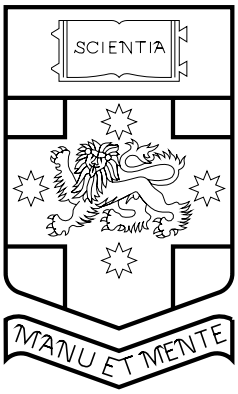
GRAMMATICAL BIAS FOR EVOLUTIONARY LEARNING



A THESIS

SUBMITTED TO THE SCHOOL OF COMPUTER SCIENCE

UNIVERSITY COLLEGE

UNIVERSITY OF NEW SOUTH WALES

AUSTRALIAN DEFENCE FORCE ACADEMY

FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

By

Peter Alexander Whigham

14 October 1996

Certificate of Originality

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgement is made in the text.

I also hereby declare that this thesis is written in accordance with the University's Policy with respect to the Use of Project Reports and Higher Degree Theses.

Peter A. Whigham

Acknowledgments

During the three years in which this work was developed I met many people who helped me in one way or another. I must start by thanking the CSIRO Division of Water Resources for allowing me leave to pursue this research and keeping my job available on my completion.

Supervising is not an easy task, and can be further complicated when the student is older and somewhat set in their ways. My most heartfelt thanks must go to Bob McKay, who managed to supervise me without creating any tension or frustration. Bob always seemed to know exactly when I should be left alone or given direction and was extremely patient and supportive. This work would not have been possible without his keen eye for detail, enthusiasm and intelligence. A good friend and mentor, Bob taught me how to do research and remain a human being. For this I am forever in his debt.

When this work commenced I had not intended to use an evolutionary framework for learning. However, once it was clear that the work was based on this field, Xin Yao immediately offered his help and became a second supervisor. Although Xin was constantly busy he always had time to stop and help, and towards the end of this thesis he became an invaluable support and critic.

The pragmatic attitude of Richard Davis ensured that this work would be in some way practical. Although the fields which were investigated diverged from Richard's expertise he always managed to give useful comments and to ensure that the work was comprehensible.

David Hoffman must be thanked for always having his door open when I wanted to talk about statistics. He had a great ability to give impromptu lessons on any mathematics that I required and has therefore been of immense help.

William Cohen visited ADFA during the Spring of 1994. His insights with using grammars led to the concept of using derivation trees to represent the population of programs. As such, he was instrumental in the development of many of the ideas that grew from this notion. I should, once again, thank Bob McKay at this stage because he emphasised the importance of this concept clearly to me.

Justinian Rosca must be thanked for his effort in supporting this work during the last year. He spent a generous amount of time correcting and improving the first paper I presented to the GP community, even though he had not met me nor was he familiar with my work. His enthusiasm gave me the encouragement to attend ML'95 and therefore meet many other people

who were helpful in this work.

Building a comprehensive bibliography is always a difficult task. William Langdon gave me a great deal of help with gathering the relevant work associated with this thesis. He sent me many papers and kept me informed of new work at a time when I most needed help.

Ken DeJong and Zbigniew Michalewicz (Mike) must be thanked for their effort at AI'95 with the postgraduate workshop. They provided some useful comments on the work and have been happy to help me with references and suggestions during the past six months.

I met John Koza during ML'95 and ICGA'95, where he offered some useful suggestions and was willing to promote my work at other workshops in my absences. Frank Francone has also been a strong supporter of my work and deserves special mention for the number of times he mentioned my ideas when dealing with the GP mailing list.

I visited Dortmund University during October, 1995, and spent several days with Peter Nordin and Wolfgang Banzhaf. Their enthusiasm and interest in new ideas helped me bring together many of the concepts that are presented in this thesis. They also forced me to drink several German beers as a cultural experiment. Thanks.

Peter Angeline must be thanked for sending me several relevant papers and for discussions during ICEC'95. His prompt responses to e-mail and suggestions for travel funding have helped in shrinking the distance between Australia and the rest of the world.

I remember a long conversation with Francis (Buster) Greene during ICGA'95. He introduced me to the field of Population Genetics and showed me the amount of overlap between GP and other fields of science. He must be thanked for opening up a number of avenues and approaches that I did not know existed.

I shared a room with Paul Darwen for nearly three years while creating this thesis. Paul was always willing to help with my work and had the uncanny ability to quote strange, historical facts that had no relevance to science. He made the insanity of the academic world somehow more stable, even though he was originally from Bowen.

Other people that I would like to mention are (in no particular order), Una-May O'Reilly, David Andre, David Mark, Andrew Frank, Dan Montello, Peter Laut and Jon Thomas. These people have all contributed in some way to this thesis and I thank them.

Finally, we come to the people who just make life possible. Trevor and Sandra Farley have been the best of friends for many years. Without their support and understanding this work could not have been completed. The humour, love and generosity that they give has helped me to maintain a positive attitude and a smile.

Last, and most important, I owe all of this work to my wife, Desley Anthony. She has somehow put up with the frustration of a poor student, the misdirection of a relived youth and the highs and lows that go with research. She has given me financial support, love and understanding beyond what I deserved. For all of the times when she could have turned away she has remained. I am eternally grateful for everything that she has done. Thankyou.

Abstract

A framework for declarative bias, based on the genetic programming paradigm(GP), is presented. The system, CFG-GP, encapsulates background knowledge, inductive bias and program structure. A context-free grammar is used to create a population of programs, represented by their corresponding derivation trees. These computer programs evolve using the principle of Darwinian selection. The grammar biases the form of language that is expressible and the inductive hypotheses that are generated. Using a formal grammar to define the space of legal statements allows a declarative language bias to be stated. The defined language may express knowledge in the form of program structure and incorporate explicit beliefs about the structure of possible solutions. Additionally, the form of the initial population of programs may be explicitly biased using a merit selection operation. This probabilistically biases particular statements generated from the grammar.

The program induction system, CFG-GP, represents search bias with three operators, namely *selective crossover*, *selective mutation* and *directed mutation*. Each of these operators allows a bias to be explicitly defined in terms of how programs are modified and how the search for a solution proceeds. Hence, both a search and language bias are declaratively represented in an evolutionary framework.

The use of a grammar to define language bias explicitly separates this bias from the learning system. Hence, the opportunity exists for the learning system to modify this bias as an additional strategy for learning. A general technique is described to modify the initial grammar while the evolution for a solution proceeds. Feedback between the evolving grammar and the population of programs is shown to improve the convergence of the learning system. The generalising properties of the learnt grammar are demonstrated by incrementally adapting a grammar for a class of problems.

A theoretical framework, based on the schema theorem for Genetic Algorithms(GA), is presented for CFG-GP. The formal structure of a grammar allows a clear and concise definition of a building block for a general program. The result is shown to be a generalisation of both fixed-length(GA) and variable-length(GP) representations within the one framework.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Where is the life we have lost in living ?
Where is the wisdom we have lost in knowledge ?
Where is the knowledge we have lost in information ?

T.S. Eliot, from *'The Rock'*

That smooth-fac'd gentleman, tickling Commodity,
Commodity, the bias of the world.

Shakespeare, from *'King John'*

1.1 Motivation

The recent growth of computer-based technologies has focussed the need to represent, store, analyse, display and manipulate information[1]. This is demonstrated by fields such as Geographic Information Systems, Database Systems, Expert Systems and Decision Support Systems. To comprehend the increasing volume and complexity of this information computer-based tools for analysis are required. The field of machine learning has developed as one approach to exploiting these information-rich domains.

There is something quite persuasive about a machine which learns from its environment. Rather than having to be explicitly programmed, it would be desirable for a machine to learn how to construct relationships(objects) based on a representation of the problem. These objects can be quite complex. For example, they may represent a programming language, or the structure of a network or a set of heuristic rules. Machine learning is fundamentally about structure - the structure of these objects and their transformations.

Some learning strategies maintain only one object at any time. Transforming one object into another is based on the representation of the problem. Because the number of possible

[1]There is a great deal of evidence that the generation and collection of such information will continue to increase. Technologies such as remote sensing and the so called *information superhighway* are evidence of this trend.

transformations from one object to another is usually too large to enumerate, forms of bias are introduced to make the transformation possible. These forms of bias are often called heuristics.

A second method for learning uses many objects at the one time. A population of objects are maintained, each with a certain fitness based on examples of the problem. Transformations between members of a population are often described in a genetic framework, since objects exchange and randomly modify components of their representation.

Formal grammars may be used to represent many different languages. Hence, objects that are based on a grammar should be able to express many different languages. Transforming the objects represented by a grammar is difficult. Each new learning task may require a new grammar and, therefore, a new language to be searched. For each new grammar there is the difficulty of not knowing how to manipulate the objects representing this language. Unfortunately, there is no general bias that will be satisfactory for all grammars.

Learning may be considered as a search for one particular object from a large set of possible objects. A learning system may be biased by representing knowledge about the problem, explicitly given by the user of the system. The representation of knowledge, separate from the learning system, is referred to as a declarative form of bias. This type of representation is useful since it gives a framework for unambiguously stating the beliefs of the user in a transparent manner. Such beliefs or partial knowledge, supplied to assist learning, are often available for problems involving natural resource domains, geological processes, biological, chemical and physical descriptions.

This dissertation describes a learning system that allows functional descriptions to be developed with declarative and learnt bias. A formal grammar is used to define the language of possible hypotheses and both the declarative and learnt bias for the system. Using the framework of evolutionary algorithms to perform the search creates a population-based system with formal characteristics and explicit biasing. This may be applied to a diverse range of problems and has formal properties that may be studied independently of any particular problem.

1.2 Formal Languages

A language is specified by an *alphabet*, a structure or *syntax* and a meaning or *semantics*. The alphabet is a finite set of symbols that compose the elements of the language. The syntax defines how these symbols may be structured to give sentences in the language. The semantics define the meaning of these sentences.

The possible sentences of a language may be finite or infinite. When the number of sentences is infinite a method is required to define how legal sentences are create with a given syntax and alphabet. The set of rules that are used to define this structure is called a *grammar*.

A grammar may be used in two ways; as a generator and as a recogniser of sentences. This dissertation will develop a system that employs a grammar to *generate* a population of programs defined by some language and to *maintain* the structure of the language during the search for

new sentences.

The class of context-free grammars[8] has been selected to define the possible sentences which are generated during the search for a functional program with certain properties. The expressive power and simplicity of context-free grammars make them suitable to represent many diverse structures in a clear and understandable manner.

1.3 Evolutionary Computation

The field of evolutionary computation techniques has been widely studied since the 1960's. This form of search technique is characterised by the use of a population of objects that compete to perform some specified task. Using biological analogies, the population of possible solutions are modified in two main ways.

- *Mutation* of an individual, normally causing a small change in the individual's representation.
- *Mating* between two or more individuals, thereby mixing the genetic material composing elements of the population.

The field of Genetic Algorithms [28], Genetic Programming [44], Simulated Annealing[38], and Evolutionary Strategies[67] have all developed with this basic concept - a population may be used to search a space of possible representations. The basic framework for evolutionary learning may be expressed as follows.

1. Generate an initial *random* population $P(0)$. Set the generation count $t = 0$.
2. Evaluate $P(t)$ creating a fitness or ranking for each population member.
3. **Repeat**
   (a) Select individuals from $P(t)$ based on their relative fitness or ranking.
   (b) Apply Genetic Operators to these individuals creating $P(t+1)$.
   (c) $t = t + 1$.
   (d) Evaluate $P(t)$ creating a fitness or ranking for each population member.

   Until *Termination Criterion* satisfied.
4. Select one or more members from $P(t)$ to represent the solution.

The representation of the population $P(t)$, the evaluation of these individuals and the genetic operators that modify these individuals determine the domain of applicability. For example, Holland's genetic algorithm[28] uses a fixed-length (typically binary) representation whereas genetic programming[44] allows functional descriptions to be represented in a tree-structured manner. The form of representation determines the type of genetic operators that may be applied.

1.4 Bias and Learning

Bias may be defined as the factors that influence a learning system to favour certain hypotheses or strategies. The application of a learning technique always involves some form of bias. Bias may be introduced in any of the following areas.

1. The problem representation.
2. The operators used to search the representation space.
3. The structural constraints of the representation.
4. The search constraints when manipulating the representation.
5. The criterion used to evaluate proposed solutions.

Although the value of declarative bias has been recognised for many years, there has been little work on applying explicit biasing techniques to evolutionary algorithms. Bias generally narrows the possible representations that a learning system may consider and therefore is an essential component for complex problems.

1.5 Statement of Thesis

This thesis describes a unified framework for learning computer programs with an explicit language bias and search bias. A population of random computer programs evolve using the principle of Darwinian selection. These programs are biased in the form of language and possible hypotheses they represent by a context-free grammar. Using a formal grammar allows declarative search operators to be defined that control how new hypotheses are created.

The use of a grammar to define language bias explicitly separates this bias from the learning system. Hence, the opportunity exists for the learning system to modify this bias as an additional strategy for learning. A general technique is described to modify the initial grammar while the evolution for a solution proceeds. Feedback between the evolving grammar and the population of programs is shown to improve the convergence of the learning system. The generalising properties of the learnt grammar are demonstrated by incrementally adapting a grammar to unseen members from a class of problems.

A theoretical framework, based on the schema theorem for Genetic Algorithms, is presented. The formal structure of a grammar allows a clear and concise definition of a building block for a general program. The result is shown to be a generalisation of both fixed-length(GA) and variable-length(GP) representations within the one framework.

1.6 Outline of Dissertation

Chapter 2 is a survey of related work with bias, evolutionary algorithms and formal analysis methods. This chapter serves as the basis for arguing the need for bias in learning and the current use of bias in evolutionary algorithms.

An approach to declarative bias with genetic programming, in a unified framework, is described in Chapter 3. The learning system, CFG-GP, uses a context-free grammar to define the language bias and structure for the problem. The use of a grammar also allows typing and closure information to be automatically maintained. Based on the derivation trees representing each program, the genetic operators of *selective crossover*, *selective mutation* and *directed mutation* are described. These operators allow an explicit search bias to be declaratively defined.

Chapter 4 demonstrates the basic properties of the learning system on three examples. The first example is the simple boolean problem of the 6-multiplexer. This is used to demonstrate the basic concepts of the system. The second example demonstrates the use of directed mutation as a search bias. A recursive solution to the function, $member(x, y)$, is evolved using a restricted LISP-like language. The mutation is used to identify tautologies and repeated patterns. The final example involves predicting the locational density of an Australian marsupial. This problem requires spatial and attribute descriptions in the language. Knowledge about the likely forms of a useful theory are used to direct the resulting hypotheses generated by the learning system.

The use of a grammar to define explicit bias for the learning system is further explored in Chapter 5. A framework is described to allow the learning system to modify the language bias (i.e. the grammar) during the evolution of a solution. This results in new productions of the grammar being created, which modify the search space defined by the initial language. This chapter concludes with an example of incremental learning, where the learnt bias of a particular grammar is used to improve the learning performance on a similar, but different, problem.

The formal properties of context-free grammars are used to define a schema theorem for CFG-GP in Chapter 6. This definition is shown to be a generalisation of genetic algorithms for single-point crossover and single-point mutation for a particular fixed-length binary grammar. A different mapping, via a grammar, demonstrates that the definition is also a generalisation of the genetic programming paradigm.

Finally, Chapter 7 summarises the research, reviews the contributions and comments on future directions. A bibliography and index is given at the end of this dissertation.

Chapter 2

Related Work

This chapter describes previous work involving learning bias, evolutionary approaches to learning and formal methods for analysing the properties of learning systems. The Genetic Programming paradigm is discussed in detail because it forms the basis of the learning system described in Chapter 3. The issues of typing, program structure and inductive bias in Genetic Programming are highlighted to show the need for declarative biasing with evolutionary learning techniques.

2.1 Introduction

This thesis brings together concepts from several fields of machine learning. The following topics will form the focus of this chapter.

- Bias, heuristics and learning.
- Grammatical bias and learning.
- Learning inductive bias.
- Evolutionary computation techniques.
- Genetic Programming.
- Formal theories of learning using evolution.

The concept of machine-based learning has been studied with great interest for many years (see [41, 91] for introductory concepts). The field of machine learning may be broadly classified according to the types of representation used for a problem, the type of operators that modify this representation and the type of information presented to the system to achieve the learning task. This thesis will describe a system that attempts to learn a simple computer program which performs some specified task. The problem is presented as a series of specific examples from which the system must attempt to construct a general theory, represented as a computer

program. This approach is a classic example of inductive reasoning[50] and has been further categorised, in the field of machine learning, as *learning from example*. The use of a programming language to represent the solution to a problem should not be construed as suggesting that all programming tasks (or even a small subset) may be automated. Rather, the domain of computer-based languages has been selected due to their flexibility and formal characteristics.

2.2 Learning and Bias

> All our experiences in AI research have led us to believe that for automatic programming, the answer lies in *knowledge*, in adding a collection of expert rules which will guide code synthesis and transformation [48].
>
> Douglas Lenat, 1983

The prospect of building inductive systems which could create any *reasonable* generalisation, with limited knowledge, was set back in 1984 by the theoretical results of Valiant[74]. He discussed two areas of complexity involved with learning. Firstly, *computational complexity*, which described the amount of computation required to find a hypothesis that is consistent with a given set of observations. Secondly, *example complexity*, which defined the number of observations required to reach a specified level of certainty (confidence) that the induced hypothesis is likely to be correct. This work implied that the *example complexity*, based on the probability of inducing an approximately correct hypothesis, was proportional to the logarithm of the size of the hypothesis space. The implication of this work was that, to ensure a problem remained tractable, the hypothesis space should be restricted.

The restriction can occur in several ways. The language that is used to represent the hypothesis may be limited in its possible expressions, for example by allowing only conjunctive normal forms or by imposing a maximum length to any inductive statement. Additionally, if an ordering of the hypothesis space is possible then it may be appropriate to limit the next generalisation step that is formed from the current hypothesis. The use of heuristics, to guide the choice of inductive statement, can also be used to remove, or enforce, possible options and thus make the search for a good generalisation less difficult. These forms of restriction are intended to allow the learning system to construct a solution to the induction problem, within certain time and space restrictions. We refer to these restrictions as forms of bias.

Theoretical work on discovering extrema of cost functions, by Wolpert and Macready[92], has demonstrated that all learning systems will perform poorly over some problems. This result, termed the No Free Lunch Theorem, implies that to solve a particular function optimisation problem a search algorithm should be tailored to the salient features of the problem. This theorem can be viewed as another argument for the use of bias with a general learning system. For a generic learning system to perform well over a broad range of problems it must be able to incorporate knowledge about the problem domain.

2.2.1 The Concept of Bias

Bias is the set of all factors that influence the selection of a particular hypothesis (generalisation). There are three major kinds of bias, based on the description language and the learning algorithm.

1. **Selection Bias:** The learning system may be biased due to a partial ordering over the hypothesis space. This enables two or more equivalent hypothesis (in terms of their performance on the available examples) to be distinguished. An example would be to select the shorter of two descriptions, when both descriptions are consistent. This type of selection has been described as bias by Schaffer[66] in relation to decision tree induction and avoiding the overfitting of training data.

2. **Language Bias:** The learning system may be restricted in the possible hypotheses that can be constructed. Using a space that has been restricted in this way implies that not all meaningful hypotheses can be constructed. This is usually achieved by using a description language in which some concepts are not describable.

3. **Search Bias:** The method by which a learning system transforms one hypothesis into another may be viewed as a search for a suitable hypothesis. Search bias refers to the factors that control this transformation. For example, a genetic algorithm has a search bias based on the genetic operators, such as crossover and mutation, and their relationship to the encoding of the problem.

2.2.2 Early Learning Systems and Bias

One of the earliest learning systems to provide explicit bias was described by Michalski[50]. He developed the STAR methodology for inductive learning. The method used explicitly stated information to constrain the possible inductive hypothesis that could be created and to locate those hypothesis that were likely to be close to the generalised solution. Michalski stressed that a machine generated hypothesis should be comprehensible to human experts. To ensure this occurred he gave heuristic rules that limited the complexity of any statement that was inductively asserted.

The learning formalism, defined by Michalski, used an extension of predicate calculus, named annotated predicate calculus, to represent the language bias. The explicit knowledge, used to constrain the possible assertions as a language bias, was expressed in several forms.

- An annotation of the predicates, variables and functions which constrained the possible range and domains where each of these constructs could be applied.

- Constraints on the description space, based on the properties and relationships amongst descriptors. These included the interdependence among values, properties such as transitivity over relations and conditions that were known to be true between different relations.

Although these forms of knowledge constrained the possible assertions it was generally possible to construct many statements that were still equivalent in their performance over a set of examples. Selection bias was introduced by using a list of preference criteria. This selected an assertion based on factors such as length of description and relative importance of correctly satisfying each observation. Michalski stated that this selection bias gave control over the generalising/specialising properties of the inductive assertions.

The search for an inductive hypothesis was viewed as a state-based search. Initially, the possible states were the set of observations. Operators transformed these states by generalising and specialising the descriptors, guided by the user-supplied knowledge. The final hypothesis was based on satisfying the observations, explicit knowledge and maximising the performance criteria.

Michalski's work emphasised the importance of knowledge in guiding inductive assertions and demonstrated that, for complex problems, structure about the solution could often be stated *without knowing the final solution.*

More recently, Pazzani et al.[58] also argued for the utility of explicit knowledge with inductive learning. The system, named FOCL(First Order Combined Learner), defined several techniques for incorporating knowledge with a system that generated function-free Horn clause rules. FOCL was based on the concept formation system FOIL[59] which learns a set of Horn clauses from positive and negative examples. The language constructions for FOIL were initially given as a set of background predicates. However, there is no way to represent knowledge to constrain the search space of the learning system.

FOCL used several methods to guide how a particular literal was selected when attempting to extend the current body of a function-free clause. Single argument constraints for a predicate, representing typing information, allowed FOCL to restrict the possible literals that could be selected for an argument with a predicate. Inter-argument constraints were represented by allowing conditions to be expressed about the relationship between arguments of a predicate. This allowed conditions such as *variable uniqueness* and the *commutability of arguments* to be expressed. Initial rules, representing a partial theory, could be expressed as a starting point for the definition of the predicate being learnt. Additionally, predicates with user-supplied definitions could be incorporated into the language. Each of these techniques was demonstrated to reduce the search space for a particular problem and therefore increase the possible applications of the learning system[58].

2.2.3 Using a Formal Grammar as Bias

The concept learning system LEX[52], first described by T. Mitchell et al., used a grammar to represent a generalised description of possible forms of a solution. This system was designed to use heuristic knowledge to aid learning about the domain of symbolic integration. The grammar described the form of legal problem states (i.e. legal hypotheses) that could occur during learning.

LEX operated in the domain of symbolic integration, using a grammar to represent the applicable conditions that may cause one integral representation to be transformed into another. The grammar was used both as a set of mappings between states and to represent generalisations of a state that may be applied to new conditions. This representation combined heuristics (i.e. guiding principles that could be applied) with a declarative representation of the search space, defined by a grammar which represented legal transformations. The grammar, for symbolic integration, represented the legal forms of integral expressions. The *sentential forms* of the grammar represented legal generalisations of an expression within the language of mathematical expressions. The grammar was context-free, where the nonterminals represented classes of concepts (such as the trigonometric functions, or a mathematical expression) and the terminals of the language represented the functions and operators of the language (e.g. the integral sign, the trigonometric functions, unbound variables).

Additionally, heuristics could be expressed in a declarative format. These heuristics represented a bias towards trying certain types of transformations between the current mathematical expression and its next transformed state as a search bias.

The concepts used in LEX were extended by Utgoff [73], where the concept description language of LEX was modified during the learning phase. This program, STABB (shift to a better bias), had the ability to *modify its language bias*. Utgoff began with a language bias that restricted the hypothesis space, defined by an incomplete concept description language. This bias was described as a strong bias, since it overly constrained the possible representation language (i.e. it was likely that the initial language bias could not express the desired generalisation that was being sought). Thus, the problem of modifying this bias was simplified to allow only a *shift of bias within a formalism*, not the more difficult problem of shifting between formalisms. Further, the method considered only weakening this bias by extending the concept description language. This was achieved by associating a combination of terms from the description language so that the terms could be selected together, thereby extending the language bias. This method was named the *RTA Method* for shifting bias and consisted of three steps.

1. *Recommend*, using heuristics to guide selection, that a new concept description should be added to the language.

2. *Translate* the recommendations into a form that is represented by the formalism of the learning system.

3. *Assimilate* the newly formed components into the concept description language in such a way as to maintain the structure of the hypothesis space.

This shift in bias occurred automatically, thereby allowing the program to change its representation when a solution was not forthcoming. The change represented a modification of the grammar that was used to define legal sentences in the language. As shown in Figure 2.1,

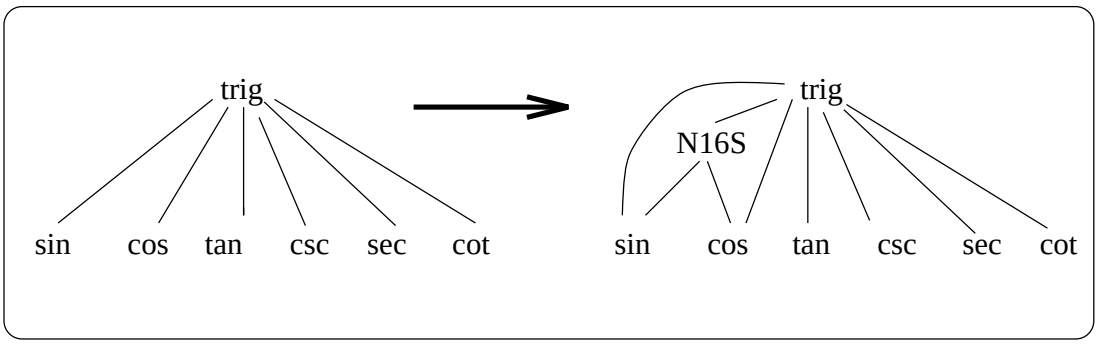


Figure 2.1: Bias before and after shift (taken from Utgoff [73] pg. 49).

STABB augments the initial grammar by creating new productions which represents a disjunction of concepts. This least disjunctive construction is used to indicate that one description is a subset of another.

The inductive system GRENDEL[10] has been developed within the domain of Inductive Logic Programming. GRENDEL used an *antecedent description language (ADG)* to generate a set of literals as a bias when learning first-order descriptions. GRENDEL used an extension of FOIL[59], a first order inductive learning system that used horn clauses as the description language. The types of knowledge, represented with FOCL[58], were extended within a single framework by using a formal representation language.

GRENDEL used an extended context-free grammar to represent the possible literals that could be expressed. This grammar differed from a CFG by the use of logical literals to represent strings in the generated language. This approach removed the constraint of a fixed alphabet from the grammar. Further, the ADG allowed conditions to be expressed for generated predicates. These conditions were used to generate many clauses from a single grammatical production. The semantics of Prolog were used to generate the possible predicate forms. The ADG was an overly-general theory, generating many possible statements *including the statement that represented the target hypothesis.* A sentence, derived from the ADG, became an operational theory that could be evaluated against the problem domain.

Cohen[10] demonstrated that antecedent grammars could be used to express many forms of background knowledge, including:

- Constraints over the use of predicates, such as argument typing and limitations on the use of certain literals.
- Knowledge of programming cliches which allowed constructs, known to be useful, to be included explicitly in the hypothesis.
- Theories of related concepts, which allowed concepts to be defined which could be used and expanded by the learning system. Hence, if a concept was known to be useful in explaining the type of problem that was being explored it could be explicitly included as part of the language that could be generated from the ADG.

- Incomplete theories, where some component of a theory is known but some components are yet to be expanded. The learning system could focus the learning towards finding definitions for these unknown predicate definitions.

- Theories which are syntactically close to the solution, where small changes to the structure of the initial hypothesis are required to generate a suitable solution. This type of approach may be viewed as searching for small perturbations in the language.

The grammatical statements generated horn clauses whose antecedents are sentences defined by the grammar. Because this language had well-defined semantics Cohen was also able to define a partial ordering over the hypothesis space. This allowed the navigation through a generalising/specialising hierarchy to be used when selecting an hypothesis.

This work showed that a grammar was useful in representing many forms of bias. This thesis will argue that, for many domains, a context-free grammar is powerful enough to express a suitable bias when applying the learning system, CFG-GP, described in Chapter 3. The need to express context-sensitive statements as bias will be considered in Chapter 7, where it is argued that an extension of CFG-GP to incorporate context-sensitive languages is relatively simple.

2.3 Evolutionary Computation

> Owing to this struggle for life, any variation, however slight and from whatever cause proceeding, if it be in any degree profitable to an individual of any species, in its infinitely complex relations to other organic beings and to external nature, will tend to the preservation of that individual, and will generally be inherited by its offspring [12].
>
> Charles Darwin, 1859

2.3.1 The Beginnings of Program Induction

The first computational attempts to use evolutionary techniques for learning appeared with the work of Friedberg[16]. The work is fascinating, not for the results that were achieved, but because of the insights that Friedberg presented, about both the concepts of machine learning and the idea of using evolution to drive a mechanical learning process. Friedberg believed that the language for expressing learning must be complex enough to allow representations that could not have been envisaged by the original designer of the system. He suggested that "..the universe of methods consist of all programs that can possibly be written for a given computer".

2.3.2 The Beginnings of Induction and Simulated Evolution

The work of Fogel et al.[15], published in 1966, considered the possibility of simulating evolution as a general technique for building robust learning systems. The structures that underwent evolution were represented as finite-state machines(FSM). The basic method for modifying any FSM was to allow random mutations to occur, which would alter both the connections between represented states and the states that could be reached. The FSM's were evaluated, based on their performance for the particular problem being solved. The FSM that achieved the highest fitness (i.e. demonstrates the best performance on the given task) was selected as the parent to create the next generation. This represented a selection bias. This work captured the basic concept of using an evolutionary technique to modify an initially random structure towards one which performed a specific task. The conclusions of this work were that the evolutionary approach to learning offered a systematic and mechanical procedure which could be applied to many induction problems that were originally thought to be the domain of only human experts.

2.3.3 The Genetic Algorithm

The concept of a genetic algorithm(GA) was first introduced by Holland[28] in 1975. This model of computation differed from Fogel et al. in the form of representation and the search operators that manipulated this representation. The GA used a chromosome-like encoding for the problem and introduced recombination operators that allowed new areas of the search space, represented by the chromosome encoding, to be explored. A balance between the *exploration* of a search space and the *exploitation* of fit partial solutions allowed the GA to be a robust method for many forms of optimisation and discovery. Holland called this method an *adaptive plan*, since the structures representing the population were gradually modified according to their previous performance.

The basic algorithm proposed by Holland, used a fitness proportional selection mechanism and "genetic" operators to create new individuals, which were represented as fixed-length binary strings. The genetic operators of crossover, mutation and inversion are shown in Figure 2.2. Single-point crossover creates new individuals by mixing the representations of two parent strings. The original Holland algorithm selected one parent, based on fitness, and the other parent at random. A random site was selected to cut each parent string and the tails of each parent were swapped, thereby creating two new strings. Mutation takes an individual string and probabilistically modifies any bit of the string. This represents a random mutation of the string and is generally considered as occurring with equal (usually very low) probability for each bit independently. Inversion is a mixing operator that occurs with one parent. Two sites are selected at random within the individual, and the bits between these sites are swapped, by replacing the last bit with the first, and so on. Each of these operators represent a search bias in terms of how the bit-strings are mixed and the relationship to the encoding represented in the strings.

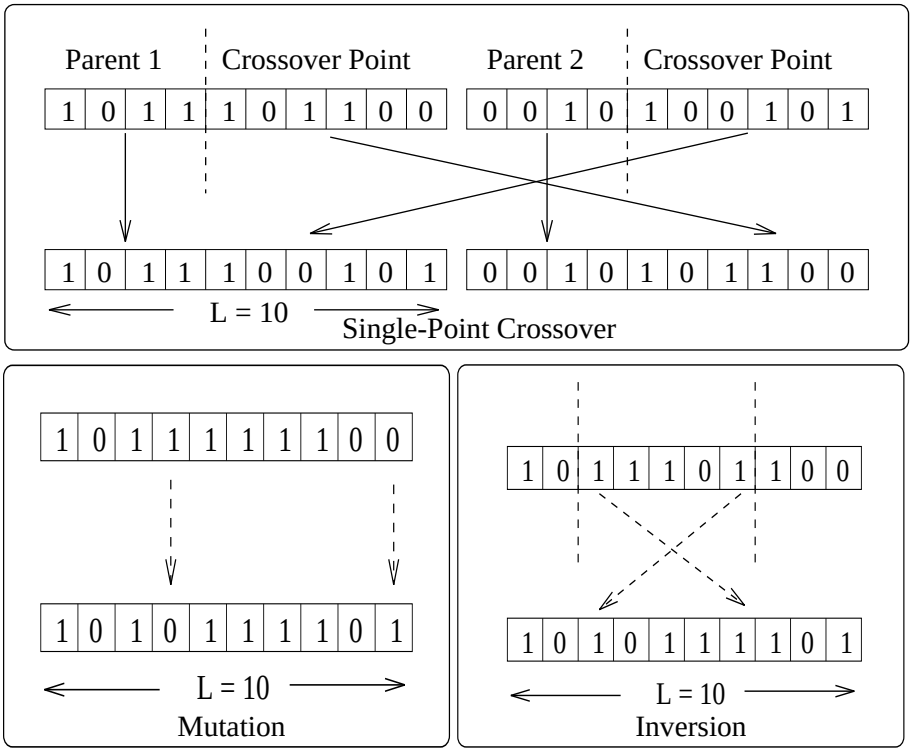


Figure 2.2: GA Operators Crossover, Mutation and Inversion.

The basic adaptive algorithm was defined as follows.

1. $t = 0$.
2. Initialise the population at random. Represent this as $P(t)$.
3. Evaluate the "fitness" of each member of $P(t)$.
4. $t = t + 1$.
5. Select a member, $A_i(t)$ from the population, $P(t)$, based on proportional fitness.
6. Apply crossover to $A_i(t)$ and $A_j(t)$ with probability p_c, where $A_j(t)$ is selected at random from $P(t)$. Select one of the resultants at random and designate it as $A^1(t)$.
7. Apply inversion to $A^1(t)$, with probability p_i, yielding $A^2(t)$.
8. Apply mutation to $A^2(t)$, with probability p_m, yielding $A^3(t)$.
9. Update $P(t)$ with $A^3(t)$, replacing a randomly selected member from $P(t)$.
10. Goto Step 3.

This basic plan demonstrates the concept of creating a new population from the previous population, using a fitness measure to determine selection of individuals and probabilistic genetic operators to create new individuals. The presented algorithm processes an initially random

population. Although there is no completion to this algorithm, some criterion for halting, such as a maximum number of generations or a certain space/time resource limit, is always defined. The fitness selection mechanism embodies the concept of "survival of the fittest" and represents a selection bias with the learning algorithm. The population members that are performing better at the given task are more likely to be selected as parents for the next generation. The individuals (i.e. binary strings) are generally decoded to some representation that allows an evaluation function to be applied to the representation. This evaluation function represents the fitness of the individual, which is then used as the selection mechanism for each generation (iteration).

Holland emphasised that this technique was a general search method, in that the genotypes (represented by the fixed-length encoding) are modified under the guidance of an evaluation function that probabilistically determines the state of the next generation of individuals in the population. The genetic algorithm has been applied successfully to many different problems, including optimisation[20, 69, 6, 5, 30], classification[37, 33], scheduling[9, 72, 40], control strategies[22, 25, 7], and neural networks[51, 90, 81]. The continued interest and success of this technique highlights the benefits of using a general, population-based method for searching complex representations in an efficient manner.

The concept of using an evolutionary search method for learning may be generalised to structures other than fixed-length binary strings. For example, messy genetic algorithms[19], rule-based (classifier) systems[28], Lisp S-expressions[42] and cellular encoding[24] represent alternative structures that have been used with evolutionary approaches.

The first comprehensive consideration of introducing problem specific knowledge into a GA framework appears to have been by Grefenstette[21], where several heuristic methods for biasing the initial population and the genetic operators of crossover and mutation are discussed. The domain of application explored was that of the Travelling Salesman Problem. Experiments were performed to show that seeding the initial population with structures that were known to be useful improved the performance of the GA. This type of bias was determined by creating the initial population with a heuristic algorithm which generated tours that used a greedy selection algorithm for determining the next city to visit from the current one. By selecting, at random, the start city the algorithm could create a large variety of possible tours.

Explicit search bias was introduced heuristically by modifying the crossover operator to avoid disrupting short tour lengths that were locally near-optimal. Although this approach was shown to promote the inheritance of short subtours and useful edges, the results did not demonstrate a statistically significant improvement in solving the travelling salesman problem. The mutation operator was used to perform a local search by reversing a randomly chosen subtour within the representation. The notion of a hillclimbing operation to supplement the search was also described. This was demonstrated to have a beneficial effect in refining solutions that were near-optimal.

The conclusions of this work were that bias must be carefully handled with GA's to avoid

premature convergence with the population. However, the positive results achieved with the travelling salesman problem demonstrated that forms of bias could be incorporated into an evolutionary search method with some success.

De Jong et al.[34] described a system, GABIL, that learnt and refined classification rules by interacting with an environment. This work was motivated by the fact that concept learning systems, with explicit bias, performed inconsistently. The genetic algorithm was used to adaptively change the bias for the system as the learning proceeded. The system represented knowledge as a set of disjunctive rules, where each rule condition could be a conjunction of variable conditions.

GABIL incorporated two forms of bias, based on the work of Michalski[50], which allowed generalisations of the current hypothesis to be performed. These operators were applied probabilistically in the same manner as other genetic search operators. GABIL was extended to allow these biasing operators to be adaptively selected. This was implemented by explicitly representing whether each bias would be applied to the particular population member. This representation was adapted along with the population member each generation. The conclusions of this work indicated that task-specific biasing improved the performance of a general learning system. Similar work which supports the use of genetic algorithms to process high-level concepts and incorporate task-specific knowledge has been presented by Janikow[29].

A population-based method for learning heuristics has been described by Wah et al.[78, 79]. The goal of this work was to create a general learning system, named TEACHER, that could automatically learn heuristics in knowledge-lean applications. There were two steps involved in this learning method. Firstly, the use of a genetics-based approach to generate and select heuristic methods that perform well over a set of training cases. Secondly, a technique for generalising selected heuristics to unseen test cases which were likely to give a similar level of performance to the training cases. The heurisitics were represented in a form that could be modified syntactically, such as a bit-string or as a set of symbols and numbers. Genetic operators were defined to modify these representations in a probabilistic manner. These heuristics represented a search bias for the learning system. The heuristics were separated from the learning method that was being applied to the particular problem. Hence, the emphasis was on coupling a method for searching for good heuristics to a general problem-solving technique.

This approach differed from Grefenstette[21] by having an explicit separation between the learning system and the heuristics used to drive the learning. Additionally, the work of Grefenstette used a static bias which could not evolve during the search for a solution. Similarly, GABIL[34] could not modify the forms of bias that were used during the search for a solution. Although the biasing techniques could be adaptively selected the methods used for each bias could not be modified during learning. The approach of Wah et al.[79] demonstrated that heuristics could be evolved to assist a general learning system and emphasised the importance of knowledge to aid learning.

Cramer[11] presented a representation for the generation of simple sequential computer

programs using GA's as the search mechanism. The approach specified two characteristics that were desirable for the induction of programs. Firstly, the encoding should allow standard genetic operators to be applied. Secondly, the representation should only be capable of producing well-formed programs during the evolution of a solution. These goals were achieved by using a list of integers to represent each program, where the list could be mapped unambiguously to a valid program for evaluation. This work was only partially successful due to the restriction of the integer representation and subsequent difficulty in defining a mapping to legal programs.

De Jong[32] discussed the possibility of using genetic algorithms to search program spaces. He believed that traditional genetic operators were inappropriate when applied to general programs for a number of reasons. Firstly, the complexity of the syntax and semantics of programming languages makes it difficult to define genetic operators that maintain these constraints. Secondly, the order dependencies, typical of procedural languages, imply that genetic operators that swap lines in a program will often render the program ineffective. The proposed solution to this dilemma was to use a program representation that did not emphasis order and complex syntax, such as a rule-based or production system.

Kitano[39] has presented a method for designing neural networks using a genetic algorithm with a graph grammatical encoding. The graph generation grammar is an extension of Lindenmayer's L-system[49] which was developed as a model and mathematical theory of plant development. The main contribution of this work was to demonstrate that evolutionary learning could be applied to a system which had a clear separation between the genotype (i.e. the production system) and the phenotype (i.e. the neural network). The work developed a family of matrices which represented the connectivity of a neural network. The main results of this work were that using a grammar generated more regular patterns in the networks and that the system converged more rapidly than a direct encoding system.

2.3.4 Genetic Programming

The field of program induction, using a tree-structured approach, was first clearly defined by Koza[44]. This field, named Genetic Programming(GP), evolved a solution in the form of a Lisp program using an evolutionary, population-based, search algorithm which extended the concepts of the fixed-length representations used with genetic algorithms. The structures were specified as a combination of functions (arity > 0) and terminals (0-arity functions) which combined to form Lisp programs. To apply GP to a specific problem, the following setup was required.

1. Define the function set, $F = \{f_1, f_2, .., f_n\}$, of functions, with arity > 0. Each function from F takes a specified number of arguments, defined as $b_1, b_2, .., b_n$.

2. Define the terminal set $T = \{t_1, t_2, .., t_m\}$ of 0-arity functions or constants.

3. Define the population size.

4. Define the maximum initial program size (depth of program tree) that may be created.
5. Define the maximum program depth which may be created in future generations.
6. Define the fitness measure used to evaluate each program.
7. Define the genetic operators that are used to modify programs each generation.
8. Define the termination criterion.

The Basic Flowchart for GP

The GP system uses an overall approach to creating and modifying structures similar to GA's and other evolutionary approaches. The following steps summarise the search procedure used with GP.

1. Create an initial population of programs, randomly generated as compositions of the function and terminal sets.
2. WHILE *termination criterion* not reached DO
   (a) Execute each program to obtain a performance (fitness) measure representing how well each program performs the designated task.
   (b) Use a fitness proportionate selection method to select programs for reproduction to the next generation.
   (c) Use probabilistic operators to combine and modify components of the selected programs.
3. The fittest program represents a (partial) solution to the problem.

Creating the Initial GP Population

Each initial program created with GP is based on the functions and terminals from $\{F \cup T\}$ that have been defined. This process commences by selecting a function, f_i, randomly from the set F. For each of the b_i arguments, this process is repeated where a random function or terminal may be selected to fill each argument position. If a terminal is selected the generation process is complete for this branch of the function. If a function is selected the generation process is recursively applied to each argument of this function. This process results in a random program, composed of functions and terminals from $\{F \cup T\}^+$. Some maximum depth of parse tree is specified to this procedure to limit the size of the initial programs. For example, using the function set $F = \{AND, OR, NOT\}$ and terminal set $T = \{a, b, c, d\}$, several programs that could be created for the initial population are shown in Figure 2.3. It is worth noting that this form of initialisation does not allow any explicit biasing over the structure (composition) of the generated programs and can be viewed as a weak language bias.

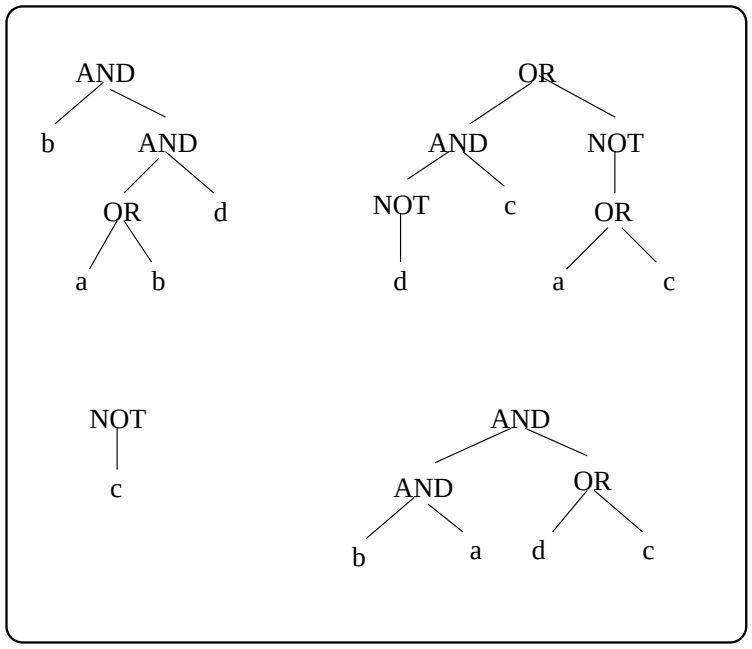


Figure 2.3: Initial GP programs, where $F = \{AND, OR, NOT\}$ and $T = \{a, b, c, d\}$.

Evaluating the fitness of each program

Each program is assigned a numerical fitness by evaluating the program against a set of test problems. These problems represent the environment that the program is attempting to learn. The fitness, for each program, is normalised so that a proportional measure may be used for selecting programs when forming the next generation. This method is described formally in Section 3.6 and is a selection bias with the learning system.

Genetic Operators defined for Lisp S-Expressions

There are two main genetic operators used with GP, namely reproduction and crossover. Reproduction selects a program, based on fitness, and copies this program identically to the next generation. Crossover selects two parent programs, based on fitness, and creates two children by swapping sub-trees between the parent programs. The crossover site within each parent is randomly selected, using a normal distribution, from any of the terminal or function sites. An example of crossover is shown in Figure 2.4. The possibility arises with crossover that a program is created which violates the maximum depth of program that has been specified by the user. If this occurs, the crossover is aborted and one of the parent programs (selected at random) is copied to the next generation. The rate of crossover is defined by specifying a percentage of the population that is to be created using this operator for each generation. This method is a search bias.

Koza defines two other, secondary, genetic operators that may be used with tree-structures. Mutation[44] operates on a single tree. A node within the tree is randomly selected and the subtree below this node is deleted. A new subtree is randomly generated from $\{F \cup T\}^+$ to

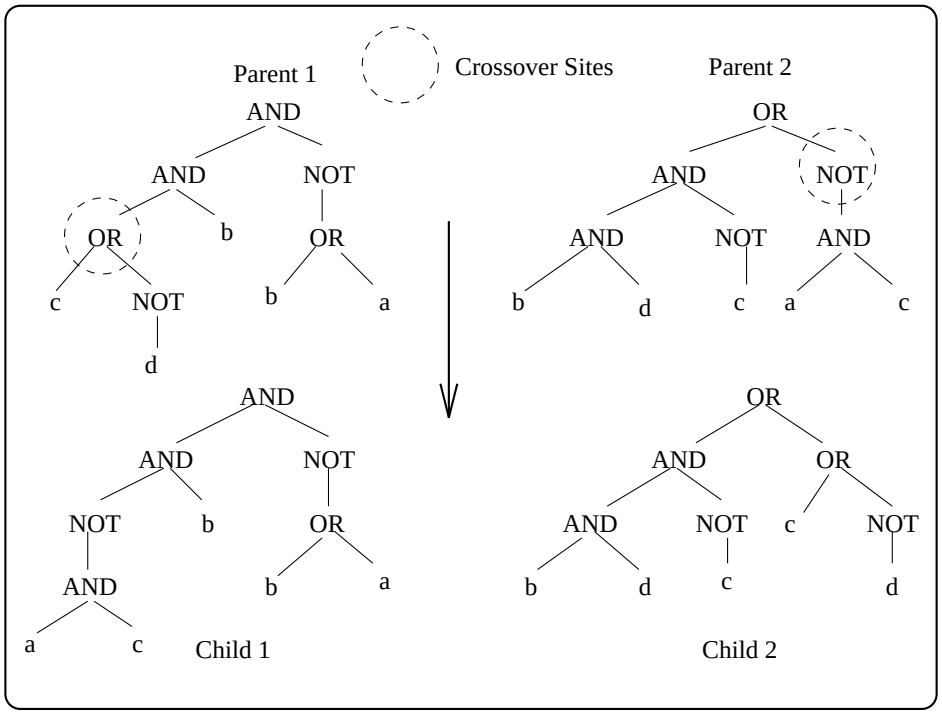


Figure 2.4: GP Crossover: Subtrees between two parents are swapped.

replace the deleted subtree. Permutation[44], based on GA inversion, permutes the order of the arguments to a randomly selected function within a program.

Termination Criterion for GP

The evolution of a program is halted when perfect fitness is achieved (i.e. the program satisfies all conditions represented by the evaluation environment) or some specified maximum number of generations have passed. The program that has the best fitness, after termination, is deemed to be the discovered solution.

Sufficiency and Closure

Koza[44] stated that two conditions must be satisfied to ensure GP could be applied to a specific problem. The concept of *sufficiency* stated that the functions and terminals (in combination) must be capable of representing a solution to the problem. The condition of *closure* stated that each "..function in the function set should be well defined for any combination of arguments that may be encountered"[43]. Thus, closure states the weakest of possible constraints on program form by forcing all functions and terminals to have essentially the same type. Although this condition is easily satisfied when boolean or mathematical functions are used, problems that involve data structures or have mixed types are difficult to express with this constraint.

Editing S-Expressions

Koza[44] suggested that editing the evolved LISP programs, to remove redundant statements, could assist the discovery of a solution. For example, a boolean expression could be simplified using DeMorgans Laws, or a mathematical expression could be reduced where multiplication by zero had occurred. These editing tasks were problem specific and could not be defined in a declarative manner. An alternative method which subsumes these concepts is described in Section 3.9.

Constrained Syntactic Structures

Many problems require some form of constrained structure in terms of the form of the resulting programs. This was noted by Koza[44], where an informal definition of *syntactic constraint* was given. He gave several examples where the program structure had to be constrained to produce valid programs.

The simplest example arose with special functions which could only occur as the root in a program tree. These functions essentially divided the program into a number of subprograms which were formed and modified in the original manner.

A more complex example was described involving the solution to a fourier series. Three syntactic constraints were specified, as follows.

1. The root of the tree must be the special function, &.
2. The only functions allowed immediately below an & are the trigonometric functions *xsin*, *xcos* and the special function, &.
3. The only functions allowed below the trigonometric functions are the mathematical functions (+,-,*,%) or a random, real-valued, constant.

These constraints were maintained by labelling each program tree node by a symbol number that defined the level in the program where the node may exist. These special symbols were used to ensure that crossover did not violate the stated syntactic constraints. Although this method worked adequately for the fourier series problem it appeared as an adhoc approach that would need to be customised for each new problem. Also, the method required an English explanation to define the structural constraints and could not be easily stated in a declarative manner explicitly included as part of the initial problem description.

To illustrate the declarative nature of a grammar, the previous syntactic constraints could be represented by the grammar, $G_{syn-con}$. This unambiguously describes, in a transparent manner, how the program is to be constructed and the arity of each function.

$$
\begin{aligned}
&G_{syn-con} = \\
&\quad \{S, \\
&\quad\ N = \{A, T\},
\end{aligned}
$$

$$
\begin{aligned}
&\sum = \{\&, xsin, xcos, +, -, *, \%, \mathbb{R}\}, \\
&P = \\
&\quad \{S \rightarrow \&\ A\ A \\
&\quad\ A \rightarrow \&\ A\ A \mid xsin\ T\ T \mid xcos\ T\ T \\
&\quad\ T \rightarrow +\ T\ T \mid -\ T\ T \mid *\ T\ T \mid \%\ T\ T \mid \mathbb{R} \\
&\quad \} \\
&\}
\end{aligned}
$$

Strongly Typed Genetic Programming (STGP)

The issue of syntactic constraints has been further investigated by Montana[54], who relaxed the closure requirement by specifying the required argument types for each function and the type of each function result and terminal. The initial, random programs were constructed so that the typing constraints were satisfied. This tended to limit the number of possible functions and terminals that could be placed in any particular argument position. The crossover operator was redefined so that the selected crossover sites between two parents had *matching syntactic types*. This ensured that swapping two subtrees could not produce a program that violated the syntactic constraints defined by the typing.

One important aspect of constraining the possible form of programs with typing is that the search space (representing the number of different programs that may be formed with some maximum depth of tree) is reduced. This increases the likelihood of discovering a program solution within some time and space constraint. For example, Haynes et al.[26] demonstrated that STGP outperformed standard GP for the problem of evolving cooperation strategies in a predator-prey environment. The conclusion of this work was that the improved performance of STGP was related to the reduced search space which occurred with the typed system. Additionally, the resulting programs created with STGP tended to be easier to understand.

The use of a typed language for constraining program forms with GP has been demonstrated to improve the performance of program induction. However typing alone cannot represent structural constraints beyond the simple level where one function or argument is constrained in its relationship to another function or argument.

Other Approaches to Constraining GP

Stefanski[70] proposed that GP should be extended to include declarative biasing techniques via the use of abstract syntax trees[68]. These trees could be used to constrain the possible forms of generated programs and to control the nature of crossover and mutation. Unfortunately, the implications of this approach do not appear to have been explored in any detail.

The author first presented work which used a context-free grammar to represent the language bias for genetic programming in November, 1994[82]. This work was inspired by the inherent difficulties associated with applying GP to a complex, spatial problem[83, 84]. The spatial

problem required a language that had several types of information, including the attribute values of locations and the spatial relationships between elements of the represented space. To overcome the closure requirements of GP, the spatial relationships were combined with the attribute and relational values, resulting in 189 terminals. This large number of terminals made it difficult for GP to perform acceptably without a large population. This result made it clear that some form of typing was required to allow GP to efficiently represent and search a language which involved some structure.

The work attempted to recreate a set of expert system rules that had generated the initial test data. Unfortunately, there was no clear way to enforce the structure of these rules in a declarative manner. Personal communications with William Cohen during September of 1994 directed the work towards maintaining the derivation trees associated with a grammar as a representation of the search space for GP. This lead to the technical report[82] which formed the starting point for the work described in this thesis. Extensions to this work implied that language bias[87] and the definition of structure[86] were important issues to be considered when applying GP to problems involving structured programs.

The work of Roston et al.[63, 64] demonstrated that a formal grammar may be used to specify constraints with GP. The authors explored the possibility of using a genetic search technique to create engineering designs. This methodology, entitled Genetic Design, was used to generate viable design alternatives using a formal grammar to specify artifact descriptions and representations. A context-free grammar was used to define the structure of the language. The grammar was used to generate the initial population of programs, however the grammar was discarded after this operation. The structure, represented explicitly by the grammar, was maintained by coupling the programs to an STGP system. Although this suffered from the same drawbacks as STGP, the use of a grammar to specify the initial system allowed the language bias to be introduced in a transparent manner. Additional bias was introduced by having constraints explicitly stated which represented the feasible design alternatives. For example, the design for a bridge must be capable of spanning the distance which the bridge is to traverse. Potential designs that failed this type of specification were never allowed to be generated in the population. This represented a search bias in the formulation of designs. However, this enforcement of feasibility was specific to each problem and was not represented in a declarative manner.

Recently there has been further interest in applying a formal grammar to control the language bias of programs generated within a GP framework. Wong et al.[93, 94] in his LOGENPRO system demonstrated that a logic grammar could be used to combine GP and ILP. Logic grammars are context sensitive and can therefore describe programming languages such as 'C', LISP and Prolog. The logic grammar provided a declarative description of valid program forms that could appear in the initial population. The genetic operators of crossover and mutation were applied to the derivation trees representing the parent programs. Using a grammar, the

language forms for ILP and GP could be represented and were shown to be a flexible representation of these learning systems. The emphasis of this work was to demonstrate that a single learning system could be defined that could be used to represent many different language forms. There was little mention of the importance of bias with this work. Additionally, the issue of ambiguity in language was presented as a problem for LOGENPRO. This occurred because the derivation trees, representing each program generated from the grammar, were discarded after the programs were created. When genetic operators were applied to programs they were parsed to recreate the derivation trees representing their formation. Additionally, LOGENPRO could not represent an explicit search bias with the genetic operators defined over the derivation trees. A site for crossover or mutation was selected at random which meant no method was available to specify where most of the search effort should be directed. The learning system, presented in Chapter 3, maintains the derivation trees which represent the population of programs. This removes the problem of dealing with ambiguous grammars, due to reparsing, and frees the user from the difficulty of determining whether a defined grammar is ambiguous.

Gruau and Whitley[24] have used a grammar tree to encode a cellular developmental process which generate a family of boolean neural networks for computing parity and symmetry. The grammar trees were modified using genetic operators that were based on the Genetic Programming paradigm[44], where the search space became the hyperspace of all possible labelled trees generated from the grammar.

Further support for using explicit bias with GP has been presented by Gruau[23], where a context-free grammar is used to shape the language bias of allowable programs. He argues strongly that bias should be viewed as a powerful tool for allowing GP to be applied more successfully. However, Gruau also removes the derivation trees for the population and must reconstruct them when the genetic operators are applied.

A method for evolving a hardware design for a particular problem has been described by Mizoguchi et al.[53]. This approach used a hardware description language, SFL(Structured Function description Language), which was defined as a set of production rules, each labelled with a unique category number. The genetic operator of crossover always swapped program components at the same category number. This ensured that the language, defined by the production rules, was not violated. This work also preserved the derivation trees that represented the population of programs and therefore closely parallels the work of this thesis. Several other genetic operators were defined to allow the representation to evolve from a simple structure to a more complex structure. Duplication operated by inserting a copy of a function block within a program, thereby introducing some redundancy to the program. Insertion was similar to duplication, however the function block was taken from a different program. Deletion was used to remove a function block from a program to allow a more compact representation to be created.

Extensions to this work[27] have described a technique for changing the rewriting system so that the hardware description language evolves. This was achieved by restricting some

production rules which removed possible derivations of the language. The intention of this process was to place more emphasis on particular program forms being generated from the grammar by changing the search bias. Few details were given for this work[27], suggesting that it was work in progress. The described approach seems overly restrictive in that some possible derivations from the language are removed. For example, the proposed transformation of an initial grammar was presented as follows[27].

$$\begin{array}{lcl} A \rightarrow D & & A \rightarrow D_1 \\ B \rightarrow D\ G & & B \rightarrow D_2\ G \\ D \rightarrow h\ i & \Longrightarrow & D_1 \rightarrow h\ i \\ D \rightarrow C\ F & & D_1 \rightarrow C\ F \\ & & D_2 \rightarrow h\ i \end{array}$$

The derivations that are possible from the nonterminal, B, have been restricted. A method that placed a probabilistic weighting to the possible derivations would seem more appropriate, since at the time of transforming the productions the best solution has not yet been discovered. Removing possible expressions may result in a language that is overly specialised to a particular suboptimal solution. The conclusion of Mizoguchi's work was that using a grammar allowed a structured development for a program to be achieved. The suggested grammar transforming operation was intended to allow a *programming style* to be adaptively created, however no work using this operation has been published to date.

Adapting GP Representations

Complex problems often become difficult to express using GP. Because there is no way to encapsulate useful components of a partial solution, it is difficult for the learning system to hierarchically compose a solution. Several approaches to this problem have previously been proposed.

The automatic definition of functions (ADF), first described by Koza[44], extends GP by allowing a solution to use subroutines that have been evolved along with the GP programs. In this approach, each individual in the population is defined by a fixed number of components. These represent the automatically defined functions, each with a predefined number of arguments, and the result returning branch of the program (typically the main body of the program). Each ADF is a complete subroutine, requiring a definition of the arguments, functions and terminals from which it is composed. A calling structure is imposed on multiple ADFs by not allowing a lower ADF to call a higher ADF. The program body is normally allowed to call any ADF with arguments defined from the terminal or function set. An example of the calling structure using ADFs is shown in Figure 2.5. Here, the program body has the function set, F, extended to include the two defined functions, ADF0 and ADF1. These functions are defined with one and three arguments, respectively. Note that ADF1 may call ADF0 and that each function has an internally defined set of functions, terminals and arguments. The functions are *not shared* between various population individuals (i.e. they are essentially internal to each defined

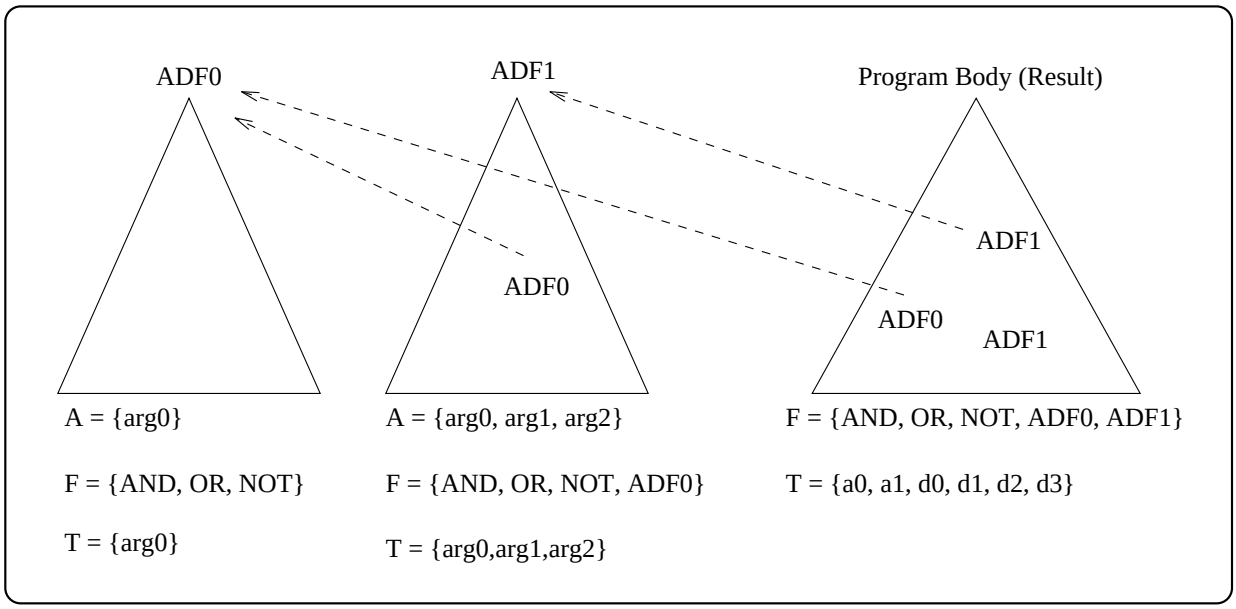


Figure 2.5: GP extended with Automatically Defined Functions.

program). The use of ADFs has been empirically shown to improve the performance of GP, however they rely on the user stating the form and structure of the functions before applying the system. One way to view this structure is that it is an extension of the constrained syntactic concepts introduced in Section 2.3.4. The success of ADFs imply that imposing structure on a program may improve the performance of the evolving solutions. This supports the notion that explicit language and search bias should be incorporated in the GP framework.

These concepts have been extended by Koza[45], where the restriction on having to initially define the structure of each ADF is relaxed. Six architecture-altering operations have been defined which allow a program to modify the form and makeup of the automatically defined functions that are used. Initial results have demonstrated that the approach is viable[46], however significant results using this approach have not yet been presented.

An alternative approach, named *module acquisition*, has been proposed by Angeline et al.[2]. This approach builds a library of modules which are functions created by selecting a subtree from the population and removing selected branches of the program. This creates a new function, with an argument list based on the branches that have been removed. The operation of *compression* is used to freeze genetic material as a module. A second operation, *expand*, takes a module and substitutes it back into a program which is using the module. This approach has the desirable feature of allowing useful components of a program to be held and used by many different programs *at the one time*.

The goal of discovering useful building blocks with GP has been studied successfully by Rosca et al.[61, 62]. Their approach, named adaptive representations(AR), uses a *bottom-up* method for discovering new subroutines that have been found to be useful. Candidate building blocks are discovered by considering the fitness of partial programs. The discovered building blocks

are defined as functions which are incorporated into the initial language. This is performed in a similar manner to descriptive generalisation[50], where a terminal is replaced by a variable which is used as an argument to the function. The evolutionary process is split into *epochs*, where an epoch is a sequence of generations where no attempt is made to discover new building blocks. An analysis of the population is made after a number of generations (i.e. after an epoch) and useful building blocks are defined as new functions. The population is then modified by retaining the fittest individuals and regenerating new individuals using the extended function set. This represents the beginning of a new epoch. The AR algorithm may be summarised as follows.

1. Discover useful building blocks by evaluating each blocks merit. Let the selected building blocks be represented by the set, B.

2. For each building block, $b \in B$:

   (a) Determine the terminal subset T_b used in b.

   (b) Create a new function f_b with arguments determined from T_b and body the block b.

   (c) Extend the GP function set with the new function f_b.

Rosca et al.[61] showed that AR could discover hierarchical representations while learning to solve a problem. In particular, problems where the solution contained symmetry or had repeated patterns were solved more quickly than standard GP approaches. The main explanation for this success was that the reusable nature of the discovered functions changed the search space of the problem, resulting in a modified language and search bias. The approach demonstrated that a shift in the representation language could transform a difficult problem into an easier one.

Zannoni et al.[95] has used a cultural algorithm to extract knowledge about the structure of individual programs generated within the GP framework. A cultural algorithm provides two distinct levels of evolution, where global information about the evolving population is represented as a set of *beliefs* which may influence the development of individuals. The system, named CAGP(Cultural Algorithm with Genetic Programming), used a GP system to represent the population with a set of beliefs represented as program segments. The beliefs were used to constrain the way programs were modified by the genetic search operators. Program segments that were found to be useful were protected from disruption when crossover or mutation was applied. This allowed the development of modules that could be considered as building blocks for a solution. The system was demonstrated to improve the rate and quality of the evolved solution, compared with a standard GP approach, for the domain of quartic polynomial symbolic regression. The conclusions of this work were that it is possible to extract information from individuals of a population that may allow the population, as a whole, to be usefully directed to a solution.

The use of adaptive techniques for determining crossover positions with GP has been demonstrated by Angeline[1]. The operator adapts values that alter the probability of crossover being performed at some particular site within a program tree. This is achieved by maintaining a parameter tree, for each program tree, which has the same structure as the program. However, at each node in the parameter tree a value is associated that represents the probability of crossover being performed at that node. These values are adaptively modified using a gaussian random noise after each crossover operation. This work demonstrates that an adaptive approach to crossover location performs at least as well as standard crossover and represents an adaptive approach to modifying the search bias for GP. The importance of this work is that it demonstrates that changing the search bias may influence the effectiveness of an evolutionary computation. In particular, if the search bias may be adaptively changed by the learning system then it is more likely to be a robust approach to learning for many different domains.

2.3.5 Formal Theories of Evolutionary Computation

John Holland first proposed the schema theorem for genetic algorithms[28], which described the effect of reproduction, single-point crossover and mutation on a population of fixed-length strings from one generation to the next. A detailed description of the schema theorem is given in Appendix C, which includes the terminology required for Chapter 6. The importance of the schema theorem lies in its description of how a GA performs the role of search and discovery. The theorem shows that schemata (building blocks) which are contained in individuals with above-average fitness will propagate exponentially in successive generations. This demonstrates the role of building blocks with evolutionary search.

A second value of the schema theorem is that it explains how implicit parallelism occurs within the population-based search of a GA. This is easily understood in terms of the number of schemata (see Appendix C for a definition) that are processed each generation within the population. For a population of size L, the number of schemata represented implicitly is $O(L^3)$. Although the evaluation of each schema is noisy, the overall statistical influence of each population member allows highly fit schemata to propagate. The implication of this theorem is that a suitable encoding scheme will improve the performance of the GA. The notion of epistasis[1] implies that a good representation will allow the search operators to explore many combinations, whilst maintaining the structure of useful components of fit individuals. This encoding problem is further emphasised with GP, since it is difficult to define subtree structures that should be closely associated to avoid disruption due to crossover. The use of ADFs and modules have been one approach to try and enforce this encoding. This further supports the concept of using explicit bias with GP and to allow structure to be defined in a transparent manner that may be modified for each new problem.

[1]Epistasis is most easily understood in terms of a programming language. For a language such as Pascal, Fortran or 'C', changing one line in a program may entirely alter the behaviour of the program. Thus, these languages have high epistasis. In the field of population genetics[14] this concept is described as linkage disequilibrium.

Other approaches to describing the behaviour of genetic algorithms have been proposed. For example, Rudolph[65] analysed the convergence properties of the GA applied to static optimisation problems. The conclusion of this study was that, for a GA to guarantee convergence to a global optimum, the best solution that has currently been discovered must be maintained in the population. Vose has demonstrated that the schema theorem may be presented in several different frameworks. One approach defines a schema as a predicate[75], which allows the functioning of a GA to be described as a constrained random walk. In an alternative approach, Vose[76] models GAs as a dynamic system in a high dimensional Euclidean space. The population of fixed-length strings is then modeled by a vector which may be expressed as a matrix. This work was extended by Battle et al.[4] which generalised the notion of schemata to cosets of subspaces generated by the columns of an invertible matrix. Each of these works offer further support for the processing of schemata as defined originally by Holland[28] and extend the understanding of schema processing for fixed-length strings.

A schema theorem for Genetic Programming has been developed by O'Reilly et al.[57]. This carefully followed the original schema theorem of Holland for GA's by developing the concept of schemata, defining length and schema order for tree-structured LISP programs. The lack of a formal structure with GP meant that it was difficult to define the concept of a schema, based on the notion of similarity between subtrees. The formal definition of schemata given by O'Reilly is stated in Section 6.8. This definition extends the fixed-length description of binary strings to tree-structured LISP S-expressions. This was achieved by defining the concept of a **tree fragment**, using a *wildcard* that matched any subtree (i.e. lisp expression). The fragment was defined as a tree that has at least one leaf that is a wildcard. The wildcard corresponded to an incomplete S-expression. The entire fragment also had a wildcard *at its root position* to represent the fact that the fragment could be fully embedded in a tree. This led to a definition for tree-structured schema which describes an unordered collection of both completely defined S-expressions and incompletely defined S-expressions. These incomplete S-expressions were represented as fragments.

The difficulty in defining a schema for arbitrary S-expressions was further complicated when attempting to define a schema theorem for GP. O'Reilly had to introduce several functions that defined how a schema was instantiated in terms of both the fixed- and variable-length components of a subtree.

Although the work successfully defined a schema theorem, equivalent in form to that proposed for a GA, the definition was complicated and difficult to follow. The goal of this work was to theoretically analyse whether a hierarchical process was evident in the way GP developed a solution. The results indicated that *no building block hypothesis* could be supported without untenable assumptions. Given that it has been empirically demonstrated that the use of ADF's improve convergence to a solution and that AR techniques allow the automatic discovery of useful building blocks the main conclusion that should be drawn is that the GP search method will exploit building blocks when the problem is naturally decomposable. However, for many

problems involving program induction, this cannot be assumed. This implies that language and search bias, as a general declarative tool, should be used to simplify the search space and complexity of the problem.

Recent work by the author[85, 88] has demonstrated that a formal grammar may be used to define a schema theorem. The concept of a schema is shown to be naturally represented as a *partial derivation tree*. This work, explored in Chapter 6, shows that using a formal structure for representing programs allows a clear theoretical statement to be made about evolutionary techniques for program induction. The work also shows that a single schema theorem may be written that defines how structures are propagated for both fixed- and variable-length representations. This is possible by demonstrating that a grammar may be written for a fixed-length binary string which has the same schema properties as a standard genetic algorithm. The flexible nature of a grammar, in that all current GP applications may be expressed by a CFG, allows variable-length structures to be represented and therefore a schema theorem, using this grammar, to capture the basic properties of the GP search method.

2.4 Conclusion

This chapter has presented the field of machine learning in terms of representation, operators and bias. The theoretical results of Valiant[74] and Wolpert et al.[92] have been used to argue that no general learning system is capable of performing well over many problem domains. The solution to this dilemma is that learning systems must have formal mechanisms to represent a bias so that any information known about the problem domain may be used to assist the search for a suitable inductive hypothesis. Three types of bias have been introduced to represent the methods available to influence a learning system. These forms of bias are selection bias, language bias and search bias.

The field of evolutionary computation, and in particular Genetic Programming, have not emphasised the use of bias. Although the concept of learning a computer program from example is a general and powerful approach, many difficulties have been found with this technique because of a lack of imposed structure. The advent of strongly-typed genetic programming and the recent use of formal grammars has arisen because of the need to represent structure and typing information within this framework. The conclusions of this development are that, for GP to be truly applicable over a wide variety of problems, explicit language and search bias is necessary to restrict the search space and make the discovery of a suitable computer program tractable.

The work of Utgoff[73], Rosca et al.[61, 62] and Angeline et al.[2, 1] have demonstrated that it is possible to adapt a language or search bias to improve the performance of a general learning system. The use of a formal system, such as a grammar, for representing bias gives the opportunity to automatically adapt this representation during the course of evolving a solution. A learning system that adapts its language and/or search bias may have an improved

performance over a system that uses a static bias.

A formal representation for a learning system allows a theoretical framework to be developed. The Schema Theorem for Genetic Algorithms has been a useful tool for understanding the mechanisms that contribute towards understanding its success and applicability. It is desirable for a learning system to have properties that can be studied, independent of any particular problem, so that a deeper understanding of the learning process may be developed.

In summary, a general learning system should allow a declarative definition of language and search bias, have the ability to adapt this bias during the search for a solution and have a theoretical representation that allows properties of the system to be studied independently of any particular problem. The following chapters described such a system.

Chapter 3

Genetic Program Induction

This chapter defines a program induction system, CFG-GP, that allows explicit bias to be described using a context-free grammar. The derivation trees associated with generated programs are maintained and used to control the subsequent structure of programs when they are transformed by genetic operators. This structure is shown in Figure 3.1. A context-free grammar is used to generate an initial population. Genetic operators are applied to the derivation trees which represent the programs. Thus, the structure defined by the grammar is maintained. These programs are evaluated against a specific problem by creating the program terminals (for evaluation) represented by the derivations trees of the grammar. This explicit separation of the *genotype* (derivation tree) and *phenotype* (program) allows the biased structural information to be maintained automatically. The declarative structure offered by a grammar allows a clear and easily modifiable definition of bias when searching for program solutions. The formal structure of a grammar allows genetic operators to be defined that give a variety of search strategies when evolving programs.

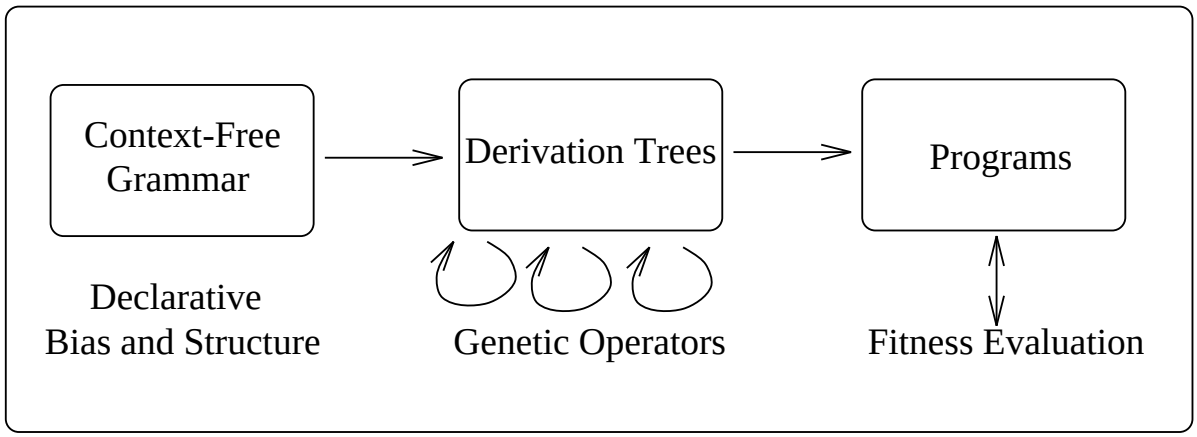


Figure 3.1: Basic Structure of the Program Induction Genetic System.

3.1 Introduction

The previous chapter has argued that bias is desirable when learning complex descriptions such as those involved in computer programs. This is particularly evident when the semantics are not explicitly defined in the structure of the language defining the programs. The ability to explicitly state a language bias and search bias allows the user of the learning system to more clearly define the form of solution being explored and to direct the search for this solution.

An explicit language bias has several advantages over the original genetic programming framework. A declarative definition of the language can provide an unambiguous statement of the arity, typing constraints and overall structure of the components that will describe the solution. The form of the initial population of programs may be explicitly biased, representing the belief of the user that certain components of the language are more likely to be important. Additionally, a separation between the language definition and the implemented functions, composing the language, allows a user to change the bias towards certain constructions without having to modify the underlying implementation of the language.

An explicit search bias allows the user to state the areas of the language where most search effort should be performed when evolving a solution. The previous chapter has emphasised the growing interest in creating mechanisms that protect certain building blocks from the disruptive effects of crossover. This type of structured protection may be expressed explicitly in our formalism by not allowing certain strings in the language to be disrupted. The ability to represent patterns in the evolving solutions and change these strings explicitly also allows the user to force the search for a solution towards areas that are believed to be beneficial.

Furthermore, using an explicit, formal system for representing the language and search bias provides an opportunity for the system to learn how to modify this bias during the search for a solution (see Chapter 5). It also permits a simpler formal analysis of the system performance than is possible for GP (see Chapter 6). This chapter describes a system with these properties.

3.2 Context-Free Grammars

A context-free grammar has been chosen to represent declarative bias and program structure. The reasons for this are as follows.

1. Context-free grammars are simple and easily understood.

2. A context-free grammar may be used to represent incomplete knowledge and general structure that is *known in advance*.

3. A grammar may be easily written to represent the current, most general description of a language that encapsulates a problem. Once this general description has been defined, it is easily modified to represent a preferred bias.

4. Genetic operations, such as crossover and mutation, are easily defined within the framework of programs represented by their associated derivation trees.

5. All previous applications of GP are expressible in a context-free language.

Chomsky[8] distinguished four general classes of grammars. Of these, the three phrase-structured grammars have been the most studied. These *generative grammars* are described as *context-sensitive*, *context-free* and *right-linear*. Of these grammars, the class of context-free grammars are the most popular, as they are simple and yet widely applicable to many problems. For example, most modern programming languages may be defined using this class of grammar. To fix our terminology a formal definition follows.

A context free grammar can be represented by a four-tuple $(N, \sum, P, S)$, where N is the alphabet of nonterminal symbols[1], $\sum$ is the alphabet of terminal symbols[2], P is the set of productions and S is the designated start symbol. The productions are of the form $x \rightarrow y$, where x is a member of N and y is any composition of symbols from $\{\sum \cup N\}$. Productions of the form

$$x \rightarrow y$$
$$x \rightarrow z$$

may be expressed using the disjunctive symbol |, as

$$x \rightarrow y \mid z.$$

When describing a grammar, the nonterminal symbols will be distinguished by *starting with a capital letter*. Terminal symbols will be defined by *lower-case letters only*. A string that is composed of (potentially) both terminal and nonterminal symbols will be represented by *lower-case greek letters*.

3.2.1 Derivation Steps and Derivation Trees

A derivation step represents the application of a production to some string which contains a nonterminal. For example, the string xAy may be transformed by the production $A \rightarrow \alpha$ to $x\alpha y$ in one step. This is represented by the symbol $\Rightarrow$, as follows.

$$xAy \overset{A \rightarrow \alpha}{\Rightarrow} x\alpha y$$

A string that is transformed by *zero or more* derivation steps is represented as follows[3].

$$xAy \overset{\star}{\Rightarrow} x\beta y$$

A string that is transformed by *one or more* derivation steps is represented as follows.

$$xAy \overset{+}{\Rightarrow} x\beta y$$

In general, a series of derivation steps may be represented by a syntax tree or parse tree. As shown in Figure 3.2 the derivation steps

[1] Nonterminal symbols are replaced by other symbols when generating a sentence in the language.
[2] The terminal symbols compose the generated sentences of the language.
[3] The $\star$ operator is known as the Kleene star.

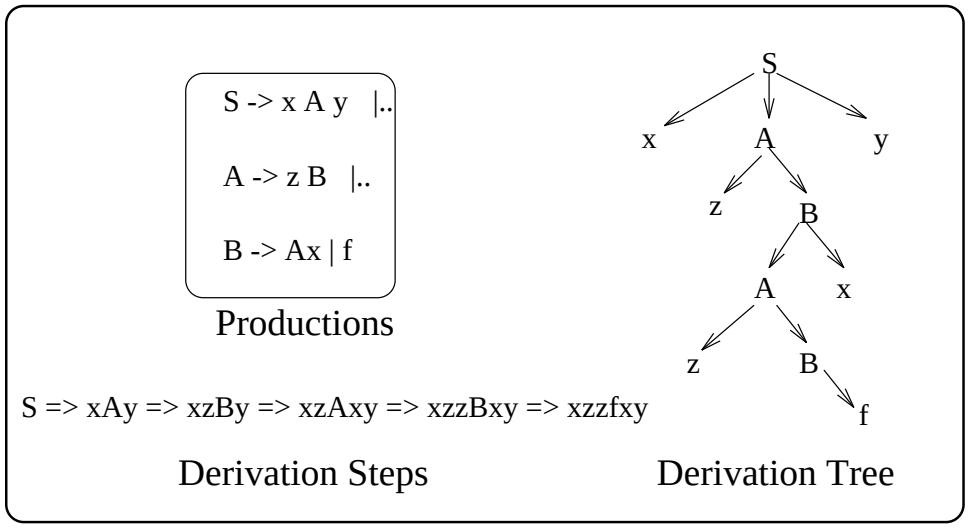


Figure 3.2: The Application of Grammar Productions represented as a Tree.

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

may be represented as a derivation tree. This representation is important because the genetic operations that modify program structures are most easily viewed as operations on these trees.

3.2.2 What a Grammar Represents

A grammar is a scheme for generating a *potentially infinite* number of strings (sentences) in a language. Hence, a grammar that defines a programming language may be considered as the most general description of any program defined by that language. All symbolic learning systems learn within a defined language. This represents the *language bias* for the learning system. The approach described in this thesis allows the language bias to be explicitly defined by tying it to a context-free grammar. The task for the learning system is to search the space of possible programs defined by this grammar and to discover appropriate programs that perform some specifiable task. The semantics of the programs, generated from a grammar, are described in Section 3.5.

3.3 The Structure of the Program Induction System

The structure of the program induction system, CFG-GP, is shown in Figure 3.3. Program derivation trees are selected based on their fitness. These trees are probabilistically modified by the specified genetic operators, thereby creating the next generation of programs. The remainder of this chapter will describe each of these components in the following order.

1. Generation of the initial population using a context-free grammar.
2. Fitness evaluation of the programs representing the population.
3. The proportional fitness selection scheme.

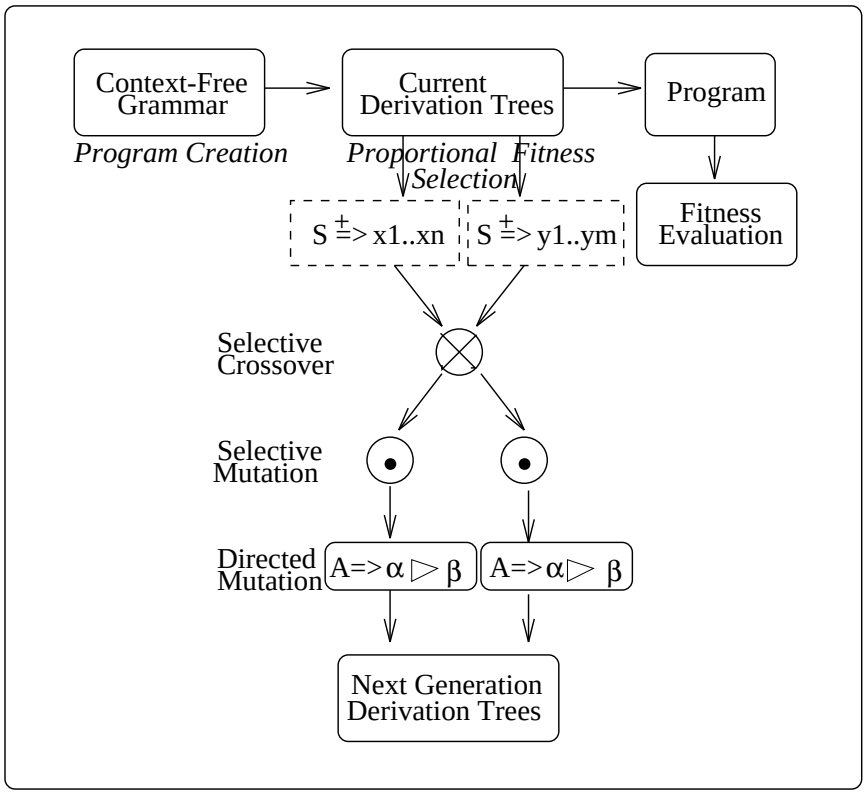


Figure 3.3: Creating the Next Generation of Programs.

4. Selective crossover.

5. Selective mutation.

6. Directed mutation.

3.4 The Initial Population

A grammar may be used as a generator of derivation trees to represent the initial population of programs. The goal of this process is to create a mixture of different programs in terms of overall size, shape[4] and functionality. The notion is to cover a large area of the possible program search space and to allow building blocks[5] to be discovered and combined to create the final solution. Each member of the initial population is forced to be structurally unique. This ensures that a measure of syntactic diversity[6] is initially defined.

Generating a population of programs, using a context-free grammar, proceeds as follows.

1. Label each production to indicate the *minimum depth of derivation tree* to create a string composed only of terminals using this production as the first derivation. This concept will be referred to as min-depth-tree.

[4]This refers to the program structure viewed as a derivation (parse) tree.
[5]Building blocks are parts of programs.
[6]Note that a semantic diversity is *not guaranteed*.

Initial Population Creation	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 4	50
CREATE MAXIMUM DEPTH 5	150
CREATE MAXIMUM DEPTH 6	150
CREATE MAXIMUM DEPTH 7	150
MAXIMUM FAILURES	100

Table 3.1: An Example of Defining the Initial Population.

2. Specify the number of programs to be created. Associated with this statement is a maximum depth of derivation tree allowed during the creation of the programs. Normally, there will be a range of these statements to allow a variety of differently shaped programs to be created.

3. Generate the programs by randomly selecting productions, commencing with the *start symbol* S and guided by min-depth-tree.

4. Enforce uniqueness for each program that is randomly generated. As many programs will be created that already exist some upper bound on the number of failures [7] must be given. If this limit is exceeded, the generation procedure is aborted. This criterion is defined so that the initial population has no repeated programs and is therefore structurally diverse.

3.4.1 Labelling the Productions

The following *bottom-up* algorithm is used to label min-depth-tree for each production from the grammar.

1. For each production of the form $A \rightarrow x_1 x_2 \ldots x_n$ where $x_1 x_2 \ldots x_n \in \sum^{\star}$,
   min-depth-tree = 1.

2. **WHILE** all productions from P have not been labelled **DO**

   (a) For each unlabelled production $A \rightarrow \alpha$, where $\alpha \in \{N \cup \sum\}^{\star}$

   For each nonterminal $B \in \alpha$, if a production $B \rightarrow \beta$ has min-depth-tree set, assign the depth for $A \rightarrow \alpha$ as the maximum of min-depth-tree$\{B\} + 1$.

3.4.2 Creating the Initial Population of Programs

The initial program population is defined by a set of parameters which specify the number of programs to be created with some upper bound on the depth of the generated derivation trees.

[7] By failure we mean that the generated program already exists in the population.

Table 3.1 shows an example of specifying an initial population. The **CREATE MAXIMUM DEPTH** parameter is used to specify the number of programs to be created with various maximum depths of derivation tree. The **MAXIMUM FAILURES** parameter defines the number of times a non-unique program may be created *in succession* during the initialisation of the population. If this limit is passed, the learning system halts. The following algorithm is used to create the population, based on the parameter **CREATE MAXIMUM DEPTH** *depth*.

1. Select the start symbol S. Label this as the current nonterminal A.
2. Select, at random, a production $P_1 \in P$ of the form $A \rightarrow \alpha$ with min-depth-tree$\leq$ *depth*.
3. Select each nonterminal $B \in \alpha$ and label B as the current nonterminal.

   $depth = depth - 1$. Repeat steps 2 and 3.

3.4.3 Selecting Productions with Further Bias

Use of the previous algorithm assumes that there is equal probability of selecting any valid production from a particular nonterminal. The domain expert may wish to express a bias towards the generation of certain program strings in the initial population. This bias is explicitly represented in CFG-GP by associating a *merit weighting* with each production. The probability of a production being selected during the creation of the initial population is now directly proportional to the merit weighting of each production. If we represent the merit weighting for a production from some nonterminal A as w_A, then the probability of selecting this production can be defined as

$$\frac{w_A}{\sum_A w_A}$$

where $\sum_A w_A$ represents the sum of the probabilities of all productions from A which satisfy the minimum depth requirements. This proportionate selection of productions allows a preference to be given in terms of the composition of strings in the initial population.

3.5 Evaluating Generated Programs

The strings represented by a context-free grammar may define many different types of languages that require different interpretations. There are two main techniques for evaluating strings generated from a grammar.

1. Interpret the derivation trees by a *preorder traversal*, thereby giving a functional representation to the structure of the language.

2. Allow the generated strings to be passed to another system for interpretation. Such a system could include a compiler, assembler or interpreter.

3.5.1 Evaluating Programs as Functions in Preorder

Given a derivation tree such that $S \stackrel{+}{\Rightarrow} x_1 \ldots x_n$, the tree is traversed in preorder fashion to the first terminal x_1. This function is evaluated. Arguments that are evaluated from x_1 are traversed in preorder until a terminal symbol is found. The arguments accessible to any function are limited by the production which generated this string. For example, the productions

$$A \rightarrow f\ x_1\ x_2 \mid g\ B$$
$$B \rightarrow h\ x_3$$

represent the functions, $f(x_1, x_2)$ and $g(h(x_3))$. The evaluation of the function, g, causes the derivation tree *rooted in* B to be traversed in preorder. This results in the function $h(x_3)$ being evaluated. The result of this function is passed *by value* to g. Note that g does not have direct access to h or x_3.

3.5.2 Passing Evaluation to an External System

Investigation of this approach is beyond the scope of this document. It is simply implemented by passing the terminal string of each derivation, via a pipe[8] or file, to the system that is used for evaluation. The evaluated fitness may be passed back to the learning system using the same mechanism.

3.6 Proportional Fitness Selection

This section describes the method by which programs are selected from the population for crossover and mutation. The selection is based on the fitness of each program in relation to the entire population. The following scheme represents a roulette wheel selection strategy which is a proportional fitness measure.

Following Koza's definition of fitness[44], the *raw fitness* **r(i,t)** is defined as the performance measure given to **program i** in **generation t**. This fitness measure is mapped to a *standardised fitness* **s(i,t)** which ranges from 0 to some number > 0. The fittest programs have the lowest value of standardised fitness. A *perfectly fit* program will have a fitness of 0. This is the measure of fitness that *must be supplied to the learning system* by the user.

This standardised fitness is scaled to a range between 0 and 1 and is referred to as *adjusted fitness* **a(i,t)**. The adjusted fitness is defined as:

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

[8] A pipe may be thought of as a communication channel.

Finally, this *adjusted fitness* is normalised in terms of the proportion of the population with certain fitness values. This gives a *normalised fitness*, **n(i,t)**, calculated as:

$$n(i,t) = \frac{a(i,t)}{\sum_{i=1}^{population} a(i,t)}$$

.

The normalised fitness has the following properties.

- **n(i,t)** ranges between 0 and 1.
- **n(i,t)** is larger for fitter individuals.
- $\sum_{i=1}^{population} \mathbf{n(i,t)} = 1.$

This *normalised fitness* is used as the basis for the selection of programs in a proportional manner. This is done as a *roulette wheel* selection in which the probability of a program being selected is proportional to its normalised fitness. There are other methods (such as rank based selection[3] and tournament selection[19]) that could be used to select programs[9]. This thesis will not consider the various merits of these techniques as the focus of this work is the representation of explicit bias. The use of proportional selection has been demonstrated to be adequate with previous GP systems[44] and will be assumed to be suitable for this current work.

3.7 Selective Crossover ⊗

The crossover search operator creates new programs by mixing the contents of two parent programs. In the context of a population, represented by derivation trees, crossover operates by swapping the derivations associated with two occurrences of the same nonterminal. By requiring that the rooted derivation trees that are swapped have matching nonterminals, we ensure that the new programs *must also be members of the language defined the grammar.* The crossover is described as *selective* because the operation may be specified to have legal crossover sites from a subset of the total nonterminals. The crossover operator may change the depth of the derivation trees associated with a program. To ensure that the created programs do not grow in an unbounded fashion, a limit is placed on the maximum depth of any program created by crossover. This parameter is referred to as **MAX-DEPTH-PROGRAMS** and reflects the maximum depth of any program that is expected to be required to produce a solution. Given two derivation trees, ρ_1 and ρ_2, selective crossover is defined as follows:

1. Randomly select a nonterminal $A \in \otimes$ from ρ_1. If no nonterminal from the set $\otimes$ exists in ρ_1, crossover is complete.

[9] Work which compares various selection mechanisms may be found in Goldberg et al.[18].

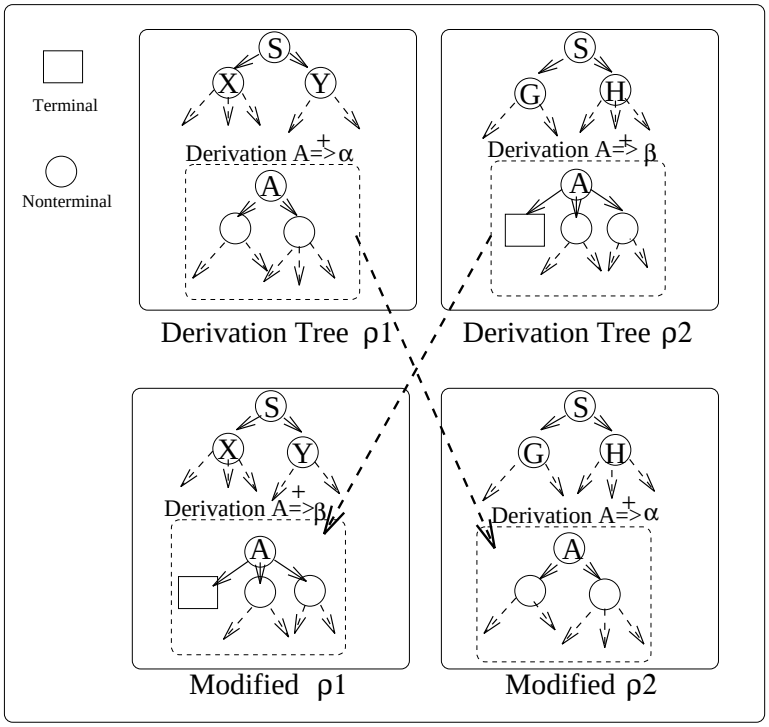


Figure 3.4: Crossover Based on Derivation Trees.

2. Randomly select a nonterminal $A \in \otimes$ from ρ_2. If no nonterminal from the set $\otimes$ exists in ρ_2, crossover is complete.

3. Swap the derivation trees below these matching nonterminals, thereby creating 2 new programs.

4. If either new derivation tree exceeds **MAX-DEPTH-PROGRAMS** the crossover is ignored. The parent derivation trees ρ_1 and ρ_2 then remain unchanged.

Figure 3.4 shows crossover viewed as an operation on derivation trees. Crossover may also be described in terms of how the final strings of a program are mixed. If we have two programs, with derivations

$$S \stackrel{+}{\Rightarrow} x_1 \ldots x_n \; A \; z_1 \ldots z_i \stackrel{+}{\Rightarrow} x_1 \ldots x_n y_1 \ldots y_m z_1 \ldots z_i$$
$$S \stackrel{+}{\Rightarrow} a_1 \ldots a_j \; A \; c_1 \ldots c_k \stackrel{+}{\Rightarrow} a_1 \ldots a_j b_1 \ldots b_r c_1 \ldots c_k$$

crossover on the nonterminal A will produce two new strings of the form

$$S \stackrel{+}{\Rightarrow} x_1 \ldots x_n \; A \; z_1 \ldots z_i \stackrel{+}{\Rightarrow} x_1 \ldots x_n b_1 \ldots b_r z_1 \ldots z_i$$
$$S \stackrel{+}{\Rightarrow} a_1 \ldots a_j \; A \; c_1 \ldots c_k \stackrel{+}{\Rightarrow} a_1 \ldots a_j y_1 \ldots y_m c_1 \ldots c_k$$

3.7.1 Parameter Specification for Selective Crossover

Table 3.2 shows an example of the specification of selective crossover. The maximum allowable depth of a derivation tree that can be created has been set to 15. There are two types of

Selective Crossover	
Parameters	Specifications
MAX-DEPTH-PROGRAMS	15
$\otimes = \{A, B, C\}$	40%
$\otimes = \{E, F\}$	50%

Table 3.2: Specification of Selective Crossover.

crossover operations; one using the nonterminals $\{A, B, C\}$, occuring with a probability of 40%, and one using the nonterminals $\{E, F\}$, occuring with a probability of 50%.

3.8 Selective Mutation ⊙

Mutation has traditionally been associated with a *small change* in the structure of the representation to which it applies. The mutation operator introduced here follows Koza's GP mutation[44]. However, as a selective operator it allows the user to be more discriminating about how a program is modified.

Mutation applies to a single derivation tree (and the program that it represents). The mutation is referred to as selective because the operation may be specified to have legal mutation sites from a subset of the total nonterminals. This set of legal mutation sites is represented by the symbol, ⊙. Mutation is portrayed in Figure 3.5 as an operation on a derivation tree using the language defined by the grammar, G. Given a derivation tree ρ_1 the selective mutation algorithm is defined as follows.

1. Randomly select some nonterminal $A \in \odot$ for mutation. If no nonterminal A exists in ρ_1, then mutation is complete.

2. Delete the derivation tree rooted in the nonterminal A.

3. Create a new derivation tree associated with A using the grammar G that has been defined. The depth of tree created is limited by the parameter **MAX-DEPTH-PROGRAMS**. This operation applies the same technique as that of generating the initial population. However, the generation commences with the nonterminal A rather than the start symbol S.

3.8.1 Parameter Specification for Selective Mutation

Specifying selective mutation is similar to that of selective crossover. The nonterminals indicating the selective sites for mutation are specified along with the probability of mutation occuring. These parameters are shown in Table 3.3, where two mutation operations are defined. One is

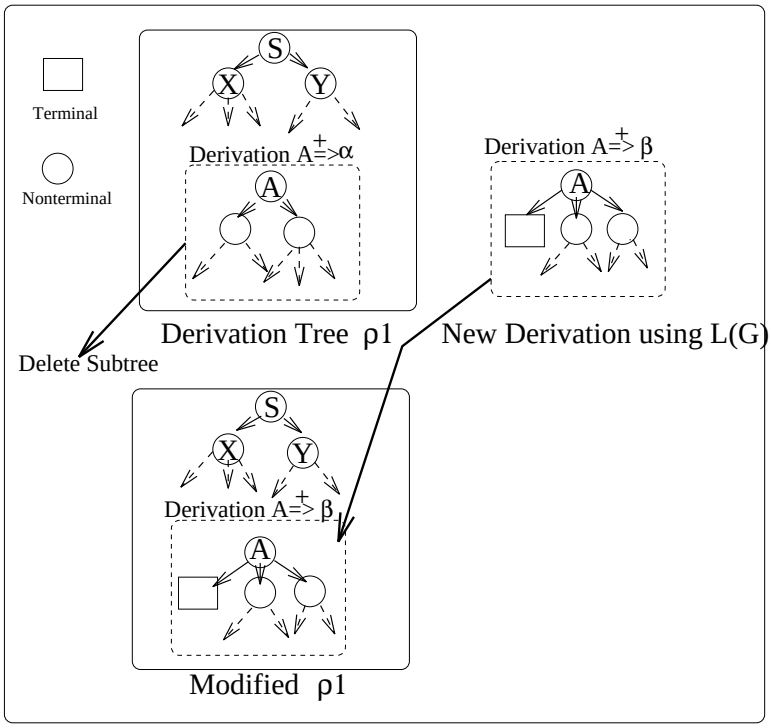


Figure 3.5: Mutation Based on Derivation Trees.

Selective Mutation	
Parameters	Specifications
MAX-DEPTH-PROGRAMS	15
$\odot = \{A, B\}$	10%
$\odot = \{C\}$	5%

Table 3.3: Specification of Selective Mutation.

defined over the nonterminals A and B with a probability of 10%, the other over the nonterminal C with a probability of 5%.

3.9 Directed Mutation $A \rightarrow \alpha$

The crossover and mutation operators perform the role of searching for a good inductive hypothesis by probabilistically modifying the current population of derivation trees. Although their selective definition allows some control over the search for new structures there are often situations where explicit control over this search is desirable. For example, if a repeated pattern is discovered in a fit derivation tree it may imply that a recursive structure could be exploited with this definition. The ability to express conditions that may suggest iteration or recursion can be used to direct a change in the derivation tree structure that reflects this condition. An example of this type of structure is shown in Section 4.2, where the recursive structure of a membership program is discovered using a directed mutation operator. The ability to express this type of search bias is only possible because of the population-based approach to learning. A probabilistic search that radically alters the structure of one population member will not affect the overall performance of the system if it is not found to be useful. If the learning system used a single program (derivation tree) to to represent the current hypothesis a change in structure of this order may cause the system to fail unless the suggested alteration was correct.

The selective mutation operator is potentially very destructive, in contrast with the normal view of mutation which implies *a small change in structure*. This is obvious by considering a selective mutation that selects the start symbol, S, as the site for mutation. If this occurs then the entire derivation tree is deleted and a new, random derivation tree is created. Thus all information which had contributed to the fitness of the derivation tree is lost. This destructive nature of selective mutation is also demonstrated theoretically in Section 6.6. Hence, a second mutation operation is defined that allows small changes to be controlled in the structure of a program. We call this operation *directed mutation*.

A directed mutation specifies that one particular production used in a program derivation should be replaced by a second production. The productions may contain both terminal and nonterminal symbols. If the replaced production contains nonterminal symbols they are randomly generated, using the grammar G, in the same manner as selective mutation. The general form of a directed mutation is

$$A \rightarrow \alpha \triangleright \beta$$

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

Note that $A \rightarrow \alpha$ and $A \rightarrow \beta$ are legal productions defined in the grammar. *Directed mutation* may be considered as a specialisation of *selective mutation* where the selection site for mutation is specified and the form of mutation is defined explicitly. An example of directed mutation is shown in Figure 3.6.

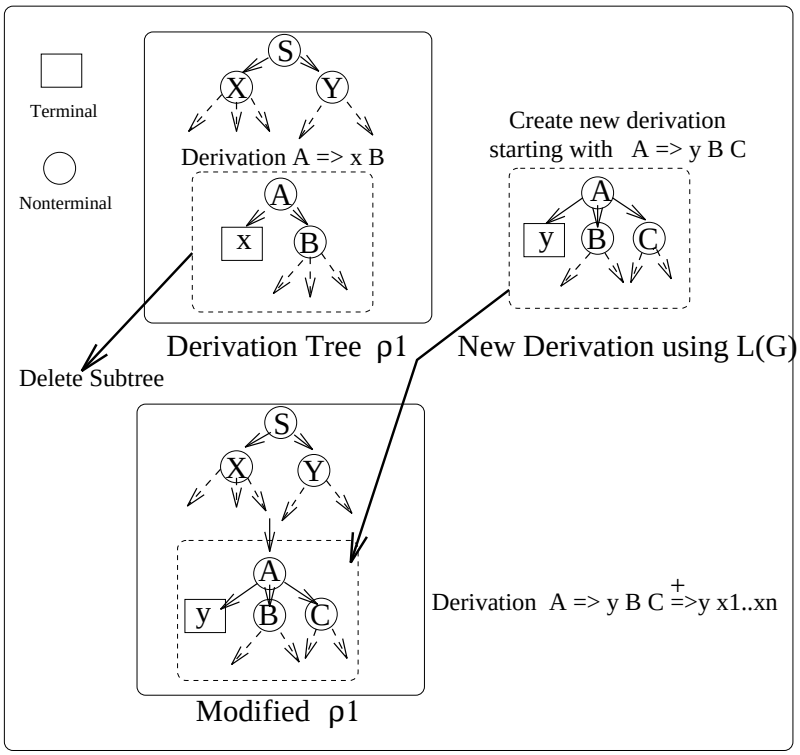


Figure 3.6: Directed Mutation $A \rightarrow x\ B \triangleright y\ B\ C$.

Given a derivation tree ρ_1, the directed mutation algorithm is defined as follows:

1. Randomly select some derivation representing the production $A \rightarrow \alpha$ from ρ_1. If no derivation of this form exists in ρ_1, then directed mutation is complete.
2. Delete the derivation tree associated with this production.
3. Create a new derivation tree associated with A using the production $A \rightarrow \beta$.
4. For each nonterminal in β, create a new derivation tree using the grammar G. The depth of each generated derivation tree is limited by the parameter **MAX-DEPTH-PROGRAMS**.

3.9.1 Parameter Specification for Directed Mutation

As shown in Table 3.4, the specification for directed mutation is similar to previous genetic operators. The parameter **MAX-DEPTH-PROGRAMS** is required as new derivations may have to be randomly generated when the substitution of one production for another is performed. The most important role of directed mutation is to permit the user to define what a *small change* should be considered to be in terms of modifying a program.

For example, the directed mutation $B \rightarrow x_1 \triangleright y_1$ allows a program to mutate the string x_1 to the string y_1. This is reminiscent of a standard genetic algorithm, single-bit mutation where one bit in a string is flipped, based on some probability.

Directed Mutation	
Parameters	Specification
MAX-DEPTH-PROGRAMS	15
$A \rightarrow B\ x \triangleright y\ C$	10%
$B \rightarrow x_1 \triangleright y_1$	5%

Table 3.4: Specification of Directed Mutation.

3.9.2 A Generalisation of Directed Mutation

The *directed mutation* operator selects one derivation and replaces it with a second derivation. This concept can be extended to allow larger derivation tree structures to be specified, thereby allowing complex patterns in a derivation tree to be recognised. For example, a derivation from A such as

$$A \overset{A \rightarrow xyC}{\Rightarrow} x\ y\ C \overset{C \rightarrow bD}{\Rightarrow} x\ y\ b\ D$$

may be replaced (*mutated*) by the derivation steps

$$A \overset{A \rightarrow Ec}{\Rightarrow} E\ c \overset{E \rightarrow xF}{\Rightarrow} x\ F\ c \overset{F \rightarrow Ga}{\Rightarrow} x\ G\ a\ c$$

This generalisation allows derivation trees of an arbitrary complexity to be selected for mutation. The derivation tree that replaces the specified mutation site may also be specified by any number of derivation steps[10]. This allows general structures to be distinguished in the population. An application of this extended mutation operator is shown in Section 4.2.

3.10 Implementation Issues

The user of the program induction system must supply the following components before the system is used.

1. A context-free grammar representing the language bias and therefore the legal derivation trees that may be created.
2. The population dynamics including population size, genetic operators, maximum depth of programs and maximum number of generations before halting.
3. A function that represents the semantics of each *terminal* defined in the grammar.
4. A function that determines the *standardised fitness* for each derivation tree, represented by a real number ≥ 0.

Our program induction system, CFG-GP, has been written in the 'C' programming language. The main loop that controls this system is as follows[11]. Note that for each generation the population size remains constant.

[10] Once a string in the language is composed *of terminals only* the derivation is complete.
[11] The psuedo code presented here reflects a 'C' style of coding.

```
cfg_initialise_user_system();       /* USER CALL */
cfg_link_user_functions();          /* USER CALL */


create_initial_pop_storage(NUM_POPULATION);
generate_initial_pop();

cfg_adjust_user_test_functions(0,MAX_GENERATIONS); /* USER CALL */

evaluate_population(NUM_POPULATION,0);

for(gen=1;gen<=MAX_GENERATIONS;++gen)
{
    modify_population(NUM_POPULATION);

    cfg_adjust_user_test_functions(gen,MAX_GENERATIONS); /* USER CALL */

    evaluate_population(NUM_POPULATION,gen);
}

if (cfg_adjust_user_test_functions(gen,MAX_GENERATIONS)) /* USER CALL */
    evaluate_population(NUM_POPULATION,gen);
```

To setup these components, the following functions must be completed[12].

1. **cfg_initialise_user_system():** This function is called when CFG-GP commences. The user must supply the code to initialise the structures used to evaluate the problem.

2. **cfg_link_user_functions():** This function is called to link each of the terminal symbols representing a function call to the address of a 'C' function. To link a symbol and its function the 'C' function **link_function_to_symbol(string,function):** is supplied. This function must be called with the string of each terminal symbol in the grammar. The function field represents the address of the function that will be associated with the string for evaluation.

3. **cfg_adjust_user_test_functions(current_gen,max_gen):** This function is called to allow the user to adjust the user environment (i.e. training data) during evolution. This

[12] The functions represented here are the 'C' interface functions. The argument types have been left unspecified.

may be required when the information used to train the learning system is modified during the evolution. A common example occurs when both training and test data is run for the same population. The final call to this function allows this situation to occur. The function must return a non-zero value to perform the last **evaluate_population(..)**. Other evaluations of this function are ignored.

4. **evaluate_program_tree(program,generation):** This function is called for each program in the population for each generation. The fitness value for the evaluated program must be set in the program structure. This function uses **eval_tree(dtree)** to evaluate the program on the defined task.

The CFG-GP system defines the following functions, which are available to the user to determine the fitness for each derivation tree.

1. **eval_tree(dtree):** This function evaluates the derivation tree **dtree** and returns a structure representing the result. The contents of the structure are defined by the user, such that there is no limitation imposed on the type of a result or the information passed between functions.

2. **get_argument(dtree,argnum):** This function returns a derivation tree associated with the argument position accessible from the particular derivation **dtree**. This is determined by the production used to create the derivation. For example, to evaluate the arguments associated with the function, f, created from the production

$$A \rightarrow f\ B\ x_2$$

the following procedure may be followed. The derivation tree associated with B is obtained by using **get_argument(dtree,1)** and then evaluating this tree using **eval_tree(..)**. Similarly, the value of x_2 is obtained by calling **eval_tree(get_argument(dtree,2))**.

3.11 Conclusion

This chapter has described a program induction system, CFG-GP, which learns computer programs defined by a context-free grammar. The use of a context-free grammar to represent declarative bias and structure allows a domain expert to narrow the possible search space and, therefore, have a greater possibility of discovering solutions. The grammar allows both the typing to be automatically maintained with program constructs and also the structure (i.e. how functions are combined) to be explicitly stated and controlled.

The programs are selected based on a roulette wheel selection scheme which gives proportional representation to programs based on their fitness. The derivation trees representing the programs are evaluated in *preorder*. This represents a functional interpretation of the language defined by the grammar.

The genetic operators of *selective crossover*, *selective mutation* and *directed mutation* are used to transform the population of derivation trees from one generation to the next. These operators allow a declarative search bias and may be used to control where in the search space most effort is concentrated when evolving a solution.

Chapter 4

Some Examples

This chapter describes two applications of the program induction system, CFG-GP. The first problem is the boolean one known as the 6-Multiplexer. It has been selected for its simplicity. The concepts of changing the language and search bias are demonstrated.

The second problem involves determining a predictor for the density of an Australian marsupial over a geographic region. This region is divided into four hundred areas. Within each of these areas, six independent variables represent conditions of the forest at that location. The "known" values of marsupial density were determined by a survey. This problem is interesting as it represents a complex spatial problem which contains noise and is unlikely to have an exact solution. The language which describes how these marsupials interact with the environment requires typed functions and allows the user to express some knowledge about expected properties of the animals, in terms of a home range and preferred habitat conditions.

Together, these problems show that the CFG-GP system is capable of a wide range of descriptions and strategies as follows.

- The initial population may be explicitly biased based on knowledge of the likely importance of certain structures in the language.
- The language bias is defined in a declarative manner.
- The search operators (crossover and mutation) may be biased to place emphasis on certain parts of the language to be explored.
- The definition of the language and search bias are defined transparently.
- The language and search operators may be easily modified. Hence, information about the problem (and its solution) may be incorporated into the declarative framework, as it is discovered.

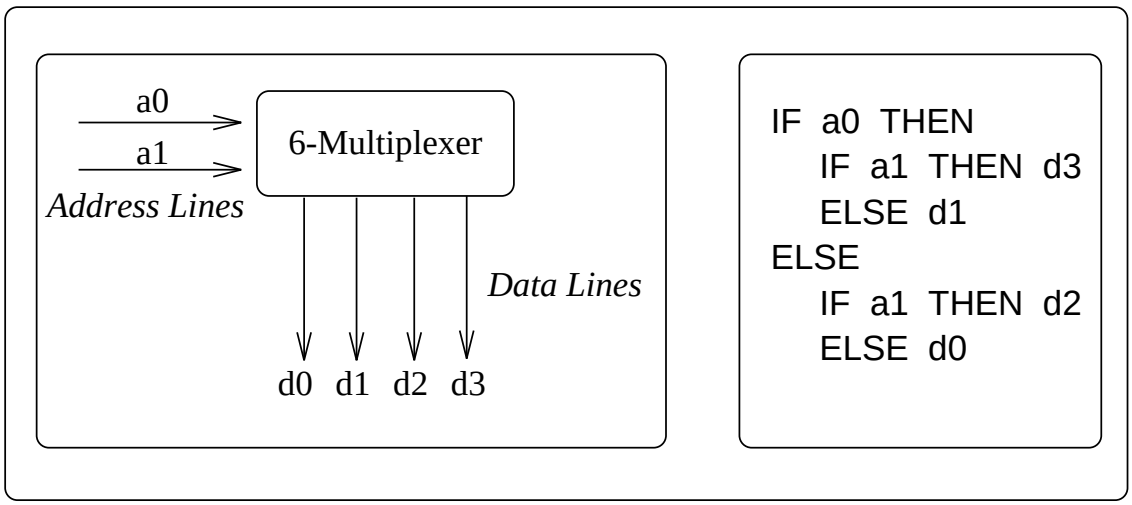


Figure 4.1: Basic Structure of the 6-Multiplexer

4.1 The 6-Multiplexer

The 6-multiplexer is a simple boolean problem where two address lines are used to select between four possible data lines. This structure is shown in Figure 4.1, along with one possible solution expressed as a series of **if-then-else** statements. The small search space and known structure for this problem make it suitable to explore how the modification of both the search bias and the language bias may improve the performance of learning.

4.1.1 The Grammar G_{6m} for the 6-Multiplexer

The language defined for this problem will include the basic boolean operators **and x y**, **or x y** and **not x**. In addition, the 3-argument function, **if x y z** is used to select between various conditions. This function evaluates the first argument, returning the second if this is true, otherwise the value of the third argument is returned. These four boolean functions will allow the biasing of particular components of the defined language to be demonstrated. The grammar G_{6m} [1] is defined as follows.

$$
\begin{aligned}
&G_{6m} = \\
&\quad \{S, \\
&\quad N = \{B, T\}, \\
&\quad \textstyle\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}, \\
&\quad P = \\
&\qquad \{S \rightarrow B \\
&\qquad\ B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T
\end{aligned}
$$

[1] The merit value for each production is set to one and is therefore not shown in this definition.

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

}

}

Although G_{6m} could be simplified to eliminate the nonterminal T, this additional structure will be used to demonstrate the use of search bias.

4.1.2 The Measure of Success

A basic statistic that may be associated with a stochastic event for a learning system is *the probability of success*, p_s. This represents the probability that a given run of the system will succeed before the set maximum number of generations is complete. This measure will be used to compare various language and search biases and is determined by running CFG-GP many times for a particular configuration of population and set of operators. To determine p_s for the 6-multiplexer, one hundred runs were performed for each configuration. The value of p_s is influenced by the following conditions.

- The initial population structure.
- The maximum depth of a program derivation tree after crossover and mutation.
- The number of generations before completion.
- The search operators (e.g. selective crossover sites).
- The language bias defined by the grammar.

This thesis will use the *probability of success* to indicate how changing search operators and grammatical structure can influence the discovery of a solution. This will be done by maintaining a constant set of population parameters and adjusting *one factor at a time*. A measure of significance is computed on the basis that the distribution of runs is approximately normal. This assumption is shown to be valid in Appendix A. The calculation of significance for each multiplexer example is shown in Appendix B.

4.1.3 Initial System Specification

The initial setup used for the 6-multiplexer is shown in Table 4.1. A population of 500 programs is used to search for a boolean expression that matches *all 64 test cases*. The initial population is created with four different maximum depth specifications, the lowest being five and the greatest eight. Although the grammar has a smallest derivation tree of depth three (see Figure 4.2), the minimum depth of 5 was selected to accommodate changes in the grammar when later language bias is demonstrated. This was necessary as

The 6-Multiplexer	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 5	200
CREATE MAXIMUM DEPTH 6	100
CREATE MAXIMUM DEPTH 7	100
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	100
MAX DEPTH PROGRAMS	8
$\otimes = \{B\}$	90%
GENERATIONS	50
FITNESS MEASURE	64 boolean cases
GRAMMAR	G_{6m}

Table 4.1: The Initial 6-Multiplexer System.

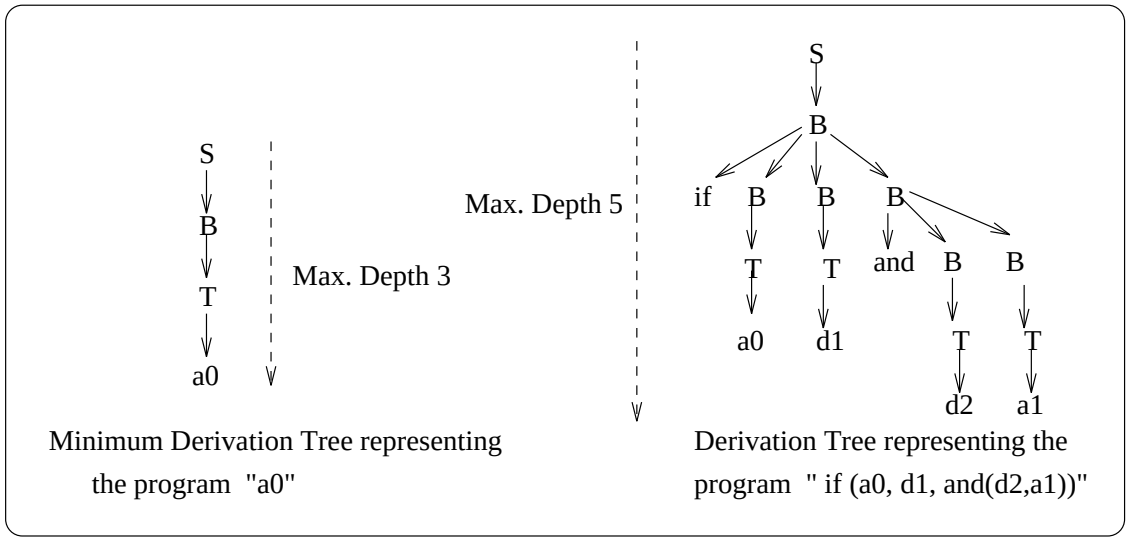


Figure 4.2: Two Example programs generated using the grammar G_{6m}.

the same population configurations must be used for each different run. In this way, the obtained probabilities of success can be meaningfully compared.

The search operator used is *selective crossover* over the nonterminal, B, with a probability of 90%. The nonterminal, T, has *not been selected* as a crossover site. Later experiments will use T to demonstrate change in the search bias. Based on the 90% probability of crossover occurring, 50 programs could be expected to be copied into the next generation. The selective crossover using B allows all possible combinations of program components to be mixed. This, however, gives a bias towards swapping internal structures, rather than the tips of derivation trees. As shown in Figure 4.2, the address and data values are derived from the nonterminal, T. Since T is *always derived* from B, crossover using B may swap individual terminals.

4.1.4 Results

Using the settings from Table 4.1, the initial *probability of success* of 34% was determined. Figure 4.3 shows the changing population structure for a typical run which succeeded after thirty generations. The boxes represent the initial (generation zero) structure of each program in the population defined by the number of terminal and nonterminal symbols in each derivation tree. The single lines represent the structure of the population after thirty generations. It is clear that the structure of the population as a whole alters significantly during evolution. This phenomenon has been recognised previously in genetic programming literature and is referred to as *bloating* or *the emergence of introns*[55]. The graph of the maximum depth of derivation tree for the population shows that the initial structure of the population, as defined by Table 4.1, is faithfully recreated. A small number of programs have been created with a maximum derivation tree depth of three or four. This occurs, since there is some probability that productions will be randomly selected which create small derivation trees (i.e. select the production $B \rightarrow T$ early in the derivation of the program). This is desirable because the initial population should be a mixture of program sizes.

Figure 4.4 shows how the standardised fitness changes during the evolution of a solution. This figure shows the best, average and worst standardised fitness for each generation throughout the evolution until a solution is discovered after thirty generations. Notably, the average fitness of the population steadily decreases and, therefore, the evolution is gradually converging to a fit solution.

4.1.5 Modifying the Initial Population Bias

The initial population structure may be changed by biasing particular productions to occur with different probabilities. This is achieved by changing the merit selection values

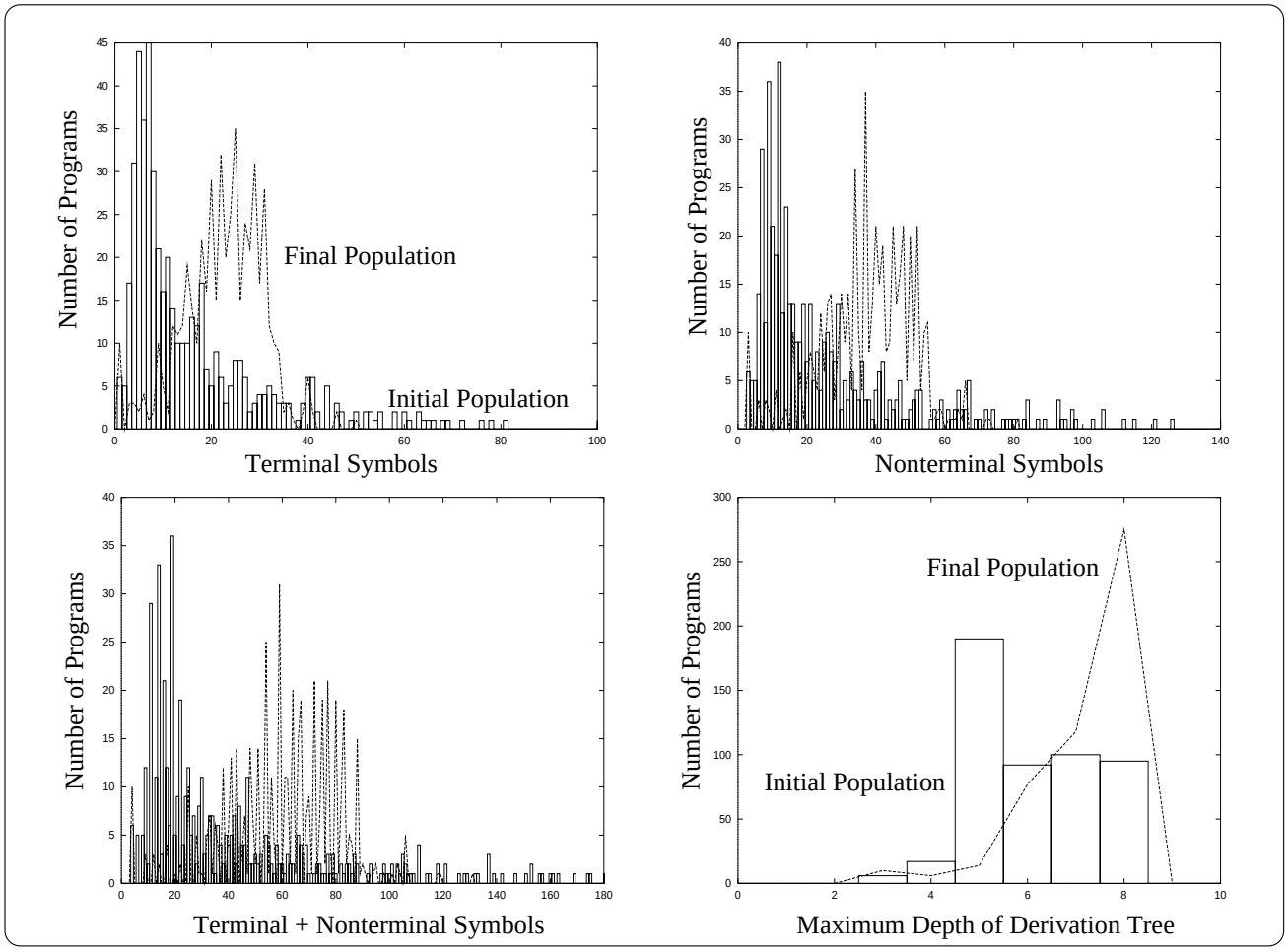


Figure 4.3: Population Statistics - At Generation 0 and after 30 Generations.

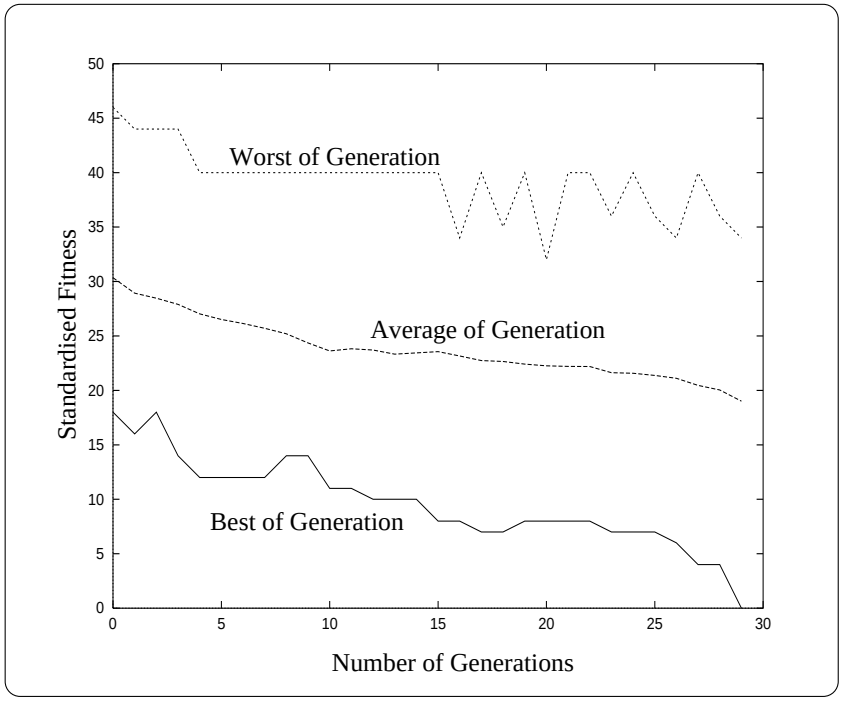


Figure 4.4: Best, Average and Worst Standardised Fitness.

for particular productions. To demonstrate this concept, p_s was determined for a modified G_{6m} (referred to as $G_{6m-popbias}$) that increased the probability of the if function occuring in the initial population. This grammar is defined as follows.[2]

$$
\begin{aligned}
&G_{6m-popbias} = \\
&\quad \{S, \\
&\quad\ N = \{B, T\}, \\
&\quad\ \textstyle\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}, \\
&\quad\ P = \\
&\qquad \{S \rightarrow \overset{1}{B} \\
&\qquad\ B \rightarrow \overset{1}{and}\ B\ B \mid \overset{1}{or}\ B\ B \mid \overset{1}{not}\ B \mid \overset{4}{if}\ B\ B\ B \mid \overset{1}{T} \\
&\qquad\ T \rightarrow \overset{1}{a0} \mid \overset{1}{a1} \mid \overset{1}{d0} \mid \overset{1}{d1} \mid \overset{1}{d2} \mid \overset{1}{d3} \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

The value of the merit selection for the if production for grammar $G_{6m-popbias}$ was selected so that there was approximately a 50% probability of selecting this production from the nonterminal, B, when creating the initial population.

The probability of success using $G_{6m-popbias}$ was found to be 60%[3]. Figure 4.5 shows that the average number of if functions for each initial program (with a particular *maximum depth of parse tree*) reflects the increase in function distribution defined by the modified merit values.

4.1.6 Discussion of Initial Population Bias

The increase from G_{6m} is a direct result of the significance of if as a function for representing the solution to the 6-multiplexer. The important issue that is demonstrated here is that modification of the function distribution in the initial population may result in improving the performance of the learning system. The merit selection values are a *declarative* method for representing these biases and are a shorthand technique for representing multiple, identical productions. The ability to change this bias in a simple and declarative manner, without altering any other components of the CFG-GP system, is one advantage that merit selection and a formal grammar has in relation to the standard GP system.

[2]The merit selection probabilities are shown above the first symbol in each production. e.g. To represent a merit selection of 4 for the function fn, the production is written as $\overset{4}{fn}$.

[3]This is a statistically significant improvement based on a 95% one-sided confidence interval (see Appendix B).

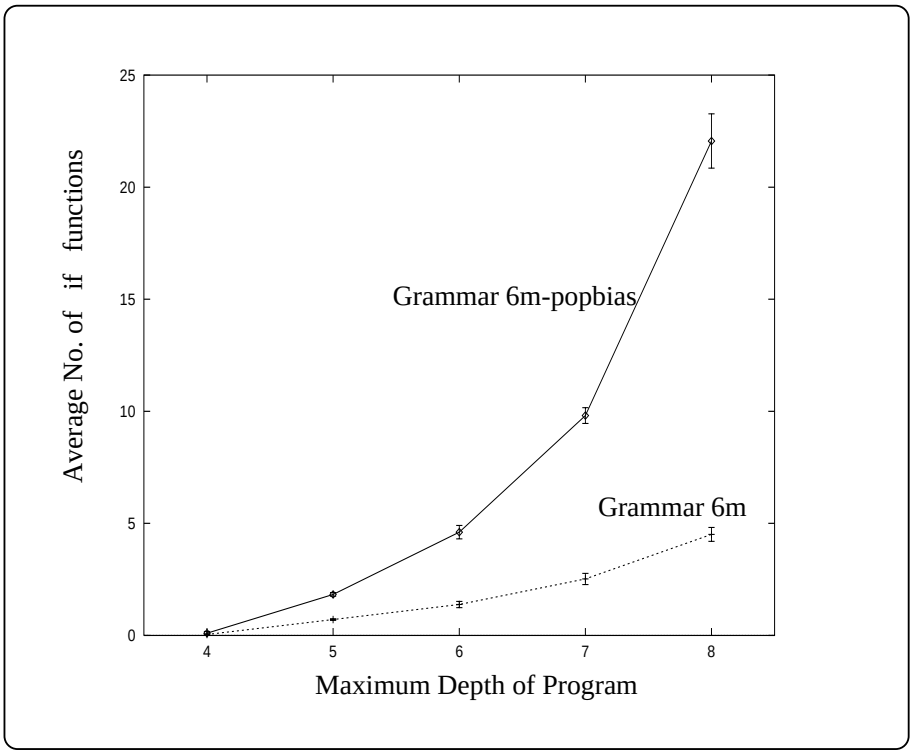


Figure 4.5: Distribution of if functions in the initial population.

4.1.7 Modifying the Search Bias

Search bias relates to how a program is modified to create a new hypothesis (program) for each generation. The selection of any program is based on its fitness in relation to the entire population. Once the program is selected, its offspring are influenced by the type of crossover and mutation operators applied. The search bias may be modified by changing these operators. Selective crossover may influence the search bias in a number of ways.

- Change of the probability of a particular selective crossover operation occuring.
- Change of the nonterminal sites where a particular selective crossover occurs.
- The addition or removal of further selective crossover operations, so that other parts of a program are swapped during evolution.

To illustrate these concepts, the previous setup defined in Table 4.1 is extended to allow an additional selective crossover operation over the nonterminal, T. This is shown in Table 4.2. This represents an increased emphasis of the search in terms of swapping single terminals between programs. For example, the program $if(a0, d1, and(d2, a1))$, shown in Figure 4.2, has a probability of 66% of selecting a single terminal for crossover using $\otimes = \{B\}$ and a probability of 80% using $\otimes = \{B, T\}$.

Using the settings of Table 4.2 the probability of success was found to be 41%. This is not statistically significant (in terms of an improvement from the base level of 34%) for a

Search Bias	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 5	200
CREATE MAXIMUM DEPTH 6	100
CREATE MAXIMUM DEPTH 7	100
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	100
MAX DEPTH PROGRAMS	8
$\otimes = \{B, T\}$	90%
GENERATIONS	50
FITNESS MEASURE	64 boolean cases
GRAMMAR	G_{6m}

Table 4.2: Search Bias with the 6-Multiplexer.

one-sided 95% confidence interval (see Appendix B).

4.1.8 Discussion of Search Bias

The 6-multiplexer is a trivial problem. The simplicity of the search space is illustrated by the (slight) improvement in success when crossover with the nonterminal, T, is introduced. This crossover is essentially a single point mutation since it can swap *terminal symbols only*. The importance of *selective crossover* is the ability to declaratively state which parts of the language are to be most extensively modified. This will be further demonstrated with the Glider Density example described in Section 4.3.

4.1.9 Modifying the Language Bias

The grammar G_{6m} does not have a bias towards any particular form of boolean expression. If, in advance, something is known about the structure of the possible solution a grammar may be defined to incorporate this knowledge in the form of an explicit bias (the merit selection weights were one other form of bias, however they did not influence the form of solutions). Note, that in the case of the 6-multiplexer we know the exact solution. The following experiments demonstrate this concept by changing the underlying grammatical definition, without changing other parameters of the learning system. As knowledge about the solution structure is introduced, the *probability of success* significantly increases. This is important, as it allows the user to explore changes to the representation language without additional functions being introduced to the language. Changing the underlying grammar allows a biasing mechanism for exploring the *structure* of potential solutions to

be developed. The following examples all use the settings of Table 4.1 to define CFG-GP, modifying *only the grammar* in each case. Since $\otimes = \{B\}$, additional nonterminals introduced to the grammar are not used as legal crossover sites.

The Grammar G_{6m-if}

The first bias in the language forces all programs to begin with an if function. The grammar is defined as follows:

$$
\begin{array}{l}
G_{6m-if} = \\
\quad \{S, \\
\quad\ N = \{B, T\}, \\
\quad\ \sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}, \\
\quad\ P = \\
\quad\quad \{S \rightarrow if\ B\ B\ B \\
\quad\quad\ B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T \\
\quad\quad\ T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3 \\
\quad\quad \} \\
\quad \}
\end{array}
$$

The probability of success for G_{6m-if} was found to be 37%[4].

The Grammar $G_{6m-if-address}$

This grammar forces all programs to begin with the if function using a first argument of either address line $a0$ or $a1$.

$$
\begin{array}{l}
G_{6m-if-address} = \\
\quad \{S, \\
\quad\ N = \{B, T, ADDRESS\}, \\
\quad\ \sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}, \\
\quad\ P = \\
\quad\quad \{S \rightarrow if\ ADDRESS\ B\ B \\
\quad\quad\ ADDRESS \rightarrow a0 \mid a1 \\
\quad\quad\ B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid T \\
\quad\quad\ T \rightarrow a0 \mid a1 \mid d0 \mid d1 \mid d2 \mid d3
\end{array}
$$

[4]This is not statistically significant at a 95% confidence interval based on the initial grammar G_{6m} (see Appendix B).

$$\}$$

$$\}$$

The probability of success for $G_{6m-if-address}$ was found to be 62%, which is a significant improvement from G_{6m} (see Appendix B).

The Grammar $G_{6m-if-then}$

This grammar forces all programs to have a top structure of two if functions. The second if function is placed in the *then* position of the first if function.

$$G_{6m-if-then} =$$

$$\{S,$$

$$N = \{B, T, IFTHEN\},$$

$$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$$

$$P =$$

$$\{S \rightarrow if\ B\ IFTHEN\ B$$

$$IFTHEN \rightarrow if\ B\ B\ B$$

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

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

$$\}$$

$$\}$$

The probability of success for $G_{6m-if-then}$ was found to be 63%.

The Grammar $G_{6m-if-address-then}$

This grammar extends $G_{6m-if-then}$ by forcing the first if function to have an address value in its first position.

$$G_{6m-if-address-then} =$$

$$\{S,$$

$$N = \{B, T, ADDRESS, IFTHEN\},$$

$$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$$

$$P =$$

$$\{S \rightarrow if\ ADDRESS\ IFTHEN\ B$$

$$ADDRESS \rightarrow a0 \mid a1$$

$$IFTHEN \rightarrow if\ B\ B\ B$$

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

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

}

}

The probability of success for $G_{6m-if-address-then}$ was found to be 80%.

The Grammar $G_{6m-if-a0-if-a1}$

This grammar extends $G_{6m-if-address-then}$ by forcing all programs to have a 2 level if structure where both $a0$ and $a1$ are tested to determine which of the remaining if branches are selected. Hence, the grammar has captured the main concept of the problem - a process that selects some action based on the values of $a0$ and $a1$.

$G_{6m-if-a0-if-a1} =$

{S,

$N = \{B, T, IFA1\}$,

$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}$,

$P =$

{$S \rightarrow if\ a0\ IFA1\ B$

$IFA1 \rightarrow if\ a1\ B\ B$

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

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

}

}

The probability of success for $G_{6m-if-address-then}$ was found to be 88%.

4.1.10 Discussion of Language Bias

Adjusting the grammar so that it more closely represents the believed solution demonstrates the use of declarative bias to modify the search space. This is important, as it gives a clear statement of bias which is external to the learning system. Explicitly stating a grammar gives an easily understood representation of the possible language constructs that are being explored. Additionally, the language bias is easily changed without altering other components of CFG-GP and so promotes the exploration of various structures by the user when a solution is not forthcoming.

Language Bias	
Grammar	Probability of Success p_s
G_{6m}	34%
$G_{6m-popbias}$	60%
G_{6m-if}	37%
$G_{6m-if-address}$	62%
$G_{6m-if-then}$	63%
$G_{6m-if-address-then}$	80%
$G_{6m-if-a0-if-a1}$	88%

Table 4.3: Summary of Language Bias.

4.1.11 Conclusion

The results obtained by modifying the grammar G_{6m} are shown in Table 4.3.

The previous sections have demonstrated the use of declarative bias in the framework of the program induction system, CFG-GP, defined in Chapter 3. The ability to express a search and language bias within a declarative framework is an addition to current evolutionary program induction techniques.

4.2 Search Bias and Recursion

4.2.1 Introduction

This section will describe an example of biasing the search space using directed mutation. Using a simplified version of LISP as the language bias, CFG-GP is required to evolve a solution to the member function. The examples use a grammar, $G_{lisp-member}$, to describe the language under consideration. The directed mutation will be used in two ways; to repair statements that are tautologies, and to detect a pattern used to indicate that a recursive call to *member* should be attempted.

4.2.2 The Grammar $G_{lisp-member}$

The grammar expresses a language which defines a single function. This function is defined as $member(x, y)$, where x is an integer and y is a list of integers. The function returns $true$ if $x \in y$ and $false$ otherwise. The grammar creates functions which are a list of conditional statements. The *member* function in the grammar represents a recursive call to the function as a whole, evaluated intensionally (i.e. by execution) rather than extensionally (i.e. from the data).

$$
\begin{array}{l}
G_{lisp-member} = \\
\quad \{S, \\
\quad\ N = \{M, EXPN, AL\}, \\
\quad\ \sum = \{cond, atom, eq, member, x, y, true, nil, car, cdr\}, \\
\quad\ P = \\
\quad\quad \{S \rightarrow M \\
\quad\quad\ M \rightarrow cond\ EXPN\ EXPN\ M \mid \epsilon \\
\quad\quad\ EXPN \rightarrow atom\ AL \mid eq\ AL\ AL \mid member\ x\ AL \mid true \mid nil \\
\quad\quad\ AL \rightarrow car\ AL \mid cdr\ AL \mid x \mid y \\
\quad\quad \} \\
\quad \}
\end{array}
$$

4.2.3 Initial System Specification

The initial setup used for the membership function is shown in Table 4.4. A population size of 1000 is used. Two selective crossover operations are defined and represent large scale crossover ($\otimes = \{M, EXPN\}$) and crossover near the tips of the derivation tree($\otimes =$

The Membership Function	
Parameters	Specifications
POPULATION SIZE	1000
CREATE MAXIMUM DEPTH 5	300
CREATE MAXIMUM DEPTH 6	300
CREATE MAXIMUM DEPTH 7	300
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	10
$\otimes = \{M, EXPN\}$	90%
$\otimes = \{AL\}$	90%
GENERATIONS	50
FITNESS MEASURE	Square of the error over 20 test functions $(0 - 400)$
GRAMMAR	$G_{lisp-member}$

Table 4.4: The Initial Membership System.

$\{AL\}$). The training set consisted of ten true and ten false membership examples, as follows.

```
member( 1, [ 1  nil ] ) :-  true
member( 1, [ 2 [ 1  nil ]] ) :-  true
member( 1, [ 2 [ 3 [ 1 nil ]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 1 nil ]]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 1 nil ]]]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 1 nil ]]]]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 1 nil ]]]]]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 1 nil ]]]]]]]] ) :- true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 9 [ 1  nil ]]]]]]]]] ) :-  true
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 9 [ 2 [ 1  nil ]]]]]]]]]] ) :-  true
member( 1, [ 6  nil ] ) :-  nil
member( 1, [ 3 [ 6  nil ]] ) :-  nil
member( 1, [ 2 [ 3 [ 6 nil ]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 6 nil ]]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 nil ]]]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 nil ]]]]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 nil ]]]]]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 9  nil ]]]]]]]] ) :- nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 9 [ 2  nil ]]]]]]]]] ) :-  nil
member( 1, [ 2 [ 3 [ 4 [ 5 [ 6 [ 7 [ 8 [ 9 [ 2 [ 3  nil ]]]]]]]]]] ) :-  nil
```

The Membership Function with Mutation	
Parameters	Specifications
POPULATION SIZE	1000
CREATE MAXIMUM DEPTH 5	300
CREATE MAXIMUM DEPTH 6	300
CREATE MAXIMUM DEPTH 7	300
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	10
$\otimes = \{M, TEST, RESULT\}$	90%
$\otimes = \{AL\}$	90%
$eq(x,x) \rhd eq(x, car(y))$	100%
$eq(y,y) \rhd eq(x, car(y))$	100%
$eq(x, car(cdr(cdr(y)))) \rhd member(x, cdr(y))$	50%
$eq(car(cdr(cdr(y))), x) \rhd member(x, cdr(y))$	50%
GENERATIONS	50
FITNESS MEASURE	Square of the error over 20 test functions $(0 - 400)$
GRAMMAR	$G_{lisp-member}$

Table 4.5: The Membership System with Directed Mutation.

The standardised fitness was based on the number of errors that occurred with the training set. The maximum error is twenty (where a stack overflow has occurred for each example) and minimum zero. This error was squared to give a selection bias towards programs that could correctly identify membership in position 1 and position 2 and etc. The resulting probability of success, based on 100 runs, was 7%.

4.2.4 Using Directed Mutation

The low probability of success for $member(x, y)$ may be due to several reasons. One reason relates to the complexity of the programs that are generated. Typically, many programs will incorporate tautologies, such as $eq(x, x)$; these clutter up the search space and limit the effectiveness of the crossover operators. Directed mutation may be used to mutate these less useful expressions into potentially useful ones, as follows.

$$EXPN \Rightarrow eq\ AL\ AL \Rightarrow eq\ x\ AL \Rightarrow eq\ x\ x$$

$$\rhd$$

$$EXPN \Rightarrow eq\ AL\ AL \Rightarrow eq\ x\ AL \Rightarrow eq\ x\ car\ AL \Rightarrow eq\ x\ car\ y$$

This was applied with a probability of 100%. The equivalent pattern for $eq(y, y)$ was also

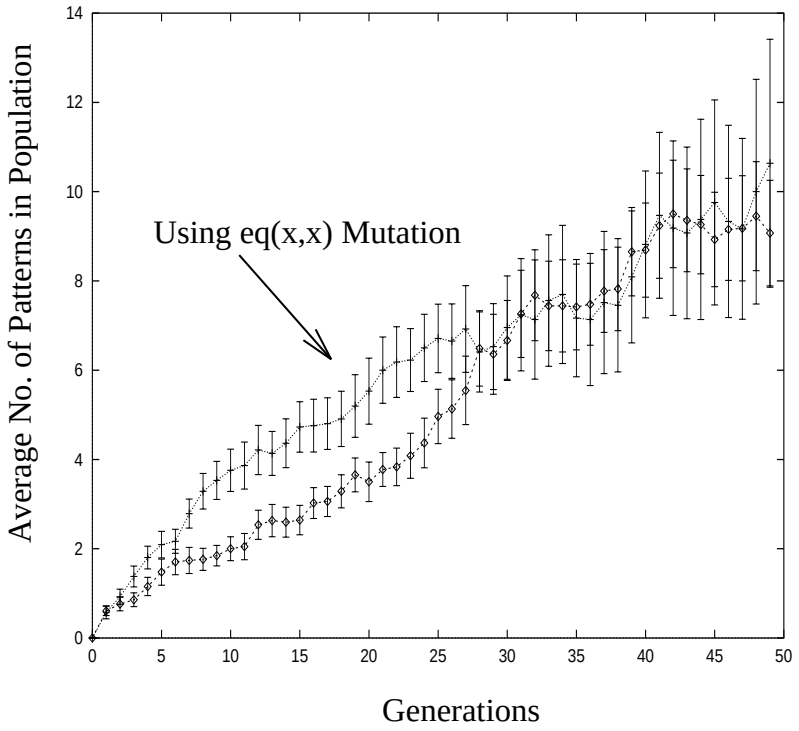


Figure 4.6: Average Number of Recursive Patterns.

defined, as follows.

$$EXPN \Rightarrow eq\ AL\ AL \Rightarrow eq\ y\ AL \Rightarrow eq\ y\ y$$

$$\triangleright$$

$$EXPN \Rightarrow eq\ AL\ AL \Rightarrow eq\ x\ AL \Rightarrow eq\ x\ car\ AL \Rightarrow eq\ x\ car\ y$$

A second cause of the low probability of success has to do with the discovery of a recursive pattern. The population gradually discovers long chains of functions (e.g. $car(cdr(cdr(cdr...))$ that attempt to step along the list to test for membership. However, the discovery of the corresponding recursive call does not occur because it is syntactically distant from the chain of functions. Directed mutation is used to alter the distance metric of the search space so that recursion is now close to these chains, as follows.

$$EXPN \Rightarrow eq\ AL\ AL \Rightarrow eq\ x\ AL \Rightarrow eq\ x\ car\ AL \Rightarrow eq\ x\ car\ cdr\ AL$$

$$\Rightarrow eq\ x\ car\ cdr\ cdr\ AL \Rightarrow eq\ x\ car\ cdr\ cdr\ y$$

$$\triangleright$$

$$EXPN \Rightarrow member\ x\ AL \Rightarrow member\ x\ cdr\ AL \Rightarrow member\ x\ cdr\ y$$

The equivalent pattern was also defined for the reverse ordering of the arguments to eq. These mutations were each applied with a probability of 50%. Such a high mutation rate was needed because the patterns do not appear very often in the population. Figure 4.6 shows the average and standard deviation of the number of recursive patterns in a

The $member(x, y)$ Function	
Mutation Operators	Probability of Success p_s
No Directed Mutation	7%
$eq(x, x) \triangleright eq(x, car(y))$	34%
$eq(x, car(cdr(cdr(y)))) \triangleright member(x, cdr(y))$	23%
Both Mutations	48%

Table 4.6: Results of Combining Directed Mutations for $member(x, y)$.

population of 1000, based on 100 runs. The lower line shows the number of patterns when no directed mutation operator is applied. The upper line shows the number of patterns when the directed mutations, $eq(x, x) \triangleright eq(x, car(y))$ and $eq(y, y) \triangleright eq(x, car(y))$, are applied. These mutations increase the likelihood of recursive patterns occuring in the population before generation 30.

The setup is shown in Table 4.5. The directed mutations were treated in two groups; the tautologies (labelled as $eq(x, x)$) and the recognition of function chains (labelled as $eq(x, car(cdr(cdr(y)))))$. The results for the various combinations of directed mutation are shown in Table 4.6. The improvement from the base case of no mutation to each combination of directed mutations is significant at the 95% one-sided confidence level (see Appendix B). The improvement from either of the single mutations to the combined mutation is also significant.

4.2.5 Discussion

The use of two directed mutations has improved the likelihood of CFG-GP discovering a definition for the $member(x, y)$ function. One possible explanation for this behaviour is that the first directed mutation seeds the population with building blocks that will create the intermediate step towards a recursive definition. The second directed mutation can exploit the increased bias towards the pattern used for recursion. The importance of the presented mutations is that they transparently express transformations which may improve the ability of CFG-GP to discover a solution. The directed mutations may also be viewed as a generalisation of the *editing operator* (see Section 2.3.4), first described by Koza[44].

4.2.6 Conclusion

This section has demonstrated the use of directed mutation to change the search bias. The mutations were shown to be able to recognise complex patterns in the population and to vary these explicitly. The mutations demonstrated two types of patterns that may be useful to identify and manipulate. The first mutation performed an editing task to

remove program components that are trivially true. The second mutation recognised a repeated pattern that was used to indicate that a recursive call should be attempted.

Directed mutation is a powerful and general operator which should be applicable in many problem domains. For example, in a spatial application the detection of repeated chains of adjacency operations may imply that a distance operator should replace these repeated functions. A second situation could occur with river modelling, where a chain of adjacencies may be mutated to an *upstream* or *downstream* function. The power of directed mutation comes from its declarative specification, probabilistic application and ability to represent and modify complex patterns based on the user-defined grammar.

4.3 The Prediction of Greater Glider Density

Each like a dormouse sleeps
In the spout of a gumtree old,
A ball of fur with a sliver coat;
Each with his tail around his throat
For fear of his catching cold.

A.B.Paterson ("Banjo")[5]

4.3.1 Introduction

Spatial information, in the form of map data from remote sensing and survey work, has increased dramatically over the last decade. The advent of geographic information systems (GIS) as a valuable tool for modelling and representing this data has further promoted the collection and use of spatial information. Machine-based learning systems offer the possibility to generalise the patterns and processes represented by this information.

Most learning systems assume that a propositional description of the problem is adequate for representation. However, the inherent *spatial relations* that exist within GIS data make it attractive to use a functional or relational description with learning.

This section describes the application of CFG-GP to predict the density of an Australian marsupial[6]. This data has been studied previously [71] using various propositional learning and regression techniques. These Australian marsupials, technically named *Petauroides volans*, provide an appropriate area of study as their abundance is a good measure of the health of mature forests in South Eastern Australia [71]. The inherent difficulty in creating good survey information about tree-dwelling marsupials means that information from one study area must be used to learn predictions for other, similar, locations.[7] The potential exists for learning systems to identify underlying patterns representing properties of the environment. The construction of classification and explanatory models is a valuable tool for the natural resource scientist.

4.3.2 The Greater Glider Study Area

As described by Stockwell et al.[71]:

[5] From the book "The Animals Noah Forgot".
[6] A brief description of the Greater Glider is given in Appendix D.
[7] This assumption implies that the *cost* of collecting the independent data is less than the work involved in collecting the relevant dependent variable. This is often true with prediction of animal habitats, as the animals involved may be difficult to locate accurately, are often nocturnal and live underground or in dense vegetation.

> This study was based in Coolangubra State Forest, located in south-eastern New South Wales of Australia. The 22,000-ha forest contains both tableland and foothill eucalypt forest and is important to wildlife, especially arboreal marsupials [36]. The study area, 1600 ha in extent, was located at Waratah Creek in tableland forest.

Figure 4.7 shows the predictor variables that were used to determine the glider density pattern. The forest inventory described the data in grid form with each designated area having dimensions of 200×200 *mtr*. The data of Figure 4.7 represents a 20×20 map of areas for each of the independent and dependent variables. Each variable was stored as a categorical value whose meaning is shown in the keys displayed in Figure 4.7.

Stockwell applied a number of different methods to analyse this data, including the following.

- Multiple regression to produce a linear model (MR).
- Principal component analysis leading to a decision table (PCA).
- Knowledge acquisition leading to a rule based expert system (KA).
- Machine induction of decision trees by the algorithms ID3, CN2 and CART.

All of these models may be thought of as dividing the data into separate classes. These systems represent learning by dividing the possible decisions based on variable values and logical combinations of these variables. The language is propositional.

ID3 attempts to find the variable that best divides the set of values into homogeneous classes. This is applied repeatedly until the goal class (i.e. glider density) is completely described. The result of applying ID3 may be viewed as a decision tree, as shown in Figure 4.8. The CN2 system uses a conjunction of values to discriminate each glider class, whereas CART uses bifurcations of a decision tree, where a number of values of a single variable may be used to divide the set of objects. This tree is then pruned to create a general description of the goal.

One observation about the glider density data is that there are limitations to the actual predictivity based upon the simple local descriptions of space that are used. To illustrate this point, Stockwell developed two additional models as a guide to validating the decision tree models. These represented the most *specific* (MSC) and most *general* (MGC) models that could be created using the language forms available to the induction techniques. These measures give an upper and lower bound to the inductive predictivity based on the available local information.

The models were trained by dividing the 400 locations into two groups. The first group of 200 locations was used to train the models. The second group of 200 was used to test

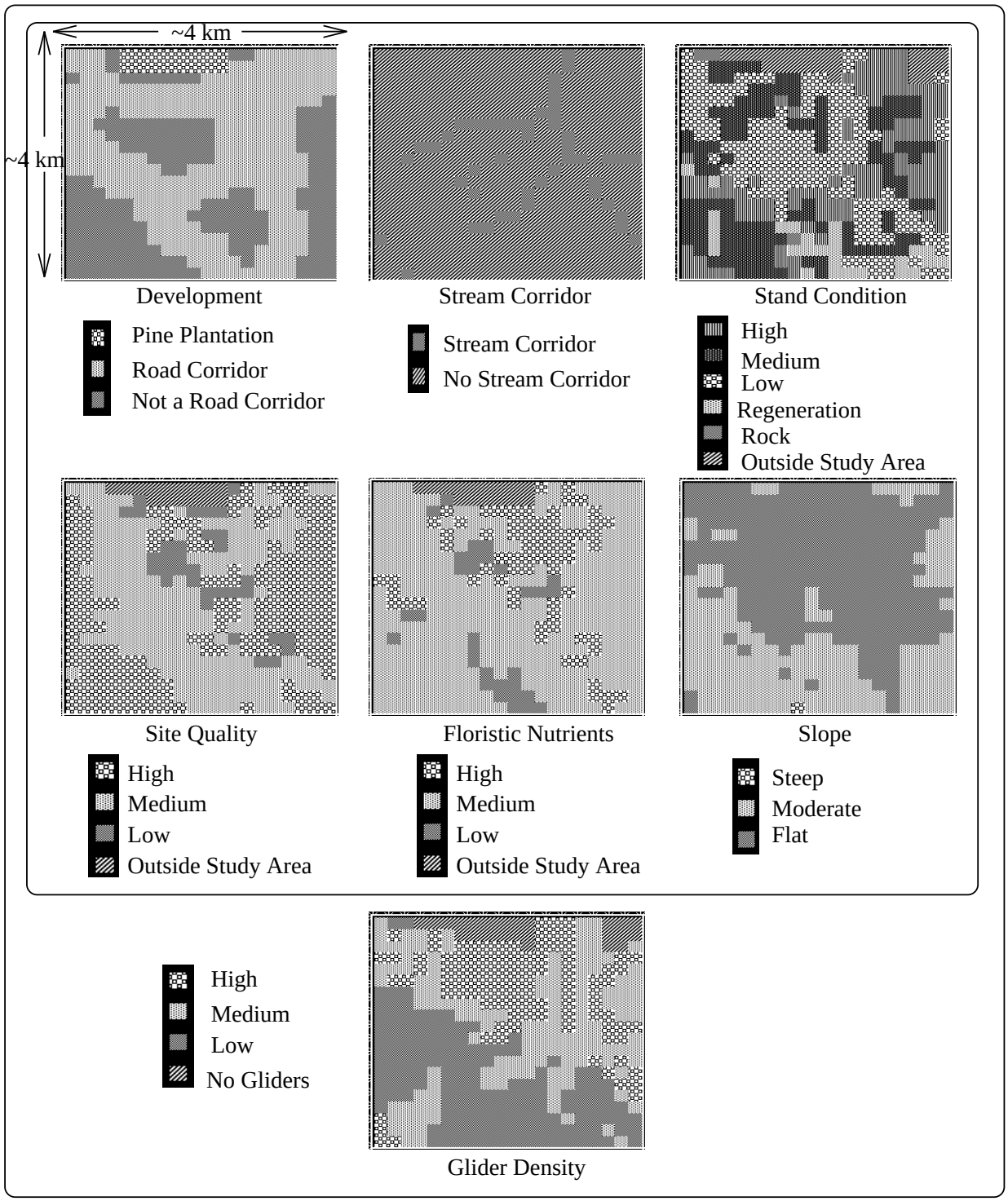


Figure 4.7: The Glider Density Independent and Dependent Data.

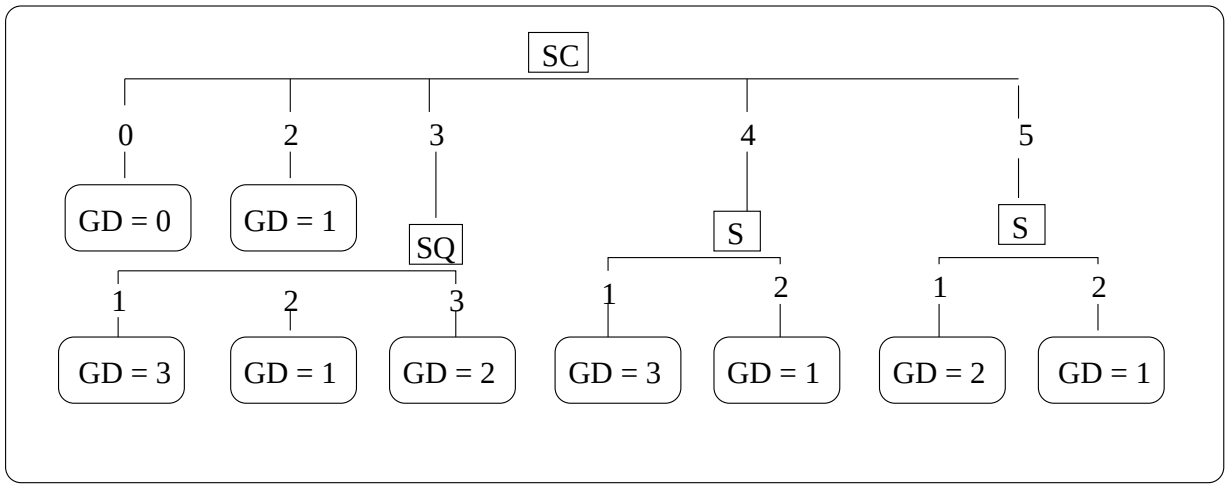


Figure 4.8: A Decision Tree created by ID3 to predict Glider Density.

how well the models applied to unseen data. This is generally accepted as a measure of how well the models have learnt a concept. The locations were divided into six random collections of 200 learning and 200 test sites. This was used to compare the predictive accuracy of each inductive technique. The measures of accuracy could then be averaged over the six random splits. Estimates of the standard deviation for each model could be obtained from the results of the six replicated random samples.

A subtle issue arises when using a single map (as described above) to create the training and test data. Normally it is desirable to use independent data for the training and test sets. This ensures that an overly-specific theory (based on the training data) will not achieve good results with the test data. The method that Stockwell has used (and that is used in this thesis) does not account for the fact that locations in space are *not independent*. This is easily understood by the fact that nearby locations in space will be related to each other due to the continuous nature of space and that geographic space tends to vary slowly. The main affect of this lack of independence will be that the error in the learnt theories, either those quoted by Stockwell or using CFG-GP, will be underestimated. Although this will mean the absolute values for the test results will be greater than would occur for truly independent training and test data, the order of significance between each learning method should remain the same. This may be justified by noting that each learning method attempts to maximise their performance based on the training data. Hence, although each method may be different in its approach to developing a theory, it is likely that each method would exploit the lack of independence with the test data in a similar manner. Since the goal of this section is to demonstrate the use of bias and functional learning, this issue is not overly significant. It should be noted, however, that the issue of independent training and test data, when using spatial information, is worthy of further research.

Model Accuracy		
Method	Training	Test
MR	-	45%
PCA	-	41%
KA	-	48%
MGC	$36.7 \pm 0.8\%$	$33.2 \pm 1.6\%$
ID3	$60.7 \pm 1.3\%$	$57.3 \pm 2.2\%$
CN2	$50.8 \pm 2.6\%$	$45.2 \pm 2.2\%$
CART	$61.2 \pm 3.5\%$	$54.8 \pm 3.9\%$
MSC	$75.8 \pm 1.2\%$	$48.3 \pm 1.6\%$

Table 4.7: Comparison of Model Accuracy for Greater Glider Density [71].

The results are shown in Table 4.7, where the training and test results indicate the percentage of correct glider density predictions for the 200 locations. The MSC classification represents the most specific description based on the training data. This will achieve the maximum possible predictive capability (on the training data) because it is the most specific description of the data classes. The result using the training data shows that approximately 25% of the glider predictions cannot be uniquely distinguished. This is due to different density classes having the same set of local independent attributes[8]. The drop in predictability of MSC to 48% for the test data shows that over-specialisation does not produce useful models with unseen data. The ID3 and CART learning methods out-perform this specialisation on the test data by finding general statements about the glider density.

4.3.3 The Grammar G_{ggd} for the Greater Glider

The previous methods have used a propositional description for classification. To commence this study, a language that approximates the decision tree format will be described. This initial language will attempt to recreate the results using similar expressions to previous work. This enables the comparison between the previous work and the learning system defined in Chapter 3.

The following grammar was used to describe the initial language, which attempted to mimic the propositional components of the previous decision tree systems.

$$
\begin{aligned}
&G_{ggd} = \\
&\quad \{S, \\
&\qquad N = \{GDRES, GDVALUE, B, EXPN, DEV, STREAM, \\
&\qquad\quad STAND, QUALITY, FLORISTIC, SLOPE, REL, DEVVAL,
\end{aligned}
$$

[8] Using all 400 surveyed areas it is only possible to uniquely distinguish 282 (70.5%) of the glider density sites. The value of 75.8% given for the MSC method is due to the separation of data into random groups of 200.

$$STVAL, STANDVAL, QUALITYVAL, FLORISTICVAL, SLOPEVAL\},$$

$$\sum = \{no\_gliders, \ldots, rock, \ldots, steep, ifelse, and, or, not, <, >, <=, >=, =\}$$

$$P =$$

$$\{S \rightarrow GDRES$$

$$GDRES \rightarrow ifelse\ B\ GDVALUE\ GDRES \mid GDVALUE$$

$$GDVALUE \rightarrow no\_gliders \mid low\_gliders \mid med\_gliders \mid high\_gliders$$

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

$$EXPN \rightarrow DEV \mid STREAM \mid STAND$$

$$EXPN \rightarrow QUALITY \mid FLORISTIC \mid SLOPE$$

$$DEV \rightarrow dev\ REL\ DEVVAL$$

$$STREAM \rightarrow st\ REL\ STVAL$$

$$STAND \rightarrow sc\ REL\ STANDVAL$$

$$QUALITY \rightarrow sq\ REL\ QUALITYVAL$$

$$FLORISTIC \rightarrow fn\ REL\ FLORISTICVAL$$

$$SLOPE \rightarrow sl\ REL\ SLOPEVAL$$

$$REL \rightarrow <|>|<=|>=|=$$

$$DEVVAL \rightarrow no\_road \mid road\_corridor \mid pine\_plantation$$

$$STVAL \rightarrow no\_stream\_corridor \mid stream\_corridor$$

$$STANDVAL \rightarrow outside\_study \mid rock \mid regeneration \mid low \mid med \mid high$$

$$QUALITYVAL \rightarrow outside\_study \mid low \mid med \mid high$$

$$FLORISTICVAL \rightarrow outside\_study \mid low \mid med \mid high$$

$$SLOPEVAL \rightarrow flat \mid moderate \mid steep$$

$$\}$$

$$\}$$

The grammar G_{ggd} will create a decision tree represented as a series of $if - then - else$ statements.

4.3.4 Initial System Specification

Table 4.8 shows the setup for determining the glider density predictions using the propositional grammar G_{ggd}. The $MAX\ DEPTH\ PROGRAMS$ parameter was set so that *at least* four levels of **if-then-else** were able to be created. This ensured that a program could be formed that described each of the four glider density classes.

The Propositional Glider Density System	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 6	200
CREATE MAXIMUM DEPTH 7	200
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	10
$\otimes = \{GDVALUE, GDRES, EXPN\}$	90%
GENERATIONS	50
FITNESS MEASURE	200 Training Sites
GRAMMAR	G_{ggd}

Table 4.8: The Initial Glider Density System

Model Accuracy over 6 Random Training and Test Sites	
Training	Test
58.5%	53.5%
58.0%	53.0%
58.5%	57.0%
57.5%	53.5%
55.5%	51.5%
59.5%	46.5%
57.9±1.4%	52.5±3.4%

Table 4.9: Results using $L(G_{ggd})$ for Glider Prediction.

4.3.5 Results

The system was run 100 times for each of the six random collections of 200 training locations. These random collections of locations are shown in Figure 4.9. From the 100 runs the program that performed best over the training examples was selected as the predictor of glider density. This program was then tested on the remaining 200 test locations to give a measure of performance on unseen data. The results for each of these six (best) runs is shown in Table 4.9. The final entry shown in this table gives the average and standard deviation for the six runs. Referring back to Table 4.7, it can be seen that the results are comparable in accuracy to the previous decision tree methods.

The glider density map produced by the best test program is shown in Figure 4.10. The percentage of correctly predicted glider values over the 400 locations is 57.7%. The associated program is shown below.

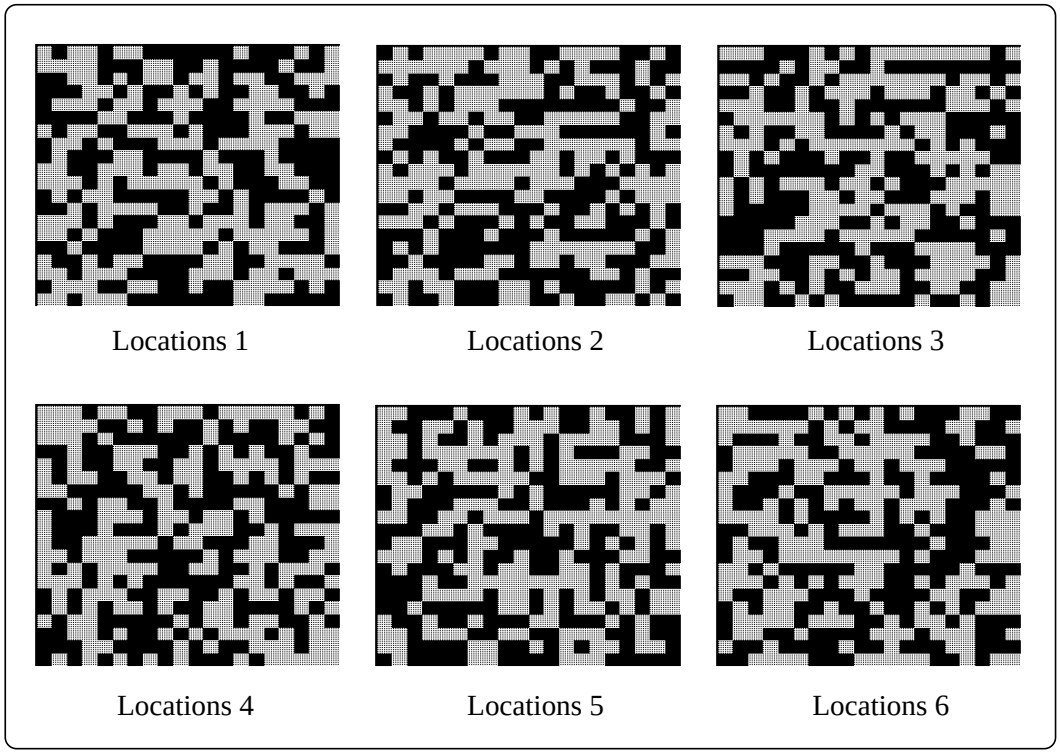


Figure 4.9: The Six Random Collections of 200 Training and Test Locations.

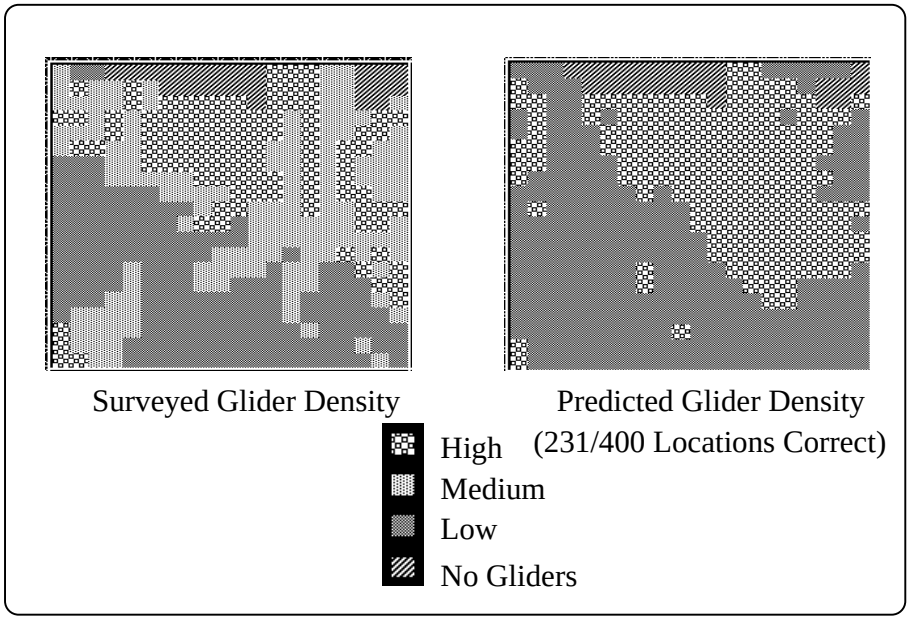


Figure 4.10: The Best Prediction of Glider Density using G_{ggd}.

```
ifelse(and(dev(=,pine_plantation),
           or(sq(<,high),dev(=,pine_plantation))),no_gliders,

ifelse(not(sl(=,flat)),low_gliders,

ifelse(sc(=,outside_study),no_gliders,

ifelse(and(fn(=,medium),sq(<,high)),low_gliders,
/* otherwise */
         high_gliders))))


Training: 58.5%
Test    : 57.0%

where sc: stand condition
      sl: slope
      st: stream corridor
     dev: development
      fn: floristic nutrients
```

The main criticism of this solution is that it does not cover all glider density classes. For example, there is no prediction for medium_glider density. A number of restrictions to the structure of the language may be made by observing the following.

- All glider density values are represented in the survey.
- The survey sites **outside the study area** for stand condition have been set to a glider density of **no_gliders**. This value is not a true reflection of the glider density and, therefore, introduces some noise into the prediction. The original work of Stockwell used the stand condition value of outside_study as a legitimate attribute for prediction. Since this occurs for all sites where no gliders occur the previous learning systems have used this attribute as the predictor for the **no_glider** density class. This is a false prediction in that there is no meaning associated with the *outside study attribute* and the actual preferred habitats of the greater glider. This has the secondary effect of creating a predictive accuracy that is greater than that actually represented by the developed theories, since no meaningful statements are made about conditions where no gliders are likely to be found. The advantage of

a declarative bias is that this noise may be explicitly stated, thus making it clear that the developed predictor of glider density does not make any statement about locations where gliders will not be found.

- The *glider density* is **low** when the *stand condition* is **rock**.
- A greater glider has certain preferred habitat sites based on non-local spatial conditions. For example, the availability of preferred food sources a short distance from suitable nesting sites.

The following section explores changes to the basic propositional language that attempt to incorporate these concepts in a declarative manner.

4.3.6 Modifying the Language Bias

The Grammar $G_{ggd-cover}$

A covering grammar $G_{ggd-cover}$ is defined which ensures that each glider class is represented.

$$
\begin{aligned}
&G_{ggd-cover} = \\
&\quad \{S, \\
&\quad N = \{IFELSE1, IFELSE2, B, EXPN, DEV, STREAM, \\
&\qquad STAND, QUALITY, FLORISTIC, SLOPE, REL, DEVVAL, \\
&\qquad STVAL, STANDVAL, QUALITYVAL, FLORISTICVAL, SLOPEVAL\}, \\
&\quad \sum = \{no\_gliders, \ldots, rock, \ldots, steep, ifelse, and, or, not, <, >, <=, >=, =\} \\
&\quad P = \\
&\qquad \{S \rightarrow ifelse\ B\ no\_gliders \text{ IFELSE1} \\
&\qquad IFELSE1 \rightarrow ifelse\ B\ low\_gliders\ IFELSE2 \\
&\qquad IFELSE2 \rightarrow ifelse\ B\ med\_gliders\ high\_gliders \\
&\qquad B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid EXPN \\
&\qquad EXPN \rightarrow DEV \mid STREAM \mid STAND \\
&\qquad\qquad \ldots \\
&\qquad SLOPEVAL \rightarrow flat \mid moderate \mid steep \\
&\qquad \} \\
&\}
\end{aligned}
$$

The language $L(G_{ggd-cover})$ gives a bias requiring functions to be created that cover some part of each glider density class. The system setup is shown in Table 4.10. This setup

The Propositional Glider Density System	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 7	200
CREATE MAXIMUM DEPTH 8	300
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	12
$\otimes = \{EXPN\}$	90%
GENERATIONS	50
FITNESS MEASURE	200 Test, 200 Training Sites
GRAMMAR	$G_{ggd-cover}$

Table 4.10: The Covering Glider Density System.

Model Accuracy over 6 Random Training and Test Sites	
Training	Test
62.0%	51.0%
60.0%	58.5%
59.5%	58.5%
59.5%	58.5%
61.0%	57.0%
61.5%	52.0%
60.6±1.1%	55.9±3.5%

Table 4.11: Results using $L(G_{ggd-cover})$ for Glider Prediction.

differs from the initial setup of Table 4.8 in that the creation parameters have an increased maximum depth. This is due to the change in grammatical structure. The results using $L(G_{ggd-cover})$ are shown in Table 4.11 for each of the six random test and training runs.

The best program over the six random collections of 200 test sites[9] is shown below. The predicted glider values for this program are shown in Figure 4.11. This program correctly predicts 59.25% of the 400 locations from the surveyed area.

```
ifelse(sc(<=,outside_study),no_gliders,

ifelse(or(and(or(dev(<=,road_corridor),sc(<,regeneration)),
              or(sl(>=,moderate),and(sq(=,medium),fn(<=,medium)))),
```

[9]The best program was defined as the one with the highest test and training score.

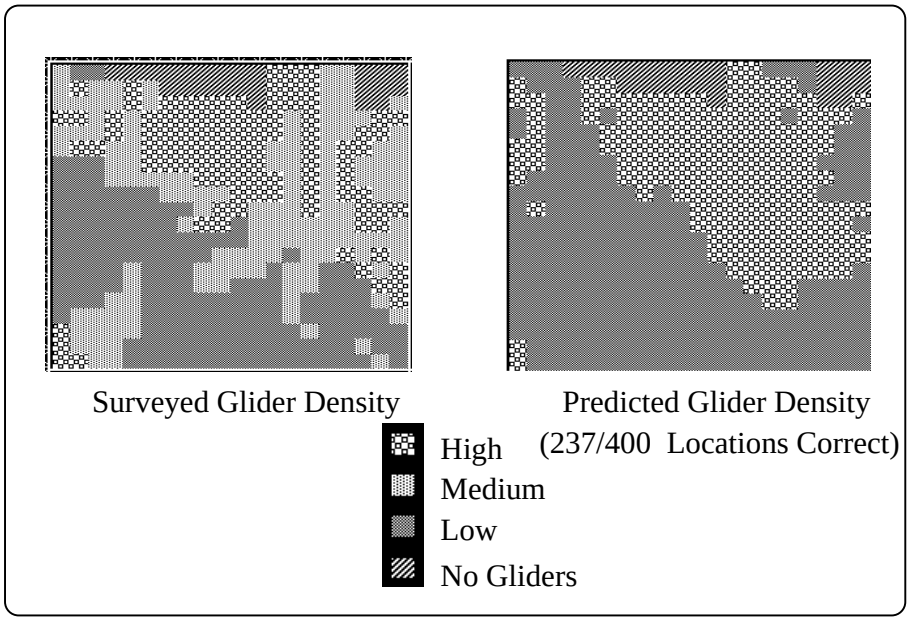


Figure 4.11: The Best Prediction of Glider Density using $G_{ggd-cover}$.

```
          sc(<,1)),low_gliders,

ifelse(and(and(or(and(fn(<=,low),sc(<,regeneration)),sl(>=,flat),
                      sc(<,rock)),sc(>=,regeneration)),med_glider,
    /* otherwise */
        high_gliders)))


Training: 60.0%
Test    : 58.5%

where sc: stand condition
      sl: slope
      sq: site quality
      st: stream corridor
     dev: development
      fn: floristic nutrients
```

There has been an overall (test) improvement from $52.5 \pm 3.4\%$ (G_{ggd}) to $55.9 \pm 3.5\%$ ($G_{ggd-cover}$). This result is due to the language having a bias that forms a covering expression over each glider density class.

The Grammar $G_{ggd-cover+no\_gliders}$

The grammar $G_{ggd-cover}$ does not explicitly state that the *stand condition* may be used to determine all locations which have *no_gliders* as a value. To demonstrate this bias the grammar $G_{ggd-cover+no\_gliders}$ is defined which explicitly forces the *stand condition* to be used as the *only determining factor* with *no_gliders*. This narrows the possible expressions for the language and forces the search to proceed over other glider density classes.

$$
\begin{aligned}
&G_{ggd-cover+no\_gliders} = \\
&\quad \{S, \\
&\quad N = \{BSC0, IFELSE1, IFELSE2, B, EXPN, DEV, STREAM, \\
&\qquad STAND, QUALITY, FLORISTIC, SLOPE, REL, DEVVAL, \\
&\qquad STVAL, STANDVAL, QUALITYVAL, FLORISTICVAL, SLOPEVAL\}, \\
&\quad \sum = \{no\_gliders, \ldots, rock, \ldots, steep, ifelse, and, or, not, <, >, <=, >=, =\} \\
&\quad P = \\
&\qquad \{S \rightarrow ifelse\ BSC0\ no\_gliders\ \text{IFELSE1} \\
&\qquad BSC0 \rightarrow sc = outside\_study \\
&\qquad IFELSE1 \rightarrow ifelse\ B\ low\_gliders\ IFELSE2 \\
&\qquad IFELSE2 \rightarrow ifelse\ B\ med\_gliders\ high\_gliders \\
&\qquad B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid EXPN \\
&\qquad EXPN \rightarrow DEV \mid STREAM \mid STAND \\
&\qquad\qquad \ldots \\
&\qquad SLOPEVAL \rightarrow flat \mid moderate \mid steep \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

The results using this grammar are shown in Table 4.12.

The best program over the six random collections of 200 test sites is shown below. The predicted glider values for this program are shown in Figure 4.12. The program correctly classifies 61.8% of the 400 surveyed locations.

```
ifelse(sc(=,outside_study),no_gliders,

ifelse(or(and(sl(>=,moderate),or(sl(>=,moderate),sq(=,med))),
       and(and(sq(=,med),sc(<,med)),
```

Model Accuracy over 6 Random Training and Test Sites	
Training	Test
60.5%	53.0%
59.5%	58.5%
61.5%	62.0%
61.0%	57.5%
61.0%	57.5%
64.0%	54.0%
61.3±1.5%	57.1±3.2%

Table 4.12: Results using $L(G_{ggd-cover+no\_gliders})$ for Glider Prediction.

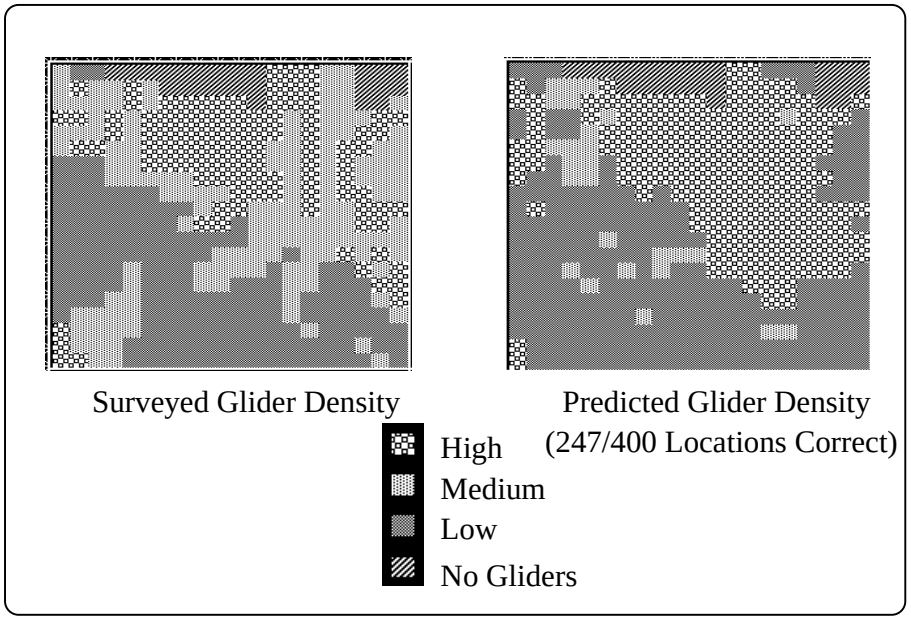


Figure 4.12: The Best Prediction of Glider Density using $G_{ggd-cover+no\_gliders}$.

```
          or(sl(>,flat),fn(<,high)))),low_gliders,

   ifelse(or(sl(>=,moderate),or(sl(>=,moderate),and(fn(<,high),
          and(or(sl(>=,moderate),st(<=,no_stream_corridor)),
          or(sl(>,flat),sq(=,med))))))),med_gliders,
 /* otherwise */
          high_gliders)))


   Training: 61.5%
   Test    : 62.0%

   where sc: stand condition
         sl: slope
         sq: site quality
         st: stream corridor
         fn: floristic nutrients
```

The Grammar $G_{ggd-spatial}$, Spatial Expressions, Typing and Bias

The previous examples have used a non-spatial description of the data. This assumes that the greater glider does not consider surrounding locations when selecting preferred habitat sites. This is obviously false with a mobile tree-dwelling marsupial that has requirements of food, water and shelter.

There are other reasons for considering spatial aspects with the glider data. An area that has highly desirable local attributes for glider habitation will possibly become overcrowded. This would force some greater gliders to move to nearby areas that do not have equally desirable local attributes, thus raising the population at these locations. This increased glider density is not due to the local attributes, but to the highly desirable attributes that are nearby. A second general effect that can be exploited, by allowing spatial aspects to be represented, is that of *surrogation between attributes*. Some attributes may be used as surrogates for attributes that have not been explicitly represented in the independent data. For example, the spatial distribution of slope gives an indication of general landscape features such as valley floors and hilltops. These landscape features have not been directed represented but are likely to be relevant attributes when predicting glider density. The ability to combine descriptions of attributes with some spatial extent is likely to allow these conditions to be (indirectly) considered.

The grammar $G_{ggd-spatial}$ is defined so that statements about the condition of locations

within some distance from some current location may be specified. The language defined by this grammar may express boolean combinations of spatial conditions. Additionally, the spatial conditions are based upon boolean combinations of the independent attributes.

$$
\begin{aligned}
&G_{ggd-spatial} = \\
&\quad \{S, \\
&\quad N = \{BSC0, SPAB, SPAEXMP, IFELSE1, IFELSE2, B, EXPN, DEV, \\
&\qquad STREAM, STAND, QUALITY, FLORISTIC, SLOPE, REL, DEVVAL, \\
&\qquad STVAL, STANDVAL, QUALITYVAL, FLORISTICVAL, SLOPEVAL\}, \\
&\quad \sum = \{no\_gliders, \ldots, rock, \ldots, steep, ifelse, and, or, not, <, >, <=, >=, =\} \\
&\quad P = \\
&\qquad \{S \rightarrow ifelse\ BSC0\ no\_gliders\ \text{IFELSE1} \\
&\qquad BSC0 \rightarrow sc = outside\_study \\
&\qquad IFELSE1 \rightarrow ifelse\ SPAB\ low\_gliders\ IFELSE2 \\
&\qquad IFELSE2 \rightarrow ifelse\ SPAB\ med\_gliders\ high\_gliders \\
&\qquad SPAB \rightarrow and\ SPAB\ SPAB \mid or\ SPAB\ SPAB \mid not\ SPAB \\
&\qquad SPAB \rightarrow spaeval\ B\ SPAEXP \\
&\qquad B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid EXPN \\
&\qquad SPAEXP \rightarrow current \mid within\ DISTANCE \\
&\qquad DISTANCE \rightarrow 1 \mid 2 \\
&\qquad EXPN \rightarrow DEV \mid STREAM \mid STAND \\
&\qquad\qquad \ldots \\
&\qquad SLOPEVAL \rightarrow flat \mid moderate \mid steep \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

The grammar $G_{ggd-spatial}$ defines a language that allows all possible boolean combinations of locations based on local and spatial conditions. It is worth noting that the spatial expressions introduce a typing constraint with the language. This language constraint is automatically handled by forcing the sites for selective crossover to be *matching nonterminals*. As an example, the following program is contained within the language $L(G_{ggd-spatial})$.

```
ifelse(sc(=,outside_study),no_gliders)
```

```
ifelse(or(spaeval(and(sc(=,regeneration),dev(>,1)),within(2)),
          sl(>,1)), 1,

      ...

 high_gliders)))
```

This program uses boolean functions to group attributes spatially and as collections of spatial functions. The ability of $L(G_{ggd-spatial})$ to express arbitrary boolean functions as part of a spatial condition make it impractical to create attributes representing each of these possible combinations.

The function *spaeval* B $SPAEXP$ evaluates the boolean expression B based on the locations represented by $SPAEXP$. If the boolean expression is true *for any* location selected by $SPAEXP$, *spaeval* is true. In the case of *current* the location where glider density is currently being determined is assumed. This allows the previous non-spatial conditionals to be expressed. The *within* $DISTANCE$ function approximates a circular spatial operation centred on the current location, as shown in Figure 4.13. For example, *within* 1 gives all adjacent locations. Further, the language is designed so that created programs have a structure that is easily read and understood. Initially, the $DISTANCE$ value has been limited to 1 or 2 for several reasons.

- The *cost of evaluation* dramatically increases as distance increases.
- The resolution of the glider data means that *within 2* gives approximately a $1km^2$ area. Although gliders may travel greater distances this description should be adequate for a good theory.
- The glider density data is inherently noisy. This noise has been introduced by the sampling methods used to collect the data and the broad classifications that are used for each attribute. As the distance function increases the number of locations selected by the *spaeval* function increases as $O(DISTANCE)^2$. The existence of some particular combination of attributes is likely to increase as $DISTANCE$ increases from the current location. These conditions, although true for the data, may be spurious as the uncertainty of each location is compounded by combining them to represent a condition for a single location. Hence, although a better predictor of glider density may be created by using larger $DISTANCE$ values, the described theory may not have a meaningful interpretation.

The setup for $G_{ggd-spatial}$ is shown in Table 4.13. There is a 90% probability of crossover occuring over a spatial boolean expression or a boolean condition. Additionally, there is

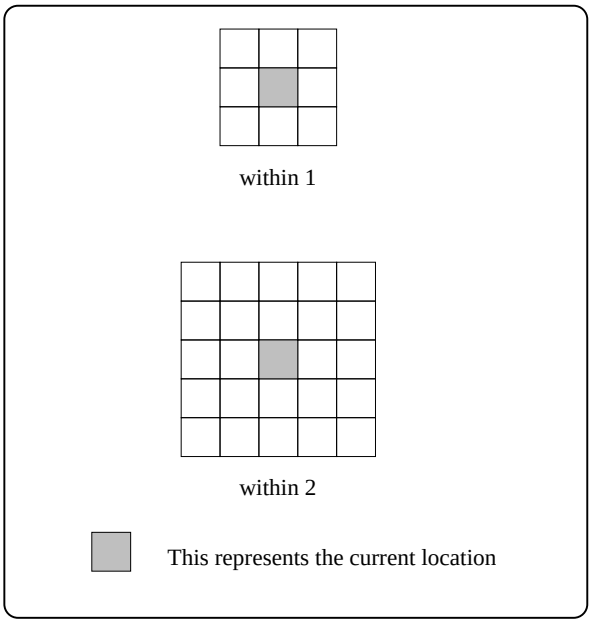


Figure 4.13: $G_{ggd-spatial}$ and the *within* $DISTANCE$ function.

The Spatial Glider Density System	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 8	300
CREATE MAXIMUM DEPTH 9	200
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	14
$\otimes = \{B, SPAB\}$	90%
$\otimes = \{SPAEXP\}$	5%
GENERATIONS	50
FITNESS MEASURE	200 Test, 200 Training Sites
GRAMMAR	$G_{ggd-spatial}$

Table 4.13: The Spatial (covering) Glider Density System.

Model Accuracy over 6 Random Training and Test Sites	
Training	Test
67.5%	62.0%
67.5%	63.0%
64.0%	64.5%
70.0%	65.0%
68.5%	67.5%
66.5%	62.5%
67.3±2.0%	64.1±2.0%

Table 4.14: Results using $L(G_{ggd-spatial})$ for Glider Prediction.

a 5% probability of crossover occuring at the $SPAEXP$ site. This crossover acts like a mutation, causing statements to change from a local to spatial expression (and vice versa).

4.3.7 Results

The best program over the 6 random collections of 200 training sites is shown below. The predicted glider values for this program are shown in Figure 4.14. The program correctly predicts 67.5% of the 400 surveyed glider density locations.

```
ifelse(sc(=,outside_study),no_gliders,

ifelse(spaeval(and(sc(=,regeneration),dev(>,no_road)),sqr2),low_gliders,

ifelse(and(or(spaeval(st(<,stream_corridor),current),
         and(spaeval(and(and(sc(=,regeneration),st(=,no_stream_corridor)),
         sc(=,regeneration),sqr2),spaeval(sc(=,regeneration),sqr2))),
         spaeval(and(sl(>=,moderate),st(<,no_stream_corridor)),sqr2)),med_gliders,

 /* otherwise */
    high_gliders)))


   Training: 70.0%
   Test    : 65.0%

   where sc: stand condition
         sl: slope
```

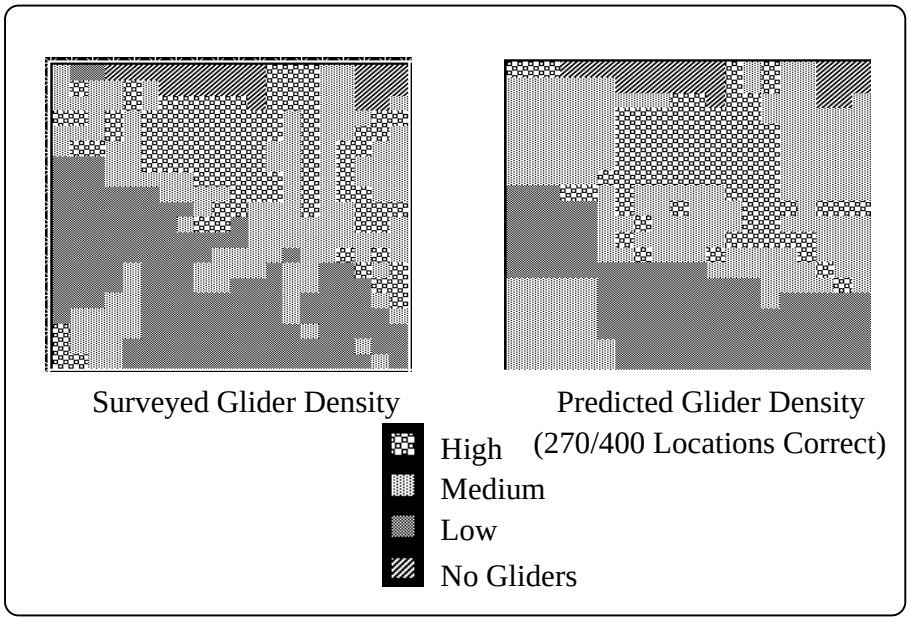


Figure 4.14: The Best Prediction of Glider Density using $G_{ggd-spatial}$.

```
dev: development
 st: stream condition
```

A number of points may be made about this result, due to the introduction of spatial functions as part of the description language.

- The prediction of glider density improves over the propositional language.
- The distance function is easily incorporated into the original G_{ggd} grammar.
- The typing constraints introduced by the spatial functions are automatically handled by the grammar and selective crossover operators.
- The resulting programs are often simpler, yet more powerful, because the included functions are more expressive.
- The resulting maps of prediction, such as figure 4.14, have a generalised form that implies the programs are representing a more fundamental theory about preferred glider habitats. For example, the previous solution implies that greater gliders are most likely to be found away from a road corridor and where the slope of the land is not too extreme. Additionally, these sites must occur within a short distance of preferred food sources.

4.3.8 Discovering a Good Solution

The previous examples have demonstrated improvements as the expressiveness of the language is extended. However, the population has been limited to 500 and the goal

The Spatial Glider Density System	
Parameters	Specifications
POPULATION SIZE	2000
CREATE MAXIMUM DEPTH 8	1000
CREATE MAXIMUM DEPTH 9	1000
MAXIMUM FAILURES	1000
MAX DEPTH PROGRAMS	15
$\otimes = \{B, SPAB\}$	90%
$\otimes = \{SPAEXP\}$	5%
GENERATIONS	100
FITNESS MEASURE	200 Test, 200 Training Sites
GRAMMAR	$G_{ggd-spatial-biased}$

Table 4.15: Extending the Population - Searching for a Good Solution.

has not been to achieve the best possible solution, but the best solution *given certain computational constraints*. The size of the search space represented by $L(G_{ggd-spatial})$ warrants a larger population and additional generations to improve the predictability of discovered programs. Table 4.15 shows the setup file for this experiment. The population has been increased to 2000 and the number of generations increased to 100. In addition, the $WITHIN$ function has been extended to allow distances up to 5 squares from the current location.

The grammar $G_{ggd-spatial-biased}$ has been biased to include two facts about the glider density survey.

(a) A *stand condition* of **outside_study** at the current location implies that *glider density* is **none** for this location.

(b) A *stand condition* of **rock** at the current location implies that the *glider density* is **low** for this location. There are other areas where *glider density* is **low** so this condition is represented as a disjunction.

The grammar $G_{ggd-spatial-biased}$ is defined as follows.

$$G_{ggd-spatial-biased} =$$

$$\{S,$$

$$N = \{BSC0, ORSPAB, SPAB, SPAEXMP, IFELSE1, IFELSE2, B, EXPN, DEV,$$

$$STREAM, STAND, QUALITY, FLORISTIC, SLOPE, REL, DEVVAL,$$

$$STVAL, STANDVAL, QUALITYVAL, FLORISTICVAL, SLOPEVAL\},$$

$$\sum = \{no\_gliders, \ldots, rock, \ldots, steep, ifelse, and, or, not, <, >, <=, >=, =\}$$

$$
\begin{aligned}
P = \\
&\{S \rightarrow ifelse\ BSC0\ no\_gliders\ \text{IFELSE1} \\
&\ BSC0 \rightarrow sc = outside\_study \\
&\ IFELSE1 \rightarrow ifelse\ ORSPAB\ low\_gliders\ IFELSE2 \\
&\ ORSPAB \rightarrow or\ ROCK\ SPAB \\
&\ ROCK \rightarrow sc = 1 \\
&\ IFELSE2 \rightarrow ifelse\ SPAB\ med\_gliders\ high\_gliders \\
&\ SPAB \rightarrow and\ SPAB\ SPAB \mid or\ SPAB\ SPAB \mid not\ SPAB \\
&\ SPAB \rightarrow spaeval\ B\ SPAEXP \\
&\ B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid EXPN \\
&\ SPAEXP \rightarrow current \mid within\ DISTANCE \\
&\ DISTANCE \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \\
&\ EXPN \rightarrow DEV \mid STREAM \mid STAND \\
&\qquad \ldots \\
&\ SLOPEVAL \rightarrow flat \mid moderate \mid steep \\
&\} \\
\}
\end{aligned}
$$

The increased evaluation time for the setup of Table 4.15 meant that only one run of the system was performed for each of the six random collections of locations. Even though the discovered programs are unlikely to be the best possible solutions for this language the results, shown in Table 4.16, are an improvement over the grammar $G_{ggd-spatial}$. The program with the best training result is shown below. Note that the spatial distance functions of $within(4)$ and $within(5)$ are both used.

```
ifelse(sc(=,outside_study),no_gliders,

ifelse(or(sc(=,rock),
          and(spaeval(sl(>,flat),within(2)),
               and(spaeval(sc(=,regeneration),within(4)),
                   spaeval(fn(>,medium),within(5))))),low_gliders,

ifelse(or(spaeval(sl(>,flat),within(2)),
          spaeval(sc(=,regeneration),within(4))),med_gliders,
/* otherwise */
```

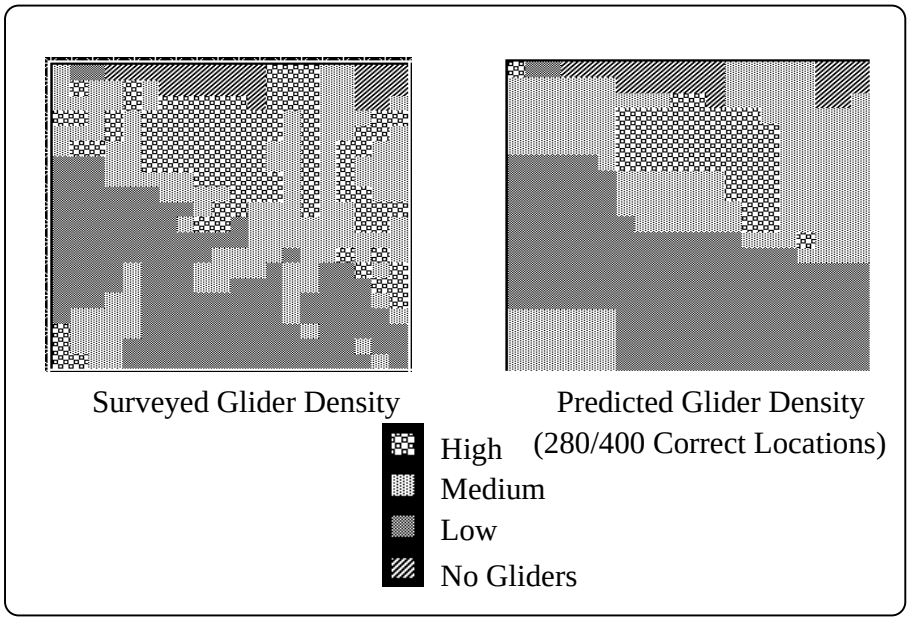


Figure 4.15: The Best Prediction of Glider Density using $G_{ggd-spatial-biased}$.

Model Accuracy over 6 Random Training and Test Sites	
Training	Test
70.0%	69.5%
72.5%	65.5%
73.0%	67.0%
71.0%	68.5%
71.0%	67.5%
73.0%	65.0%
71.8±1.3%	67.2±1.7%

Table 4.16: Results using $L(G_{ggd-spatial-biased})$ for Glider Prediction.

```
high_gliders)))

Training: 73.0%
Test    : 67.0%

where sc: stand condition
      sl: slope
      fn: floristic nutrients
```

The predictions for this program over the study area is shown in Figure 4.15. The program correctly classifies 70% of the 400 surveyed locations.

Model Accuracy		
Method	Training	Test
MR	-	45%
PCA	-	41%
KA	-	48%
MGC	$36.7 \pm 0.8\%$	$33.2 \pm 1.6\%$
ID3	$60.7 \pm 1.3\%$	$57.3 \pm 2.2\%$
CN2	$50.8 \pm 2.6\%$	$45.2 \pm 2.2\%$
CART	$61.2 \pm 3.5\%$	$54.8 \pm 3.9\%$
MSC	$75.8 \pm 1.2\%$	$48.3 \pm 1.6\%$
$L(G_{ggd})$	$57.9 \pm 1.4\%$	$52.5 \pm 3.4\%$
$L(G_{ggd-cover})$	$60.6 \pm 1.1\%$	$55.9 \pm 3.5\%$
$L(G_{ggd-cover+no\_gliders})$	$61.3 \pm 1.5\%$	$57.1 \pm 3.2\%$
$L(G_{ggd-spatial})$	$67.3 \pm 2.0\%$	$64.1 \pm 2.0\%$
$L(G_{ggd-spatial-biased})$	$71.8 \pm 1.3\%$	$67.2 \pm 1.7\%$

Table 4.17: Model Accuracy for Greater Glider Density [71].

Model Significance using Student's t-test							
Method	MGC	ID3	CN2	CART	MSC	$L(G_{ggd})$	$L(G_{ggd-cover})$
MGC	-	-	-	-	-	-	-
ID3	99%	-	99%	-	99%	95%	-
CN2	99%	-	-	-	-	-	-
CART	99%	-	99%	-	99%	-	-
MSC	99%	-	95%	-	-	-	-
$L(G_{ggd})$	99%	-	99%	-	99%	-	-
$L(G_{ggd-cover})$	99%	-	99%	-	95%	-	-
$L(G_{ggd-cover+no\_gliders})$	99%	-	99%	-	99%	90%	-
$L(G_{ggd-spatial})$	99%	99%	99%	99%	99%	99%	99%
$L(G_{ggd-spatial-biased})$	99%	99%	99%	99%	99%	99%	99%

Table 4.18: Model Significance for Greater Glider Density - Part 1.

4.3.9 Summary and Discussion

A summary of the previous languages used to explore solutions of glider prediction are shown in Table 4.17. A Student's t test [47], for significance of the difference between means with known variance, has been applied to these results, as shown in Table 4.18 and 4.19. These tables should be read from left to right. For example, from Table 4.18, the language defined by $L(G_{ggd-cover})$ is a significant improvement from MGC and CN2, with 99% confidence. Additionally, the result for $L(G_{ggd-cover})$ is a significant improvement from MSC with 95% confidence. Entries in the table where a "-" occur indicate one of two conditions - either there is less than 90% confidence for a difference between the compared

Model Significance using Student's t-test			
Method	$L(G_{ggd-cover+no\_gliders})$	$L(G_{ggd-spatial})$	$L(G_{ggd-spatial-biased})$
MGC	-	-	-
ID3	-	-	-
CN2	-	-	-
CART	-	-	-
MSC	-	-	-
$L(G_{ggd})$	-	-	-
$L(G_{ggd-cover})$	-	-	-
$L(G_{ggd-cover+no\_gliders})$	-	-	-
$L(G_{ggd-spatial})$	99%	-	-
$L(G_{ggd-spatial-biased})$	99%	95%	-

Table 4.19: Model Significance for Greater Glider Density - Part 2.

means or the mean for the left-hand column method is less than the cross-column method being compared.

The accuracy of the learnt programs improve as more complex spatial relations are introduced into the language. Additionally, the explicit statement of some parts of the search space (for example, stating that glider density is zero for locations outside the study area) allows the search effort to be concentrated where it is most desirable. The result for the method using $L(G_{ggd-spatial-biased})$ is a significant improvement, with a confidence of 99%, against all other methods except $L(G_{ggd-spatial})$, where the confidence interval is significant at the 95% level. Both languages that have introduced spatial relationship are significant improvements over all other learning methods with a confidence of 99%. The introduction of spatial relationships can clearly be seen to significantly improve the quality of the theory created using CFG-GP. This example has demonstrated that typing, language bias and search bias may be usefully defined with CFG-GP for a complex spatial problem.

4.4 Conclusion

This chapter has demonstrated that declarative bias may be expressed for several problems with the learning system, CFG-GP. The ability to obtain good results using this system is dependent upon the ability to write a grammar that captures the underlying belief of the user about the structure of a solution. Although it may not always be possible to write a grammar that has an appropriate bias it is always possible to begin with a grammar that has no explicit bias towards any particular program form. This grammar merely states the typing constraints for each function and argument without expressing any further bias. If the system cannot discover a solution with this grammar an analysis of the partial solutions may suggest certain structures that are likely to be useful in creating a better

solution. These structures may then by introduced explicitly into the grammar and the system rerun. This iterative process may be continued to gradually increase the bias for the system and therefore improve the performance for a particular problem.

Chapter 5

Learning Inductive Bias

This chapter describes a method for automatically learning how to modify the bias represented explicitly in the initial grammar, G. New productions are discovered from an analysis of the fittest program during the evolution of a (partial) solution. A selected derivation from this program is used to modify the current grammar each generation. The grammar is modified in two ways, namely, replacing nonterminals with terminals, in existing productions, and creating new productions that represent underlying structure. New programs are introduced each generation into the population using this modified grammar, using an operation referred to as $REPLACEMENT$. The feedback between the evolving solutions and the grammar is demonstrated to improve the performance of CFG-GP.

The chapter concludes by presenting a form of incremental learning. A grammar that has been learnt for the 6-multiplexer is shown to improve the performance of the learning system applied to the 11-multiplexer. This demonstrates that the learnt grammar is capable of being biased towards a class of problems.

5.1 Introduction

The selection of an appropriate bias, represented by G, is a fundamental task when applying CFG-GP. The bias influences both the form and success of the created programs. Specifying an appropriate language bias is certainly one of the most important consideration when trying to solve a problem for which no solution is known. The declarative modification of bias has been demonstrated in Chapter 4, where knowledge of the structure of a solution guided changes to the initial grammar, G. This manual biasing could occur, *even when the solution was not known*, by manually examining fit programs and incorporating useful components of previously discovered partial solutions explicitly into the initial grammar.

A system that learns how to modify its own inductive bias (i.e. learns how to modify G) gives an additional approach to discovering useful components of a language and to improve the performance of CFG-GP. The ability to change the language bias may also simplify the learning task and allow a grammar, learned from a simple problem, to be applied to a similar, yet more complex, problem. In this manner, a difficult problem may be approached by using graded examples that allow the grammar to be biased towards the general patterns that occur for a class of problems. Further, the automatic discovery of a useful language bias alleviates the requirement of knowing this bias in advance. Thus, a weak bias, represented by G, may be modified by the learning system, to allow a problem (and potentially a class of problems) to be encapsulated by the modified grammar and thus learnt more easily.

Modifying the bias, represented by G, will be demonstrated in two ways. Viz.,by replacing nonterminals with other nonterminals and terminals in existing productions, and by including additional productions that reflect the underlying structure of the solution.

5.2 The Grammar, G, Viewed as a Bias

The language represented by a grammar may be considered as *the most general description* of a solution. This is clear by considering the definition of $L(G)$, as follows.

$$L(G) = \{x \mid x \in \sum^{\star}, \, and \; S \stackrel{+}{\Rightarrow} x\}$$

The sentences represented by $L(G)$ contain all derivable strings from the grammar, G. Hence, if the grammar G allows the construction of a program that would represent the solution, $L(G)$ contains the solution[1].

There are an infinite number of grammars that represent the same language. However, for the purpose of using a grammar as the language bias for CFG-GP, some of these grammars will be more likely to generate a solution than others. This arises from the mechanisms CFG-GP uses to create the population from the initial grammar (see Section 3.4). The influence on the performance of CFG-GP, using an appropriate grammar, has been demonstrated in Section 4.1. The goal of automatically modifying G is to find an appropriate emphasis, for particular strings in $L(G)$, which allow the learning system to more easily solve a particular problem.

5.3 Learning to Modify a Context-Free Grammar

Each generation, the current grammar will be modified to create a new grammar. Defining the grammar, at generation t, as G^t, there will be a progression of grammars that are

[1] In Genetic Programming, this condition is termed sufficiency[44]: that the set of functions and terminals used to evolve a solution *must be capable of solving the problem.*

created. If the initial grammar (i.e. the initial language bias) is represented as G^0, then we wish to construct a series of grammars over t generations, as follows.

$$G^0, G^1, G^2, .., G^t$$

There are two conditions that are imposed on the series of grammars.

(a) $G^0 \subset G^1 \subset G^2, .., \subset G^t$.

This condition states that the grammar created at generation t includes the grammar that existed at generation $t-1$. Further, the extension to the grammar G^t must not remove components that existed in G^{t-1}.

(b) $L(G^0) = L(G^1) = L(G^2), .., = L(G^t)$.

This condition states that the learned grammars must represent the same language that is expressible from the initial grammar, G^0. This follows directly from the definition of G as a generalised description of the solution space. Although a grammar may be extended to represent a certain language bias, all initially derivable strings from $L(G^0)$ must still exist, to avoid removing the solution from the language[2].

These conditions *do not change the language represented by* $L(G)$, however the sentences most likely to be generated from CFG-GP, using the grammar, are changed. The goals of this learning process are as follows.

(a) **Condense derivation steps into grammar productions:**Learn to create new productions that are essentially previous productions where a nonterminal has been replaced by other nonterminals or terminal symbols. For example, given the derivation step, $if\ B\ B\ B \overset{B \to a0}{\Rightarrow} if\ a0\ B\ B$, a new production could be created, $B \to if\ a0\ B\ B$. This represents the discovery that $a0$ is a useful terminal to be placed in the first argument position for if.

(b) **Change the emphasis of particular productions:**Learn to bias the grammar by modifying the *merit selection* values for particular productions. For example, if the production, $B \overset{1}{\to} if\ a0\ B\ B$, exists *and the same production is discovered as useful*, then the merit selection for this production should increase to reflect its relative importance when selecting productions from the nonterminal, B. Thus, the merit selection value is incremented to create the production $B \overset{2}{\to} if\ a0\ B\ B$.

(c) **Generate new levels in the grammar:** Learn to create new productions of the form $X \to \alpha$, where the nonterminal, X, has not previously existed. This

[2]This approach is essentially opposite to Utgoff[73], where the goal was to *weaken a strong bias*. In the description here a weak bias (represented by the initial grammar G) is gradually strengthened by introducing new productions and modifying the merit selection values. Although no potential sentence from $L(G)$ is removed, the technique used to create the initial population (see Section 3.4) can exploit the changed grammar so that generated programs are biased towards the structure of a solution. This differs from Utgoff in that he modified the language bias, whereas this approach, by maintaining $L(G)$, changes only the search bias. However, the relationship between Utgoff's work and our own closely parallels the difference in approach between an overly specialised language representation and an overly general representation.

type of production is intended to capture structural generalisations that have been discovered. For example, the 6-multiplexer has one form of solution where two levels of the if function must be combined to distinguish the four data lines. Two new productions could be created to represent this statement, as follows.

$$B \xrightarrow{1} if\ B\ N_1\ B$$
$$N_1 \xrightarrow{1} if\ B\ B\ B$$

The nonterminal N_1 has been created and included as part of the definition of G. This new production from N_1 can now be involved in later changes to the grammar. The combination of these two productions represents a bias towards creating programs that have a two-tiered if structure and, therefore, represents the structure for one form of solution for the 6-multiplexer.

The following steps are involved when attempting to create the grammar G^{t+1}, from the grammar G^t, after evaluating the population of derivation trees at generation t.

(a) Identify the program derivation tree that will contribute to changes in the grammar.

(b) Identify the program derivations that are to be refined as part of the next-generation grammar, G^{t+1}.

(c) Incorporate the proposed changes into G^t, thereby creating G^{t+1}.

(d) Incorporate the grammar, G^{t+1}, into the population of programs that represent the population in generation $t+1$[3].

Each of these points will now be discussed in further detail.

5.3.1 Identifying Program Individuals

Rosca and Ballard[60] have studied identifying which parts of a program contribute towards a solution. They concentrated on two criteria, namely *fit blocks* and *frequent blocks*. Their goal was to determine building blocks that could be used to extend the function set within the framework of Genetic Programming. This extended function set was then used to create new members of the population that could exploit these discovered functions. They concluded that *fit blocks* were the most useful measure to determine a building block. The work of Rosca and Ballard discovered useful building blocks by analysing how parts of programs contributed to the fitness of an entire program, using an information measure.

We take a contrary approach which has more of an evolutionary flavour. The main assumption is that a fit program will contain (some) relatively fit components. In terms of G, this implies that a fit program will contain (some) relatively fit derivations which can be exploited in changes to the underlying grammar. Given this assumption, it is

[3]Note that any production (and therefore any derivation) in G^t remains valid in the grammar G^{t+1}.

reasonable to conclude that selecting the fittest program from each generation will be a useful guide to select productions from G that can be used to change the underlying bias. When more than one program is equally fit, the program which has the least depth of parse tree will be selected. This is based on the argument that the smaller of two equally fit programs will be a more concise definition of the (partial) solution and, therefore, the possibility of selecting a useful production will be increased. When both fitness and size cannot distinguish one program, from the population, an arbitrary choice from these candidate programs will be made.

5.3.2 Identifying Program Productions

Once a program has been selected the problem remains of determining which derivation from this program should be selected to direct the modification of G^t. This problem is made particularly difficult due to the epistatic nature of programs. It is impossible to conclude, with certainty, that particular parts of a program are the main contributors to the fitness of the entire program. In fact, it is often a meaningless statement to consider which individual statements of a program contribute to the overall fitness. This difficulty has been highlighted by the theoretical work of O'Reilly[56]. The linkage between statements and structure of a program generally create the overall properties of the program.

Previous work[73] that has incorporated a learnt bias have used a language where a partial ordering could be defined. This partial ordering, over the hypothesis space, enabled the language to be changed in a directed manner. However, in general, the language defined by a context-free grammar does not represent a partial ordering over the hypothesis space[4].

The approach described here assumes that any terminal, from the selected program derivation tree, may contribute to developing a useful bias with the grammar G^{t+1}. Therefore, a terminal from this program *will be selected at random.* Although a random selection of terminal may create a production which does not contribute towards a solution, the proportional fitness selection pressure will not propagate these changes and so their influence will not be emphasised.

Let the randomly selected terminal, from the fittest derivation tree ρ, be x. There are two possible forms of derivation that created $x \in \rho$.

(a) $\alpha\ A\ \beta \overset{A \rightrightarrows x}{\Rightarrow} \alpha\ x\ \beta$

Here, the terminal, x, is created from a production which contains *only this teminal.* Figure 5.1 shows two examples where the selected terminal, x, satisfies this condition.

[4] An ordering is only possible where the semantics of the language are known (for example, with a CNF or DNF language). The CFG-GP system does not assume any explicit meaning to the terminal strings defined by the user-defined language, since the grammar *only defines the syntax of the language.* The semantics of the language are discovered by executing the generated programs, thereby giving a fitness measure to each program.

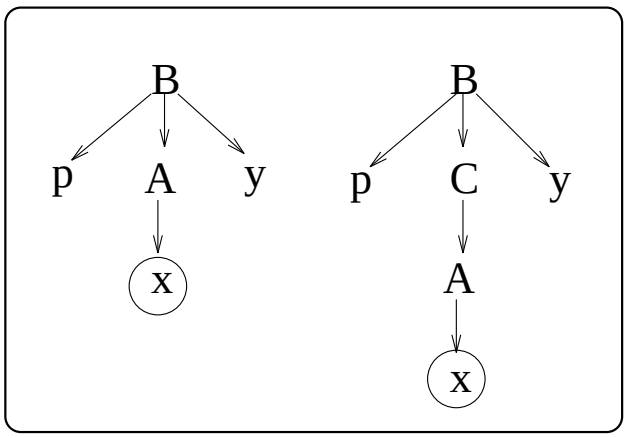


Figure 5.1: Terminal Sites derived from the production $A \rightarrow x$.

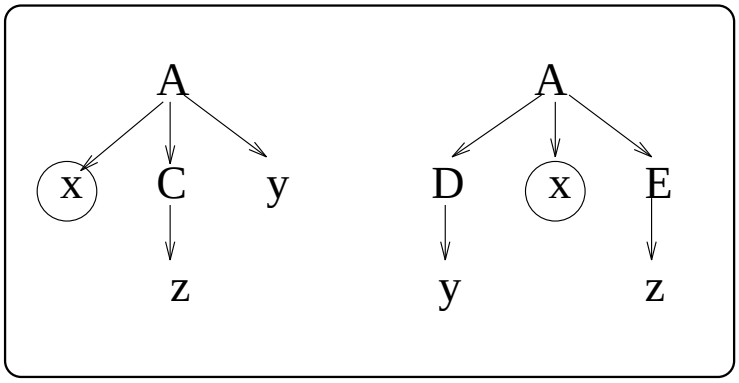


Figure 5.2: Terminal Sites derived from the production $A \rightarrow \mu\ x\ \xi$.

(b) $\alpha\ A\ \beta \overset{A \rightarrow \mu x \xi}{\Rightarrow} \alpha\ \mu\ x\ \xi\ \beta$

Here, the terminal, x, is created from a production that contains some other terminals and/or nonterminals. Formally, this implies that at least one of μ or $\xi \in \{N \cup \sum\}^+$. Figure 5.2 shows two examples where the selected terminal, x, satisfies this condition.

5.3.3 Incorporating New Productions into the Grammar

Once a terminal from a derivation tree has been selected, the production associated with this terminal can be used to direct a change to G^t. The two forms of derivation that created x are handled in different ways. Each of these methods will now be described.

Creating Productions from $A \rightarrow x$

A new production is determined by propagating x up the derivation tree until a level is reached where other nonterminal and/or terminal nodes exist. Two examples of this propagation are shown in Figure 5.3. In both cases, the terminal, x, has been selected as the site to create a new production for G^{t+1}. The first example propagates x up one level

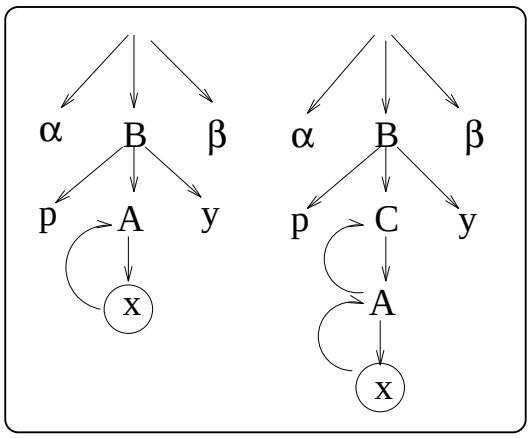


Figure 5.3: Propagating the terminal, x, up a derivation tree.

in the tree, whereas in the second example x must be propagated up the derivation tree two levels. This demonstrates the requirement of moving up the derivation tree until a level is reached where other terminals or nonterminals exist. Once this level is reached a new production may be inferred by reading across the tree, *substituting the next level of terminal or nonterminal* for the current nonterminal in this position. For example, based on Figure 5.3, the left example derivation tree was created by applying the following derivations.

$$\alpha\ B\ \beta \overset{B \to pAy}{\Rightarrow} \alpha\ p\ A\ y\ \beta \overset{A \to x}{\Rightarrow} \alpha\ p\ x\ y\ \beta$$

Propagating x up the derivation tree creates the production $B \overset{1}{\rightarrow} p\ x\ y$, which is inserted into G^{t+1}. The new production is given a *merit selection value* of 1 and the productions from B extended by this production.

The second example from Figure 5.3 shows the case where two levels of derivation must be climbed to arrive at a production where other nonterminals and/or terminals exist. The following derivation steps have created the terminal x, based on this example.

$$\alpha\ B\ \beta \overset{B \to pCy}{\Rightarrow} \alpha\ p\ C\ y\ \beta \overset{C \to A}{\Rightarrow} \alpha\ p\ A\ y\ \beta \overset{A \to x}{\Rightarrow} \alpha\ p\ x\ y\ \beta$$

A new production is created that substitutes A for the nonterminal C. The production $B \overset{1}{\rightarrow} p\ A\ y$ is inserted into G^{t+1}. The new production is given a *merit selection value* of 1 and the productions from B extended by this new production.

For each of the previous examples, if the new production already exists in G^t, then the merit selection value of the matching production is incremented. In this case no new production is created.

Creating Productions from $A \rightarrow \mu\ x\ \xi$

The derivation step, $\overset{A \to \mu x \xi}{\Rightarrow}$, is used to form a new production. This is achieved by creating a new nonterminal, say N_i, and creating the following production.

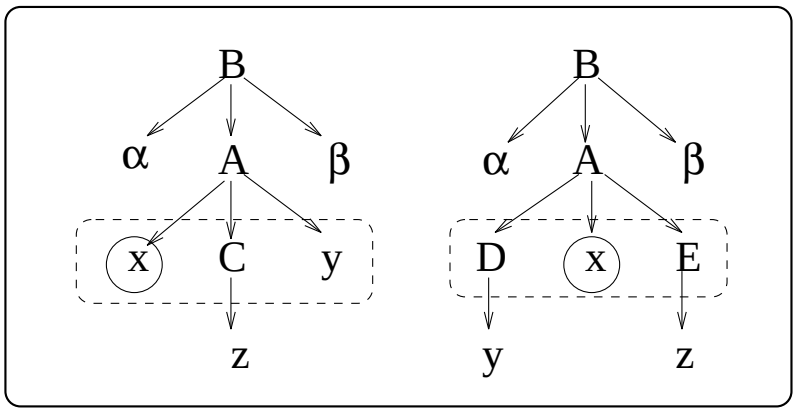


Figure 5.4: Creating a New Production from $A \rightarrow \mu\ x\ \xi$.

$$N_i \stackrel{1}{\rightarrow} \mu\ x\ \xi$$

The nonterminal, N_i, is then linked to the previous level of derivation by creating a new production, substituting N_i for the previous nonterminal, A. For example, the derivations, shown in Figure 5.4, demonstrate two cases where a terminal has been selected where other terminals and/or nonterminals exist as part of the same derivation. For the left-most example, the derivation steps that created x are shown below.

$$\stackrel{B \rightarrow \alpha A \beta}{\Rightarrow} \alpha\ A\ \beta \stackrel{A \rightarrow x C y}{\Rightarrow} \alpha\ x\ C\ y \Rightarrow$$

Two new productions are created as a result of this situation. A new nonterminal, say N_i, is created. The following two productions are then inserted into G^t, to create G^{t+1}.

$$B \stackrel{1}{\rightarrow} \alpha\ N_i\ \beta$$

$$N_i \stackrel{1}{\rightarrow} x\ C\ y$$

In a similar manner, using the right-most example from Figure 5.4, the following productions would be created.

$$\mathrm{B} \stackrel{1}{\rightarrow} \alpha\ N_i\ \beta$$

$$N_i \stackrel{1}{\rightarrow} D\ x\ E$$

The purpose of these new productions is to allow structure to be explicitly expressed. Although the new production from N_i is a copy of a previous production from G^t, the new production represents a bias towards creating programs where x is combined with the production from which it was created. In this manner, the composition of functions and arguments may be explicitly expressed in the grammar and the new grammar reflects underlying patterns of structure that are discovered during the search for a program solution.

Let the set of created nonterminals be $\{N_1, N_2, ..N_j\}$. If the production that is proposed from the new nonterminal, N_i, already exists from one of the created nonterminals, say N_f, then the merit selection for the production, from N_f, is incremented and no new

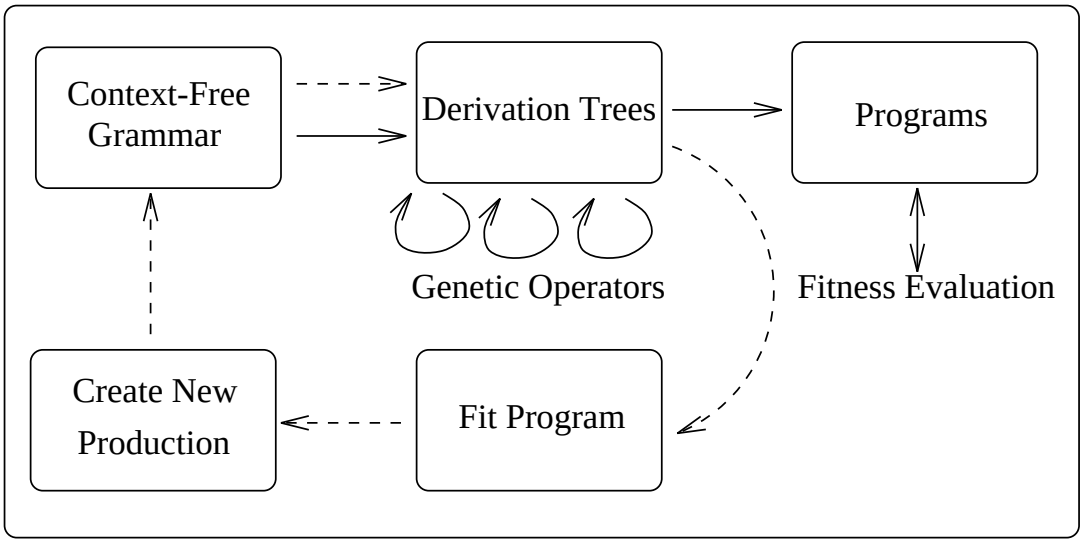


Figure 5.5: Creating New Programs using a modified Grammar.

production from N_i is created. Similarly, if the production, which includes N_i, already exists, then the merit selection for this production is incremented and no new production is created. Note that the new production from N_i now becomes part of the grammar and is available for further modification in later generations.

5.3.4 Incorporating the Modified Grammar back into the Population

As shown in Figure 5.5, the modified production is inserted into G^t and new programs are generated from this modified grammar, in generation $t+1$. The new programs are created in the same manner as the initial population, however there is *no check for uniqueness* between the inserted programs and the current population. This is not required (or desirable) as the bias introduced by the modified grammar should be encouraged within the population. If changes to the grammar create fit programs these are propagated throughout the population due to the fitness selection pressure. This reintroduction of programs into the population will be referred to as a $REPLACEMENT$ operation and specified by a percentage probability of occurrence, in the same manner as selective crossover and selective/directed mutation. Note that the generated programs, using $REPLACEMENT$, are limited in depth by the *MAX DEPTH PROGRAMS* parameter.

5.4 Experiments

There are two main purposes of learning to modify the initial grammar, G^0. Firstly, to improve the performance of the learning system for a particular problem and, secondly, to allow a grammar to be learnt that generalises from one class of problems to another.

The following experiments (Section 5.4.1) show that the learnt changes to G^0 *are beneficial* and that the evolved grammar has been biased to represent components of the solution. This shows that a form of *specialisation* has occurred with the grammar, allowing partial components of the solution to be expressed in the grammar explicitly.

5.4.1 The 6-Multiplexer with $REPLACEMENT$

The multiplexer problem will be used to demonstrate how learning to modify G^0 can improve the *probability of success*. The grammar, $G_{6m-address}$, will be used as the initial language bias. This grammar is defined as follows.

$$
\begin{aligned}
&G_{6m-address} = \\
&\quad \{S, \\
&\quad\ N = \{B, DATA, ADDRESS\}, \\
&\quad\ \textstyle\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\}, \\
&\quad\ P = \\
&\qquad \{S \rightarrow B \\
&\qquad\ B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid DATA \mid ADDRESS \\
&\qquad\ DATA \rightarrow d0 \mid d1 \mid d2 \mid d3 \\
&\qquad\ ADDRESS \rightarrow a0 \mid a1 \\
&\qquad \} \\
&\quad \}
\end{aligned}
$$

This grammar is essentially the same as G_{6m}(see Section 4.1), however $G_{6m-address}$the data and address values are distinguished by the nonterminals $DATA$ and $ADDRESS$. This will allow the learnt grammar to treat the group of address and data values independently. Note, however, that there is no initial bias for treating the $ADDRESS$ and $DATA$ productions differently.

The setup using $G_{6m-address}$ is shown in Table 5.1. The $REPLACEMENT$ operator occurs with a probability of 10%. Hence, given a population size of 500, there will typically be about 50 programs created from $G^t_{6m-address}$, each generation. This implies that the number of programs created from crossover will decrease from approximately 450 (Section 4.1.1) to approximately 405, based on a 90% probability of crossover. These changes in the population dynamics will affect the *probability of success*. Since this measure will be used to determine whether there is a significant improvement when using $REPLACEMENT$, a base-line value for p_s must be determined. This is found by running the system 100 times, using the setup of Table 5.1, where $G^0_{6m-address}$ *is not modified* during the evolution.

The 6-Multiplexer	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 5	200
CREATE MAXIMUM DEPTH 6	100
CREATE MAXIMUM DEPTH 7	100
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	100
MAX DEPTH PROGRAMS	8
$\otimes = \{B\}$	90%
REPLACEMENT	10%
GENERATIONS	50
FITNESS MEASURE	64 boolean cases
GRAMMAR	$G_{6m-address}$

Table 5.1: The 6-Multiplexer System using REPLACEMENT.

The REPLACEMENT Operator	
Method (10% *REPLACEMENT*)	Probability of Success p_s
Grammar Not Modified	31%
Grammar Modified	50%

Table 5.2: *REPLACEMENT* and the 6-Multiplexer, using $G_{6m-address}$.

Hence, the *REPLACEMENT* operator merely creates new programs, using the initial grammar, each generation. Another way to view *REPLACEMENT*, *when the grammar is not modified*, is that it is a *selective mutation*, where $\odot = \{S\}$.

The resulting *probabilites of success* are shown in Table 5.2. The original setup, which does not modify $G_{6m-address}$, has a *probability of success* of 31%. When the two previous grammar modifying operations are applied to $G^t_{6m-address}$, throughout the evolution of a solution, the probability of success improves to 50%. This significant increase in p_s (see Appendix B), when the grammar is modified, implies that the feedback occuring with the learnt grammar is beneficial for this problem.

Four learnt grammars will now be shown, to give some idea of the types of constructions that have been discovered. The first grammar, $G^{12}_{6m-address}$, was created from the shortest successful run of CFG-GP, which succeeded after 12 generations[5].

$G^{12}_{6m-address} =$

[5]If no *merit selection value* is shown for a production it has the value one.

$\{S,$

$N = \{B, DATA, ADDRESS, N_1, N_2\},$

$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$

$P =$

$\{S \rightarrow B$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid DATA \mid ADDRESS$

$B \rightarrow \overset{3}{if}\ ADDRESS\ B\ B \mid \overset{2}{and}\ DATA\ B \mid if\ ADDRESS\ B\ DATA$

$B \rightarrow not\ DATA \mid if\ B\ DATA\ N_2 \mid if\ B\ B\ DATA$

$B \rightarrow if\ ADDRESS\ N_1\ B \mid if\ B\ DATA\ B$

$DATA \rightarrow d0 \mid d1 \mid d2 \mid d3$

$ADDRESS \rightarrow a0 \mid a1$

$N_1 \rightarrow and\ DATA\ B$

$N_2 \rightarrow if\ ADDRESS\ B\ B$

$\}$

$\}$

The grammar, $G^{12}_{6m-address}$, has discovered some relevant new productions that have a sensible bias for the 6-mulitplexer. For example, the most likely production to be selected from the nonterminal, B, is as follows.

$$B \overset{3}{\rightarrow} if\ ADDRESS\ B\ B$$

This embodies the concept that an address value should be in the first argument position for the if function. This concept has also been represented by the production created from the new nonterminal, N_2. In general, the grammar does appear to have some constructions that would be a useful bias when learning the 6-multiplexer.

The grammar $G^{31}_{6m-address}$ was created from a successful run of CFG-GP, after 31 generations, and shows what happens when the grammar has been allowed to evolve further.

$G^{31}_{6m-address} =$

$\{S,$

$N = \{B, DATA, ADDRESS, N_1, N_2, N_3, N_4, N_5\},$

$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$

$P =$

$\{S \rightarrow B \mid \overset{2}{N_4} \mid N_2$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid DATA \mid ADDRESS$

$B \rightarrow \overset{5}{or}\ DATA\ B \mid \overset{2}{if}\ B\ B\ DATA \mid \overset{2}{if}\ ADDRESS\ B\ DATA$

$B \rightarrow \overset{2}{and}\ ADDRESS\ B \mid if\ B\ DATA\ DATA \mid and\ B\ N_4 \mid not\ N_4$

$B \rightarrow if\ ADDRESS\ DATA\ B \mid if\ B\ DATA\ B \mid if\ N_3\ B\ B \mid or\ B\ N_2$

$B \rightarrow if\ N_4\ B\ B \mid if\ B\ d0\ B \mid if\ B\ N_4\ B \mid if\ B\ N_1\ B$

$DATA \rightarrow d0 \mid d1 \mid d2 \mid d3$

$ADDRESS \rightarrow a0 \mid a1$

$N_1 \rightarrow or\ ADDRESS\ B \mid or\ N_2\ B \mid or\ B\ N_2 \mid or\ B\ B$

$N_2 \rightarrow if\ B\ B\ N_5 \mid if\ B\ B\ B$

$N_3 \rightarrow not\ ADDRESS \mid not\ B$

$N_4 \rightarrow and\ B\ B$

$N_5 \rightarrow if\ B\ DATA\ B$

}

}

The grammar, $G^{31}_{6m-address}$, has increased in size from $G^{12}_{6m-address}$. This is not surprising as there were another 19 grammar modifications applied before the run halted. The grammar has created 5 new nonterminals, which predominantly bias towards using the if function. In fact, from the start symbol, S, there is now a 37% probability of selecting the if function as the top-level function for a randomly created program[6]. Additionally, the use of $ADDRESS$ as the first argument with if has been discovered.

The grammar $G^{49}_{6m-address}$ was created from a successful run of CFG-GP, after 49 generations. This grammar was selected from the *latest finishing successful run*, from the 100 runs that were performed.

$G^{49}_{6m-address} =$

$\{S,$

$N = \{B, DATA, ADDRESS, N_1, N_2, N_3, N_4, N_5\},$

$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$

$P =$

$\{S \rightarrow B \mid N_2$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid DATA \mid ADDRESS$

$B \rightarrow \overset{5}{if}\ N_1\ B\ B \mid \overset{4}{or}\ DATA\ B \mid \overset{3}{if}\ B\ B\ DATA \mid \overset{3}{if}\ B\ B\ N_2 \mid \overset{2}{if}\ N_3\ B\ B$

[6]The original grammar had a probability of 17% for selecting the if function initially.

$B \rightarrow \overset{2}{if}\ B\ ADDRESS\ B \mid \overset{2}{if}\ B\ B\ N_4 \mid \overset{2}{or}\ ADDRESS\ ADDRESS \mid not\ N_5$

$B \rightarrow \overset{2}{if}\ B\ B\ ADDRESS \mid or\ ADDRESS\ B \mid or\ B\ N_1 \mid if\ ADDRESS\ B\ B$

$B \rightarrow or\ N_1\ B \mid or\ B\ N_3 \mid if\ B\ N_4\ B \mid if\ B\ B\ N_1 \mid and\ DATA\ B$

$B \rightarrow if\ B\ N_3\ B \mid or\ a1\ B \mid or\ B\ N_2 \mid if\ B\ DATA\ B \mid and\ N_1\ B$

$B \rightarrow and\ N_1\ B \mid if\ B\ N_2\ B \mid or\ ADDRESS\ a1 \mid and\ B\ N_4 \mid if\ N_1\ N_1\ B$

$B \rightarrow if\ B\ B\ d0 \mid if\ B\ ADDRESS\ DATA \mid if\ N_1\ ADDRESS\ B$

$DATA \rightarrow d0 \mid d1 \mid d2 \mid d3$

$ADDRESS \rightarrow a0 \mid a1$

$N_1 \rightarrow and\ ADDRESS\ B \mid and\ B\ B$

$N_2 \rightarrow if\ B\ B\ B$

$N_3 \rightarrow or\ ADDRESS\ B$

$N_4 \rightarrow if\ N_1\ B\ B$

$N_5 \rightarrow if\ B\ ADDRESS\ B$

}

}

The grammar, $G^{49}_{6m-address}$, has evolved to a considerable extent. Certainly, some of the new productions are not useful, however it has learnt that the two-tiered *if* structure is important. This is shown with the following two productions.

$$B \rightarrow if\ B\ N_4\ B$$

$$N_4 \rightarrow if\ N_1\ B\ B$$

The final grammar, $G^{50}_{6m-address}$, was created by a run that *did not succeed*. This grammar was selected from the failed group of runs that had the *worst fitness* (a raw fitness of 16) after 50 generations.

$G^{50}_{6m-address} =$

$\{S,$

$N = \{B, DATA, ADDRESS, N_1, N_2, N_3, N_4, N_5\},$

$\sum = \{and, or, not, if, a0, a1, d0, d1, d2, d3\},$

$P =$

$\{S \rightarrow B \mid \overset{20}{N_1} \mid N_5$

$B \rightarrow and\ B\ B \mid or\ B\ B \mid not\ B \mid if\ B\ B\ B \mid DATA \mid ADDRESS$

$B \rightarrow \overset{4}{if}\ B\ DATA\ B \mid \overset{3}{or}\ a0\ DATA \mid \overset{2}{and}\ B\ N_2 \mid \overset{2}{or}\ ADDRESS\ d0 \mid or\ B\ DATA$

$B \rightarrow if\ B\ DATA\ DATA \mid or\ B\ d0 \mid or\ ADDRESS\ DATA \mid if\ B\ B\ DATA$

Language Bias Comparisons	
Grammar	Probability of Success p_s
G_{6m}	34%
G_{6m-if}	37%
$G_{6m-if-address}$	62%
$G_{6m-if-then}$	63%
$G_{6m-if-address-then}$	80%
$G_{6m-if-a0-if-a1}$	88%
$G^{12}_{6m-address}$	69%
$G^{31}_{6m-address}$	82%
$G^{49}_{6m-address}$	71%
$G^{50}_{6m-address}$	1%

Table 5.3: Results of Learnt Language Bias applied to the 6-multiplexer.

$B \rightarrow or\ B\ N_4 \mid and\ ADDRESS\ B \mid if\ B\ N_3\ N_4 \mid if\ B\ N_3\ B$

$B \rightarrow if\ N_5\ N_3\ B \mid and\ B\ ADDRESS \mid if\ ADDRESS\ DATA\ B$

$DATA \rightarrow d0 \mid d1 \mid d2 \mid d3$

$ADDRESS \rightarrow a0 \mid a1$

$N_1 \rightarrow if\ a0\ d3\ d2 \mid if\ a0\ DATA\ d2 \mid if\ ADDRESS\ DATA\ d2$

$N_1 \rightarrow if\ ADDRESS\ DATA\ DATA \mid if\ B\ DATA\ DATA$

$N_2 \rightarrow or\ B\ DATA$

$N_3 \rightarrow or\ DATA\ B \mid or\ ADDRESS\ B \mid or\ B\ B$

$N_4 \rightarrow and\ B\ B$

$N_5 \rightarrow if\ B\ DATA\ B$

}

}

This grammar has converged to a partial solution based on the address line, $a0$. The most obvious indication of this convergence is the production, $S \xrightarrow{20} N_1$, which gives a probability of 20/22 that the productions from N_1 will be selected to begin the program definition. Although the productions from N_1 all use the if function, 2 of the 5 productions have $a0$ explicitly stated in the first argument position, and have the data values already assigned. This means that programs, created from the modified grammar, will tend to converge to the partial solution, $if(a0, d3, d2)$. Hence, this grammar is not a good representation of the underlying structure for the 6-multiplexer since it has converged to a partial solution.

To demonstrate that the learnt grammars have embodied useful components of the multiplexer solution, each grammar was used as the language bias for the original setup for

the 6-multiplexer (Section 4.1). The results are shown in Table 5.3, along with the original results using various human-biased grammars. The results show that the grammars, created from successful runs, have been biased towards a solution for the 6-multiplexer. The extremely poor performance of $G^{50}_{6m-address}$ shows what happens if the grammar and population converge to a partial solution. This implies that an evolved grammar will be most useful if it is created when the hypothesis space is being searched in a broad fashion and has not converged. The grammar, $G^{50}_{6m-address}$, has also performed poorly due to the crossover operator, $\otimes = \{B\}$. The initial population that is created with this grammar has few legal sites for crossover (i.e. few B nonterminals). Hence, there is little searching performed by the population since crossover often fails to find legal sites within the selected derivation trees of the population.

The previous results have shown that it is possible to learn modifications to a context-free grammar that represent components of a problem that are useful. This form of specialising the search bias, as the evolution proceeds, also improves the performance of the learning system. Although these results are quite positive, a more important issue exists with the concept of modifying the initial grammar, G^0. Namely, whether is it possible to evolve a grammar that can generalise from one problem to another, similar, problem. This is the question that is now investigated.

5.5 Incremental Learning

The goal of incremental learning is to build a system which gradually adapts to a particular class of problems. This means structures that are learnt from a simple problem may be applied to progressively more complex problems *from the same class*. This section will demonstrate that a learnt grammar not only focusses towards a particular problem, but may be used as a good bias for a more difficult problem which has similar structure. The following steps will be followed to illustrate this process.

(a) Learn a series of grammars for the 6-multiplexer.

(b) Select one grammar from this series to be applied to the 11-multiplexer. This selection will be performed by using a small random training set from the 11-multiplexer. The results of applying this training set will give an indication of the performance of each grammar. The best performing grammar, from this series, will be used as the biased grammar for the 11-multiplexer.

(c) Test the 11-multiplexer using the selected learnt grammar and an unbiased grammar. The resulting best, average and worst fitness over 100 runs will be used to indicate the improved performance of the learnt grammatical bias.

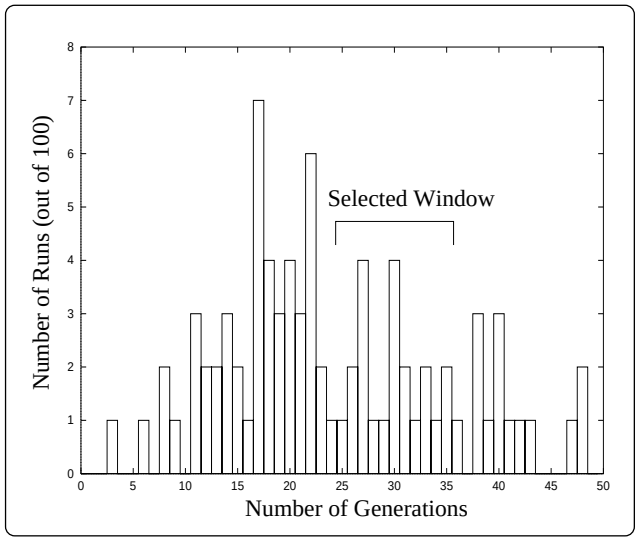


Figure 5.6: The Generation Distribution of Successful runs using $G_{6m-ifonly}$.

5.5.1 Learning the Initial (Generalised) 6-multiplexer Grammar

The selected grammar to be used with the 6-multiplexer was extremely simple. This meant that there was a possibility of this unbiased grammar being applied to the 11-multiplexer with some success, given a limitation on population size and generations. The grammar will be referred to as $G_{6m-ifonly}$, and is defined below.

$$
\begin{aligned}
&G_{6m-ifonly} = \\
&\quad \{S, \\
&\quad\ N = \{B, DATA, ADDRESS\}, \\
&\quad\ \textstyle\sum = \{if, a0, a1, d0, d1, d2, d3\}, \\
&\quad\ P = \\
&\quad\quad \{S \rightarrow B \\
&\quad\quad\ B \rightarrow if\ B\ B\ B \mid DATA \mid ADDRESS \\
&\quad\quad\ ADDRESS \rightarrow a0 \mid a1 \\
&\quad\quad\ DATA \rightarrow \mid d0 \mid d1 \mid d2 \mid d3 \\
&\quad\quad \} \\
&\quad \}
\end{aligned}
$$

The grammar was applied with the CFG-GP setup shown in Table 5.1. The small search space, as defined by $G_{6m-ifonly}$, meant that the probability of success was very high. Over the 100 runs, p_s was determined as 83%. The frequency distribution, based on the number of generations to discover a solution, is shown in Figure 5.6.

The goal of applying this process is to create a suitable grammar to be applied to the 11-multiplexer. The extension to the 11-multiplexer requires the additional terminal value for the address line, $a2$, and the data lines, $d4$, $d5$, $d6$ and $d7$, to be introduced into the learnt grammar. The nonterminals $DATA$ and $ADDRESS$ were used so that the grammar that was learnt could distinguish between these concepts.

5.5.2 Selecting the Biased Grammar from the 6-multiplexer

Based on the results of Section 5.4.1, learnt grammars gradually become more focussed towards the solution as the number of grammar modifying operations (i.e. the number of generations) increases. This is a direct consequence of the grammar modifying operations trying to strengthen the initially weak bias. When applying a learnt grammar to a new, more difficult, problem the grammar *must not be overly focussed*. A grammar that generates solutions specifically for the 6-multiplexer will tend to converged to this partial solution when applied to the 11-multiplexer. The solution to the 6-multiplexer will incorrectly classify 512 examples when the 11-multiplexer is tested. This figure of 512 gives a base-line measure of the minimum expected result that should be achieved by the biased grammar.

The grammars that were created by successful runs, discovering a complete solution between 25 and 35 generations inclusive, were selected as the initial set of grammars. This range was chosen since the evolved grammars had been changed (i.e. each grammar had between 25 and 35 modifications) but had not become *overly focussed* to the 6-multiplexer. This selection gave 21 candidate grammars. Each selected grammar was applied to a randomly selected subset of 30 training examples from the 11-multiplexer, using the population setup from Table 5.1. Once the training set had been solved or 50 generations had passed, the best program that had been discovered was applied to the remaining 2018 cases of the 11-multiplexer problem. The resulting fitness, applied to this test set, was assumed to represent a measure of how well the grammar was performing at generating solutions to the 11-multiplexer problem. This process was repeated 100 times to give an average performance measure for each grammar. The training example scatter is shown in Figure 5.7, where 30 values for the 11-multiplexer have been selected at random. The selected values are shown by an unbroken line, where the spread of values represent the data space for the 11-multiplexer. The scatter of the 64, 6-multiplexer values, for this data space are shown by broken lines.

The best performing grammar, from the 21 selected grammars, had an average fitness (i.e. average misclassification error over 2018 test cases) of 611 ± 128, a best solution with error 382 and a worst error of 974. This selected grammar, referred to as $G^{27}_{6m-biased}$, was

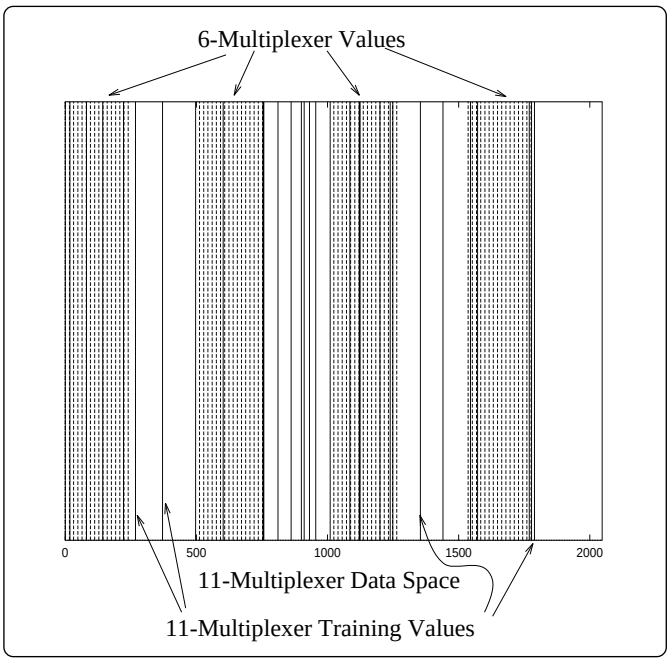


Figure 5.7: The Distribution of Training Cases Used to Select the Biased Grammar.

created after 27 generations and is shown below.

$$
\begin{aligned}
&G^{27}_{6m-biased} = \\
&\quad \{S, \\
&\quad N = \{B, DATA, ADDRESS, N_1, N_2, N_3, N_4, N_5, N_6\}, \\
&\quad \sum = \{if, a0, a1, d0, d1, d2, d3\}, \\
&\quad P = \\
&\quad \{S \rightarrow B \mid N_1 \\
&\quad B \rightarrow \overset{2}{if}\ a1\ DATA\ B \mid if\ B\ B\ d0 \mid if\ a0\ d0\ B \mid if\ a1\ B\ N_1 \\
&\quad B \rightarrow if\ ADDRESS\ DATA\ DATA \mid if\ ADDRESS\ d3\ B \mid if\ a0\ B\ B \\
&\quad B \rightarrow if\ ADDRESS\ B\ DATA \mid if\ a0\ DATA\ N_1 \mid if\ a0\ DATA\ B \\
&\quad B \rightarrow if\ N_3\ B\ DATA \mid if\ a1\ B\ B \mid if\ B\ N_2\ B \mid if\ ADDRESS\ B\ B \\
&\quad B \rightarrow if\ ADDRESS\ DATA\ B \mid if\ B\ ADDRESS\ B \mid if\ B\ B\ DATA \\
&\quad B \rightarrow if\ B\ B\ B \mid DATA \mid ADDRESS \\
&\quad N_1 \rightarrow \overset{2}{if}\ a0\ B\ B \mid \overset{2}{if}\ ADDRESS\ B\ DATA \mid if\ ADDRESS\ B\ B \\
&\quad N_1 \rightarrow if\ ADDRESS\ B\ N_6 \mid if\ ADDRESS\ N_5\ B \\
&\quad N_2 \rightarrow if\ B\ N_4\ B \mid if\ B\ B\ B \\
&\quad N_3 \rightarrow if\ B\ B\ DATA \\
&\quad N_4 \rightarrow if\ a0\ B\ B
\end{aligned}
$$

The 11-Multiplexer	
Parameters	Specifications
POPULATION SIZE	500
CREATE MAXIMUM DEPTH 5	200
CREATE MAXIMUM DEPTH 6	100
CREATE MAXIMUM DEPTH 7	100
CREATE MAXIMUM DEPTH 8	100
MAXIMUM FAILURES	100
MAX DEPTH PROGRAMS	10
$\otimes = \{B\}$	90%
REPLACEMENT	10%
GENERATIONS	50
FITNESS MEASURE	2048 boolean cases
GRAMMAR	$G^{27}_{6m-biased-11m}$

Table 5.4: The 11-Multiplexer System.

$N_5 \rightarrow if\ B\ ADDRESS\ B$

$N_6 \rightarrow if\ a0\ DATA\ N_1$

$ADDRESS \rightarrow a0 \mid a1$

$DATA \rightarrow \mid d0 \mid d1 \mid d2 \mid d3$

}

}

The previous method for selecting the grammar was undertaken to avoid introducing any human bias into the selection procedure. Apart from the use of a generation window to select a subset of the 6-multiplexer grammars, the selection of the final grammar was automatic.

5.5.3 Applying the Grammar, $G^{27}_{6m-biased-11m}$, to the 11-Multiplexer

The grammar, $G^{27}_{6m-biased}$, was applied to the 11-multiplexer, by extending the $ADDRESS$ and $DATA$ productions to include the additional terms introduced by the 11-multiplexer. This, extended grammar, is referred to as $G^{27}_{6m-biased-11m}$. The setup, using this grammar, is shown in Table 5.4. The only change for this setup, from the initial 6-multiplexer example, is that the $MAX\ DEPTH\ PROGRAMS$ parameter has been increased to 10, to account for the additional complexity of the 11-multiplexer.

The 11-Multiplexer Results			
Grammar	Best	Average	Worst
$G_{11m-ifonly}$	448	668 ± 72	768
$G^{27}_{6m-biased-11m}$	128	333 ± 94	512

Table 5.5: Incremental Learning: Results for the 11-Multiplexer.

The complete 2048 test cases were used as the fitness measure and the crossover operator was set to select nodes in the derivation tree labelled by the nonterminal, B. This is essentially the same setup as that used to train the grammar with the 6-multiplexer. The measure of improvement was derived from applying the unbiased grammar, $G_{11m-ifonly}$, using the setup of Table 5.4. This grammar is an extension of $G_{6m-ifonly}$, where the $ADDRESS$ and $DATA$ productions have been extended to handled the additional values associated with the 11-multiplexer. The results, for each of these grammars, is shown in Table 5.5, where the best, average (with standard deviation) and worst result over 100 runs is shown.

These results clearly show that the learnt grammar, $G^{27}_{6m-biased-11m}$, outperforms the original, unbiased grammar. This implies that the learnt grammar represents useful properties from one, simple problem, that may be extended to more complex problems from the same class. It is worth noting that the base line performance, namely 512 incorrectly classified examples for the 6-multiplexer solution applied to the 11-multiplexer, was the worst result with the grammar ,$G^{27}_{6m-biased-11m}$. This implies that $G^{27}_{6m-biased-11m}$ is, at worst, producing the partial result based on the 6-multiplexer and, at best, using this as a stepping stone to the next level of complexity which captures more information about the 11-multiplexer problem.

5.6 Discussion

The feedback process, introduced in Section 5.3.4, shows one method of modifying a declarative language bias to improve the convergence of the program induction system, CFG-GP. The $REPLACEMENT$ operator allows CFG-GP to perform the search for a program on a second level, by explicitly capturing useful components of fit programs. The selection pressure, via fitness proportionate selection, will exploit these discovered components if they give an advantage to generated programs. This process has been shown to generally improve the performance of CFG-GP. However, this feedback may also cause a premature convergence to a sub-optimal solution, as has been seen with the grammar, $G^{50}_{6m-address}$. This type of problem can be minimised by reducing the probability of $REPLACEMENT$ occurring. Premature convergence can also occur since the search

bias is not adjusted to reflect the new nonterminals that are introduced into the language. Eventually, the learnt grammar may create programs where there are *no legal crossover or mutation sites*. These static programs may propagate through the population and cause the search process to slow down.

Learning to modify the search bias, with Genetic Programming, has been investigated by Angeline[1], where each node in a program had an adaptive value for the likelihood of crossover occurring *at that site*. A similar approach may be possible with CFG-GP, where the declarative crossover sites may be adjusted as the evolution proceeds. A simple extension would be to include any new nonterminals, N_i, into the set representing the legal crossover sites, $\otimes$, thus allowing the learnt building blocks to be swapped between fit programs, whilst maintaining the structure that has been previously found to be useful.

The random selection of a terminal site may be improved by using information measures to select an appropriate terminal to direct the modification of the grammar. Techniques, such as those demonstrated by Rosca and Ballard, may offer some direction when trying to create such a measure. The random selection of a terminal site was found to be appealing as there were no assumptions made about the structure or execution style of the generated programs. Hence, although other selection methods may have improved the performance of the $REPLACEMENT$ operator[7], it was felt that the method should not rely on heuristics that were problem or implementation specific.

5.7 Conclusion

This chapter has demonstrated a framework for automatically learning to modify an initially weak language bias. The described process gradually modifies this initial bias by exchanging information between the evolving grammar and the fittest program individual, each generation. This technique has been demonstrated to improve the convergence of CFG-GP for the 6-multiplexer. The modified grammar, if used as the initial language bias for the same problem, has been shown to improve the performance of CFG-GP. This shows that the learnt grammar has been modified to become biased towards solving this particular problem. Additionally, the learning system has been shown to be capable of creating a grammatical bias which may be applied between examples for a class of problems. This has significant implications when applying CFG-GP, since the importance of creating a good language bias has been weakened. Although selecting a good initial grammar is still of paramount importance, the learning system may assist the user to discover useful constructs in the language. This will often allow a solution to be developed more easily.

[7] Early experiments found that selecting the *deepest, left most terminal* gave the best performance results with the multiplexer. This was a result of the preorder execution of the programs and the form of the problem that was being solved. Hence, the improvement was not felt to be a generalised characteristic and was ignored.

Chapter 6

A Schema Theorem for Program Induction

This chapter describes a schema theorem for the program induction system, CFG-GP, defined in Chapter 3. A definition of a schema is presented that uses the notion of a *partial derivation tree* to represent components of a program. This definition allows a schema theorem to be developed for the search operators of *selective crossover*, *selective mutation* and *directed mutation*. The flexible nature of a grammar allows this theorem to describe both fixed-length and variable-length program structures. This is demonstrated by defining a grammar that represents a fixed-length binary string and showing that the disruption to schemata, for this grammar, is equivalent to both single-point crossover and single-point mutation for a genetic algorithm. Additionally, a mapping to a grammar is described that shows how this theorem may be used as a representation of Genetic Programming.

6.1 Introduction

The *schema theorem*[1] for genetic algorithms (GA), first proposed by John Holland[28], attempts to explain the search behaviour for this class of adaptive algorithms. This theorem describes how similarity templates are propagated during the evolution of a population of binary strings. These similarity templates are referred to as *schemata* and represent partitions of the search space, often referred to as hyperplanes. The power of the genetic algorithm is argued to have been derived from the *implicit parallelism* which occurs when a population of strings is evaluated. Many different hyperplanes are evaluated in an implicit manner each time a string from the population is evaluated. The evaluation of a population of strings gives a statistical measure of the fitness for each hyperplane.

[1] The *schema theorem* is also known as the *Fundamental Theorem of Genetic Algorithms*.

Although there has been some criticism of this explanation, the formulation of the schema theorem has been useful in guiding discussion about the properties and expected performance when applying a GA to a variety of problem domains. A brief introduction to the concepts and terminology commonly used when describing this theorem is presented in Appendix C.

The formulation of the schema theorem for CFG-GP will proceed as follows. The definition of a similarity template, H, for a context-free grammar will be presented. The disruption to this template, due to the genetic operators described in Chapter 3, will than be calculated. Using these expressions, a schema theorem will be presented, based on the dynamics of the CFG-GP system. The general nature of this definition will be demonstrated by presenting a grammar that mimics the fixed-length characteristics of a GA. Additionally, a second grammar will be defined that allows the system to capture many of the properties of the Genetic Programming formalism.

6.2 The Definition of H for a Context-Free Grammar

A similarity template for a program, defined by a context-free grammar, is represented by a *partial derivation tree rooted in some nonterminal.* This allows any component of a program to be considered as a template. The definition of H for binary strings requires a special symbol to be introduced, representing the *don't care* condition. This is not necessary for the grammatical definition of H, since every nonterminal in a *partial derivation* implicitly represents all legal strings that are derivable from the nonterminal.

Definition 1 *A schema H for a context-free grammar is the (partial) derivation tree $A \stackrel{*}{\Rightarrow} \theta$, where $A \in N$ and $\theta \in \{N \cup \sum\}^*$.*

For example, consider the following derivation steps.

$$S \stackrel{S \rightarrow and\, B\, B}{\Rightarrow} and\ B\ B \stackrel{B \rightarrow x}{\Rightarrow} and\ x\ B \stackrel{B \rightarrow y}{\Rightarrow} and\ x\ y$$

The following schemata are represented by the derivation tree associated with these steps.

$S \Rightarrow$, $B \Rightarrow$, $S \stackrel{+}{\Rightarrow} and\ B\ B$, $S \stackrel{+}{\Rightarrow} and\ x\ B$, $S \stackrel{+}{\Rightarrow} and\ B\ y$, $S \stackrel{+}{\Rightarrow} and\ x\ y$, $B \stackrel{+}{\Rightarrow} x$, $B \stackrel{+}{\Rightarrow} y$.

The next three sections describe the probability of disrupting $A \stackrel{*}{\Rightarrow} \theta$, for each of the genetic operators described in Chapter 3. For simplicity, the assumption will be made that any particular schema *exists only once* for each derivation tree in the population. (This assumption will be relaxed in Section 6.6 to give the complete schema theorem for CFG-GP.)

6.3 Schema Disruption due to Selective Crossover

The *selective crossover* operator randomly selects a nonterminal site for crossover from the set $\otimes$. The disruption to $A \stackrel{*}{\Rightarrow} \theta$ occurs with a probability based on the number of legal sites from $\otimes$ within the schema. However, the legal crossover sites *on the frontier of* θ must be discounted, due to the lack of disruption occuring to the schema at these nonterminals.

Proposition 1 *The probability of the schema* $A \stackrel{*}{\Rightarrow} \theta$*, within derivation tree* $\rho \in D_i(G)$*, being disrupted during* selective crossover*, where* $\otimes \subseteq N$*, is defined as follows.*

$$\chi_{\otimes} = \begin{cases} \dfrac{\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} - \mid \theta \mid_{\otimes} -1}{\mid \rho \mid_{\otimes}} & A \in \otimes \\ \dfrac{\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} - \mid \theta \mid_{\otimes}}{\mid \rho \mid_{\otimes}} & A \notin \otimes \end{cases}$$

Here,$\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes}$ *represents the number of nonterminals* $\in \otimes$ *that occur in the schema,* $\mid \theta \mid_{\otimes}$ *is the number of nonterminals* $\in \otimes$ *at the frontier of the schema,* $\mid \rho \mid_{\otimes}$ *is the number of nonterminals* $\in \otimes$ *in the total derivation tree* rooted in S.

Note that if $\mid \rho \mid_{\otimes} = 0$, then $\chi_{\otimes} = 0$.

6.4 Schema Disruption due to Selective Mutation

The *selective mutation* operator randomly selects a nonterminal site for mutation from the set $\odot$. The disruption to $A \stackrel{*}{\Rightarrow} \theta$ occurs with a probability based on the number of legal sites from $\odot$ within the schema. However, the legal crossover sites *on the frontier of* θ must be discounted, due to the lack of disruption occuring to the schema at these nonterminals. An additional term for disruption occurs with *selective mutation*, since selecting *any path on the derivation tree* that leads to the schema, will remove this schema from the derivation tree.

Proposition 2 *The probability of the schema* $A \stackrel{*}{\Rightarrow} \theta$*, within derivation tree* $\rho \in D_i(G)$*, being disrupted during* selective mutation*, where* $\odot \subseteq N$*, is defined as follows.*

$$\psi_{\odot} = \begin{cases} \dfrac{\mid S, .., A \mid_{\odot} + \mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} - \mid \theta \mid_{\odot} -1}{\mid \rho \mid_{\odot}} & A \in \odot \\ \dfrac{\mid S, .., A \mid_{\odot} + \mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} - \mid \theta \mid_{\odot}}{\mid \rho \mid_{\odot}} & A \notin \odot \end{cases}$$

Here, $\mid S, .., A \mid_{\odot}$ *represents the number of nonterminals* $\in \odot$ *in the derivation tree from the start symbol to the nonterminal A,* $\mid \theta \mid_{\odot}$ *is the number of nonterminals* $\in \odot$ *at the*

frontier of the schema and $|\ \rho\ |_{\odot}$ *is the number of nonterminals* $\in \odot$ *in the total derivation tree* rooted in S.

Note that if $|\ \rho\ |_{\odot} = 0$, then $\psi_{\odot} = 0$.

6.5 Schema Disruption due to Directed Mutation

The *directed mutation*, $B \rightarrow \alpha \triangleright \beta$, selects, at random, the root of a derivation step, using production $B \rightarrow \alpha$ and replaces it with the derivation using production $B \rightarrow \beta$. The disruption to a schema occurs when the schema is contained within the derivation $B \rightarrow \alpha$ which is selected for directed mutation. Additionally, a schema will be disrupted if the derivation selected for *directed mutation* appears on the path from the start symbol S to the head of the schema, A. This may be viewed as a special case of *selective mutation*.

Proposition 3 *The probability of the schema* $A \stackrel{*}{\Rightarrow} \theta$, *within derivation tree* $\rho \in D_i(G)$, *being disrupted during* directed mutation $B \rightarrow \alpha \triangleright \beta$, *is defined as follows.*

$$\varphi_{B \rightarrow \alpha \triangleright \beta} = \frac{|\ S, .., A\ |_{B \rightarrow \alpha} + |\ A \stackrel{*}{\Rightarrow} \theta\ |_{B \rightarrow \alpha}}{|\ \rho\ |_{B \rightarrow \alpha}}$$

Here, $|\ S, .., A\ |_{B \rightarrow \alpha}$ *represents the number of derivations* $B \rightarrow \alpha$ *that appear on the path from* S *to the head of the schema,* A, $|\ A \stackrel{*}{\Rightarrow} \theta\ |_{B \rightarrow \alpha}$ *represents the number of derivations, in the schema, which have used the production* $B \rightarrow \alpha$, $|\ \rho\ |_{B \rightarrow \alpha}$ *is the total number of derivations* $B \rightarrow \alpha$ *that occur in the derivation tree* ρ.

Note that if $|\ \rho\ |_{B \rightarrow \alpha} = 0$, then $\varphi_{B \rightarrow \alpha} = 0$.

6.6 A Schema Theorem for CFG-GP

The GA schema theorem implicitly assumes that all members of a population *are the same size*. This cannot be assumed when using the grammatical description of programs, and results in two observations[57, 88].

- A schema $A \stackrel{*}{\Rightarrow} \theta$ may exist more than once in a population member $\rho \in D_i(G)$.
- The probability of disrupting a schema will depend on the size of the population member ρ which contains the schema. This is clear from the previous definitions of disruption. As the size of the derivation tree ρ increases, the probability of disruption will generally decrease.

Before presenting the schema theorem for CFG-GP, these variations must be incorporated into the basic definition of χ, ψ and φ.

The average fitness of a schema, within a population $D_i(G)$ of derivation trees, is calculated as follows.

$$\hat{f}\,(A \stackrel{*}{\Rightarrow} \theta) = \frac{\sum_{\rho \in D_i(G)} f(\rho).\mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

where $\sum_{\rho \in D_i(G)}$ indicates a sum over the n derivation trees that exist in the population, $f(\rho)$ is the fitness of ρ and $\mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$ represents the number of schema in the derivation tree ρ.

The average disruption to a schema, due to *selective crossover*, within a population $D_i(G)$ of derivation trees, is calculated by summing the number of schema in each ρ, multiplied by the probability of disruption for this schema for each ρ. This total disruption must be divided by the number of schema that exist in the entire population to give an average measure of disruption. Since each instance of the schema will have the same probability of disruption, within a derivation tree ρ, the average disruption due to *selective crossover* is defined as follows.

$$\bar{\chi} = \frac{\sum_{\rho \in D_i(G)} \chi.\mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

The average disruption to a schema, due to *selective mutation*, is more complex then the previous average disruption for *selective crossover*. Why? The disruption to a schema for *selective mutation* must account for the path, $\mid S, .., A \mid_{\odot}$, from the root of the derivation tree to the head of the schema, A. Two identical schema in the same derivation tree, ρ, will have different paths to their commencing nonterminal, A, and therefore may have different probabilities of disruption. The average disruption *due to selective mutation* is defined as follows.

$$\bar{\psi} = \frac{\sum_{\rho \in D_i(G)} \sum_{A \stackrel{*}{\Rightarrow} \theta \in \rho} \psi}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

Here, the summation over ψ indicates that the value of ψ *for each instance* of the schema $A \stackrel{*}{\Rightarrow} \theta$ which occurs in the derivation tree ρ must be summed to give the average disruption for the derivation tree ρ.

In a similar manner, the average disruption of a schema, *due to directed mutation*, is defined as follows.

$$\bar{\varphi} = \frac{\sum_{\rho \in D_i(G)} \sum_{A \stackrel{*}{\Rightarrow} \theta \in \rho} \varphi}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

From the previous statements the variable-length nature of the programs generated from a grammar may be incorporated into the description of schema disruption. The following assumptions are now made, to allow the schema theorem for programs to be simplified.

(a) The possibility of a schema being reintroduced during *selective crossover*, *selective mutation* or *directed mutation* will be ignored. The schemata that are created when the mutation operators generate new derivation trees based on the grammar are similarly excluded.

(b) The schemata that are explicitly introduced by the *directed mutation* production $B \rightarrow \beta$ will be ignored.

(c) The program induction system has been defined to allow many different crossover and mutation operations. For example, there can be several different $\otimes$ sets defined, each with a different probability of being applied. To simplify the schema theorem only one *selective crossover*, *selective mutation* and *directed mutation* operator will be allowed. The probability of each of these operations occuring will be represented by p_c, p_m and p_d, respectively.

In developing a schema theorem for CFG-GP, the expected number of instances of $A \stackrel{*}{\Rightarrow} \theta$ which will be propagated to the next generation, due to proportional fitness selection, must first be calculated. Let $H = A \stackrel{*}{\Rightarrow} \theta$ and $m(H,t)$ represent the number of schema in the population at generation t.

Lemma 1 $$m(H,t+1) = m(H,t)\frac{\hat{f}\,(H,t)}{\bar{f}\,(t)}$$

where $\hat{f}\,(H,t)$ *is the average fitness of schema* H *in the population at generation* t*, and* $\bar{f}\,(t)$ *is the average fitness of all programs in the population at generation* t*.*

Proof: Let n be the number of programs (derivation trees) in the population. The number of times a single derivation tree ρ instantiates a schema is given by $\mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$. The number of instances of a schema in the total population is therefore given by

$$\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$$

A derivation tree, ρ, with fitness $f(\rho)$ is reproduced by fitness proportionate selection with probability:

$$\frac{n.f(\rho)}{\sum_{q \in D_i(G)} f(q)}$$

Therefore, the number of instantiations of a schema in the next generation, due to ρ, will be:

$$\frac{n.f(\rho)}{\sum_{q \in D_i(G)} f(q)}. \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$$

Hence, the total number of instantiations of $A \stackrel{*}{\Rightarrow} \theta$, due to the total population in generation $t+1$, will be:

$$\sum_{\rho \in D_i(G)} \frac{n.f(\rho)}{\sum_{q \in D_i(G)} f(q)}. \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$$

Now, the average fitness of the population in generation t is given by:

$$\bar{f}\ (t) = \frac{1}{n} \sum_{q \in D_i(G)} f(q)$$

Thus,

$$m(H, t+1) = \frac{1}{\bar{f}\ (t)}. \sum_{\rho \in D_i(G)} f(\rho). \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}$$

$$= m(H,t) \frac{1}{\bar{f}\ (t)}. \frac{\sum_{\rho \in D_i(G)} f(\rho). \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

Since

$$\hat{f}\ (A \stackrel{*}{\Rightarrow} \theta) = \frac{\sum_{\rho \in D_i(G)} f(\rho). \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}{\sum_{\rho \in D_i(G)} \mid \rho \mid_{A \stackrel{*}{\Rightarrow} \theta}}$$

the previous statement may be rewritten as

$$m(H, t+1) = m(H,t) \frac{\hat{f}\ (H,t)}{\bar{f}\ (t)}$$

□

Theorem 1 *The schema theorem for program induction, with a population size of* n*, using* selective crossover, selective mutation, directed mutation *and fitness proportionate selection, where* $H = A \stackrel{*}{\Rightarrow} \theta$*, may be stated as follows.*

$$m(H,t+1) \geq m(H,t)\frac{\hat{f}(H,t)}{\bar{f}(t)} \times \left\{ (1 - p_c\ \bar{\chi}_{\otimes}(t))(1 - \frac{1}{n} m(H,t)\frac{\hat{f}(H,t)}{\bar{f}(t)})(1 - p_m\ \bar{\psi}_{\odot}(t))(1 - p_d\ \bar{\varphi}_{B \rightarrow \alpha \triangleright \beta}(t)) \right\}$$

Proof:

From **Lemma 1**, the expected number of schema reproduced to the next generation is

$$m(H,t)\frac{\hat{f}(H,t)}{\bar{f}(t)}$$

Considering the disruptive impact of *selective crossover*, the upper bound on the probability that a schema will be disrupted has been previously shown to be $\bar{\chi}_{\otimes}(t)$ for generation t. Therefore, the lower bound on a schema surviving due to *selective crossover* will be $(1 - p_c\ \bar{\chi}_{\otimes}(t))$. In a similar manner, the lower bound of schema survival due to *selective* and *directed mutation* is $(1 - p_m\ \bar{\psi}_{\odot}(t))$ and $(1 - p_d\ \bar{\varphi}_{B \rightarrow \alpha \triangleright \beta}(t))$, respectively. These disruptions may be independently treated and are therefore combined as shown in **Theorem 1**. The additional term,

$$(1 - \frac{1}{n} m(H,t)\frac{\hat{f}(H,t)}{\bar{f}(t)})$$

represents the effect of selecting both derivation trees (programs) for crossover using fitness proportionate selection, in a manner similar to that shown with the GA Schema Theorem of Appendix C.

□

6.7 A Representation of Fixed-Length Schemata

This section demonstrates the general nature of the grammatical definition of schemata that has been previously presented. This will be achieved by defining a grammar for fixed-length binary strings and showing that the disruption to the schema, $A \stackrel{*}{\Rightarrow} \theta$, is equivalent to the disruption of the GA schema, H. The concepts of *defining length,* $\delta(H)$, and *schema order,* $o(H)$, for a GA are introduced in Appendix C. These GA concepts will

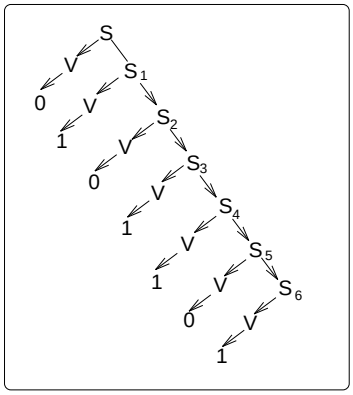


Figure 6.1: An Example Derivation Tree for 7-bit String using GA_7.

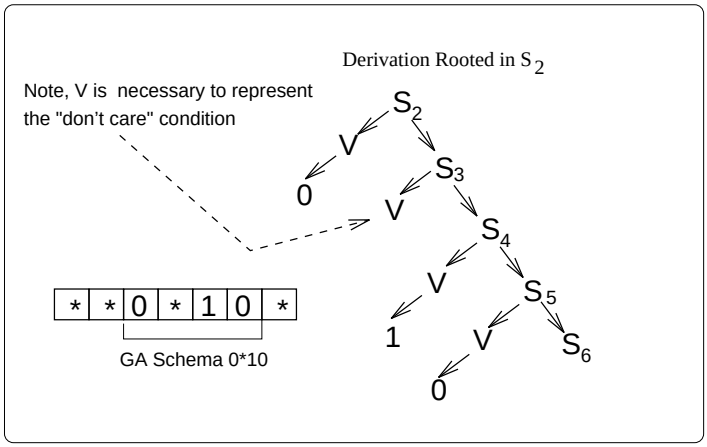


Figure 6.2: Equivalent Schema Representations between GA_l and a Binary String.

be matched to components of the grammar, so that expressions about disruption between the grammar and a fixed-length binary string may be made.

A grammar GA_l, which represents the language of fixed-length binary strings of length $l \geq 2$, may be defined as follows.

$$
\begin{aligned}
GA_l = \{ & S, N = \{V, S_1, \ldots, S_{l-1}\}, \textstyle\sum = \{0, 1\}, \\
& P = \{\ S \rightarrow V\ S_1 \\
& \qquad S_1 \rightarrow V\ S_2 \\
& \qquad S_2 \rightarrow V\ S_3 \\
& \qquad \ldots \\
& \qquad S_{l-1} \rightarrow V \\
& \qquad V \rightarrow 0 \mid 1\ \} \\
& \}
\end{aligned}
$$

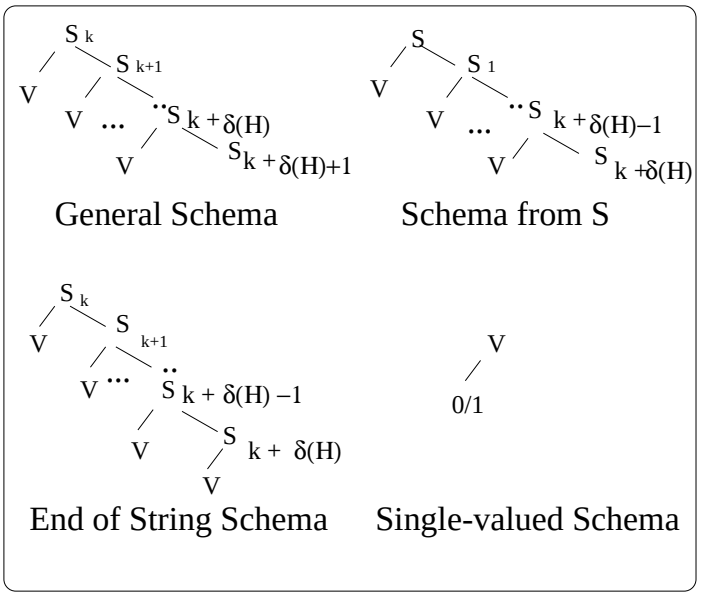


Figure 6.3: The Four Classes of Grammatical Schemata for GA_l.

Figure 6.1 shows an example of a derivation tree for a 7-bit string using GA_7. The grammar uses the nonterminal, V, so that schemata may be represented which do not have all 0 and 1 values explicitly defined. This is shown in Figure 6.2, where the represented schema may be written as follows.

$$S_2 \stackrel{+}{\Rightarrow} 0\ V\ 1\ 0\ S_6$$

The nonterminal, S_6, is used to represent *the rest of the string*. The nonterminals V and S_6 are at the *frontier of the derivation*.

6.7.1 Schemata Disruption for GA_l

The schemata represented within the grammar, GA_l, may be generalised to four classes, as shown in Figure 6.3. By calculating the disruption to schemata, for each of these classes, all possible schema disruptions may be established for the genetic operators of *selective crossover* and *selective mutation*. The sets, $\otimes$ and $\odot$, will be selected so that equivalent genetic operators to those of a GA, for crossover and mutation, may be formed. The disruption to the schemata of GA_l and an equivalent fixed-length binary string will then be shown to be identical. To commence this analysis, the GA schema properties for defining length and schema order are matched to the grammar, GA_l. This will allow the expressions, $\chi_{\otimes}$ and $\psi_{\odot}$, to be directly compared with the genetic algorithm disruption.

The GA defining length, $\delta(H)$, represents the length of the schema in terms of the number of nonterminals S_k. The GA schema order, $o(H)$, represents the number of derivation steps which have applied the productions $V \rightarrow 0$ or $V \rightarrow 1$. Using the grammar GA_7 as an example language, these types of schemata have the following interpretation.

- The *General Schema* from S_k represents any schema that does not commence with the start string, S and does not include the final nonterminal S_6. For example, the schema represented by the derivation tree $S_2 \stackrel{+}{\Rightarrow} 0\mathrm{V}1\mathrm{S}_5$ is one form of *General Schema*.
- The *Schema from S* represents any schema that commences with the start symbol. For example, the schema represented by the derivation tree $S \stackrel{+}{\Rightarrow} 0\mathrm{V}1\mathrm{S}_3$ is one form of *Schema from S*.
- The *End of String Schema* represents any schema that includes the final nonterminal S_6, in the string definition. For example, the derivation tree $S_4 \stackrel{+}{\Rightarrow} 1\ V\ 1$ is one form of *End of String Schema*.
- The *Single-valued Schema* represents the simplest form of schema, namely a single bit from the binary string. For example, the derivation tree $V \stackrel{V \rightarrow 0}{\Rightarrow} 0$ is one form of *Single-valued Schema*.

The *selective crossover* operator, using GA_l, may be restricted to a GA single-point crossover by restricting $\otimes$, as follows.

$$\otimes = \{S_1, \ldots, S_{l-1}\}.$$

The *selective mutation* operator, using GA_l, may be restricted to a GA single-point, bit-flipping, mutation by restricting $\odot$, as follows[2].

$$\odot = \{V\}.$$

For the grammar GA_l, the following values are defined for the number of legal crossover and mutation sites involved in a complete derivation tree, ρ.

$$\mid \rho \mid_{\otimes} = l - 1$$

$$\mid \rho \mid_{\odot} = l$$

Based on Figure 6.3, the following statements may be made about the schemata components. Substituting these values into **Proposition 1** or **Proposition 2**, as appropriate, results in the original GA values for disruption of H, due to single-point crossover and bit-flipping mutation. Each of these substitutions will now be shown, where the goal is to show that the disruption, due to crossover, is $\frac{\delta(H)}{l-1}$, and that of mutation is $\frac{o(H)}{l}$.

General Schema from S_k

For the *selective crossover* operator, applied to the *General Schema from S_k*, the head of the schema $A \in \otimes$. The number of crossover sites $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} = \delta(H) + 2$ and the number of crossover sites at the frontier of the schema, $\mid \theta \mid_{\otimes} = 1$, giving $\chi_{\otimes} = \frac{\delta(H)}{l-1}$.

[2]The assumption is made here that *selective mutation* will not reintroduce the string that has been removed due to mutation. This assumption could be explicitly stated by using *directed mutation* for the two cases, $V \rightarrow 0 \rhd 1$ and $V \rightarrow 1 \rhd 0$. For the purpose of demonstrating that the schema disruptions are equivalent, this was not deemed necessary.

For the *selective mutation* operator, applied to the *General Schema from* S_k, the head of the schema $A \notin \odot$. The number of mutation sites $\mid S, \ldots, A \mid_{\odot} = 0$, $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} = \delta(H) + 1$ and $\mid \theta \mid_{\odot} = \delta(H) + 1 - o(H)$, giving $\psi_{\odot} = \frac{o(H)}{l}$.

Schema from S

For the *selective crossover* operator, applied to the *Schema from S*, the head of the schema $A \notin \otimes$. The number of crossover sites $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} = \delta(H) + 1$, and the number of crossover sites at the frontier of the schema, $\mid \theta \mid_{\otimes} = 1$, giving $\chi_{\otimes} = \frac{\delta(H)}{l-1}$.

For the *selective mutation* operator, applied to the *Schema from S*, the head of the schema $A \notin \odot$. The number of mutation sites $\mid S, \ldots, A \mid_{\odot} = 0$, $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} = \delta(H) + 1$ and $\mid \theta \mid_{\odot} = \delta(H) + 1 - o(H)$, giving $\psi_{\odot} = \frac{o(H)}{l}$.

End of String Schema from S_k

For the *selective crossover* operator, applied to the *End of String Schema from* S_k, the head of the schema $A \in \otimes$. The number of crossover sites $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} = \delta(H) + 1$, and the number of crossover sites at the frontier of the schema, $\mid \theta \mid_{\otimes} = 0$, giving $\chi_{\otimes} = \frac{\delta(H)}{l-1}$.

For the *selective mutation* operator, applied to the *End of String Schema from* S_k, the head of the schema $A \notin \odot$. The number of mutation sites $\mid S, \ldots, A \mid_{\odot} = 0$, $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} = \delta(H)+1$ and $\mid \theta \mid_{\odot} = \delta(H) + 1 - o(H)$, giving $\psi_{\odot} = \frac{o(H)}{l}$.

Single-Valued Schema from V

For the *selective crossover* operator, applied to the *Single-Valued Schema from V*, the head of the schema $A \notin \otimes$. The number of crossover sites $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\otimes} = 0$, and the number of crossover sites at the frontier of the schema, $\mid \theta \mid_{\otimes} = 0$, giving $\chi_{\otimes} = 0$.

For the *selective mutation* operator, applied to the *Single-Valued Schema from V*, the head of the schema $A \in \odot$. The number of mutation sites $\mid S, \ldots, A \mid_{\odot} = 1$, $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} = 1$ and $\mid \theta \mid_{\odot} = 0$, giving $\psi_{\odot} = \frac{1}{n} \equiv \frac{o(H)}{l}$.

6.7.2 Discussion of Fixed-Length Grammars

The previous section demonstrates that the disruption to schemata, given by $\chi_{\otimes}$ and $\psi_{\odot}$, may be specialised to the genetic algorithm schema for H. This occurs for the grammar GA_l with $\otimes = \{S_1, \ldots, S_{l-1}\}$ and $\odot = \{V\}$. The disruption due to *selective mutation* is determined as $\frac{o(H)}{l}$, whereas the genetic algorithm disruption to mutation is independent of the length of the string. This occurs because the GA mutation operator is applied with a probability based on *any individual bit from a string* being flipped. This

independent mutation is not defined for *selective mutation*. The difference between these two mechanisms accounts for the differing measures of disruption. If *selective mutation* is defined such that the probability of mutation occurring, at any nonterminal $\in \odot$, is equal and independent, then the disruption to the GA schemata and the context-free grammar, GA_l, is identical.

6.8 CFG-GP as a Generalisation Of Genetic Programming

A schema theorem for genetic programming(GP) has been described by O'Reilly[57]. This extends the fixed-length description of binary strings to tree-structured LISP S-expressions. This was achieved by defining the concept of a **tree fragment**, using a *wildcard* that matched any subtree[3]. The fragment was defined as a tree that has at least one leaf that is a wildcard. The wildcard corresponded to an incomplete S-expression. The entire fragment also had a wildcard *at its root position* to represent the fact that the fragment could be fully embedded in a tree. This led to a definition for tree-structured schema which describes an unordered collection of both completely defined S-expressions and incompletely defined S-expressions. These incomplete S-expressions were represented as fragments. The formal definition for a GP-schema was stated by O'Reilly, as follows[56].

Definition 2 *A* **GP-schema** H *is a set of pairs. Each pair is a unique S-expression tree or fragment (i.e. incomplete S-expression tree with some leaves as wildcards) and a corresponding integer that specifies how many instances of the S-expression tree or fragment comprise* H*.*

Given this definition of **GP-schema**, a corresponding analogy to *defining length* and *schema order* for arbitary tree structures was than developed. The lack of a formal structure with GP programs made the schema theorem difficult to express. For example, several functions had to be introduced to describe how these schemata were represented in the population of programs. It is worth noting the difference in complexity between the **GP-schema** definition and the grammatical schema, $A \stackrel{*}{\Rightarrow} \theta$. The formal structure of a grammar has allowed the structure of components of a program to be easily and clearly stated.

The generalisation of CFG-GP to genetic programming will be shown by defining a mapping between GP constructs and a grammar. The equivalent program forms and number of sites where disruption to schemata occur will be used to argue that the created structures are identical. If the program structures and the operators that change these structures, each generation, are identical then the space of programs, and their transformations, are

[3]The *wildcard* performed the same role as $\star$ for the schema description with genetic algorithms.

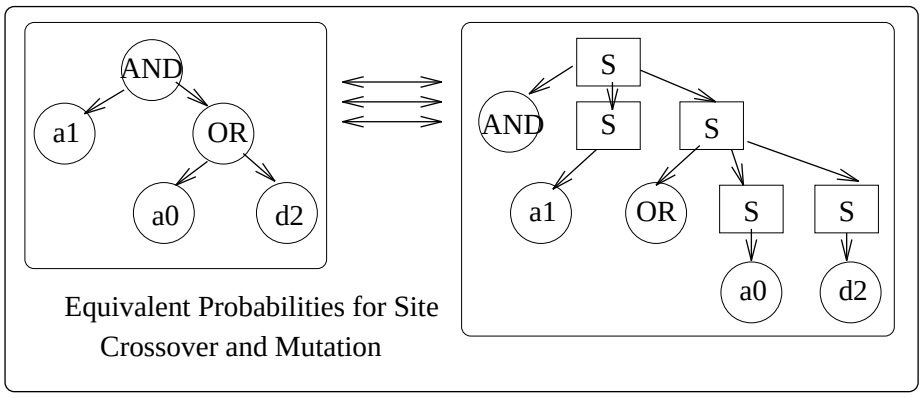


Figure 6.4: Equivalent Structures between Genetic Programming and the language $L(GP_{t,f})$.

the same. Hence, the disruption to schemata for CFG-GP and GP, under these circumstances, will be equivalent.

Given the set of GP terminals $GP_t = \{t1, t2, \ldots, ti\}$ and GP functions $GP_f = \{f1, f2, \ldots, fj\}$ (with number of arguments $a_1, a_2, \ldots, a_j$) we define the grammar $GP_{t,f}$ as follows.

$$
\begin{aligned}
GP_{t,f} = \{&S, \mathrm{N}=\{\ \},\\
&\textstyle\sum = \{t1, t2, \ldots, ti, f1, f2, \ldots, fj\},\\
&P = \{\\
&\quad S \rightarrow f1\ S\ \ldots\ S \mid \ldots \mid fj\ S\ \ldots\ S\\
&\quad S \rightarrow t1 \mid t2 \mid \ldots \mid ti\\
&\quad \}\\
&\}
\end{aligned}
$$

Each GP function fj has a_j number of S's in the production defined for fj. The language, $L(GP_{t,f})$, represents any function that may be represented using the Koza-style GP definition[4].

As shown in Figure 6.4, there is an equal number of sites for crossover or mutation using the GP definition and the grammatical definition with the grammar $GP_{t,f}$. In fact, the crossover defined by GP and *selective crossover*, using $\otimes = \{S\}$, create the same new programs when applied. This implies that the schemata represented by these two descriptions will be disrupted to the same extent[5]. Additionally, mutation may be described by setting $\odot = \{S\}$. This gives the GP mutation operator described by Koza[44], which was not considered in the original schema definition for GP presented by O'Reilly[57]. A schema theorem, for genetic programming, may now be stated as a restricted version of the previous grammatical schema theorem, as follows.

[4]This assumes that no automatically defined functions or other imposed structures are allowed.

[5]This ignores the GP bias that gives 90% crossover to internal nodes. This form of search bias is assumed to be constant over any function or terminal site.

Theorem 2 *The GP schema theorem, using mutation and crossover, is defined when the following conditions are true.*

(a) $\chi_{\otimes}$ *is as in* **Proposition 1**.

(b) $\psi_{\odot}$ *is as in* **Proposition 2**.

(c) $H = A \stackrel{*}{\Rightarrow} \theta$.

(d) $G = GP_{t,f}$.

(e) $\otimes = \{S\}$.

(f) $\odot = \{S\}$.

6.9 Discussion

The genetic technique of mutation is normally viewed as a *local* syntactic search. Crossover is normally viewed as a *global* search. However, with the grammatical genetic definition, *selective mutation* is more disruptive than *selective crossover*. This is a result of the additional term used with disruption, due to mutation, which includes the path from the start symbol to the head of the schema. Since this term, $\mid S, \ldots, A \mid_{\odot}$, will always be greater or equal to zero, the disruption will always be equal or greater. However, as $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot}$ becomes large $\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} \gg \mid S, \ldots, A \mid_{\odot}$. Hence, when the schemata become large, the disruptions, due to crossover and mutation, are equal to a first approximation. Extending this further, if $\otimes = \odot$ then, for large schemata, $\chi_{\otimes} = \psi_{\odot}$. This result partially accounts for the general tendency to omit mutation when applying genetic programming techniques. If the function and terminal sets for GP are not large (which is often the case), then the initial population will contain enough building blocks to support the evolution of a solution. The disruptive nature of mutation will not be a beneficial contributor to the search. The introduction of *selective mutation* and *directed mutation* with CFG-GP avoids this problem, allowing a grammar to be defined where mutation performs the role of a *local search operator*, with minimal disruption.

The independent, bit-level, mutation of GA's may be modelled by assuming that any nonterminal $\in \odot$ has *equal probability* of being mutated in the manner described by *selective mutation*. In this case, the definition for ψ is independent of the size of the derivation tree which contains the schema.

Proposition 4 *The probability of a schema* $A \stackrel{*}{\Rightarrow} \theta$*, within derivation tree* $\rho \in D_i(G)$*, being disrupted during* selective independent mutation*, where the probability of mutation is* p_m*, will be:*

$$(1 - p_m)^{\psi_{\odot}}$$

where

$$\psi_{\odot} = \begin{cases} \mid S, .., A \mid_{\odot} + \mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} - \mid \theta \mid_{\odot} - 1 & A \in \odot \\ \\ \mid S, .., A \mid_{\odot} + \mid A \stackrel{*}{\Rightarrow} \theta \mid_{\odot} - \mid \theta \mid_{\odot} & A \notin \odot. \end{cases}$$

The subsequent definition for the schema theorem, using *selective independent mutation*, requires this definition of ψ to be substituted for **Proposition 2**. This definition of ψ would then give identical values for the disruption of H for a genetic algorithm, when using the grammar GA_l.

6.9.1 Maintaining Schema

To ensure that fit schemata are propogated to future generations, when using the genetic operators of *selective crossover* and *selective mutation*, the following expression must be minimised.

$$\frac{\mid A \stackrel{*}{\Rightarrow} \theta \mid_{\{\otimes \cup \odot\}} - \mid \theta \mid_{\{\otimes \cup \odot\}}}{\mid \rho \mid_{\{\otimes \cup \odot\}}}$$

One possibility is to modify grammar productions to reduce the number of nonterminals $\in \{\otimes \cup \odot\}$ in the derivation tree representing a schema. For example, given the schema $A \stackrel{+}{\Rightarrow} bc$ with the following derivation steps

$$\alpha\ A\ \beta \stackrel{A \rightarrow Bc}{\Rightarrow} \alpha Bc\beta \stackrel{B \rightarrow b}{\Rightarrow} \alpha bc\beta,$$

an alternative production $A \rightarrow bc$, substituted to create the same program string, would improve the chances of the schema surviving. The derivation steps could then be modified to the following single derivation step.

$$\alpha\ A\ \beta \stackrel{A \rightarrow bc}{\Rightarrow} \alpha bc\beta$$

The technique involved with learning grammatical bias, as described in Chapter 5, is one method for developing a grammatical language where the building blocks are discovered and protected by their explicit definition within the defined (and learned) grammar.

Alternatively, the value of $\mid \rho \mid_{\{\otimes \cup \odot\}}$ must increase to protect each schema in a derivation tree. This may account for the condition of *bloating* that is observed with genetic programming[55]. The growth in program depth, of the average population member, appears to go unchecked unless an explicit bound on the maximum depth of a program is enforced. The use of *explicitly introducing introns* into the population has been shown to improve the performance of GP for some problems[55] and therefore suggests that the size of each population member has a direct relationship to the performance and likelihood of success for the population, as a whole.

For example, the grammar GA_l could be extended using a nonterminal C with production

$C \rightarrow \epsilon$. The productions could then be modified as follows.

$$P = \{S \rightarrow CVS_1 \mid CVS_2C \mid \ldots$$

These changes to GA_l do not affect the overall language defined by $L(GA_l)$. They do, however, change the possible intermediate forms of schemata. Future work is required to describe how changes to the grammar, G, and the sets $\otimes$ and $\odot$, relate to building and maintaining schemata, and to the overall performance of CFG-GP. It may also be possible to extend this work into the field of *fitness landscapes*, where a landscape may represent a particular grammar and the operators that apply to this landscape, based on $\otimes$ and $\odot$. These concepts are left for future work.

6.10 Conclusion

This chapter has shown the definition for a schema theorem for the program induction system, CFG-GP. The schema theorem has been shown to subsume the GA and GP schema definitions under certain conditions[6]. Thus, the schema theorem for CFG-GP allows both fixed- and variable-length languages to be defined and represented under the one framework. An analysis of the schema disruption properties have allowed some explanations to be proposed for certain concepts that have arisen in the field of program induction. In particular, the concepts of *bloating*, search bias (in terms of $L(G)$ and the bias defined by the sets $\otimes$ and $\odot$) and learnt bias (in terms of changing the definition of G to protect building blocks of $L(G)$) have been clarified by the implications of the disruption to schemata defined by χ, ψ and φ.

The presentation of a system of learning that incorporates general grammatical definitions and restrictions to crossover and mutation sites suggests further work in investigating forms of G,$\otimes$, $\odot$ and $B \rightarrow \alpha \triangleright \beta$, that form languages with varied properties.

[6] Although the properties of these learning systems differ from CFG-GP (Chapter 3), the basic structures that are manipulated are shown to be equivalent.

Chapter 7

Conclusions

7.1 Introduction

This thesis has presented a unified framework for representing language bias and search bias using an evolutionary learning system. The use of a formal grammar to represent the language bias has allowed the search space to be declaratively represented and to allow a declarative specification of several search techniques. The learning system, CFG-GP, has been applied to a number of artificial problems and a natural resource problem, namely the prediction of Greater Glider density. The results of this work show that CFG-GP may be successfully applied to a learning problem that requires typing, functional structure and where some knowledge about the likely structure of a good solution is known.

The motivation for this work was presented in Chapter 2, where it was argued that bias is an important component of learning systems. Although the early work on bias focussed on systems which are point-based, work using bias with population-based systems has shown that bias is important irrespective of the learning method.

The theoretical results of Valiant[74] and Wolpert et al.[92] have been used to argue that bias is necessary if a learning system is to be applied to a broad range of problem domains. Although the No Free Lunch theorem implies that all learning systems will perform poorly over some domains, the applicability of a system in a given domain should be improved by allowing the language and search methods to be declaratively stated. This promotes exploration of possible solutions without having to change the internals of the learning system and therefore allows expert knowledge to be used to direct the search in an unambigous manner.

The ability of a learning system to shift bias during the search for a solution extends the learning strategies available to the system. For a difficult problem, the ability to narrow the search space may allow the discovery of a solution that would otherwise have been overlooked. The work presented in Chapter 5 has shown that it is possible to adapt the initial grammar to modify the search space and represent generalised properties of the problem. Modifying the

grammar explicitly changes the search bias. Additionally, a modified grammar may be used as input to a subsequent application of CFG-GP. This has been shown by the system learning a biased grammar for the 6-multiplexer which was applied successfully to the 11-multiplexer. This separation of the language specification and the population members allows a declarative method for representing properties that have been discovered as useful by CFG-GP during the evolution of a solution. A further advantage is that the modified grammar may be analysed by the user to assist in creating a grammar that is biased towards promising program structures.

A theoretical analysis, following Holland's Schema Theorem for Genetic Algorithms[28], has been presented in Chapter 6. This work is important in that it demonstrates a unified description for both fixed-length and variable-length structures and their likely propagation during evolution, based on the CFG-GP framework.

7.2 Contributions of this Thesis

Formal Grammars for Language Bias: The use of formal grammars to represent language bias in a genetic programming framework came to us independently of other work. However, we cannot claim priority for this concept as a number of other authors published similar ideas during the same time period. Most other approaches do not maintain the derivation trees to represent the population, preferring to parse the programs to create a derivation tree when required. This has led to problems with ambiguity when parsing was required.

Representing Programs Explicitly by their Derivation Trees: Preserving the derivation tree to represent each population member arose from discussions with William Cohen, but we appear to have been anticipated in this by Mizoguchi et al.[53].

Merit Selection as Bias: The probabilistic representation of productions in the grammar appears to be entirely original within the GP framework.

Selective Search Operators: The generality of the search operators selective crossover and selective mutation appears to be entirely original. This generality arose from the schema theorem work, where the ability to specify particular sites for mutation and crossover was required to permit the modelling of GA operators. The ability to declaratively specify how the search for new programs is to proceed has been empiricially demonstrated in this thesis to be a very useful addition to the GP framework.

Directed Search Operator: The directed mutation operator appears to be entirely original. This operator allows many different partial programs to be represented and modified in a declarative framework. Additionally, the mutation can alter the search space so that syntactically distant constructions may be closely associated.

Learning a Grammar as Search Bias: The method of evolving the initial grammar, described in Chapter 5, appears to be entirely original. Hemmi et al.[27] have proposed a method

for learning how to modify a grammar, however their details were minimal and it appears to describe work in progress. The framework we have presented is quite general and does not remove the possibility of expressing any sentence that was originally derivable from the grammar. The ability to create new nonterminals and to adjust the merit selection values for productions also appears to be original.

Schema Theorem for Program Induction: There is some overlap between O'Reilly's work[56] and the schema theorem described in Chapter 6. However, the grammatical description that has been derived leads to a simpler and more usable formulation of the schema theorem for program induction. In addition, the framework allows both fixed-length and variable-length structures to be described in a unified manner.

Framework for Bias: A framework for representing language and search bias within an evolutionary context has not been previously presented. This framework allows users of the GP paradigm to view their problem in a different manner. For example, they should consider the fitness measure as a selection bias, the function and terminal set as a language bias and crossover as a search bias. This may lead to more emphasis being directed towards controlling and representing these biases.

7.3 Future Work

This section will describe some directions that could be explored as a result of this thesis. The short-term work includes extending the language and search bias to allow greater control over the generated hypotheses. More speculative work includes exploring grammatical forms and their properties in relation to the schema theorem and explicitly representing partial programs as members of the evolving population.

7.3.1 Extending the Language and Search Bias

The work presented here has used a context-free language. Wong et al.[93, 94] have developed a similar learning system which uses a restricted class of context-sensitive languages. A context-sensitive language uses productions of the form $\alpha \rightarrow \beta$, where $\alpha \in \{N \cup \sum\}^+$ and $\beta \in \{N \cup \sum\}^*$. Is it difficult to extend the CFG-GP system to a CSG-GP system? The required changes to CFG-GP would include the following.

- The generation of the initial population would have to be modified so that min-depth-tree, from α, was considered when determining min-depth-tree for the entire production.
- The definition of selective crossover $\otimes$ and selective mutation $\odot$ sets would be extended to allow $\{\sum \cup N\}^+$ to represent crossover and mutation selection sites.

- The definition of directed mutation would be extended to $\alpha \rightarrow \beta \triangleright \gamma$, where $\alpha \in \{\sum \cup N\}^+$. This extends directed mutation merely by having a more specific description of the location to commence mutation.

Each of these modifications is easily incorporated into the CFG-GP framework because the derivation trees are maintained as representations of the population. The main function that would need to be written is one that matches an arbitary combination of terminals and non-terminals at some level of a derivation tree. There are no changes to the evaluation (fitness) method or the overall structure of the original system.

A further issue for research involves describing the forms of bias that cannot be described using a context-free grammar. There is an assumption that a context-sensitive grammar is necessary for some problems. Work is required to eulicidate the forms of bias and types of problems that require this additional expressiveness.

7.3.2 Further Knowledge Representations

The situation may arise where a program is being evolved which can be described in detail apart from several features. These features represent the program components to be searched. The ability to be able to express a partially written program and link a grammar to the components that must be discovered would be a useful addition to the CFG-GP framework. This allows the opportunity to create complex program structures with several distinct components being evolved together. This type of framework would be useful when studying learning theories and co-evolution.

For example, the structure of a sort program could be given, where the inner loop is to evolve based on the grammar, G_{sort}, as follows.

```
int *sort(list, length)
int *list, length;
{
    int i, temp;

    for(i=0;i<length-1;++i)
    {
        temp=list[i];

          L(G_sort)

    }
    return(list);
}
```

The grammar, G_{sort}, could be defined as follows.

$$
\begin{aligned}
&G_{sort} = \\
&\quad \{S, \\
&\qquad N = \{STATEMENTS, STAT1, STAT, VAR, REL\}, \\
&\qquad \sum = \{temp, list[i], list[i+1], if, ;, <, >, =, ==\} \\
&\qquad P = \\
&\qquad\quad \{S \rightarrow STATEMENTS \\
&\qquad\qquad STATEMENTS \rightarrow STAT1 \ ; \ STATEMENTS \mid STAT1 \\
&\qquad\qquad STAT1 \rightarrow if \ (\ STAT \ ) \ \{STAT\}; \mid STAT \ ; \\
&\qquad\qquad STAT \rightarrow VAR \ REL \ VAR \mid VAR = VAR \\
&\qquad\qquad VAR \rightarrow list[i] \mid list[i+1] \mid temp \\
&\qquad\qquad REL \rightarrow < \mid = \mid == \mid > \\
&\qquad\quad \} \\
&\quad \}
\end{aligned}
$$

It may be desirable for some applications to explicitly express constraints in terms of a combination of some terminals in the language. For example, a function that tested two arguments for equality is trivially true if both arguments are syntactically identical. We may wish explicitly define the patterns that cannot be initially generated. A set of explicit constraints to avoid this situation could be defined in a manner similar to the work of Cohen[10], where a language was used to represent constraints about which conditions could be meaningfully generated. The main issues with this form of explicit bias include the selection of a language to represent these constraints and a method to handle these constraints during crossover and mutation. It is worth noting that a directed mutation may also be used to detect and repair program components. It is currently not clear whether using explicit constraints would provide any advantages not already subsumed by the directed mutation operator.

7.3.3 Grammatical Forms and Schemata

The theoretical study of grammars and schema propagation leads to several lines of new research. For example, there are an infinite number of grammars with different schematic representations, which define the same language. This has been exemplified in Chapters 4 and 5, where the grammar has been modified but the underlying language has remained the same. Nevertheless, the emphasis on particular strings generated by the language and on restrictions to the search space defined by the genetic operators, influence the performance of CFG-GP.

For example, the grammar GA_l (Section 6.7) could be extended using a nonterminal C with production $C \rightarrow \epsilon$. The productions could then be modified as follows.

$$P = \{S \rightarrow CVS_1 \mid CVS_2C \mid \ldots$$

These changes to GA_l do not affect the overall language defined by $L(GA_l)$. They do, however, change the possible intermediate forms of schemata. Future work is required to describe how changes to the grammar, G, and the sets $\otimes$ and $\odot$, relate to building and maintaining schemata, and to the overall performance of CFG-GP.

7.3.4 Learning Crossover and Mutation Sites

The work of Angeline[1] has shown that it is possible to adapt the probability of crossover occuring at any particular location within a program. He maintained a tree for each program, where the GP terminals and functions had been replaced by a probability of crossover occuring at each location in the program. This probability was modified using a gaussian noise function, independent of the performance of the programs created by the crossover operation.

A simple extension to CFG-GP would be to allow each crossover and mutation set to adapt the sites where these operators are most likely to apply. For example, the initial $\otimes$ and $\odot$ sets could include all of the nonterminals in the grammar. Initially there would be no preference in selecting which nonterminals were used as the crossover or mutation sites. Subsequent selections for $\otimes$ and $\odot$ would be based on a fitness proportionate measure, much like the *merit selection* for productions. The main issue that arises from this idea is how to update the probabilities during the evolution. Angeline[1] used a gaussian noise function to modify the probabilities of any function or terminal in a GP tree being selected for crossover. This would appear to be one potential avenue to explore. A second approach to updating the probabilities of particular nonterminals would be to use the subsequent fitness of individuals (created from applying a genetic operator at some particular nonterminal) as feedback to the worth of any selected nonterminal site. If this method was successful it may be possible to demonstrate how a building block approach is working with CFG-GP (or if a building block approach occurs at all). This would be achieved by plotting the relative probability of each nonterminal during the evolution of a solution. A grammar, written so that there were several levels of different nonterminals in the final program derivation tree, could be used to show how a program was constructed. A building block hypothesis would predict that the highest relative probabilities of crossover would occur initially towards the bottom of the derivation tree, and gradually move up towards S as a solution was discovered.

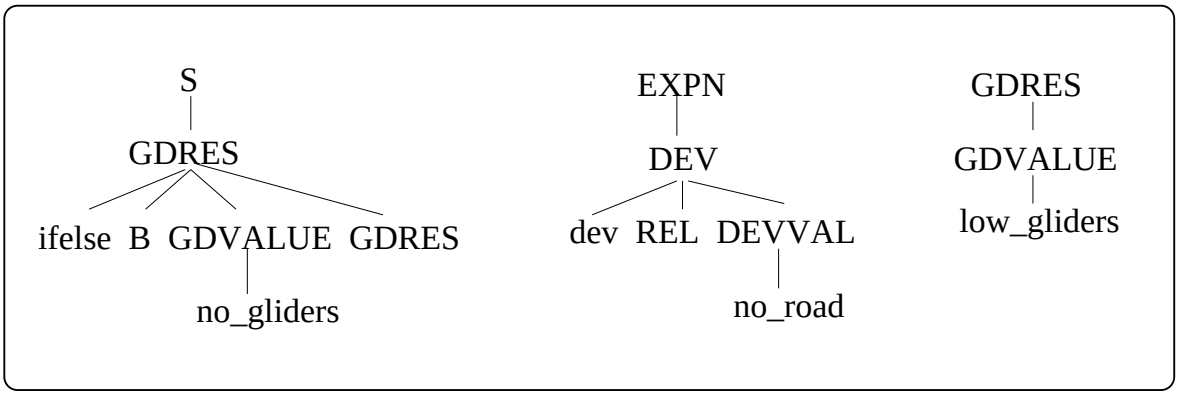


Figure 7.1: Partial Derivation Trees generated using the grammar G_{ggd}.

7.3.5 Extending Directed Mutation

The directed mutation operator could be extended to permit more general specification of contexts in which it is applied. For example, the detection of tautologies in Section 3.9 required both $eq(x, x)$ and $eq(y, y)$ to be specified. A framework based on regular expressions, or indeed a more general representation, would allow the directed mutation to more clearly define the patterns that are recognised and mutated.

7.3.6 Representing Partial Programs Explicitly as $A \stackrel{*}{\Rightarrow} \theta$

The schema definition, $A \stackrel{*}{\Rightarrow} \theta$, presented in Chapter 6, represents the notion that a partial program has been represented for each schema. An extension of this concept would be to explicitly allow these partial programs to exist in the population and to compete as building blocks. These partial programs would be represented as partial derivation trees. Figure 7.1 shows some partial derivation trees in a population generated using the grammar, G_{ggd}.

A number of issues arise when considering the representation of partially-executable programs (derivation trees).

- How can a fitness for a partial program be determined when it cannot be evaluated on some members of the dataset?
- What are the problems with a potential lack of diversity due to many partial derivation trees having the same (poor) fitness?
- What are the most appropriate genetic operators to apply with these representations?
- How do we ensure that a complete program is created as the final result?

Evaluating a partial program may be approached in several ways. The use of a lazy evaluation method[77] could be used to ensure that a program that could possibly terminate will do so. A second method would be to randomly generate a derivation tree, from the grammar, below each nonterminal which is a tip of the partial derivation tree. Although this random generation

would create a great deal of noise it would ensure that every partial program could be evaluated to give a fitness measure. Various methods to evaluate partial derivation trees would have to be studied to understand the implications of this representation.

A second issue arises with partial programs, namely how to ensure that a complete program is eventually created. A form of annealing could be used to gradually put pressure on the evolving population to represent complete programs. This could be achieved by using a function in the fitness measure that penalised programs that could not be evaluated. This function would become more important as the number of generations increased, thereby directing the selection pressure towards programs that could be evaluated for more of the training cases. The difficulties introduced by evaluating partial programs suggest that other approaches to composing complete programs, using partial derivation trees, may be worth considering.

One such framework would be to use a population of partial programs to construct complete programs for evaluation. Selecting a partial derivation tree with S as the root will represent the first component of the program. For each nonterminal at the tip of this tree, a partial derivation tree which has a matching root nonterminal could be used to extend the program. This process of adding partial trees continues until all nonterminals have been completely derived to terminal strings. The derivation tree, rooted in S, now represents a complete program that may be evaluated. The fitness of this resultant program is used to give credit to each of the partial derivation trees that were used in the program. This credit could be done in a manner similar to a bucket-brigade algorithm[17]. This would allow the selection of partial programs to be based on a proportional fitness measure when selecting which components are used to build a program for evaluation. The work of fitness sharing and niche formation[13] may also be applicable to this work, since these methods consider how to manipulate and combine partial solutions.

An interesting question arises with this approach. If the diversity of the population guarantees that a suitable program can be constructed from the partial programs, do we require any form of crossover or mutation? The evolutionary pressure would come about from the competition between partial programs based on their current fitness from credit assignment.

7.3.7 Languages and Fitness Landscapes

The study of fitness landscapes[31] has been useful in understanding the relationship between a problem search space and the operators that navigate through this space. In the context of a landscape it may be possible to relate a particular grammar (and therefore language) to a form of fitness landscape. The genetic operators and the relationship to the sets $\otimes$ and $\odot$ may lead to a better understanding of how the construction of a grammar and the search bias is suitable for certain problem domains. The theoretical directions of most interest with these ideas would be to explore how a landscape changed when different styles of language were introduced.

7.3.8 Extending the Schema Theorem to Context-Sensitive Languages

The definition of a schema, $A \stackrel{*}{\Rightarrow} \theta$, can be extended to a context-sensitive language by defining a schema as $\alpha \stackrel{*}{\Rightarrow} \theta$, where $\alpha \in \{\sum \cup N\}^+$. A review of Chapter 6 shows that there is no explicit assumptions about the form of the left-hand side of a schema. The schema theorem is essentially a counting exercise. The manner in which the schema structures are created is not relevant to the subsequent analysis. Therefore, the extension of the theorem to context-sensitive languages is only concerned with the form of crossover and mutation sites that have been extended by this language; this appears to be relatively straightforward.

7.3.9 Probabilistic Grammars

Merit selection (see Section 3.4.3) has been used to represent the prior probabilities of any particular production being selected from a nonterminal. The limitation of this work is that it assumes that the probabilities are constant, independent of the context in which the nonterminal is rewritten. Previous work with probabilistic L-systems[35] suggest that a set of homogeneous Markov chains (i.e. autonomous probabilistic semi-automata) may be used to represent the changing structure of a grammar as a sentence is constructed. The advantage of this approach is that contextual information could be embedded in the grammar and potentially learnt during the evolution of a solution.

7.4 Conclusion

This thesis has presented research into bias and evolutionary techniques for learning. The field of genetic programming has been used as the framework to develop a program induction system, CFG-GP, which incorporates explicit language and search bias. This work demonstrated that it is possible to learn how to modify the search bias, represented in the definition of the initial grammar, during the evolution of a solution. Additionally, the definition of a schema theorem, based on CFG-GP, has been shown to unite both fixed-length and variable-length representations.

Appendix A

Testing the Assumption of Normality

To determine whether it is valid to assume that the results of a series of 6-multiplexer runs are *normally distributed*, the following experiment was performed. Here, the setup for the initial grammar G_{6m} was appied 25 times to obtain a spread of p_s. A cumulative distribution of the probability of success for these runs is shown in Figure A.1. The data displayed as a frequency distribution is shown in Figure A.2.

Using a modified form of the Kolmogorov D_n test, first developed by Lilliefors, an estimate of the probability that p_s is from a normal distribution was calculated as follows. The value of D_{max} was obtained by computing the Standard Distribution Function,$F_n(x)$, for the *standardised values* $z_i = (x_i - \hat{\mu})/\hat{\sigma}$ and taking F_* as the standard normal cumulative distribution function. Table A.1 shows the results for the 25 independent calculations of p_s. The average $\hat{\mu} = 34.76$ and standard deviation $\hat{\sigma} = 4.87$.

After looking up all $F_*(z_i)$ in the cumulative normal distribution table ,D_{max} was found to be 0.093. The hypothesis, H_0, is that the distribution of p_s is from the normal distribution. Setting the Type I error to 5% gives, for a sample size of 25, a critical value of 0.173. Since the observed value = $0.093 < 0.173$ the hypothesis may be accepted.

Significance testing based on this assumption is described in Appendix B.

Standardised Values for p_s over 25 runs				
-1.7998	-1.3889	-1.3889	-1.1834	-1.1834
-1.1834	-0.7725	-0.3616	-0.3616	-0.1561
-0.1561	0.0493	0.0493	0.2548	0.2548
0.2548	0.4602	0.4602	0.6657	0.6657
0.6657	1.0766	1.4875	1.6930	1.8984

Table A.1: The Standardised Values of z_i using G_{6m}.

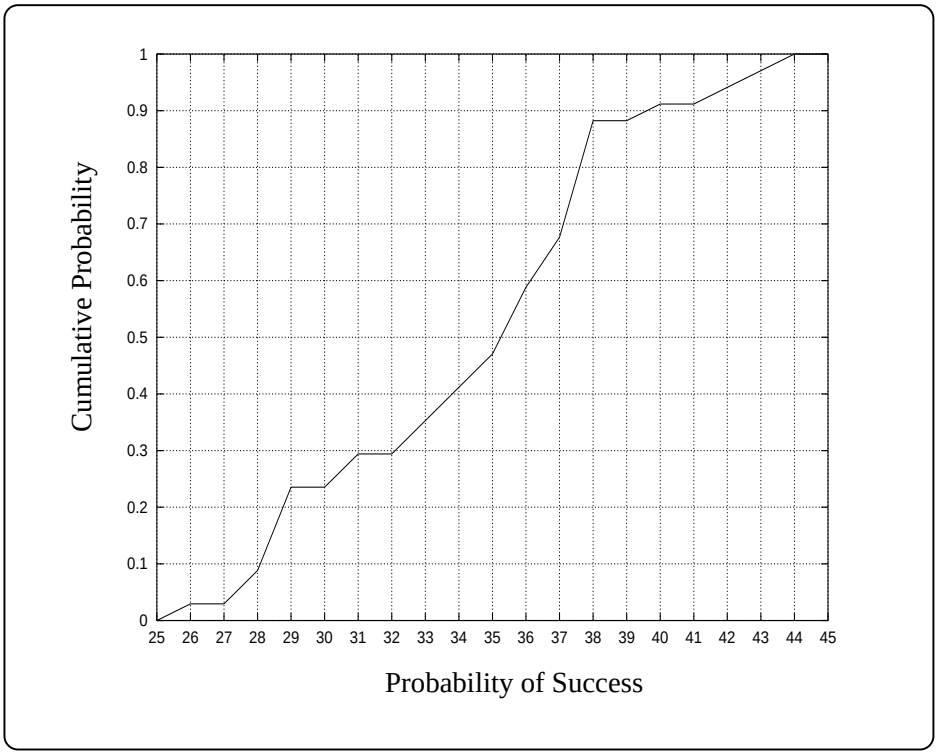


Figure A.1: Cumulative Distribution for the grammar G_{6m}.

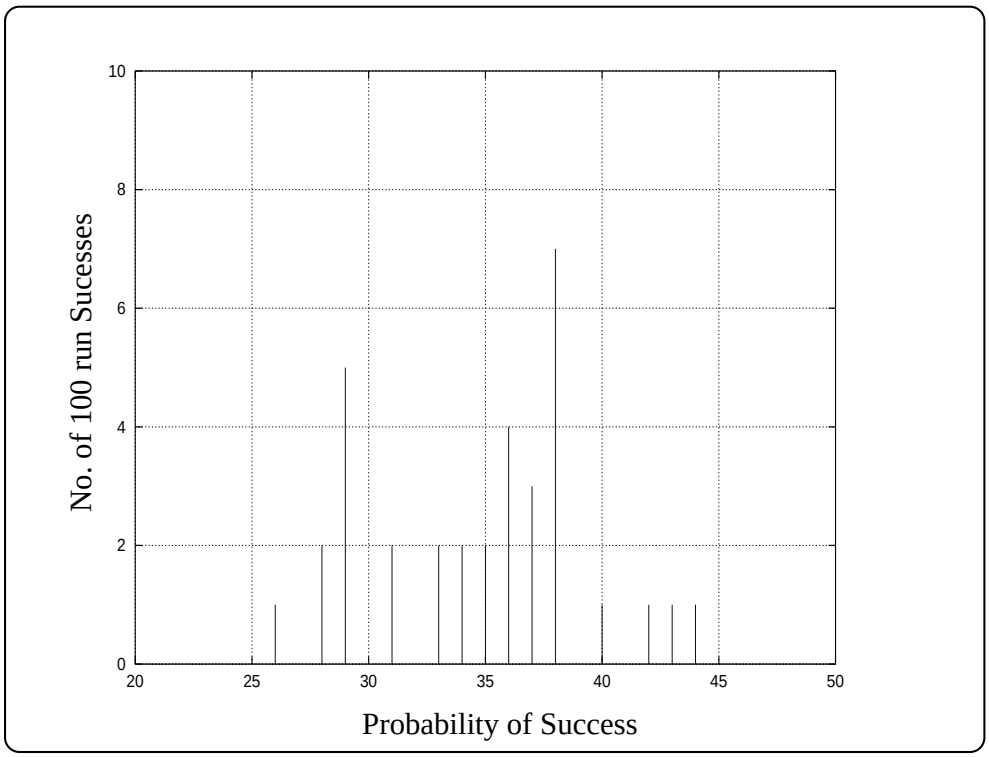


Figure A.2: Frequency Distribution for the grammar G_{6m}.

Appendix B

Significance Testing

To give a measure of the significance of the results performed with the various setups of the CFG-GP we have used a confidence interval measure. This assumes that each run is independent, and that the overall performance may be described as a normal distribution. These assumptions have been shown to be valid from Appendix A.

The confidence interval for two proportions, $p1$ and $p2$, is given by [80]:

CONFIDENCE INTERVAL FOR $p1 - p2$; $n1$ *AND* $n2 \geq 30$; $A(1-\alpha)100\%$ *confidence interval for the difference of two binomial parameters, p1 - p2 is approximately:*

$$(\hat{p1} - \hat{p2}) - z_{\alpha/2} \sqrt{\frac{\hat{p1}\hat{q1}}{n1} + \frac{\hat{p2}\hat{q2}}{n2}} < p1 - p2 < (\hat{p1} - \hat{p2}) + z_{\alpha/2} \sqrt{\frac{\hat{p1}\hat{q1}}{n1} + \frac{\hat{p2}\hat{q2}}{n2}}$$

where $\hat{p1}$ *and* $\hat{p2}$ *are the proportion of successes in random samples of size n1 and n2, respectively,* $\hat{q1} = 1 - \hat{p1}$ *and* $\hat{q2} = 1 - \hat{p2}$ *, and* $z_{\alpha/2}$ *is the value of the standard normal curve leaving an area of* $\alpha/2$ *to the right.*

Let $p1$ and $p2$ be the true *probabilities of success*, we can find a 95% one-sided confidence interval for the true interval between any two probabilities using the above formulae, with $z_{0.05} = 1.645$. For each of the 6-multiplexer experiments, $n1$ and $n2$ equal to 100. The values for $\hat{p1}$ and $\hat{p2}$ represent the *probabilities of success*.

Figure B.1 shows three significance plots using the one-sided confidence interval. The graphs representing the interval with $p2 = 0.34$ crosses the $0 - axis$ when the probability is approximately 0.45. Hence, values of p_s, compared with a base probability of 34 (grammar G_{6m}) must be above this value before the results are statistically significant. This implies that the grammar G_{6m-if} does not improve the performance of CFG-GP for the 6-multiplexer. However, all other grammar extensions are significant (see Table 4.3).

Modifying the search bias (Section 4.1.7) resulted in a *probability of success* of 41% and therefore is not a significant result.

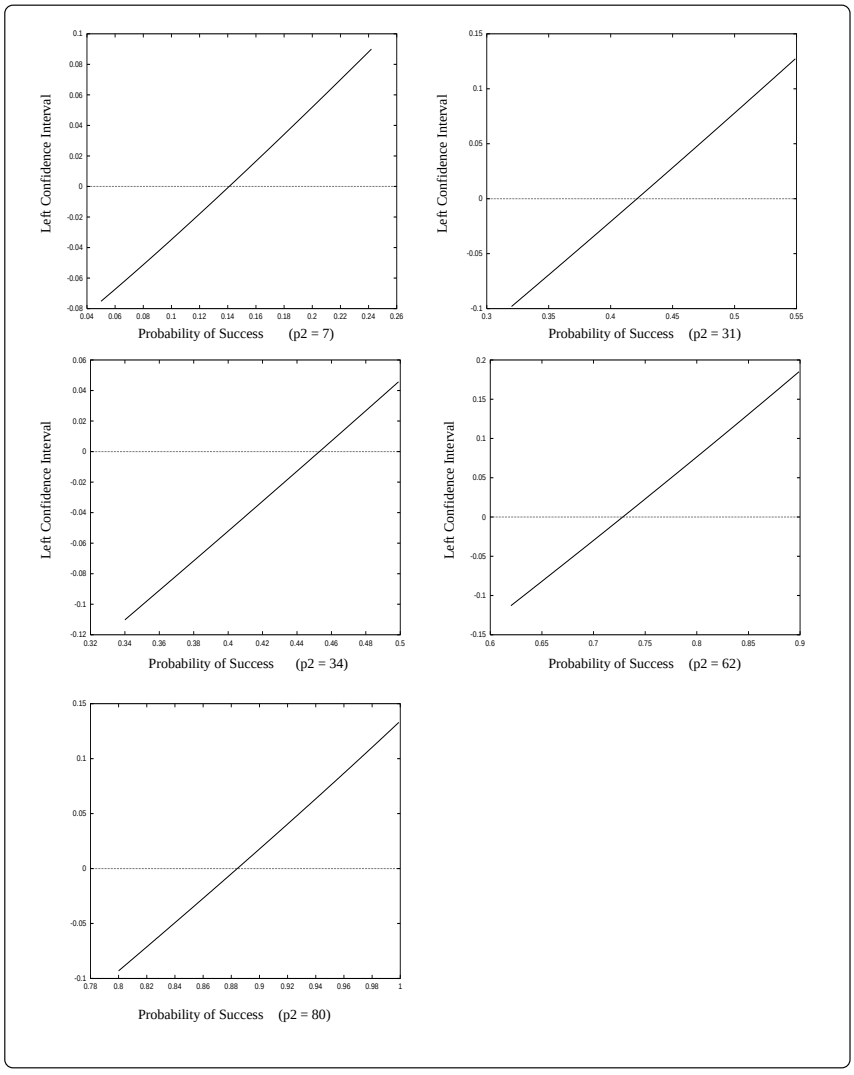


Figure B.1: Confidence Intervals for the 6-Multiplexer and Directed Mutation.

The graph representing the interval $p2 = 0.62$ shows that values above approximately 0.74 are significant. This implies that there are significant improvements from the grammar $G_{6m-if-address}$ to the grammars $G_{6m-if-address-then}$ and $G_{6m-if-a0-if-a1}$.

The graph representing the interval $p2 = 0.80$ shows that values above approximately 0.88 are significant. Hence, there is some evidence that modifying the grammar $_{6m-if-address-then}$ to $G_{6m-if-a0-if-a1}$ is significant.

The graph representing the interval $p2 = 0.31$ shows that values above approximately 0.42 are significant. Hence, the use of $REPLACEMENT$ with the 6-multiplexer, described in Section 5.4.1, improves the algorithm by a significant degree.

Appendix C

The Schema Theorem for Genetic Algorithms

C.1 Introduction

This appendix gives a brief introduction to the schema theorem for Genetic Algorithms, first proposed by John Holland[28]. Initially, the concept of a schema will be introduced for fixed-length binary strings. The affect of schemata on the genetic operators of reproduction, crossover and mutation will then be presented. This leads to an expression, referred to as the Schema Theorem for Genetic Algorithms, which describes how schemata are propagated from one generation to the next. There are many possible forms and complexities of this theorem. For the purpose of this thesis a simple version is presented, which demonstrates the basic principles behind this form of anaylsis.

C.2 The Concept of Similarity

Goldberg[17] defines a schema for fixed-length structures as follows.

> A schema, H, is a similarity template describing a subset of strings with similarities at certain string positions.

For a fixed-length binary representation the alphabet for this language is $\{0, 1\}$. This alphabet is extended to $\{0, 1, \star\}$, where $\star$ matches either 0 or 1. The use of $\star$ as a *don't care* symbol allows schemata to be explicitly represented and forms the basis for describing how the space, defined by these templates, is searched while a solution is evolved.

There are two properties used to characterise the properties of a schema. These are defined as follows.

- *The Defining Length* $\delta(H)$ is the number of bits between the first and last bits within H. Hence, this represents the length or extent of H.
- *The Schema Order* $o(H)$ is the number of fixed positions (i.e. the number of non-$\star$ positions) in H.

For example, the schema $0 \star 10 \star 1$ has a defining length of 6 and a schema order of 4.

C.3 Schemata and Reproduction

The first step in developing the schema theorem involves the consideration of the effect that selection has on H from one generation to the next. Assuming a *roulette wheel* selection strategy[1], the propagation of H will be proportional to the average fitness of the population samples containing H, in relation to the average fitness of the entire population. If the number of samples of schema H at generation t is given by $m(H,t)$, then the number of schema given in the next generation may be stated as follows.

$$m(H,t+1) = m(H,t)\frac{f(H)}{\bar{f}}$$

Here, $f(H)$ represents the average fitness of the samples in the population containing schema H, $\bar{f}$ is the average fitness of the entire population.

This equality states that schemata in the population, with above-average fitness, will receive exponentially increasing representations from one generation to the next.

C.4 Schemata and Single-Point Crossover

The operation of single-point crossover is normally applied probabilistically to the population of binary strings, thereby creating new population members in the next generation. The schema theorem describes how crossover disrupts H for the next generation. Normally, there are two assumptions that are made to simplify the definition of $m(H,t+1)$ when dealing with crossover.

(a) There is some possibility that the action of crossover will create a new population member that contains H, which did not previously exist in either of the parent strings. This gain is ignored.

(b) The conservative estimate is made that whenever crossover cuts a schema, the schema is disrupted. Hence, the possibility of reintroducing H, due to crossover, is ignored.

[1] The roulette wheel selection strategy has been described in Section 3.6

Given the previous assumptions, for a probability of crossover p_c, the following inequality describes the porpagation of H to the next generation.

$$m(H,t+1) \geq (1-p_c)m(H,t)\frac{f(H)}{\bar{f}} + p_c m(H,t)\frac{f(H)}{\bar{f}}(1-disruptions)$$

This equation may be expressed in a simplier form by rearranging terms, as follows.

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}} \times (1-(p_c \times disruptions))$$

The *disruption* of H due to crossover is easily calculated by considering the *defining length* of H in relation to the entire length of the binary string. For a binary string of length l, the probability of H being disrupted will be $\frac{\delta(H)}{l-1}$. Let n be the number of strings in the population. The propagation of H to the next generation, due to single-point crossover, where both parents are selected based on their proportional fitness[2], may be written as follows.

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}} \times \left\{ \; 1 - p_c\frac{\delta(H)}{l-1}(1-\frac{1}{n}m(H,t)\frac{f(H)}{\bar{f}}) \; \right\}$$

C.5 Schemata and Mutation

Mutation is normally defined as a low-probability operator that randomly flips the bit value for any position in a population member. This bit-flipping mutation is applied as a probability referring to the chance that *any particular bit* in a string will be changed. Let p_m be the probability of mutation. The probability that some schema H will survive disruption due to mutation is then given by $(1-p_m)^{o(H)}$.

C.6 A Statement of the Schema Theorem for Genetic Algorithms

Theorem 3 *The schema theorem for the genetic algorithm with single-point crossover, bit-flipping mutation and roulette wheel selection may be stated as follows[89].*

$$m(H,t+1) \geq m(H,t)\frac{f(H)}{\bar{f}} \times \left\{ \; 1 - p_c\frac{\delta(H)}{l-1}(1-\frac{1}{n}m(H,t)\frac{f(H)}{\bar{f}}) \; \right\}(1-p_m)^{o(H)}$$

[2]The original work of Holland[28] selected the first parent based on fitness and the second parent randomly.

Appendix D

What is a Greater Glider?

The Greater Glider, an Australian marsupial, is related to the possum family. The Glider family is tree-dwelling, ranging in size from something akin to a mouse, to that of a small long-tailed cat.

All marsupial gliders possess the ability to glide, often distances up to 50 metres, using a volplaning membrane or skinfold which extends from the fifth "finger" or wrist to the ankle. This skin is continuous with the body, thereby creating two wings which enable the glider to sail through the air in a graceful and controlled manner. The tails, often bushy, give some directional control. They have been described as "long-handled frying pans", during flight, which gives some idea of the wing extent and body shape of these animals.

The main habitat of these marsupials are the eucalyptus trees of South Eastern Australia. They are essentially nocturnal, spending the daytime curled up in lofty hollow limbs in forested areas. During the evening they travel, by gliding, up to several kilometers in search of food and water. The main diet is essentially the new tips of leaves from selected eucalypt trees. Hence, although they are easily caught, these marsupials are often difficult to breed and maintain in captivity.

Warm blooded, pouched and furry, the Greater Glider is a shy and quiet animal. Their preference for high quality vegetation makes them ideal surrogates for indicators of the condition of native forests. If a Greater Glider population exists in some location, then the quality of forest in this location is likely to be high.

Bibliography

[1] P. Angeline. Two self-adaptive crossover operators for genetic programming. In P. Angeline and K.E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 3. Cambridge, MA, USA:MIT Press, 1996.

[2] P.J. Angeline. Genetic programming and emergent intelligence. In K.E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 6. MIT Press, 1994.

[3] J.E. Baker. Adaptive selection methods for genetic algorithms. In *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, pages 100–111, 1985.

[4] D.L. Battle and M.D. Vose. Isomorphisms of genetic algorithms. *Artificial Intelligence*, 60:155–165, 1993.

[5] K. Bennet, M.C. Ferris, and Y.E. Ioannidis. A genetic algorithm for database query optimization. In R.K. Belew and L.B.Booker, editors, *Proceedings of the Fourth International conference on Genetic Algorithms*, pages 400–407. Morgan Kaufmann Publishers, Inc. San Mateo, California, July 1991.

[6] M.F. Bramlette. Initialization, mutation and selection methods in genetic algorithms for function optimization. In R.K. Belew and L.B.Booker, editors, *Proceedings of the Fourth International conference on Genetic Algorithms*, pages 100–107. Morgan Kaufmann Publishers, Inc. San Mateo, California, July 1991.

[7] A. Chipperfield and P.J. Flemming. Gas turbine engine controller design using multiobjective genetic algorithms. In *Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA '95)*, pages 214–219. Institution of Electrical Engineers, London, 1995.

[8] N. Chomsky. Three models for the description of language. *IEEE Transactions on Information Theory*, 2(3):113–124, 1956.

[9] G.A. Cleveland and S.F. Smith. Using genetic algorithms to schedule flow shop releases. In J.D.Schaffer, editor, *Proceedings of the Third International conference on Genetic Algorithms*, pages 160–169. Morgan Kaufmann Publishers, Inc. San Mateo, California, June 1989.

[10] 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.

[11] N.L. Cramer. A representation for the adaptive generation of simple sequential programs. In J.J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algorithms and their Applications*, pages 183–187. J.J. Grefenstette, July 1985.

[12] C. Darwin. *The Origin of Species by means of Natural Selection.* Penguin Books (Reprinted 1975), 1859.

[13] K. Deb and D.E. Goldberg. An investigation of niche and species formation in genetic function optimization. In J.D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50. Morgan Kaufmann Publishers,Inc. San Mateo, California, June 1989.

[14] W.J. Ewens. *Mathematical Population Genetics.* Springer-Verlag Berlin Heidelberg, 1979.

[15] L.J. Fogel, A.J. Owens, and M.J. Walsh. *Aritificial Intelligence Through Simulated Evolution.* John Wiley Sons, Inc., 1966.

[16] R.M. Friedberg. A learning machine: Part i. *IBM Journal*, 2:2–13, January 1958.

[17] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley, 1989.

[18] D.E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. In G. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. Morgan Kaufmann, 1991.

[19] D.E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms:motivation, analysis and first results. *Complex Systems*, 3(5):493–530, 1989.

[20] J.J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Trans. on Systems, Man and Cybernetics*, 16(1):122–128, 1985.

[21] J.J. Grefenstette. Incorporating problem specific knowledge into genetic algorithms. In L. Davis, editor, *Research Notes in Artificial Intelligence:Genetic Algorithms and Simulated Annealing*, chapter 4. Morgan Kaufmann Publishers,Inc., Los Altos, California, 1987.

[22] J.J. Grefenstette. A system for learning control strategies with genetic algorithms. In J.D.Schaffer, editor, *Proceedings of the Third International conference on Genetic Algorithms*, pages 183–190. Morgan Kaufmann Publishers, Inc. San Mateo, California, June 1989.

[23] F. Gruau. On using syntactic constraints with genetic programming. In P. Angeline and K.E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 19, pages 402–417. USA:MIT Press, 1996.

[24] F. Gruau and D. Whitley. Adding learning to the cellular developmental of neural networks: Evolution and the baldwin effect. *Journal of Evolutionary Computation*, 1(3):213–233, 1993.

[25] A.T. Hatjimihail and T.T. Hatjimihail. Design of statistical quality control procedures using genetic algorithms. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorithms*, pages 551–557. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[26] T. Haynes, R. Wainwright, S. Sen, and D. Schoenefeld. Strongly typed genetic programming in evolving cooperation strategies. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorithms*, pages 271–278. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[27] H. Hemmi, J. Mizoguchi, and K. Shimohara. Development and evolution of hardware behaviours. In R. Brooks and P. Maes, editors, *Proceedings of Artificial Life IV*, pages 371–376. MIT Press, 1994.

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

[29] C.Z. Janikow. A knowledge-intensive genetic algorithm for supervised learning. *Machine Learning*, 13(2/3):33–72, Nov/Dec 1993.

[30] C.Z. Janikow. A genetic algorithm for optimizing fuzzy decision trees. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorithms*, pages 421–428. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[31] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, Computer Science Dept., University of New Mexico, Albuquerque, New Mexico, May 1995.

[32] K.A. De Jong. On using genetic algorithms to search program spaces. In J.G. Grefenstette, editor, *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms.* Erlbaum, 1987.

[33] K.A. De Jong and W.M. Spears. Learning concept classification rules using genetic algorithms. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, volume 2, pages 651–657. Morgan Kaufmann Publishers,Inc., August 1991.

[34] K.A. De Jong, W.M. Spears, and D.F. Gordon. Using genetic algorithms for concept learning. *Machine Learning*, 13(2/3):5–32, Nov/Dec 1993.

[35] H. Jurgensen. Probabilistic l systems. In A. Lindenmayer and G. Rozenberg, editors, *Automata, Languages, Development*, pages 211–225. North-Holland Publishing Company, 1976.

[36] R.P. Kavanagh. Seasonal changes in habitat use by gliders and possums in south-eastern new south wales. *Possums and Gliders*, pages 527–543, 1984.

[37] J.D. Kelly, Jr. and L. Davis. A hybrid genetic algorithm for classification. In *Proceedings of the Twelfth International Conference on Artificial Intelligence*, volume 2, pages 645–650. Morgan Kaufmann Publishers,Inc., August 1991.

[38] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[39] H. Kitano. Designing neural networks using genetic algorithms with graph generation system. *Complex Systems*, 4:461–476, 1990.

[40] S. Kobayashi, I.Ono, and M.Yamamura. An efficient genetic algorithm for job shop scheduling problems. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorithms*, pages 506–511. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[41] Y. Kodratoff and R. Michalski, editors. *Machine Learning: An Artificial Intelligence Approach*. Morgan Kaufmann Pub. San Mateo, California, 1983.

[42] J.R. Koza. Concept formation and decision tree induction using the genetic programming paradigm. In H.P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, pages 124–129. Springer-Verlag, 1990.

[43] J.R. Koza. The genetic programming paradigm:genetically breeding populations of computer programs to solve problems. In *Dynamic, Genetic and Chaotic Programming*, chapter 10, pages 203–323. John Wiley and Sons, Inc., 1992.

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

[45] J.R. Koza. Architecture-altering operations for evolving the architecture of a multipart program in genetic programming. Technical Report STAN-CS-94-1528, Dept. of Computer Science, Stanford University, Stanford, California, U.S.A., 1994.

[46] J.R. Koza. Two ways of discovering the size and shape of a computer program to solve a problem. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorith ms*, pages 287–294. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[47] E. Kreyszig. *Advanced Engineering Mathematics*. John Wiley and Sons Inc., 4 edition, 1979.

[48] Douglas Lenat. The role of heuristics in learning by discovery: Three case studies. In R.S. Michalski, J.G. Carbonell, and T.M Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 9, pages 243–306. Springer-Verlag, 1984.

[49] A. Lindenmayer. Development systems without cellular interactions, their languages and grammars. *Theoretical Biology*, 30:455–484, 1971.

[50] Ryszard S. Michalski. A theory and methodology of inductive learning. In R.S. Michalski, J.G. Carbonell, and T.M Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, chapter 4, pages 83–134. Springer-Verlag, 1984.

[51] G.F. Miller, P.M.Todd, and S.U.Hegde. Designing neural networks using genetic algorithms. In J.D.Schaffer, editor, *Proceedings of the Third International conference on Genetic Algorithms*, pages 379–384. Morgan Kaufmann Publishers, Inc. San Mateo, California, June 1989.

[52] T.M. Mitchell, P.E. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristics. In *Machine Learning: An Artificial Intelligence Approach*, chapter 6, pages 163–190. Springer-Verlag, 1984.

[53] J. Mizoguchi, H. Hemmi, and K. Shimohara. Production genetic algorithms for automated hardware design through an evolutionary process. In *Proceedings of the First IEEE Conference on Evolutionary Computation.* Piscatawa, NJ, USA:IEEE Press, June 1994.

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

[55] P. Nordin, F. Francone, and W. Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In J. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, 1995.

[56] Una-May O'Reilly. *An Analysis of Genetic Programming.* PhD thesis, Ottawa-Carleton Institute for Computer Science, Carleton University, Ottawa, Ontario, Sept. 1995.

[57] Una-May O'Reilly and Franz Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In *Foundations of Genetic Algorithms 3.* Morgan Kaufmann Pub., 1994.

[58] M. Pazzani and D. Kibler. The utility of knowledge in inductive learning. *Machine Learning*, 9(1), June 1992.

[59] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, August 1990.

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

[61] J.P. Rosca and D.H. Ballard. Genetic programming with adaptive representations. Technical Report 489, Computer Science Dept., The University of Rochester, N.Y., February 1994.

[62] J.P. Rosca and D.H. Ballard. Causality in genetic programming. In L.J. Eshelman, editor, *Proceedings of the Sixth International conference on Genetic Algorith ms*, pages 256–263. Morgan Kaufmann Publishers, Inc. San Francisco, California, July 1995.

[63] G. Roston and R. Sturges. A genetic design methodology for stucture configuration. *ASME Advances in Design Automation*, DE 82:73–90, 1995.

[64] G.P. Roston. *A Genetic Methodology for Configuration Design*. PhD thesis, Carnegie Mellon University, December 1994.

[65] G. Rudolph. Convergence analysis of canonical genetic algorithms. *IEEE Transactions on Neural Networks*, 5(1):96–101, January 1994.

[66] C. Schaffer. Overfitting avoidance as bias. *Machine Learning*, 10:153–178, 1993.

[67] H. Schwefel. *Numerical Optimization of Computer Models*. J. Wiley, Chichester; New York, 1981.

[68] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

[69] W. Siedlecki and J. Sklansky. Constrained genetic optimization via dynamic reward-penalty balancing and its use in pattern recognition. In J.D.Schaffer, editor, *Proceedings of the Third International conference on Genetic Algorithms*, pages 141–150. Morgan Kaufmann Publishers, Inc. San Mateo, California, June 1989.

[70] P.A. Stefanski. Genetic programming using abstract syntax trees. Notes from the Genetic Programming Workshop, ICGA'93, 1993.

[71] D. Stockwell, S.M. Davey, J.R. Davis, and I.R. Noble. Using induction of decision trees to predict greater glider density. *AI Applications in Natural Resource Management*, 4(4):33–44, 1990.

[72] G. Syswerda and J. Palmucci. The application of genetic algorithms to resource scheduling. In R.K. Belew and L.B.Booker, editors, *Proceedings of the Fourth International conference on Genetic Algorithms*, pages 502–508. Morgan Kaufmann Publishers, Inc. San Mateo, California, July 1991.

[73] Paul Utgoff. *Machine Learning of Inductive Bias*. Kluwer Academic Publishers, 1986.

[74] L.G. Valiant. A theory of the learnable. *ACM*, 27(11):1134–1142, 1984.

[75] M.D. Vose. Generalizing the notion of schema in genetic algorithms. *Artificial Intelligence*, 50:385–396, 1991.

[76] M.D. Vose. Punctuated equilibria in genetic search. *Complex Systems*, 5:31–44, 1991.

[77] J. Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *Computer and System Sciences*, 9(3):332–354, August 1974.

[78] B.W. Wah. Population-based learning: A method for learning from examples under resource constraints. *IEEE Transactions on Knowledge and Data Engineering*, 4(5):454–474, October 1992.

[79] B.W. Wah, A. Ieumwananonthachai, L. Chu, and A.N. Aizawa. Genetics-based learning of new heuristics: Rational scheduling of experiments and generalisation. *IEEE Transactions on Knowledge and Data Engineering*, 7(5), October 1995.

[80] R.E. Walpole. *Introduction to Statistics*. The Macmillan Co. New York, Collier-Macmillan Ltd., 1978.

[81] P.R. Weller and R. Summers. Using a genetic algorithm to evolve an optimum input set for a predictive neural network. In *Genetic Algorithms in Engineering Systems: Innovations and Applications (GALESIA '95)*, pages 256–258. Institution of Electrical Engineers, London, September 1995.

[82] P.A. Whigham. Context-free grammars and genetic programming. Technical Report CS20/94, Dept. of Comp. Sci., University College, University of New South Wales, November 1994.

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

[84] P.A. Whigham. Genetic programming and spatial information. In C. Zhang, J. Debenham, and D. Lukose, editors, *Proceedings of the 7th Australian Joint Conference on Artificial Intelligence: AI'94*, pages 124–131. World Scientific, November 1994.

[85] P.A. Whigham. Grammatical genetic learning and schemata:restated. Technical Report CS13/95, Dept. of Comp. Sci., University College, University of New South Wales, 1995.

[86] P.A. Whigham. Grammatically-based genetic programming. In J.Rosca, editor, *Proceedings of the Workshop on Genetic Programming:From Theory to Real-World Applications*, pages 33–41. Morgan Kaufmann Pub., July 1995.

[87] P.A. Whigham. Inductive bias and genetic programming. In *First International Conference on Genetic Algorithms in Engineering Systems:Innovations and Applications*, pages 461–466. UK:IEE, September 1995.

[88] P.A. Whigham. A schema theorem for context-free grammars. In *Proceedings of the 1995 IEEE International Conference on Evolutionary Computation*, volume 1, pages 178–182. Piscatawa, NJ, USA:IEEE Press, December 1995.

[89] D. Whitley. A genetic algorithm tutorial. Technical Report CS-93-103, Computer Science Dept., Colorado State University, 1993.

[90] D. Whitley, S. Dominic, and R. Das. Genetic reinforcement learning with multilayer neural networks. In R.K. Belew and L.B.Booker, editors, *Proceedings of the Fourth International conference on Genetic Algorithms*, pages 562–570. Morgan Kaufmann Publishers, Inc. San Mateo, California, July 1991.

[91] P.H. Winston. *Artificial Intelligence*. Addison-Wesley, Reading, 2 edition, 1983.

[92] D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe, NM, 87501, February 1996.

[93] M.L. Wong and K.S. Leung. Applying logic grammars to induce sub-functions in genetic programming. In *Proceedings of the 1995 IEEE Conference on Evolutionary Computation*, pages 737–740. USA:IEEE Press, December 1995.

[94] M.L. Wong and K.S. Leung. Combining genetic programming and inductive logic programming using logic grammars. In *Proceedings of the 1995 IEEE Conference on Evolutionary Computation*, pages 733–736. USA:IEEE Press, December 1995.

[95] E. Zannoni and R.G. Reynolds. Extracting design knowledge from genetic programs using cultural algorithms. In *Preprint Proceedings of the Fifth Annual Conference on Evolutionary Programming*, 1996.

Index