.9 Initial Results

We wish to first demonstrate that using a proportionate selection of productions is an effective mechanism for improving the refinement of grammars. We selected the REPLACEMENT operator (no MUTATION) to study the effect of fitness proportionate selection of productions and the initial viability of grammatical refinement.

The system used $REPLACEMENT = 50$ so that 10% of the population was recreated each generation. The *probability of success* for this arrangement, without refinement, was found to be 22%. The refinement operator was then introduced using

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

the same settings, and the *probability of success* was found to increase to 52%.

The introduction of a *fitness proportionate selection* for production selection further increased the success probability to 66%.

The following grammar was created by a *successful* run of the CFG-GP, after 41 generations.

$S \Rightarrow < 1 > B$
$B \Rightarrow < 7 > if\ a1\ d3\ B\ | < 4 > or\ d1\ B$
$B \Rightarrow < 3 > if\ a0\ d1\ d3\ | < 3 > if\ a0\ d1\ B$
$B \Rightarrow < 3 > if\ a0\ B\ B\ | < 2 > or\ a1\ B$
$B \Rightarrow < 1 > if\ a0\ d1\ d2\ | < 1 > if\ a1\ d1\ B$
$B \Rightarrow < 1 > if\ a1\ B\ B\ | < 1 > T$
$B \Rightarrow < 1 > and\ d1\ B\ | < 1 > if\ a0\ d1\ d0$
$B \Rightarrow < 1 > or\ d0\ B\ | < 1 > and\ d2\ B$
$B \Rightarrow < 1 > and\ a1\ B\ | < 1 > or\ B\ a1$
$B \Rightarrow < 1 > and\ B\ B\ | < 1 > or\ B\ B$
$B \Rightarrow < 1 > not\ B\ | < 1 > if\ B\ B\ B$
$T \Rightarrow < 1 > a0\ | < 1 > a1$
$T \Rightarrow < 1 > d0\ | < 1 > d1$
$T \Rightarrow < 1 > d2\ | < 1 > d3$

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.

7 Incremental Learning

At the completion of a CFG-GP run, using the previous grammatical operators, we would expect that the new grammar would represent a bias towards the form of the program 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 previous grammar that had been created whilst solving the 6-multiplexer. The *fitness* of each production was loaded as part of the initial grammar, as well as all productions that formed the complete grammar. This technique represents a form of incremental learning, in that information from a previous run of the program is passed onto a subsequent run, with an expected improvement in performance.

The *probability of success* using the previous 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.

8 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. The use of fitness proportionate selection for grammatical productions allowed an improvement in the performance of the refinment operator, and 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 systems.

9 Acknowledgements

The author is indebted to William Cohen for suggesting the use of grammars for bias which led to this work. Thanks must also go to Bob McKay for discussions and support.

References

1. G.P.Roston. *A Genetic Methodology for Configuration Design.* PhD thesis, Carnegie Mellon University, 1994.
2. John H. Holland. *Adaptation in Natural and Artificial Systems.* MIT Press, second edition, 1992.
3. Dana Ballard Justinian Rosca. Learning by adapting representations in genetic programming. In *The IEEE Conference on Evolutionary Computation*, pages 407–412. Morgan Kaufmann Pub., June 1994.
4. John R. Koza. *Genetic Programming:on the programming of computers by means of natural selection.* A Bradford Book, The MIT Press, 1992.
5. David J. Montana. Strongly typed genetic programming. Technical Report BBN 7866, Bolt Beranek and Newman, Inc.,Cambridge, MA 02138, 1994.
6. U. O'Reilly and F. Oppacher. An experimental perspective on genetic programming. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340. Elsevier Science Publishers B.V., 1992.
7. P.E.Utgoff. *Machine Learning of Inductive Bias.* Kluwer Academic Publishers, 1986.
8. D.A.Gustafson W.A. Barrett, R.M. Bates and J.D.Couch. *Compiler Construction: Theory and Practice.* Science Research Assoc, Inc., 1986.
9. P.A. Whigham. Context-free grammars and genetic programming. Technical Report CS20/94, University College, University of New South Wales, 1994.