An Analysis of Genetic Programming Sub-Tree Swapping Crossover with Applications

Stephen Dignum

A thesis submitted for the degree of Doctor of Philosophy

Department of Computing and Electronic Systems

University of Essex

September 2008

Contents

List of Figures

List of Tables

Acknowledgements

Professor Riccardo Poli for his expert supervision, notably his help in interpreting the many results presented to him. Often what looked like disaster became triumph with his additional inquiring eye. The suggestion of a PhD topic that I have found to be both interesting and rewarding. His permission to use TinyGP long before it became publicly available, a tool that has saved many hours in the preparation of results presented here.

My fiancée Hannah Jarvis, for her support, extensive proof reading and providing a life outside of academia.

Edgar Galván-López, for discussions on all things GP, particularly many insightful thoughts regarding neutrality, not to mention his excellent code debugging skills.

The Engineering and Physical Sciences Research Council (EPSRC) for providing the funding that made this PhD possible.

The University of Essex, particularly the Department of Computing and Electronic Systems (CES), for their ongoing support which includes the provision of highly relevant training and progress monitoring that I

have always found to be very useful. The department has also provided a substantial travel bursary that has enabled my attendance at, and contribution to, many highly regarded conferences.

Finally, my parents who have provided me with much support and encouragement and made valiant attempts to understand my work.

Abstract

Genetic Programming (GP) is one of a number of biologically inspired search techniques known collectively as Evolutionary Algorithms (EAs). These algorithms use the metaphor of Darwinian Evolution to discover solutions to problems that humans, and/or other search methods have found difficult to solve. GP differs from the other main classes of EAs in that it specifically seeks to produce solutions that are executable computer programs.

Considering the large amount of books, papers and articles on GP, over 5,000 items in the official GP Bibliography, relatively few have addressed the problem of understanding the very basic biases of GP operators, i.e., how they sample program spaces.

This thesis begins to address this lack of knowledge by considering GP's defining variation operator, sub-tree swapping crossover. It first analyses crossover's bias with regard to program sampling in terms of program length, providing a number of empirically verified theoretical models.

With this knowledge in hand, the thesis investigates how length bias affects GP runs, particularly with regard to the sampling of unique programs and bloat. From this work a new bloat theory is presented, *Crossover-Bias*, and a method, *Sampling Parsimony*, to directly alter the rate of resampling and hence control bloat is created.

To counteract the length bias of crossover a new technique is introduced, *Operator Equalisation*, which enables length classes to be sampled according to pre-determined probability distributions. This provides essential information regarding GP runs and can be shown to improve GP performance.

We then turn our attentions to the sampling of programs within length classes and its implications for structural convergence within GP. From this work we show that sub-tree swapping crossover will sample programs with a frequency determined by arity proportions, our length work being a specialisation of this process. A new theoretical model based on *arity histograms* is then provided.

Chapter 1

Introduction

The programming of computers has long been seen to be the preserve of humans. To determine a set of instructions in order to accomplish a task is primarially a design activity based upon expert knowledge. Genetic Programming [Koza, 1992] provides a method to automate this process so that machines can discover, with minimal human intervention, programs that will satisfy a given set of requirements.

This thesis uses the idea of finding such a group of instructions as a *search* process as defined in the discipline of Artificial Intelligence (AI) [Russell and Norvig, 2003]. We know all the available program components and how they can be structured. However, we do not know the correct combination required to reach our solution. We have to, therefore, search through the space of possible combinations of program components in order to find such a solution. For all but the simplest of programs we do not have sufficient computing power available to search through all the possible combinations, a problem known as *combinatorial explosion*. An *informed* search is, therefore, required, one that will concentrate the search in promising areas thereby making best use of computational resources.

Darwinian Evolution [Darwin, 1859] is the inspiration for the search method employed by GP. At its most basic, the essential components of this process used by GP can be described as follows: we have a population of individuals that can reproduce, i.e., make copies of themselves. During reproduction random copying errors, *mutations*, will be made affecting certain individual *traits*. Mutations will in turn be passed on to the children of those children. If there is competition for resources individuals more suited to the environment will live longer and produce more children. Their traits will, therefore, become more frequently observed in the population, a process known as *natural selection* [Ridley, 1993].

Although not strictly required for evolution to take place, *sexual* reproduction is a key element in the evolution of higher animals and a fundemantal ingredient in GP. Sexual reproduction allows the *genes* that determine traits to be taken from either parent and recombined (or crossed over). The benefits of sexual reproduction have been debated vigorously. For example, recombination can be seen as method to speed evolution as a kind of macro-mutation, many benficial traits being combined in one operation. An alternative theory is that it allows destructive or negative traits to be removed quickly rather than waiting for a copying error to take place. The sexual reproduction operator in GP is known as *crossover* and is the primary variation method used in typical GP implementation [Poli et al., 2008a].

It is important to note that one could use one of a number other AI search methods. In fact, one could argue that documented successes in GP should be at least empirically compared to other such methods. A more satisfying approach would be to determine which method is most likely to be sucessful with a certain problem or set of problems based upon a theoretical framework of defining characteristics. Thereby, avoiding painstaking

trial and error. This is particularly relevent when one considers the No Free Lunch theorem (NFL) [Wolpert and Macready, 1997]. This can be described, at its simplest, as follows: there is no algorithm that will solve, on average, *all* classes of problems better than any other algorithm. It is important, therefore, to match an appropriate algorithm to the problem at hand.

This thesis takes a first step in the creation of such a framework by clarifying the biases of GP's *sub-tree swapping crossover* [Koza, 1992]. This operator works through a process of creating children by directly swapping groups of parent program components, known as *sub-trees*.

Of course we are not limited to simply mathematically describing the biases of our operator, no matter how useful this undoubtedly is. We can also see how operator biases affect GP runs. With this knowledge we can make recommendations as to experimental set-ups and even suggest new operators to counter or alter such biases.

With this in mind, the remainder of the thesis is structured as follows:

Chapter 2 provides an introduction to AI and search, describing how GP uses Darwinian Evolution to search the program space. Particular attention is paid to problem representation, notably the use of *syntax trees* that can be directly executed by an interpreter (or compiled for faster execution). Such trees contain nodes with connections between them. The number of connections a type of node may accept is termed its *arity*. Trees can be manipulated by sub-tree swapping crossover, which as its name suggests swaps parts of trees between individual programs. An analysis

of the biases associated with sub-tree swapping crossover form the major component of this thesis. Recent research into the nature of program spaces is also discussed in this chapter, notably regarding distributions of fitness and how they scale in relation to program size. Within the typical tree based GP representation, program length and size relate to total nodes within a program and are synonymous. Consideration is also given to the phenomenon of *bloat*, the growth of program size during a GP run without a significant return in terms of program fitness.

Chapter 3 provides a number of hypothetical models, generalisations and approximations (with experimental verification) to explain how sub-tree swapping crossover samples the search space in terms of program length. It is found that under repeated application, sub-tree swapping crossover will distribute a population in terms of length according to a *Lagrange Distribution of the Second Kind*. This is a family of distributions which has two parameters. The first is an average of internal tree node arities in the population. The second is the mean program length at generation zero. Without careful adjustment this distribution has a strong bias to sample small program length classes.

In Chapter 4 we look at how our mathematical models can help us describe GP search notably in relation to the sampling of unique programs and sampling by length classes. It is shown that the length bias of sub-tree swapping crossover is compounded with the combinatorial explosion of possible programs as length increases, i.e., it becomes increasingly difficult to sample specific larger programs using this operator. Without the sampling of such classes of programs we are unlikely to discover new solutions of sufficient sophistication to solve all but the simplest of problems. Interestingly mean program length is shown to have an effect on the degree of this sampling

bias towards smaller programs (variance in length values increases with the mean). This chapter also examines how crossover interacts with initialisation leading to a new bloat theory, *Crossover-Bias*. Length limits are also shown to increase the degree of sampling bias which can in turn speed bloat instead of slowing it down.

Chapter 5 looks at the effect of program resampling caused by sub-tree swapping crossover's bias to sample smaller programs. A new method to control the rate of resampling by altering program fitness is introduced. The method which we call *Sampling Parsimony* is shown to have a direct effect on program growth, an effect that can be explained in terms of the Crossover-Bias bloat theory.

Chapter 6 introduces a novel technique, *Operator Equalisation*, which can counteract the length bias produced by crossover or any other GP variation operator. This takes the form of a *wrapper* that can be placed around existing reproduction code which enables GP to sample according to predetermined length distributions. This technique can provide valuable information regarding the nature of GP search and the problems to which it is applied. It is also shown that for certain problems, GP search can be improved by the simple alteration of length bias obtained by this technique.

To complete our picture we look in Chapter 7, at how node labels, *alleles*, diffuse within length classes to enable us to determine how programs will be sampled within those classes. For mixed-arity representations, trees that contain internal nodes which accept differing numbers of inputs, sampling is not uniform within classes but determined by arity proportions within the population as a whole. With this in mind, our length work is shown, in Chapter 8, to be a specialisation of a broader model that predicts sampling frequencies from arity proportions or *histograms*. A theoretical model that

predicts the sampling of programs according to this mix is produced. This work explains why a population of programs, when GP with sub-tree swapping crossover is applied, is highly unlikely to converge to a single structure even if program fitness has converged.

Finally, Chapter 9 summarises the results from Chapters 3 to 8 and discusses the contributions made by this thesis. It also lists a number of avenues for future work based upon the findings presented here.

Chapter 2

Genetic Programming and Crossover

2.1 Artificial Intelligence, Automatic Programming and Search

Artificial Intelligence is a broad, multi-discipline subject, normally associated with the automation of human intelligence [Cawsey, 1997] or an idealised, *rational* version of this [Russell and Norvig, 2003].

Sub-divisions of AI are often defined in terms of their applications, for instance: *expert systems* [Jackson, 1999], *natural language processing* [Jurafsky and Martin, 2000], *machine vision* [Snyder and Qi, 2004], etc. We can also add *automatic programming* [Koza, 1992] to this list, i.e., to enable a computer to build other computer programs to a desired quality with minimal human intervention.

Central to all AI applications is the concept of *knowledge representation* [Callan, 2003], how we store, manipulate and draw inferences from information about the world that interests us [Davis et al., 1993]. For automatic

programming we are interested in the storage and manipulation of components of computer programs, primarily functions, their inputs and a sequence of activation.

A common problem found in all areas of AI is that of having to search through a number of potential solutions efficiently. For automatic programming our search consists of investigating different combinations of program components and determining their ability to solve a pre-defined problem. Unfortunately, the number of potential solutions will increase exponentially with each new feature to be considered, a problem termed *combinatorial explosion*. One soon reaches limits where problems become intractable if search is addressed with only a simple enumeration of all potential solutions. Efforts can be made to address this using a specific type of *blind search* (one that addresses only the problem definition) that may be more suited to a problem at hand. Examples in AI literature include *breadth first search*, *depth limited search*, etc. Another possible solution is to reduce the *search space*, the set of all possible solutions, by introducing certain constraints making areas of the space, or combinations of features, known to be invalid, unavailable. Although both of these ideas should be considered, one is still likely to face an intractable number of potential solutions for all but the very simplest of problems.

An alternative technique is to use an *informed search* strategy, one that uses knowledge beyond simply the problem definition. This is achieved using a *heuristic function*, one that provides guidance during the search by exploiting additional information about the problem. For example, we may, with automatic programming, wish to first use components judged more likely to be useful to the problem. This would of course be a human generated rule, i.e., counter to our aim of reducing human involvement.

If there are regularities in the search space that can be exploited, a more sophisticated approach would be to learn heuristics dynamically, or from experience as the search takes place [Russell and Norvig, 2003]. AI provides a number of techniques to achieve this by allowing relatively good sub-optimal solutions (according to an objective measure) to be retained and then altering those solutions in order to try to obtain a better solution. The most common of these methods is *hill-climbing*. Here, a number of solutions are generated from our previous best solution, by making normally small alterations. The best, newly generated solution, is retained and the process continues until a solution cannot be improved, i.e., we reach the top of the hill. A common variant of hill climbing is *simulated annealing* [Kirkpatrick et al., 1983], where potential solutions can be accepted, even if not the highest ranked, according to a certain probability that decreases with each successive round of solutions.

One can easily imagine how hill-climbing and simulated annealing can be applied to generating computer programs. We would generate a random solution and then make a number of random amendments, e.g., adding new components, re-ordering etc, thereby creating a number of new programs. Each program would then be run and the result of which assessed using an objective measure, the best being selected for the next round. This process would continue until the program could not be improved.[1]

It is important to note, the best solution found may not be the optimal solution, i.e., it may not be possible to alter the solution to improve it, whilst a better solution could have been obtained using a different starting point or having been presented with a different set of amendments during the search. Such solutions are termed *local optima*, whilst the best solutions

[1]Variations of this approach are described in [O'Reilly and Oppacher, 1994, O'Reilly and Oppacher, 1995].

in the search space are called the *global optima*.

We are not limited to analysing one solution at a time we could look at groups of solutions and carry out the process in parallel. Such a process is called *beam search*. This offers the ability to identify potentially more successful searches and to concentrate resources on them by abandoning those less successful. A modification of this process is to make additional amendments using groups of components found in other successful solutions with the expectation that they will produce a better improvement. At this point, our search method has begun to resemble the process of Darwinian evolution and is indeed the approach taken by a group of search methods collectively termed *Evolutionary Algorithms* (EAs) [Bäck and Schwefel, 1993, Eiben and Smith, 2003]. In addition, the swapping of components from successful individuals is particularly relevant to two sub-types of EAs called *Genetic Algorithms* (GAs) [Holland, 1975] and Genetic Programming. The following sections discuss Darwinian evolution, EAs, GA and GP in further detail.

Finally, it is important to discuss the No Free Lunch theorem. This theorem arises from a series of results, presented in [Wolpert and Macready, 1997], that show that there cannot exist a search algorithm, that will on average, be superior to all other search algorithms for all classes of problems. In effect, there is no *super* search algorithm that can be selected with the confidence it will always be better than any other search algorithm for any problem presented. It is important to note that these results also indirectly show that some search algorithms will perform better than others for certain classes of problem. One can also see that although a certain algorithm may be very successful in solving a highly specific problem, other algorithms may be more successful at solving a broader range of

problems, albeit not to the same degree [Schwefel, 2000]. One should, therefore, match a potential search algorithm to a problem. Characterising the biases of search algorithms, as attempted for GP in the following chapters, is an essential part of this process.

2.2 Darwinian Evolution

Spurred on by the independent discovery of his theory by Alfred Russel Wallace, Darwin published *On the Origin of Species by Means of Natural Selection* [Darwin, 1859] to explain how species adapt over time to exploit the niches in the environments that they inhabit. The theory can be summarised as follows:

- A species consists of a population of individuals that can reproduce with variation.
- More offspring are created than can survive given the resources of the environment.
- Those offspring that are most adapted to the environment are more likely to survive and hence produce more offspring.
- Therefore, favourable variations will be preserved and species will evolve, i.e., adapt to exploit their environment.

Darwin termed this continual process of preservation of favourable variations, *Natural Selection*, a term deliberately chosen to contrast to *Artificial Selection*, the process of breeding animals for traits deemed desirable by humans.

It is important to note that within the commonly used term to describe this process, *survival of the fittest*, *fittest* is employed to describe the *best*

adapted individuals to the environment. To illustrate this, one can easily imagine a case where increased physical strength would become a disadvantage with regard to survival if the resources required to maintain such strength were to become scarce.

Although the term *evolution* is only used at the very end of Darwin's book, a degree of gradual change is required for such adaptation to take place. Individuals already carry with them the ability to survive in a particular environment, major departures are far more likely to be disadvantageous in terms of survival than smaller changes [Dawkins, 2006a].

Darwin's theory was developed without any biological knowledge of genetics, i.e., how traits were inherited or modified. Indeed, it was not until almost a century later that Watson and Crick [Watson and Crick, 1953] identified the structure of the deoxyribonucleic acid (DNA) molecule which carries the genetic information used during reproduction. DNA having been identified as being the material of inheritance only a decade previously with the publication of [Avery et al., 1944].

Genetics is a young subject and major discoveries are still being made [Silver, 2007]. We can, however, draw some simple facts regarding the mechanism of inheritance. First, DNA replicates to produce more DNA. Second, DNA is transcribed into ribosenucleic acid (RNA) which is translated into proteins. Third, these proteins go on to affect the development of the offspring in conjunction with the growth environment.[2]

DNA is made up of a series of chemical bases: adenine, guanine, cytosine and thymine. Adenine and thymine pair with each other as do guanine and cytosine to produce the well known double helix structure. Each sequence of

[2]This sequence of events is commonly termed *the central dogma of genetics*. There can also be a number of alternate flows between DNA and RNA. However, this one-way model of information transfer has been the primary influence on the development of EAs.

three pairs contains the information to produce an amino acid, the building blocks of protein [Jones, 2000].

The term *gene* has traditionally been used to define a region of a DNA molecule which, through the various intermediate stages discussed, affects the development of a particular aspect of an organism [Dawkins, 2006b]. The development of a trait can be influenced by many genes, and genes can affect many traits. A strand of DNA is called a *chomosome* whilst gene positions on the chomosome are termed *loci*, the varying gene values are called *alleles*.

Variation can be achieved through simple copying errors, or mutations, of genetic material that will affect the eventual development of an individual so that it will exhibit different traits from its parent.

In higher biological life forms, variation in the form of sexual reproduction also takes place. Here, genetic material is taken from both parents so that a child will receive a mix of traits from each of its parents. Further variation takes place in that the genetic material contained in egg/sperm cells donated by each parent has been created by recombining (or crossing over) inherited genetic material from each parent's parents.

The evolutionary advantages of recombination in biological systems are still being debated, however the competing theories can broadly be divided into the following categories:

- Recombination is a form of macro mutation where multiple advantageous mutations that arose in separate individuals can be assembled quickly in a single individual [Dawkins, 2006b, page 44]. We are equally likely to produce an individual that has multiple disadvantageous mutations but this individual's combination of traits will soon be removed in successive generations by natural selection.

- To remove harmful mutations quickly by not having to wait for an opposite mutation event to take place. This evolutionary advantage is commonly termed *Muller's Ratchet* [Felsenstein, 1974].

- As a defence against parasites. Individuals are constantly being altered to keep ahead of other species that may evolve to take advantage of certain combinations of a host species traits. Popularly known as *The Red Queen Theory* [Ridley, 1994].

This section has described Darwinian evolution and the very basic mechanisms of inheritance. In the next section, we look at how the processes presented here can be abstracted and used to define a broad set of search algorithms.

2.3 Evolutionary Computation

From the previous section we can see how the process of Darwinian evolution can be viewed as a specialised form of beam search. We have a population of individual solutions which have varying degrees of survival success based upon fitness. At each step, or generation, variation is applied to more successful individuals using either small alterations and/or by recombining components, i.e., mutation or crossover.

If we add a method to create our initial population and a stopping condition to decide when to end our search, we have the essential components of an Evolutionary Algorithm. To illustrate this, skeleton pseudo code that describes most typical forms of EAs is presented in Figure 2.1.

EA populations are initialised with individuals that have been created randomly or *seeded* with individuals that are known to possess some benefit regarding the problem at hand. Stopping conditions typically include:

```
Initialise population

Evaluate fitness of population

while( stopping condition not met )
  for( each new population member )
    Select parent(s) based upon fitness
    Mutate and/or recombine parents to produce child
    Determine child fitness
    Add child to new population according to insertion policy
  endfor
endwhile
```

Figure 2.1: Skeleton pseudo code for a typical evolutionary algorithm.

finding an acceptable individual, reaching a pre-set maximum number of generations, or convergence of individuals' structure or fitness within the population.

Although the structure of the algorithm remains consistent amongst EAs, the implementation of components is highly reliant on the representation chosen for an individual. The major sub-types of EAs primarily divide in terms of individual representation. Evolution Strategies (ES) [Rechenberg, 1973] uses real value vectors, Evolutionary Programming [Fogel et al., 1966], finite state machines, GA, typically a fixed length string/vector with a predefined alphabet and classical GP employs syntax trees.

In order to gauge the success of an individual, it is passed to a *fitness function* which will normally return a numerical value indicating to what degree an individual solves a particular problem. This allows individuals to be compared when being selected as potential parents, the more successful individuals having a greater probability of selection.

With typical EA implementations there is little or no distinction be-

tween the representation that is to be manipulated and the eventual solution. Within traditional GP, for example, the syntax tree representation is executed without modification. Where a mapping does exist the representation is termed a *genotype* and the eventual solution is called a *phenotype* in order to be consistent with evolutionary terminology. Although typically the exception, interesting work has been carried out using a distinction between genotypes to be manipulated and phenotypes to be evaluated. For example, in GA, vectors have been used to generate neural nets to control mobile robots [Michel, 2001], whilst in GP, syntax trees have been used to generate electronic circuits [Koza et al., 2003], both using intermediate mapping phases.

It is important to note that EAs only use the metaphor of Darwinian evolution; no attempt is made to replicate the exact biological processes found in nature. GA and GP, however, as their names suggest, make an attempt to utilise some similarities to genetics in their implementation. GA in particular uses the analogy of a chromosome as a potential solution with parts of the string implementation being thought of as genes which in turn contain individual locations, or loci, that may contain one specific value, an allele, selected from a specific alphabet.

Although the analogy to genetics is severely restricted, certain mathematical results within evolutionary biology, have been found to be useful in analysing the mechanisms of EAs. For example, Price's Selection and Covariance Theorem [Price, 1970], which predicts the change of frequency of a gene within a population, in one generation, has been applied to gene frequencies in both GA and GP [Langdon and Poli, 2002], and used to help understand the evolution of solution size (program length) within GP [Poli and McPhee, 2008].

Recently it has been suggested that computer scientists should look beyond the constraints of traditional EAs providing algorithmic analogues to new findings in molecular biology, particularly with regard to factors that affect genotype/phenotype mapping. A new broader field named *computational evolution* has been proposed [Banzhaf et al., 2006]. Although such research holds great potential, this thesis concentrates on one existing EA technique, GP, taking the view that it is a search method in the AI sense, and, as such, should be compared to other alternative methods within that sphere.

2.4 Genetic Programming

After almost 20 years of research, as one would expect, there are an enormous number of alternatives to the basic algorithm and operators used in GP, see [Poli et al., 2008a] for a recent summary. We present here an outline of the basic model as described in Koza's original work [Koza, 1992]. This is the basis of all theoretical and experimental work presented in the thesis and has been deliberately chosen so as to provide a common reference for GP researchers who will be familiar with the founding work. It is also important to note that tree-based variable length representations used by Koza are still the mainstay of mainstream GP experimentation.

Both GA and GP differ from other EAs in their extensive use of the crossover as a means of variation [Eiben and Smith, 2003]. GP is inspired from GA but it is important to note that GP is a superset of GA[3] in spite of being the younger technique. Where they differ, is in terms of representation. GAs will commonly use fixed length vectors (or strings) designed to contain

[3]It is straightforward to construct typical GA implementations using GP components. See also work extending GA theoretical analysis to GP, for example [Poli and Langdon, 1998b]).

```
Initialise population

Evaluate fitness of population

while( stopping condition not met )
  for( each new population member )
    choose( reproduction method )
      Replication   : select one parent and copy
      Mutation      : select one parent, copy and mutate
      Recombination : select two parents and recombine components
    endchoose
    Determine child fitness
    Add child to new population according to insertion policy
  endfor
endwhile
```

Figure 2.2: Skeleton pseudo code for classical GP.

inputs to a function, whilst classical GP uses variable length syntax trees that can be actioned, using an interpreter, on input data and the result compared to test data provided. In effect, GAs seek to optimise the output of a function by discovering appropriate input values, whilst GP seeks to provide a function that maps inputs to outputs.

The basic GP algorithm is a specialised form of that described in figure 2.1. Commonly in GP, however, crossover, replication (copying without alteration) and mutation are applied mutually exclusively, according to fixed probabilities [Koza, 1992, Koza, 1994]. Also, often, only one child is produced as a result of recombination. Skeleton pseudo code of the classical GP algorithm is shown in figure 2.2.

GP experiments are normally run with stopping criteria based upon a maximum number of generations or the discovery of an acceptable solution. A stopping condition commonly found in GA, structural convergence of solutions, is not used in GP [Banzhaf et al., 1998, page 278]. We discuss the

reasons for this and suggest alternatives in Chapter 7.

Syntax tree representation and sub-tree swapping crossover can be seen as the defining elements of GP [Eiben and Smith, 2003]. Note, Koza did not use mutation in his early work [Koza, 1992, Koza, 1994] to show that GP was not performing a simple random search. This thesis, therefore, concentrates on search biases inherent in the choice of this representation and variation operator. Syntax trees, and the crossover operation applied to them, are described in the following two sub-sections.

2.4.1 Syntax Trees

With the application of variation operators such as crossover and mutation, it is important that our chosen representation will produce valid offspring. This is necessary for two reasons. Firstly, that resources are not wasted on checking the validity of child programs, and secondly, to reduce the overall size of search space. The use of syntax trees ensures that offspring will be syntactically valid, i.e., that they can at the very least be parsed and interpreted.

In graph theory, trees are defined as *a connected graph without circuits* [Truss, 1999], i.e., a graph in which all nodes are connected and for which there is only one route to each node from any other node in the graph/tree. Trees are *non-reentrant* i.e. child nodes have only one parent.

For GP purposes, syntax trees contain two distinct types of nodes called *functions* and *terminals*. These are often alternatively termed *internal nodes* and *external nodes* (or *leaves*) respectively. The term *non-terminal* is also often used to denote a function/internal node. As their name suggests, functions accept inputs and produce an output, whilst terminals return values only.

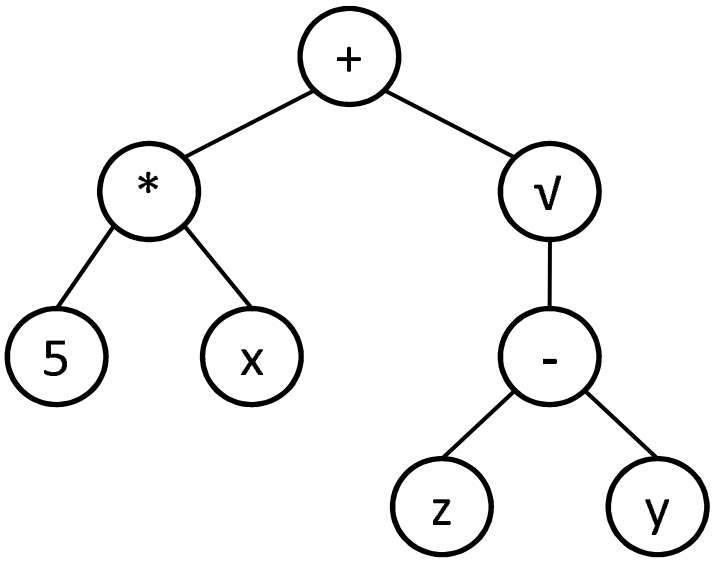


Figure 2.3: Pictorial representation of a syntax tree for the algebraic expression: $(5 * x) + \sqrt{z - y}$. Internal nodes are used for the operators $*$, $+$, $\sqrt{}$ and $-$. External nodes are used for the constant 5 and the variables x, y and z. GP program length/size is 8 nodes.

The total number of nodes in a syntax tree used to represent a GP program is termed *program length* although the equivalent term *program size* is often used. Within this thesis the terms are used interchangeably.

Figure 2.3 shows a pictorial representation of a syntax tree for the algebraic expression: $(5 * x) + \sqrt{z - y}$.

Such trees can be implemented in a number of ways. The ECJ System [Luke, 2008] uses a pointer, or linked list type structure. These are conceptually easy to understand, although there is an overhead in the storage of pointers (one for each function argument).

Other systems, such as TinyGP [Poli et al., 2008a, Appendix B], employ array based implementations using prefix or postfix notations. These can be efficiently stored as variable length structures in programming languages such as C, C++ and Java.

Table 2.1: A comparison of different array based storage representations for the algebraic expression: $(5 * x) + \sqrt{z - y}$ as shown in Figure 2.3.

Method	Representation
Postfix Notation	$5x*zy-\sqrt{}+$
Prefix Notation	$+*5x\sqrt{}-zy$
S-Expression	$(+(*5x)(\sqrt{}(-zy)))$

Koza originally used Lisp *S-Expressions* [Koza, 1992, Koza, 1994] to represent GP trees. These are syntactically the same as prefix notation but with parentheses or brackets to indicate the beginning and end of a sub-tree. As the arities of functions are known in advance, the interpreter can parse the tree without ambiguity even without brackets. There is, therefore, no need for the storage of such visual clues when implementing a GP system. A GP system implementing an S-Expression representation need only add appropriate brackets to an individual's representation when outputting data.

Table 2.1 shows a comparison of different array based storage representations for a sub-tree for the expression in Figure 2.3.

Interestingly, XML (eXtensible Markup Language) [XMLCoordinationGroup, 2008] constructs have the same syntactic structure as S-Expressions and can be parsed in a similar way. XML is unlikely to be used as an internal storage representation, i.e., one that will be directly manipulated and interpreted, due to its reliance on textual tags. However, GP systems could readily make use of the myriad XML software tools available to store, examine and retrieve individuals or populations of individuals in light of these similarities.

Although the use of syntax tree representation allows GP to avoid syntactic problems, the practitioner must also be aware of implementation issues regarding the choice of functions and terminals. For example, Koza [Koza, 1992, pages 81–86] states that for GP to work effectively the

function and terminal sets must have a property termed *closure*. This can be broken down into two further properties: *type consistency* and *evaluation safety* [Poli et al., 2008a]. The first ensures that functions can accept all values returned by other nodes. This can be illustrated by looking at a function set that includes arithmetic functions and logical operators such as an IF statement. Obviously a boolean output cannot be returned to an arithmetic operator, e.g., multiplication. This particular problem could be solved by returning a numerical value, e.g., a -1 for FALSE and +1 for TRUE. An alternative method to ensure *type consistency* is to build initialisation and variation operators that will guarantee that types are correctly matched; *strongly typed GP* [Montana, 1995] is one such method.

The second property, evaluation safety, concerns runtime issues where a program cannot be evaluated correctly. For example, a common problem is found in arithmetic problems where a division function is desired. If during program evaluation the input to the denominator parameter of the division function is zero, a mathematical error will occur. This is often solved with a bespoke *protected division* operator that returns a numerical value for such an occurrence. Often 1 is used; in Chapter 6 a function called PDIV replaces division with an operator which returns the numerator if the magnitude of the denominator is less than 0.001.

Of course one must also choose functions and terminals that will enable GP to solve the problem at hand, a property termed *sufficiency* [Koza, 1992, pages 86–88]. This is normally a human activity based on knowledge and experience although useful functions and terminals, or combinations of those nodes, can be discovered automatically. Identifying important program components falls under an activity called *trait mining* [Tackett, 1995]. More recently the discovery of important GP tree *fragments* has been attempted us-

ing a specialised algorithm [Smart et al., 2007]. The dynamic discovery and reuse of important program components, in an attempt to add scalability to GP, has been investigated by a number of researchers. The most well known technique, *Automatically Defined Functions* (ADFs) [Koza, 1994], divides a tree into function defining branches and result producing branches, the latter being able to call the former multiple times. Other methods designed to introduce modularity to GP experimentation include Koza's *Encapsulation Operator* [Koza, 1992] and component library production methods [Angeline and Pollack, 1992, Rosca and Ballard, 1996].

Although obviously an important subject, within this thesis standard test problems have been used that satisfy both *closure* and *sufficiency* properties in order that we may concentrate on specific search bias issues.

Finally, it is worth pointing out, that numerous variants of tree-based representations exist. Common alternatives include: *Linear GP* [Nordin, 1994], *Graph Based GP* [Teller and Veloso, 1996, Poli, 1999] and *Cartesian GP* [Miller, 1999]. We shall, however, concentrate on the classic syntax tree representation for the analysis presented in this thesis.

2.4.2 Sub-Tree Swapping Crossover

Sub-tree swapping crossover is, as its name suggests, a recombination operator that swaps sub-trees between donating parents. The process can be described as follows: first, two parents are presented to the operator, having been chosen by an appropriate selection method. Two crossover points are then chosen, one within each parent, according to a node selection strategy. The sub-tree rooted at the selected crossover point in the first parent is removed and replaced with the sub-tree rooted at the crossover point in the

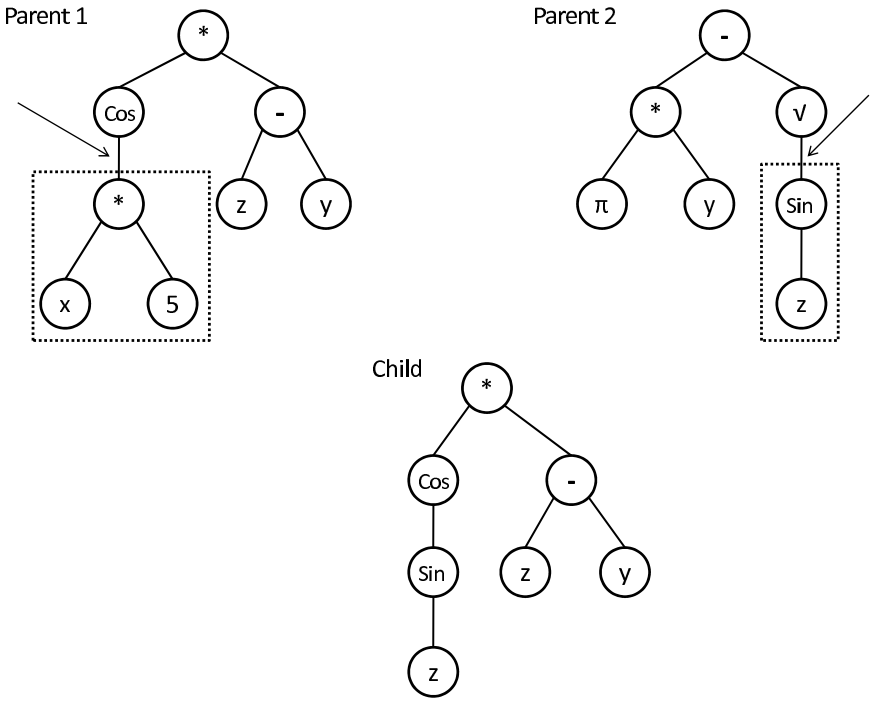


Figure 2.4: Pictorial representation of sub-tree swapping crossover. A crossover point, represented by an arrow, is chosen in each of the parents. The sub-tree below the crossover point in the first parent, designated by a dotted box, is then replaced by that of the second parent to produce the child program.

second parent.[4] The resulting tree then becomes the *child* of these programs and is placed into the population.

Figure 2.4 illustrates sub-tree swapping crossover for two selected parents.

The first parent providing the root of the new program, i.e., with the sub-tree component excised, is termed the *root donating* parent. The second parent provides the new sub-tree component for the child and is called the *sub-tree donating* parent. In some implementations, the second parent will receive the sub-tree removed from the first and a second child will also be

[4]Note, as normally there is a possibility of parents being reselected to produce additional children, sub-tree swapping crossover works on copies of parents.

produced.

Practically, for pointer based systems, this process is relatively straightforward to implement, the pointers to the selected sub-trees are simply swapped between parents. Sub-tree swapping crossover is also relatively easy to achieve when GP is implemented using Lisp like S-Expressions or prefix notation. One selects crossover points as normal, records their position and then traverses each sub-tree from that point until no more nodes can be read, the final position being the end position of a continuous *string* to be excised. Both *sub-strings* are then swapped between parents. If two children are required, the excised string from one parent will need to be stored temporarily. Note, it is also possible to carry out sub-tree swapping crossover using a GP representation based upon postfix notation, although the process is slightly more complicated and requires the use of a *stack* structure, see [Banzhaf et al., 1998, pages 326–327] for a description.

Common node selection strategies include *uniform selection*, where all nodes have an equal probability of being selected for crossover and *90/10 selection*, where internal nodes are chosen with a 90% chance of selection and terminals the remaining 10%. 90/10 selection is often used to limit the production of single node children normally associated with low fitness; Koza also points out that such a policy also limits simple terminal swapping, a process similar to point mutation [Koza, 1992]. Uniform selection is used predominantly in this thesis although 90/10 selection is addressed in Section 3.5. Both methods have been termed as *standard crossover* in various GP publications leading to some confusion. It is more useful to see these forms of crossover as variants of standard crossover which can be defined more broadly as a sub-tree swapping crossover operator that chooses crossover points by using a pre-determined probability distribution based on a func-

tion/terminal distinction only. Interestingly, if we extended our definition to node return type we could also include strongly typed crossover.

Using alternative node selection strategies, there are also a number of more sophisticated sub-tree swapping crossover operators designed to achieve a number of theoretical goals. The most common of these are a broad class of operators called *homologous* crossover operators. The reasoning behind these crossovers is to swap genetic material from similar positions within the parent trees. This is analogous to the recombination of genes during biological reproduction and also similar to that found in typical GA implementations, i.e., *one-point*, *two-point* and *uniform* crossover methods commonly used with binary vector representations [Beasley et al., 1993]. Examples of such operators in GP include *context-preserving crossover* [D'haeseleer, 1994]; here crossover points were constrained to have same coordinates. GP *one-point crossover* [Poli and Langdon, 1998b] and *uniform crossover* [Poli and Langdon, 1998a] use the notion of a common region where an algorithm is used to identify the limits to a common tree shape from the parents' root nodes to allow node selection only from both parents in this area. Uniform crossover also allows the swapping of multiple nodes from each of the parents within the common region so that genetic material can be mixed to a greater degree than that found by simply swapping a single sub-tree.

Differing methods can have a dramatic effect on the sampling of program size. For example, in [Langdon and Poli, 2002] it was shown that one-point crossover will restrict program growth as child programs cannot increase in depth beyond that of their parents; a maximum depth is therefore set by the initial generation. Also, as its name suggests, *size fair crossover* [Langdon, 2000] is designed to prevent excessive code growth by choosing a

crossover point in the first parent as normal whilst only selecting a node in the second parent for which its sub-tree is not *unfairly* large.

In this thesis, we have restricted our analysis to standard sub-tree swapping crossover, which, although by no means the full story of sub-tree swapping crossover, forms the basis of most other methods and will provide a common starting point for their analysis.

Finally, it is important to note that the crossover methods presented here take no account of individual fitness either of the parents or the resultant child.[5] The recombination exercise is purely a mechanical process applied to a *mating pool* of individuals presented by selection. This distinction becomes significant when we discuss a new theory for bloat in Section 4.5. Bloat, an important subject in GP, is discussed in the following section.

2.5 Bloat

The tendency for programs to grow rapidly during a run has been observed from the early days of GP research. In light of the bias of traditional GP initialisation methods to produce relatively small programs, some program growth is expected if programs are to display necessary levels of sophistication to solve complicated problems. The term bloat is used to describe program growth without a *significant* return in terms of program, i.e., that programs have increased to such a size that the resources required to evaluate them, now outweigh any, relatively small, improvements made in fitness. Although it is important to distinguish bloat from pure program growth, it remains a significant problem in GP experimentation and as such has

[5]There is no reason why recombination operators may not address eventual fitness of a child. A number of non-destructive operators have been devised that will not return a child that is less fit than a parent, see [Soule and Foster, 1997, Langdon et al., 1999]. More recently, context-aware crossover [Majeed and Ryan, 2007] examines the effect on child fitness of potential donated sub-trees.

received a considerable amount of attention.

Numerous theories to explain bloat have been put forward. *Replication Accuracy* [McPhee and Miller, 1995] argues that the success of an individual depends on its ability to be functionally similar to its parents. As variation operators on average tend to reduce fitness, larger programs have an evolutionary advantage in that they can reduce the effects of such operators. *Removal Bias* [Soule and Foster, 1998b] divides program components into active and inactive code in terms of their effect on fitness. Inactive code tends to be towards the bottom of a tree and is, therefore, smaller than active code. Programs that excise inactive code will have an evolutionary advantage in that they will have a similar fitness to their parents. Inactive code will be relatively smaller, on average, than the code it is replaced with; therefore, programs will grow.

The *Nature of Program Search Spaces Theory* [Langdon et al., 1999] is based on research that shows that after a certain size threshold, the distribution of program fitnesses will converge. As there are more larger programs than smaller ones with the same fitness, search methods will simply find more longer representation solutions than smaller ones.

A further bloat theory, *crossover-bias* [Dignum and Poli, 2007], based on research presented in this thesis regarding the interaction between selection and length distributions presented by sub-tree swapping crossover, is described in Section 4.5.

Bloat has traditionally been controlled by the application of program length or depth limits, i.e., children are prevented from being created if they exceed specific, pre-defined, lengths (number of nodes) or tree depths. Practically, this is achieved by returning one of the parents or by repeating the process until a child with an acceptable length is produced. The application

of length limits in relation to bloat is analysed further in Section 4.6.

In addition to applying length or depth limits, more sophisticated methods to control bloat have also been suggested [Langdon et al., 1999, Soule and Foster, 1998a, Luke and Panait, 2006]. The most common of these is *parsimony pressure* [Koza, 1992, Zhang and Mühlenbein, 1993, Zhang and Mühlenbein, 1995, Zhang et al., 1997].[6] With this method, a value is added to the fitness function that will penalise programs as they increase in size, the penalty typically being a function of program length. Recent work has extended this method to dynamically alter the penalty in light of changing experimental characteristics [Poli and McPhee, 2008]. Inspired from such techniques, an alternative parsimony method based on the resampling of programs is presented in Chapter 5.

Rather than altering all program fitnesses to control bloat the *Tarpeian* method [Poli, 2003] sets the fitness of a certain proportion of larger programs, determined stochastically, to the minimum fitness, in effect creating dynamic *fitness holes* in a problem's fitness landscape. As the implementation consists of a wrapper around evaluation code, unnecessary fitness calculations can be avoided.

In order to combat bloat, one can also ensure that variation operators are biased to sample certain program lengths, for example, size fair crossover described in the previous section or size fair mutation [Langdon, 2000, Crawford-Marks and Spector, 2002]. *Shrink Mutation* [Angeline, 1996] is an extreme approach where sub-trees are replaced by a randomly selected terminal ensuring program size reduction.

A new method to directly control the sampling of program size, hence, preventing bloat is presented in Chapter 6. In the next section we discuss

[6] Parsimony is used in the sense of being careful when giving.

the importance of program length to GP experimentation.

2.6 GP Search and Program Length

By choosing a syntax tree representation for our potential solution, we have in effect chosen the nature of our search space: the set of all distinct syntax trees up until some length limit. This limit may be directly applied through a specific length limit, indirectly through a depth limit, or be simply a computational barrier, i.e., a length after which the tree can no longer be processed.

For problem specific reasons, one can limit the space further by ignoring certain program types, known from previous research to contain low quality solutions. However, one can say that program length will at some point become the limiting factor of our search space.

For syntax tree representations, the number of programs of a particular length grows exponentially as program length increases [Langdon and Poli, 2002]. As discussed in Section 2.1, if we were to choose programs using a blind method such as random search or an enumeration, we would soon come up against the problem of combinatorial explosion. A more sophisticated search method is, therefore, required.

Strong theoretical and empirical evidence has been provided to show that beyond a certain problem dependent length threshold, the distribution of functionality for S-Expressions, i.e., our syntax trees, will, without side effects, remain constant [Langdon and Poli, 2002].[7] As fitness is derived within GP directly from functionality, fitness will also reach a constant distribution. Ideally, to conserve resources, one would like to sample from, or

[7] This work forms part of a wider investigation into the convergence of distribution of fitness landscapes for a variety of computer types, see [Langdon, 2002a, Langdon, 2002b, Langdon, 2003, Langdon and Poli, 2006].

at least up until, the shortest length at which an acceptably high fitness value can be obtained. Whether a search algorithm such as GP is capable of finding such a solution without sampling some larger classes is, of course, debatable. Indeed [Langdon et al., 1999] argues that there may be smoother fitness gradients leading to solutions at larger program sizes.

In light of our chosen syntax tree representation, understanding the bias to sample certain lengths by our operators is imperative if we are to understand the success of GP in the discovery of solutions for particular classes of problems. This thesis, therefore, pays particular attention to the sampling of program length for our chosen search operator – sub-tree swapping crossover.

A number of theoretical results already exist in this area, notably using schema theory. Schemata are templates that describe genotypic similarities. Using schemata it is possible to produce equations that calculate the probabilities of particular classes of programs reaching succeeding generations. Schemata were originally used by Holland [Holland, 1975] to provide a theoretical basis for the analysis of GA and have been extended by GP researchers to study the mechanisms of GP. Of particular interest is [Poli and McPhee, 2001] where schemata were used to understand the effects of the application of sub-tree swapping crossover in the case of linear-GP, i.e., only terminals and functions of arity 1 were used. Under these conditions the operator was found to distribute the population, by length, according to a Gamma distribution whose parameters could also be calculated. Also of note are [Poli and McPhee, 2003a, Poli and McPhee, 2003b] where a Cartesian node reference system is used to define program spaces. This representation enabled manipulation of schema equations to predict the evolution of program size with and without the application of fitness

for mixed arity trees for symmetric[8] sub-tree swapping crossover. Results from this research area are used throughout this thesis, particularly in the following chapter.

Recent research, presented in [Keijzer and Foster, 2007], has looked at the distribution of tree sizes undergoing repeated application of sub-tree swapping crossover, using an analysis of the affect on *visitation length*. This is the sum of the number of nodes traversed in a tree to visit every node starting from the root node. Empirical evidence is provided to support the conjecture that any crossover bias affects the distribution of tree sizes but has no affect on the sampling of program shapes. We provide further evidence for this conjecture in Chapters 3, 7 and 8.

2.7 GP and Machine Learning

Finally, although this thesis treats GP as primarily a search algorithm, one could also view GP, or any algorithm that improves its performance over time, as a form of *Machine Learning* [Mitchell, 1997]. Within this discipline, GP typically falls within the sphere of *supervised learning* where a group of training instances consisting of inputs matched with correct outputs are provided. The aim of the process is to learn the correct function (or set of rules) to match input to output data. Symbolic Regression problems (fitting a mathematical formulae to observed data) and their typical presentation in GP experimentation [Koza, 1992] are a good example of this. GP can also be applied to *unsupervised learning*, where correct outputs are not provided and the system looks for patterns within the input data. One can see such learning in typical data mining techniques where similarities in data sets are discovered, for an example see [Freitas, 2002, Chapter 7]. Banzhaf et al.

[8] Crossover points are selected independently within each parent.

[Banzhaf et al., 1998] argue that many GP applications are closely related to *reinforcement learning*. Here, although correct outputs are not provided, a function to determine quality of solutions discovered is used and the resulting values fed back to the system. As any complex fitness function could fall into this category, one could imagine a fitness function that also penalises increased program length, such as common parsimony pressure methods as described in Section 2.5, to be a form of reinforcement learning.

2.8 Summary

In this chapter we have described how the discovery of computer programs, that perform pre-defined tasks, can be achieved using search techniques as defined in the discipline of AI. Blind search techniques are faced with the problem of combinatorial explosion, an inability to generate and evaluate all potential solutions as more components are added. One must, therefore, use information about the search space (heuristics), in order to guide the search so that resources are directed to investigate areas with a higher likelihood of solution discovery. This can be achieved dynamically by evaluating generated solutions and reusing components from those that have been most successful. If a population of potential solutions are generated and components are recombined using stochastic methods, our search becomes analogous to Darwinian evolution. As their name suggests, evolutionary algorithms are a class of search methods that use this analogy. One of these methods is GP which, although many variants now exist, traditionally evolves syntax trees – structures that can be evaluated by execution using an interpreter or compiler without further modification. This form of GP typically uses a specialised recombination operator called sub-tree swapping crossover, which swaps sub-trees between parent programs.

In order to successfully match a search method to a potential problem, we need to examine the biases of its associated operators. If one wishes to analyse GP, a good starting point is to investigate sub-tree swapping crossover. Given GP's choice of syntax tree representation, of particular interest to our analysis will be how our operator samples program size. This is the subject of the next chapter.

Chapter 3

Program Length Distributions

3.1 Introduction

With the advent of a greater understanding of program search spaces—for example we now know that the functionality of programs reaches a limit as program length increases [Langdon and Poli, 2002, Langdon, 2002b, Langdon, 2003]—acquiring knowledge on how GP operators sample program length classes has become more and more urgent. Ideally, we would like to sample the length class where the smallest optimal programs can be found. Unfortunately, in general: a) one does not know where solutions (let alone the most compact ones) are, and b) genetic operators present specific length biases which are often unknown or only partially known and, therefore, are difficult to direct and control.[1] In any case, a characterisation of operator bias is needed in understanding how GP will sample the search space in the first instance before technically-sound, problem-specific, modifications can

[1] Chapter 6 presents a generalised technique called operator equalisation designed to address both of these issues.

be made.

In this chapter, we provide an exact characterisation of the limiting distribution of tree sizes towards which sub-tree swapping crossover with uniform selection of crossover points, when acting on its own, pushes the population. Obtaining this type of result is complex, so we initially limit our attention to the case where the primitive set includes only terminals and primitives of one other arity. Trees of this type are described as a-ary trees, a being the common arity of non-terminal primitives, i.e., functions/internal nodes. A probability distribution is derived in Section 3.3 detailing the expected proportion of trees with a specific number of internal nodes when the population is at equilibrium. The resulting distribution is then generalised in Section 3.4 to describe the distribution for mixed arity trees. Our generalised internal node distribution is then modified in Section 3.5 to describe the 90% function/10% terminal node selection policy (see Section 2.4.2) commonly used by GP practitioners. In all cases empirical evidence is provided to verify the distributions proposed.

In Section 3.6, we transform the distributions for a-ary and mixed arity trees to complete program length distributions, i.e., those that include terminals as well as internal nodes. Strong empirical evidence is provided for the a-ary tree distribution, whilst the mixed arity distribution is shown to be an accurate approximation of experimental results with only minor discrepancies. The reasons for these discrepancies are analysed in Chapters 7 and 8.

Results presented in this chapter point to a consistent bias for standard sub-tree swapping crossover to excessively sample smaller programs under typical GP experimental conditions. The effects of such a bias on GP search are analysed in detail in Chapters 4 and 5 of this thesis.

3.2 Mathematical Preliminaries

3.2.1 Branching Processes and Lagrange Distributions

In probability theory, a *Galton-Watson process* [Watson and Galton, 1875], is a discrete-time branching process that models a population in which each individual in a generation produces some random number of descendants before its demise. The probability of producing a descendants, p_a, is fixed. This leads to a family tree like structure.[2]

Branching processes have at least one application in GP: if no limits are placed on tree creation, e.g., length or depth, the tree shapes produced by the GROW method [Koza, 1992] often used to initialise populations and to perform sub-tree mutation, obey such a branching process. In this case a is the arity of the primitives and p_a the probability of choosing primitives of arity a when constructing new random trees.

The distribution of tree sizes for a branching process follows a *Lagrange distribution* [Consul and Shenton, 1972, Good, 1975]. More precisely, the probability of a process leading to a total of ℓ individuals being generated is

$$\Pr\{L = \ell\} = \begin{cases} 0 & \text{if } \ell = 0, \\ \frac{1}{\ell}\mathcal{C}(t^{\ell-1})\left\{(g(t))^{\ell}\right\} & \text{for } \ell = 1, 2, 3, \cdots, \end{cases} \tag{3.1}$$

where $g(t) = \sum_a p_a\, t^a$ is the probability generating function of the distribution p_a and $\mathcal{C}(t^m)$ denotes the coefficient of t^m in the expansion of $(g(t))^{\ell}$.

If one considers a process where only nodes of arity a and 0 are allowed

[2]Note, the original work [Watson and Galton, 1875] was designed to answer the question of why family surnames disappear.

(i.e., $p_0 + p_a = 1$), then $g(t) = p_0 + p_a t^a$. So, for $\ell > 0$ we have

$$\mathcal{C}(t^{\ell-1})\left\{(g(t))^\ell\right\} \;=\; \mathcal{C}(t^{\ell-1})\left\{(p_0 + p_a t^a)^\ell\right\} = \mathcal{C}(t^{\ell-1})\left\{\sum_{k=0}^{\ell}\binom{\ell}{k} p_0^{\ell-k}\, p_a^k\, t^{ak}\right\}$$

the final bracketed term being the binomial expansion of $(p_0 + p_a t^a)^\ell$. Since $\mathcal{C}(t^{\ell-1})$ will pick out the coefficient of the power of t for which $\ell - 1 = ak$, i.e., $k = \frac{\ell-1}{a}$, we then have

$$\Pr\{L = \ell\} = \begin{cases} \frac{1}{\ell}\binom{\ell}{\frac{\ell-1}{a}}(1-p_a)^{\ell-\frac{\ell-1}{a}}\, p_a^{\frac{\ell-1}{a}} & \text{if } \ell-1 \text{ is a multiple of } a, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2}$$

Note that, since only arity 0 and arity a primitives are allowed, a tree with $\ell - 1$ nodes has $n = \frac{\ell-1}{a}$ internal nodes and $\ell = an + 1$. So, we can rewrite the previous equation as

$$\Pr\{N = n\} = C_{\mathcal{T}}\,(a, n)\;(1 - p_a)^{(a-1)n+1}\, p_a^n, \tag{3.3}$$

where

$$C_{\mathcal{T}}\,(a, n) = \frac{1}{an+1}\binom{an+1}{n} \tag{3.4}$$

is a *generalised Catalan number* [Hilton and Pederson, 1991] which represents the number of possible different trees with n internal nodes of arity a (and, of course, $(a-1)n+1$ leaves). This formulation can be interpreted as saying that in a branching process all trees of a particular size have identical probability of being created, and this probability depends only on how many nodes/primitives of each kind the tree contains. This also means that the different parts of the trees created by a branching process are uncorrelated.

3.2.2 Moments of the Tree-Size Distribution in a Branching Process

It is possible to compute the moments of Lagrange distributions starting from the cumulants g_i of the probability density functions generated by the power series in t of $g(t)$ [Good, 1975, Consul and Shenton, 1972]. The mean progeny produced by a branching process is

$$E[L] = \frac{1}{1 - g_1} \tag{3.5}$$

and the variance is

$$Var[L] = \frac{g_2}{(1 - g_1)^3} \tag{3.6}$$

where $g_1 = E[A]$ and $g_2 = Var[A]$, A being a stochastic variable representing a node's arity. Since trees contain only arity 0 and arity a nodes, we can easily compute these two cumulants

$$g_1 = \sum_k k p_k = a p_a \tag{3.7}$$

$$g_2 = E[A^2] - (E[A])^2 = a^2 p_a - (a p_a)^2 = a^2 p_a (1 - p_a) \tag{3.8}$$

So, the mean tree size in our branching process is

$$E[L] = \frac{1}{1 - a p_a} \tag{3.9}$$

and the variance is

$$Var[L] = \frac{a^2 p_a (1 - p_a)}{(1 - a p_a)^3}. \tag{3.10}$$

From these two, we then obtain then the second non-central moment

$$E[L^2] = Var[L] + (E[L])^2 = \frac{(a - 1) a p_a - a^2 p_a^2 + 1}{(1 - a p_a)^3} \tag{3.11}$$

Note that (3.9) matches the formula for the mean size of programs built by the GROW method reported in [Luke, 2000] and that (3.9) is a special case of it.

3.3 The Distribution of Program Lengths for A-ary Trees under Sub-Tree Swapping Crossover

In the absence of selection, if a population of GP trees undergoes repeated crossovers, it tends to a limiting distribution of sizes and shapes. This is the result of the specific bias of sub-tree swapping crossover.[3] Effectively after a while, every node in every individual in the population will have been placed at its particular position as a result of one or multiple crossover events. So, any correlations present in the shapes in the initial generation will have been broken by crossover.

As we saw in the previous section, complete decorrelation in the different parts of a tree is a characteristic of branching processes. Within the class of trees of a given size, each shape is equally likely. So, one can postulate that the limiting distribution of tree sizes under repeated crossover will be one where the same happens. That is, we assume that at the fixed-point, the shape distribution is

$$\Pr\{\text{Shape with } n \text{ nodes of arity } a\} = w(n,a)\,(1-p_a)^{(a-1)n+1}\,p_a^n \quad (3.12)$$

where $w(n,a)$ is an appropriate sequence of weights to be determined and p_a is a parameter, also to be determined. So, the probability of picking a

[3]Naturally stochastic effects, such as drift, mean that in any finite population there is still random variation. However, in large populations these effects can be neglected.

tree with n internal nodes from the population is

$$\Pr\{n\} = C_{\mathcal{T}}\,(a,n)\,w(n,a)\,(1-p_a)^{(a-1)n+1}\,p_a^n \tag{3.13}$$

What constraints do we have on the parameters $w(n,a)$ and p_a? Firstly, they must be such that the distribution of shapes is indeed a probability distribution. In particular we require

$$\sum_{n\geq 0} \Pr\{n\} = 1. \tag{3.14}$$

Secondly, it is well-known that on average sub-tree swapping crossover does not alter the mean size of program trees in a population [Poli and McPhee, 2003b]. So, we also require that

$$\sum_{n\geq 0} (an+1)\Pr\{n\} = \mu_0, \tag{3.15}$$

where μ_0 is the average size of the individuals in the population at generation 0. Thirdly, we require (3.13) to be a generalisation of the results reported in [Poli and McPhee, 2001, McPhee et al., 2001, Rowe and McPhee, 2001, Poli and McPhee, 2003b] for 1-ary trees, which we rewrite here as

$$\Pr\{\ell\} = \ell r^{\ell-1}(1-r)^2, \tag{3.16}$$

where

$$r = (\mu_0 - 1)/(\mu_0 + 1). \tag{3.17}$$

We can do this by setting $a = 1$ in (3.13) and $\ell = n+1$ and so $\Pr\{\ell\}$ in (3.16)

is the same quantity as $\Pr\{n\}$ in (3.13). Equating the results we obtain

$$w(n,1)\,(1-p_1)\,p_1^n = (n+1)(1-r)^2 r^n \tag{3.18}$$

since $C_{\mathcal{T}}\,(1,n) = 1$. The most natural match between r.h.s. and l.h.s. of (3.18) appears to be one where $p_1 = r$ and $w(n,1) = (n+1)(1-p_1)$.

This last constraint completely rules out that $w(n,a)$ be constant. Indicating that the length distribution under subtree crossover cannot be purely the result of a branching process (i.e., it is not Lagrangian). Instead, it suggests that $\Pr\{\ell\}$ is the product between the frequency provided by a branching process and the length ℓ of programs. So, we postulate that in general

$$w(n,a) = (an+1)f(p_a) \tag{3.19}$$

where $f(p_a)$ is a function of p_a to be determined.

With this assumption, we impose constraint (3.14), i.e., that probabilities sum to 1, obtaining

$$f(p_a) = \frac{1}{\sum_{n\geq 0}(an+1)C_{\mathcal{T}}\,(a,n)\,(1-p_a)^{(a-1)n+1}\,p_a^n}. \tag{3.20}$$

The denominator of this equation is (by definition) $E[aN+1] = \sum_n (an+1)\Pr\{N=n\}$, where $\Pr\{N=n\}$ is given in (3.3). So, it is the expected length of the trees generated by a branching process where arity a nodes are used with probability p_a and arity 0 nodes used with probability $1-p_a$. So, from (3.9) we have

$$f(p_a) = (1-ap_a), \tag{3.21}$$

and so $w(n,a) = (an+1)(1-ap_a)$. As a result, we can now explicitly write

the *tree-size distribution at the crossover fixed-point*:

$$\Pr\{n\} = (1 - ap_a)\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n \tag{3.22}$$

where we used the explicit expression of $C_{\mathcal{T}}(a,n)$ in (3.4). This is the fixed-point tree-size distribution we were looking for. This distribution belongs to a family of distributions called *Lagrange distributions of the second kind* [Janardan and Rao, 1983, Janardan, 1987].

We can now impose constraint (3.15) to infer the value of p_a:

$$\begin{aligned}
\mu_0 &= \sum_{n\geq 0}(an+1)\Pr\{n\} \\
&= \sum_{n\geq 0}(an+1)(1-ap_a)\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n \\
&= (1-ap_a)\sum_{n\geq 0}(an+1)\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n \\
&= (1-ap_a)\sum_{n\geq 0}(an+1)\frac{an+1}{an+1}\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n \\
&= (1-ap_a)\sum_{n\geq 0}(an+1)^2\frac{1}{an+1}\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n \\
&= (1-ap_a)\sum_{n\geq 0}(an+1)^2 C_{\mathcal{T}}(a,n)\,(1-p_a)^{(a-1)n+1}\,p_a^n \\
&= (1-ap_a)\sum_{n\geq 0}(an+1)^2\Pr\{N=n\} \qquad \text{(from (3.3))} \\
&= (1-ap_a)E[L^2] \qquad \text{(by definition)}
\end{aligned}$$

and so, from (3.11),

$$\begin{aligned}
\mu_0 &= (1-ap_a)\frac{(a-1)ap_a - a^2p_a^2 + 1}{(1-ap_a)^3} \\
&= \frac{(a-1)ap_a - a^2p_a^2 + 1}{(1-ap_a)^2}.
\end{aligned}$$

By solving this equation for p_a we obtain

$$p_a = \frac{2\mu_0 + (a-1) - \sqrt{((1-a) - 2\mu_0)^2 + 4(1-\mu_0^2)}}{2a(1+\mu_0)} \tag{3.23}$$

which, encouragingly, for $a = 1$ collapses to the familiar $p_a = \frac{\mu_0 - 1}{\mu_0 + 1}$ (see (3.17)).

In order to verify empirically the distribution proposed, a number of runs of a GP system in Java was performed. A relatively large population of 100,000 individuals was used in order to reduce drift of average program size and to ensure that enough programs of each length class were available. The FULL initialisation method [Koza, 1992] was used. With this method, initial trees included $\mu_0 = d + 1$ primitives for $a = 1$ and $\mu_0 = \frac{1-a^{d+1}}{1-a}$ primitives for $a > 1$. Each run consisted of 500 generations.

Histograms were collected from the final generations (in order to ensure the effects of our chosen initialisation method have been washed-out[4]), each bin being the number of internal nodes contained within a program. The results were averaged over twenty runs. As we can see in Figures 3.1 to 3.3, there is strong empirical evidence to support our proposed distributions particularly for the first 100 classes where there is less statistical noise. It is also important to note that both the theoretical and experimental distributions show that there is a distinct bias to sample smaller programs a subject addressed in Chapters 4 and 5.

Finally, in Figure 3.4, we validate our distribution against the Linear-GP case where all internal nodes have an arity of 1. Our theoretical analysis in this section has shown that this is a special case as the value for p_a is simplified considerably. As we can see from both our theoretical and

[4]It would of course be interesting to see how may generations are required to negate length effects of chosen initialisation methods. This is discussed in Section 9.2.

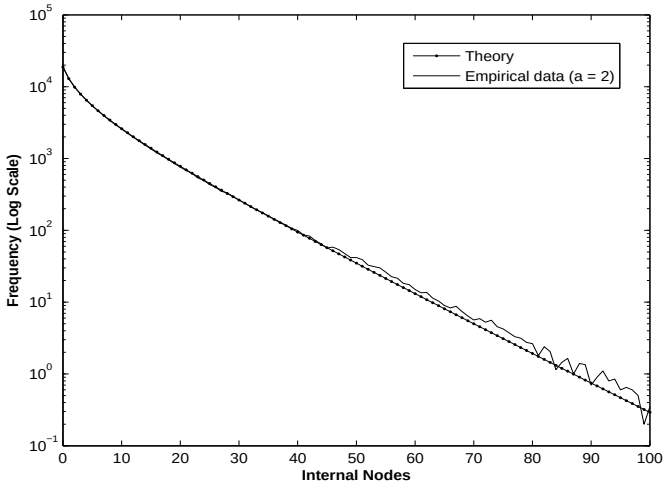


Figure 3.1: Comparison between theoretical and empirical internal node distributions for trees created with arity 2 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 15.00$, mean size after 500 generations $\mu_{500} = 14.19$). Population Size = 100,000.

empirical distributions the sampling peak has moved from very smallest number of internal nodes. The rapidly declining rate of the sampling of smaller programs is, however, evident after this point.

3.4 Generalisation of Program Length Distributions for Mixed Arity Trees

Before attempting to extend the theoretical analysis to mixed arity trees it is useful to consider if the a-ary tree equation in (3.22) could be generalised using standard mathematical techniques. The advantages of such a generalisation for branching processes are explained in [Haccou et al., 2005] but can be sumarised here as *the model may in addition to being easier to obtain, be simpler to understand, have fewer parameters and may also be computationally more tractable.*

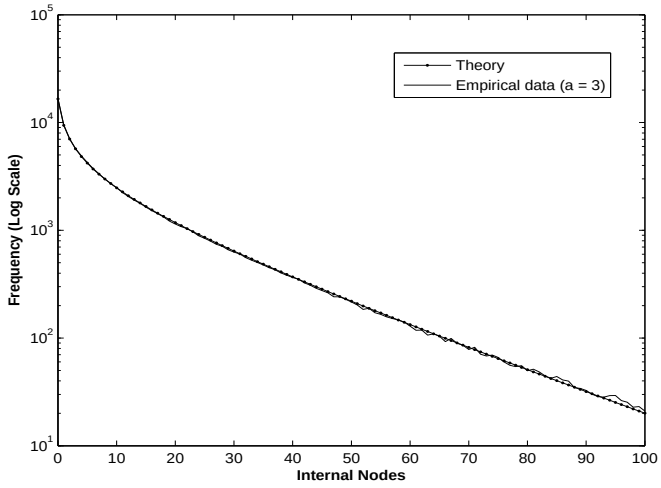


Figure 3.2: Comparison between theoretical and empirical internal node distributions for trees created with arity 3 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 40.00$, mean size after 500 generations $\mu_{500} = 39.13$). Population Size = 100,000.

In order to generalise, first, rather than viewing a as simply the identical arity of a particular set of functions, we can choose to use this as the expected number of children of a non-terminal picked from a function set, i.e. an average arity. An average a, which we will call $\bar{a}$ to avoid confusion, can be calculated for mixed function arities from experimental initialisation parameters as follows

$$\bar{a} = E(arity(F)) = \sum_{f} arity(f)P(F = f) \tag{3.24}$$

where f is a non-terminal to be used in a GP experiment, $arity(f)$ is a function returning the arity of the non-terminal f, and $P(F = f)$ is the probability that a particular non-terminal f will be selected as a non-terminal node by the tree inititialisation procedure. For traditional FULL and GROW initialisation methods non-terminals are chosen with equal prob-

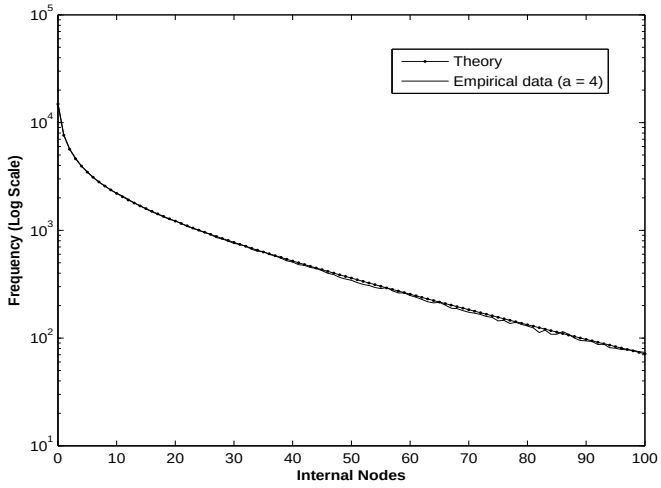


Figure 3.3: Comparison between theoretical and empirical internal node distributions for trees created with arity 4 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 85.00$, mean size after 500 generations $\mu_{500} = 79.99$). Population Size = 100,000.

ability [Luke, 2000]. Alternatively, $\bar{a}$ can be given simply by the calculation of non-terminal average arity from the initial population. For larger populations these will, of course, be almost identical.

With this new definition of a, we have the problem that the term $an+1$ in the binomial coefficient $\binom{an+1}{n}$ in (3.22) may be non-integer. So, the next step is to alter the definition of binomial coefficient so that it will also work with non-integer data values, as demanded by our new average arity version of a. This can be done simply by using the Gamma function as an alternative for the factorials used, i.e., $\Gamma(n+1) = n!$ [Ghahramani, 1996]. As a result we can rewrite the binomial coefficient as follows

$$\binom{n}{k} = \frac{n!}{(n-k)!k!} = \frac{\Gamma(n+1)}{\Gamma(n-k+1)\Gamma(k+1)} \tag{3.25}$$

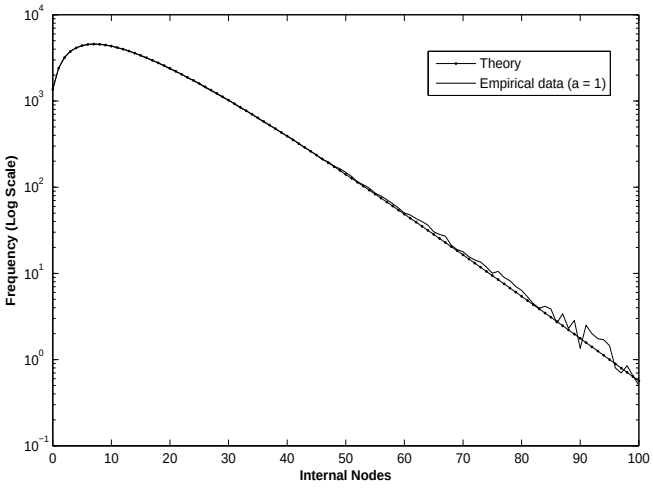


Figure 3.4: Comparison between theoretical and empirical internal node distributions for trees created with arity 1 functions and terminals only, initialised with FULL method (depth = 15, initial mean size $\mu_0 = 16.00$, mean size after 500 generations $\mu_{500} = 16.15$). Population Size = 100,000.

Therefore

$$\binom{an+1}{n} = \frac{\Gamma(an+2)}{\Gamma(an-n+2)\Gamma(n+1)} \tag{3.26}$$

which, substituted into (3.22), gives us the distribution

$$Pr\{n\} = (1-\bar{a}p_{\bar{a}})\frac{\Gamma(\bar{a}n+2)}{\Gamma((\bar{a}-1)n+2)\Gamma(n+1)}(1-p_{\bar{a}})^{(\bar{a}-1)n+1}p_{\bar{a}}^{n} \tag{3.27}$$

Using the same experimental set-up as Section 3.3[5] a comparison of theoretical predictions and observed results, averaged over twenty runs, is shown in Figures 3.5–3.7. Figure 3.5 is particularly interesting since it represents the behaviour of the primitive set for the Artificial Ant problem [Koza, 1992]. The other two figures represent hypothetical function sets where an even spread of primitive arities are present. In all cases the match between predictions obtained from Equation 3.27 and empirical data is striking (note,

[5]The FULL initialisation method is again used. In this case functions are chosen with uniform probabililty.

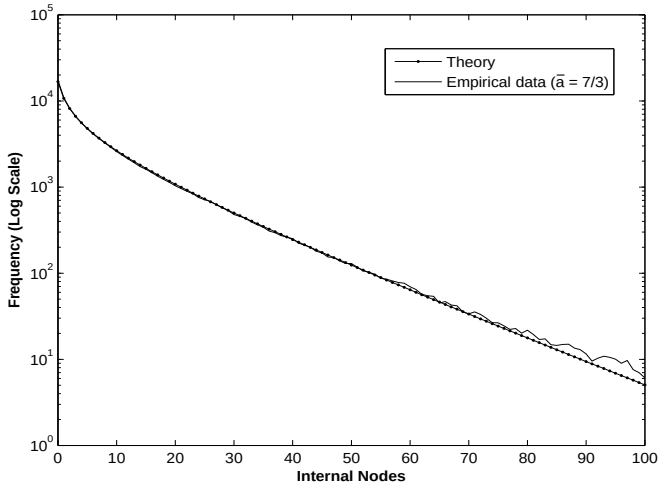


Figure 3.5: Comparison between theoretical and empirical internal node distributions for trees created with arity 2, 2 and 3 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 21.48$, mean size after 500 generations $\mu_{500} = 23.51$). Population Size = 100,000.

that sampling noise is more marked in the longer-length classes because there are fewer programs in each).

As we can see from both our predicted and observed results there is a distinct bias towards the sampling of smaller programs as was found for a-ary trees. The use of mixed arity function sets does not alter this bias.

3.5 Crossover with 90% Function / 10% Terminal Crossover-Point Selection Policy

The theoretical results on the evolution of size during GP runs developed in [Poli and McPhee, 2001, Poli and McPhee, 2003b, Poli et al., 2007] and in the previous sections assume that crossover points are selected uniformly at random out of the set of all primitives in a tree (including terminals). This assumption simplifies the, already complex, mathematics required to

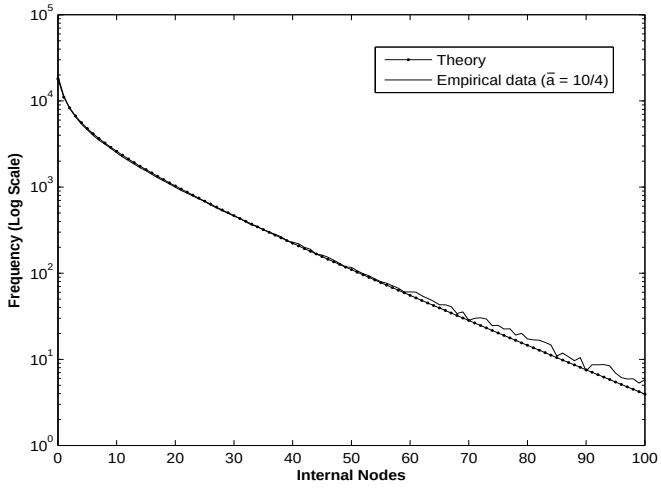


Figure 3.6: Comparison between theoretical and empirical internal node distributions for trees created with arity 1, 2, 3 and 4 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 25.38$, mean size after 500 generations $\mu_{500} = 23.72$). Population Size = 100,000.

obtain results in this area. However, GP researchers and practitioners commonly use a 90%-function/10%-terminal crossover-point selection policy [Koza, 1992, pg114]. We will call this policy 90/10 for brevity. It is, therefore, interesting to investigate to which extent the theory for uniform crossover point selection applies to this, non-uniform case. In this section, we present an analysis of this issue.

The starting point for our analyis is that in the 90/10 policy crossover points are still uniformly distributed within each class of nodes (internal vs. terminals). So, 90/10 crossover may still have the symmetries required to model the tree population using a branching process as was done in Section 3.3. This, in turn, suggests that some form of modified Lagrange distribution of the second kind might still be applicable to describe the limiting distribution of tree sizes for 90/10 crossover.

We are not in a position to say what the exact size distribution under

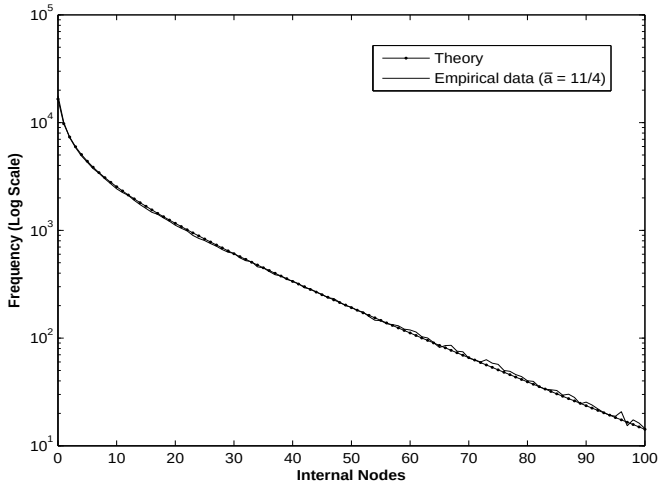


Figure 3.7: Comparison between theoretical and empirical internal node distributions for trees created with arity 1, 3, 3 and 4 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 32.12$, mean size after 500 generations $\mu_{500} = 33.29$). Population Size = 100,000.

90/10 crossover would be. However, we note that the 90/10 policy has a considerable effect on the proportion of programs of size 1, (i.e., with no internal nodes). This is because such programs are only created if the crossover point in the root donating parent is the root node itself and the crossover point in the subtree donating parent is a terminal. In the 90/10 policy the probability of a leaf node is artificially set to 10%, which, with typical primitive sets, is notably smaller than with uniform selection of crossover points. Naturally, in the same conditions, the probability of the root node being chosen grows, but not enough to compensate for the drop the probability of selecting terminals. So, their product – the probability of creating size 1 programs – is much smaller than in the uniform case. Naturally, if there is a drop in the frequency of size 1 programs, there must be a corresponding increase in the other length classes.

On the basis of these observations, it is clear that the distribution of tree

sizes under 90/10 crossover cannot be a pure Lagrange distribution of the second kind. However, we might expect to see a "Lagrange-like" distribution for programs with one or more internal nodes, i.e, with a size greater than one. We conjecture that the following family of distributions may provide a reasonable first order approximation of the true limiting distribution of sizes for sub-tree swapping crossover with a 90/10 node selection policy:

$$\Pr_{90/10}\{n\} = \begin{cases} \alpha & \text{if } n = 0, \\ (1-\alpha)\frac{\Pr\{n,\bar{a},p_{\bar{a}}\}}{1-\Pr\{0,\bar{a},p_{\bar{a}}\}} & \text{otherwise,} \end{cases} \tag{3.28}$$

where α is a constant and $\Pr\{n, \bar{a}, p_{\bar{a}}\}$ stands for the extension to the Lagrange distribution of the second kind in Equation (3.27). In this formula the denominator $1-\Pr\{0, \bar{a}, p_{\bar{a}}\}$ has the effect of normalising the numerator $\Pr\{n, \bar{a}, p_{\bar{a}}\}$ in such a way to make it a probability distribution for $n \geq 1$.[6] That is, $\sum_{n\geq 1}\left(\frac{\Pr\{n,\bar{a},p_{\bar{a}}\}}{1-\Pr\{0,\bar{a},p_{\bar{a}}\}}\right) = 1$. Then the multiplication by $(1-\alpha)$ ensures that $\Pr_{90/10}\{n\}$ is a probability distribution for any value of $\alpha \in [0, 1]$, $\bar{a} > 0$ and $p_{\bar{a}} \in [0, 1/\bar{a}]$.

To corroborate this conjecture we performed GP runs with the same configuration as in Section 3.3 except that this time we used 90/10 crossover. Figure 3.8 shows a comparison between empirical size distributions observed at generation 500 in our runs and corresponding $\Pr_{90/10}\{n\}$ modified Lagrange distributions for the case of trees made up with primitives of average arity $\bar{a} = 1.5$, 2 and 2.5 (see caption of Figure 3.8 for additional information). The theoretical models were obtained by setting α equal to the number of programs with no internal nodes and finding the value of $p_{\bar{a}}$ which minimised the mean square error between empirical data and Equation (3.28). Naturally, our choice of α guarantees a perfect fit for $n = 0$. Its value,

[6] Naturally $\Pr\{n, \bar{a}, p_{\bar{a}}\}$ is already a probability distribution for $n \geq 0$.

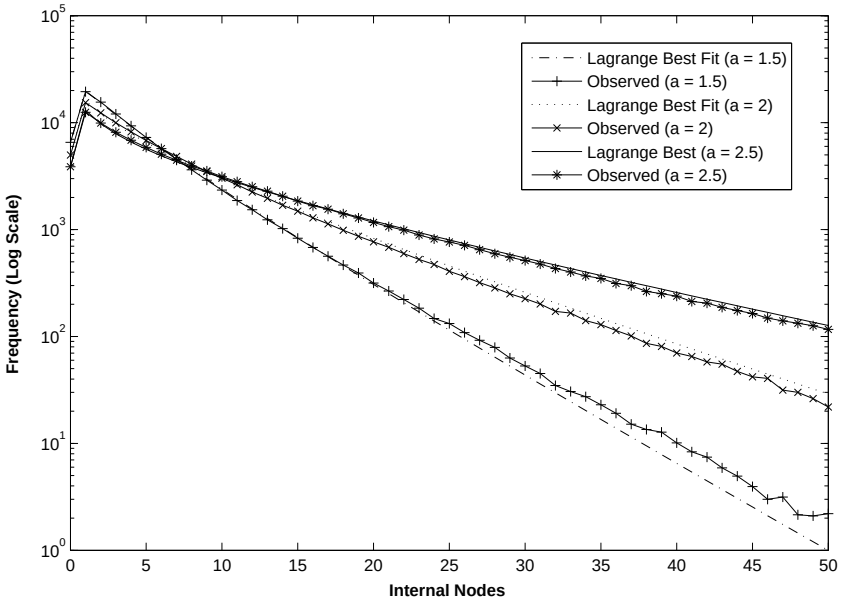


Figure 3.8: Comparison between empirical size distributions and modified Lagrange size distributions obtained by best fit for trees made up with primitives of average arity 1.5, 2 and 2.5 initialised with FULL method (depth = 3, initial mean sizes $\mu_0 = 8.12$, $\mu_0 = 15.0$ and $\mu_0 = 25.37$, respectively) and manipulated by subtree crossover with 90%/10% node selection policy (mean sizes after 500 generations $\mu_{500} = 8.06$, $\mu_{500} = 14.86$ and $\mu_{500} = 26.68$, respectively).

however, influences the scaling of the whole distribution for $n > 0$. It is, then, remarkable to see that such choice allows a very accurate match between our conjectured theoretical distribution and the distribution observed in real runs.

These results suggest that many of the implications and applications of the size bias of sub-tree swapping crossover with uniform selection of crossover points discussed in the following chapters carry over to the case of 90/10 crossover.

3.6 Complete Program Length Distributions

In this section, our objective is to extend equations (3.22) and (3.27) to allow us to predict program length distributions that include terminals as well as non-terminals. As a first step in this direction, let us look at what we can say for certain regarding the relationship between number of internal nodes and program length. First, we know for a-ary trees where the arities of all nodes in the tree are the same, the length, ℓ, of a program can be expressed exactly in terms of the number of its internal nodes, n, using the following equation:

$$\ell = a \times n + 1, \tag{3.29}$$

where a is the (fixed) arity of the internal nodes. Therefore, rearranging Equation (3.29) to obtain internal nodes in terms of length, i.e.,

$$n = \frac{\ell - 1}{a}, \tag{3.30}$$

and substituting this into Equation (3.22), we obtain that, for a-ary trees,

$$\mathrm{Pr}_l\{\ell\} = \begin{cases} \mathrm{Pr}\{\frac{\ell-1}{a}\} & \text{if } \ell \text{ is a valid length (i.e., } \frac{\ell-1}{a} \text{ is a non-negative integer),} \\ 0 & \text{otherwise,} \end{cases} \tag{3.31}$$

where $\mathrm{Pr}_l\{\ell\}$ is the limiting distribution of program lengths. This distribution applies, for example, to Boolean function induction problems where often all functions are binary and symbolic regression problems where often only the standard four arithmetic operations are used.

Figure 3.9 shows an observed plotted length distribution for 2-ary trees, with invalid (even) lengths removed, compared to that predicted by Pr_l. The observed values are averages over twenty independent runs with populations

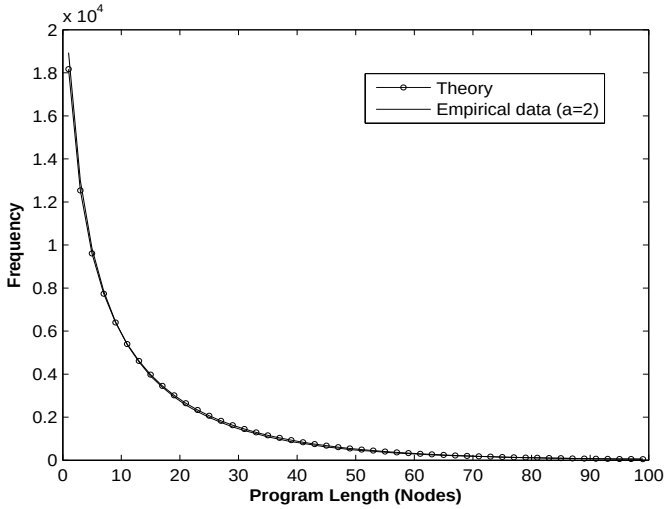


Figure 3.9: Comparison between theoretical and empirical program length distributions for 2-ary trees initialised with FULL method (depth = 3, initial mean size $\mu_0 = 15.0$, mean size after 500 generations $\mu_{500} = 14.19$). Invalid even lengths are ignored.

of 100,000 individuals run for 500 generations.[7] As we can see there is a very close fit between the two curves.

Our next step is to extend the generalised formula for mixed-arity trees (Equation (3.27)) so as to predict length distributions rather than internal node distributions. We know that for a program length of 1, a single terminal, there will always be 0 internal nodes. Therefore, the predicting single node programs is a simple one-to-one mapping with the generalised formula for 0 internal nodes. However, other lengths can be obtained by different combinations of internal nodes of different arities. For example, one can obtain programs of length 3 by using one internal node of arity 2 or two internal nodes of arity 1.

As a first approximation, we will assume that we can still estimate the expected number of internal nodes in a tree of length ℓ by applying Equa-

[7]These and all other experimental parameters were chosen as in Sections 3.3 and 3.4 for ease of comparison.

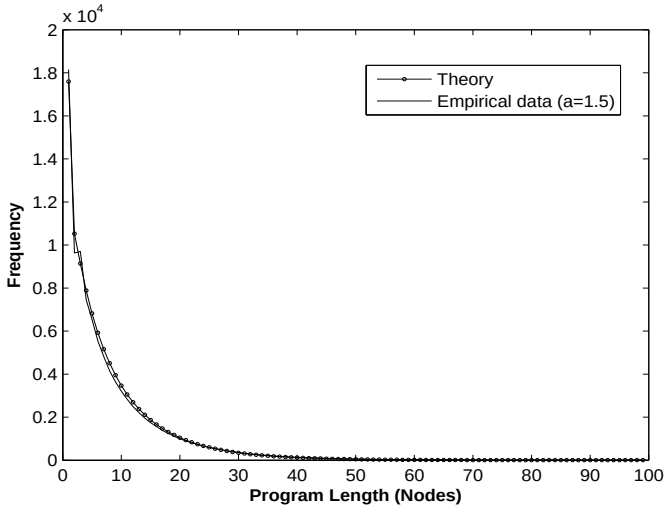


Figure 3.10: Comparison between theoretical and empirical program length distributions for trees created with arity 1 and 2 functions initialised with FULL method (depth = 3, initial mean size $\mu_0 = 8.13$, mean size after 500 generations $\mu_{500} = 8.51$). All lengths are valid.

tion (3.30), simply using $\bar{a}$ instead of a. We can then substitute the result into Equation (3.27) to obtain the approximate distribution of lengths we are looking for. Naturally, between the variable ℓ and the variable n there is a difference in scale (the factor $\bar{a}$). So, we will need to normalise the values produced by Equation (3.27) to ensure the new distribution sums to 1.

Putting all of this together, we obtain an approximate model of the limiting distribution of program lengths in the case of primitive sets of mixed arities. Namely:

$$\Pr_v\{\ell\} = \begin{cases} \Pr_g\{0\} & \text{if } \ell = 1, \\ \frac{\Pr_g\{\frac{\ell-1}{\bar{a}}\}}{\bar{a}} & \text{if } \ell \text{ is a valid length greater than 1.} \end{cases} \tag{3.32}$$

Note, we do not require $\frac{\ell-1}{\bar{a}}$ to be an integer.

Since there were approximations in the original derivation of Equa-

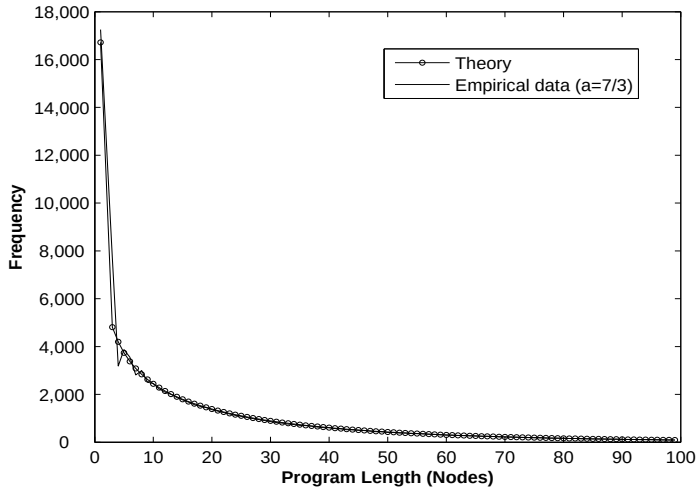


Figure 3.11: Comparison between theoretical and empirical program length distributions for trees created with arities 2, 2 and 3 functions initialised with FULL method (depth = 3, $\mu_0 = 21.48$, $\mu_{500} = 23.51$). Invalid length 2 is ignored.

tion (3.27), and we added further approximations in the derivation of Equation (3.32), one might wonder whether the model is sufficiently accurate to be of practical use. Figure 3.10 shows observed and theoretical values of the limiting length distribution experiment set up for internal nodes of arities of 1 and 2 where all lengths are valid, whilst Figure 3.11 compares the theoretical and empirical distribution obtained in a GP run with the primitive set of the Artificial Ant problem, which has internal nodes arities of 2, 2 and 3, for IF-FOOD-AHEAD, PROGN2 and PROGN3, respectively. Note, with this choice there is no way of generating programs of length $\ell = 2$. Finally, Figure 3.12 shows the results of using arities of 1, 2, 3 and 4. Note that in order to highlight the fit for larger and less common programs we used a log scale for frequency.

As one can see, Equation (3.32) accurately models the distributions observed in real runs in all cases, with only minor deviations at the very short

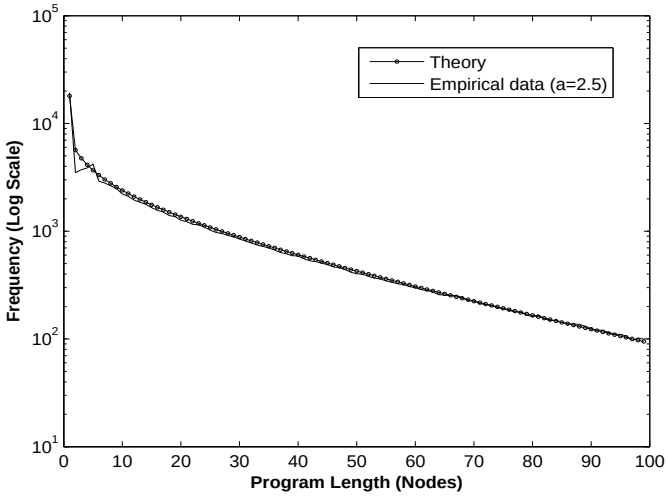


Figure 3.12: Comparison between theoretical and empirical program length distributions for trees created with arities arities 1, 2, 3 and 4 functions initialised with FULL method (depth = 3, $\mu_0 = 25.38$, $\mu_{500} = 23.72$). All lengths are valid.

program lengths where some of the assumptions behind the model are less applicable.[8] However, generally both the model and the actual runs show that in almost all cases sub-tree swapping crossover will sample with a higher frequency smaller programs.

3.7 Conclusions

In this chapter we have provided an exact model for the limiting distribution of a-ary tree sizes for sub-tree swapping crossover, with uniform selection of crossover points, on a flat fitness landscape. Then we have generalised this model to predict lengths for mixed arity populations. The generalised model was then used to approximate the size distribution obtained with sub-tree swapping crossover with a 90% function /10% terminal node selection

[8] Curing the slight mismatches for earlier lengths would require a more accurate estimation of number of internal nodes of each arity for small ℓ. We investigate more precise models in Chapter 8.

policy. Finally, we turned our attention to complete programs, i.e., with terminals, the previous models being expressed in terms of the number of internal nodes, and found that an accurate model of program lengths could be produced for a-ary trees while a strong approximation can be created for mixed arity trees.

All models suggest that this form of crossover has a distinct bias to sample smaller programs. This bias has a number of significant effects regarding GP search which are investigated in the following chapters.

Chapter 4

Program Sampling, Initialisation and Bloat

4.1 Introduction

Now that we have a number of formulae to predict our distributions of program lengths under sub-tree swapping crossover, the next step is to see how we can apply this to practical GP applications. In Sections 4.2 and 4.3 we analyse how crossover will sample the program search space and the importance of average program size. We then (Section 4.4) look at how initialisation can combine with crossover to affect both fitness and program size during a GP run. In light of this, we provide in Section 4.5 a new bloat theory, *Crossover-Bias*, that explains how selection and sub-tree swapping crossover combine in a process that will progressively sample larger and larger programs. Finally, in Section 4.6 we study empirically the effects of applying size limits and find that such limits can speed growth towards the limit.

4.2 Sampling of Unique Programs

As was done for Linear-GP in [Poli and McPhee, 2001, Poli et al., 2002], we can now compute the expected resampling probability for unique programs of different sizes. In particular, let us imagine that our GP system operating on a flat landscape is at the fixed point distribution and let $\mathcal{F}$ and $\mathcal{T}$ be the sizes of the function and terminal sets, respectively. Let us further assume that all functions in $\mathcal{F}$ are of arity a. Since, there are $\mathcal{F}^n\mathcal{T}^{(a-1)n+1}$ different a-ary programs with n internal nodes in the search space, it is possible to compute, using Equation (3.22), the average probability $p_{\text{sample}}(n)$ that each of these will be sampled by sub-tree swapping crossover, namely

$$p_{\text{sample}}(n) = \frac{(1-ap_a)}{\mathcal{F}^n\mathcal{T}^{(a-1)n+1}}\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\,p_a^n. \qquad (4.1)$$

It is easy to study this function and to conclude that, for a flat landscape, using this form of crossover GP will sample a particular short program much more often than it will sample a particular long one. For example, when $\mu_0 = 15$ as in Figure 3.1, GP will heavily resample short programs. Assuming, we have a boolean problem with five functions and four terminals[1] the same program of length 1 is resampled on average every 22 crossovers, every 638 crossovers for programs of length 3, and every 16,653 crossovers for length 5. As program size grows the sampling probability drops dramatically. For example, the resampling rate for unique programs of length 15 is 1 in over 128 billion (see Table 4.1 for details). It is important to consider, therefore, that not only is the practitioner faced with an expected combinatorial explosion of unique programs as length increases but this inability for any operator to sample particular larger unique programs is exasperated by

[1]For example a 4 Bit Even Parity Problem with a function set consisting of AND, OR, NAND, NOR and XOR.

Table 4.1: Unique program (UP) sampling probabilities, by program length, for sub-tree swapping crossover applied to a flat fitness landscape as predicted by Equation (4.1). A problem with five boolean functions and four terminals has been used as an illustration. $\mu_0 = 15$, internal nodes are calculated using Equation (3.30).

Length	Length Probability	UP Count	UP Sampling Probability
1	0.18169764	4	4.5424E-02
3	0.12534415	80	1.5668E-03
5	0.09607632	1,600	6.0048E-05
7	0.07732465	32,000	2.4164E-06
9	0.06401091	640,000	1.0002E-07
11	0.05397082	12,800,000	4.2165E-09
13	0.04609649	256,000,000	1.8006E-10
15	0.03974959	5,120,000,000	7.7636E-12

sub-tree swapping crossover's bias to sample smaller programs. In the next section we look at altering experimental parameters to help reduce this bias.

4.3 Sampling of Program Size

With some problems we may have an initial idea of likely program lengths that may be required to provide an acceptable solution. For example, this knowledge may range from knowing that a minimum length is required in that enumerated search has been unsuccessful up to a length when the search was found to be intractable, or simply that previous attempts using specific designs or other search algorithms had provided some initial success at certain solution lengths. Also, for classical test problems, a great deal of information is available about the distribution of program functionality (including fitness) as the length of programs is varied (see, for example, [Langdon and Poli, 2002]). So, it is possible to infer from such distributions what are the best length classes on which to concentrate the search for solutions. Our first step is to see how we can use the parameters of the limiting distribution of sub-tree swapping crossover to at least begin to sample a

significant number of programs of a particular size.

To begin, the a parameter and its mixed arity equivalent, $\bar{a}$, are derived from our function set which we assume is fixed for the experiment, i.e., that all functions are deemed likely to be required to solve the problem. We could of course alter our initialisation procedure to sample more of a certain function, hence, altering our average arity. However, such an approach is limited by the functions available. We could only skew initialisation towards sampling arity 2 or 3 functions for the Artificial Ant problem, for example. Little could be done with the Parity problem we have presented in Table 4.1 as all functions have an arity of 2. With this in mind another alternative could be to alter the functions so they have larger arities but ignore certain inputs. This would, however, create large parts of the parse tree that would be ignored, the affects of such a measure on a GP run would need to be analysed separately.

We are, therefore, left with our values for p_a and $p_{\bar{a}}$ which are in turn derived from a and $\bar{a}$ respectively, which we do not want to change, and the mean program size of the initial generation μ_0. This latter value can be directly altered by manipulating the parameters of our initialisation method in order that significantly large trees are produced.

To illustrate this if we start with the Artificial Ant problem, we have three functions: IF-FOOD-AHEAD, PROGN2, PROGN3 with arities of 2, 2, and 3 respectively. As seen in Figure 3.5 our value of $\bar{a}$ for this group of arities is 7/3. Knowing this we can alter the value of μ_0 to enable the sampling by crossover of different program length spaces. Figures 4.1 and 4.2 show how varying μ_0 alters the crossover length distribution. We can see that there is always a consistent higher sampling of smaller programs. However, starting from a reasonable base figure relatively small increases in

μ_0 allow larger programs to be sampled more consistently.

As an illustration, the Artificial Ant problem is known to have no ideal solutions before a program size of 11 [Langdon and Poli, 2002], if we were to ensure that a percentage of programs sampled were to have at least 5 internal nodes[2] for a μ_0 value of 5, 10, and 20 crossover would sample 16%, 36%, and 54% of the population respectively for that program size or greater.

Thus, if we initialised the population so that the length distribution is a Lagrange distribution of the second kind (as we will discuss in the following section), we could perform an informed choice of what is the best μ_0 to use to maximise the chance of solving a problem. Naturally, if, instead, we initialised the population with a distribution that is very different from a Lagrange distribution of the second kind, as is the case, for example, for FULL or RAMPED initialisation, then it would take a number of generations before sub-tree swapping crossover transforms the size distribution into something resembling such a distribution. However, at that point, a good choice of μ_0 would start paying off, i.e., effectively there is a race. If one can find a solution before crossover reshapes the distribution, then the limit distribution may not matter so much. For example, if one initialised all the individuals in the population at a reasonable program length, and then chose a very large population, there is a chance that a solution might be found very quickly, i.e., before crossover shifts the mode of the distribution away from the optimal. However, whenever runs take tens of generations, then it might make sense to initialise so that the peak of the initial distribution is beyond the optimal length, so that when crossover reshapes the length distribution, the peak will do a scan of the best area.

[2]This estimate is derived from $length = \bar{a}n + 1$.

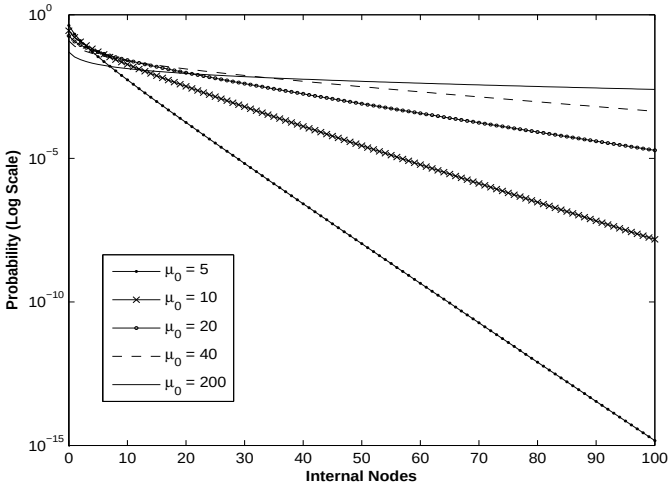


Figure 4.1: Comparison of probability distributions, derived from Equation (3.27), for different μ_0 values ($\bar{a}$ is fixed at 7/3) for the Artificial Ant problem. Note: the use of a logarithmic scale for the probability axis.

4.4 Initialisation and Crossover

By choosing to apply GP to a particular problem we have assumed that both fitness based selection and crossover are likely to provide an efficient search method to provide a solution. Normally, a GP initialisation method (e.g. GROW, FULL, RAMPED, etc) takes no account of the eventual distributions 'desired' by either crossover or selection (or any other GP operator such as mutation). Naturally, eventual fitness values are not known in advance of a GP experiment. However, we now have evidence that sub-tree swapping crossover will, with repeated application, distribute the population according to a predictable distribution. Our next step is to investigate the possible benefits or disadvantages of initialising a population by length to take account of crossover.

We could of course write an algorithm to initialise the population according to the eventual predicted distribution desired by crossover. The most

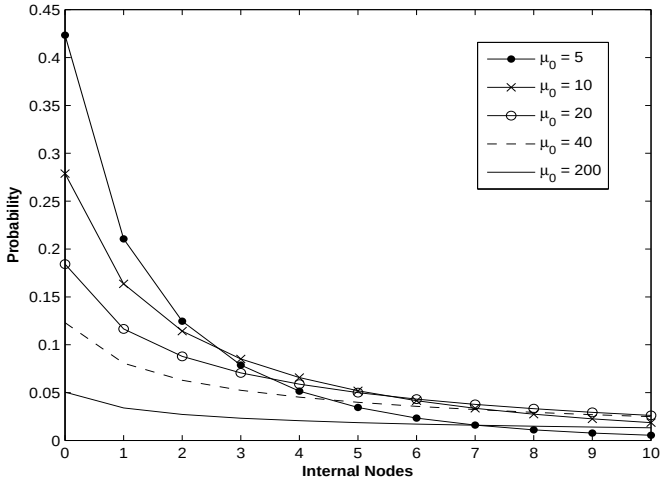


Figure 4.2: Comparison of probability distributions, derived from Equation (3.27), for different μ_0 values ($\bar{a}$ is fixed at 7/3) for the Artificial Ant problem. Early internal node counts are shown.

straightforward programatical method, however, is to simply run some prior generations of a GP experiment without fitness, thereby allowing crossover to distribute program lengths without having to endure the cost of fitness calculations.

To test this idea we took two out-of-the-box problems from the ECJ evolutionary toolkit [Luke, 2008], the Artificial Ant and 4 Bit Even Parity as previously discussed, making adjustments to remove mutation, to ensure uniform selection of crossover points, and to prevent a depth limit being applied. A population size of 1024 individuals was used and the results were averaged over a hundred runs. All experiments were initialised using the FULL method with a depth of 3. Each experiment looked at the effect of running GP with zero, twenty or fifty initial generations where a constant fitness value was applied before allowing the experiment to continue as normal. Naturally, the flat-fitness phase was much faster than normal, since no fitness evaluation was required. During this phase crossover was free to

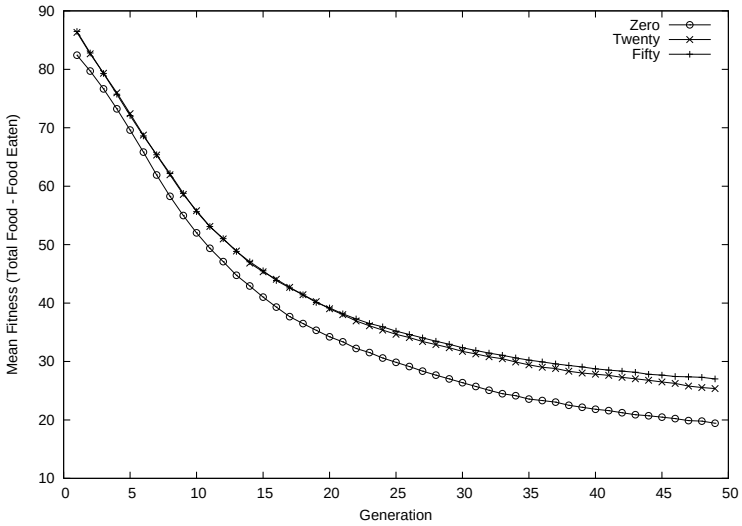


Figure 4.3: Comparison of mean fitness values for populations with zero, twenty and fifty prior generations of crossover without fitness for the Artificial Ant problem. Note, values for prior generations where the fitness function returns a constant value are not shown.

distribute the population towards its limit size distribution.

As we can see in Figures 4.3 and 4.4, there is noticeable degrading in mean fitness for the populations with initial crossover-only generations compared to those where fitness is applied straight away. This is also seen in Figures 4.5 and 4.6 where best fitness has been recorded. If we look at Figures 4.7 and 4.8 we can see there is much greater variation in individual fitness for the fitter populations.

The reason for this effect is in the sampling of smaller programs produced by 'Lagrange-like' initialisation. A population distributed by length through the application of crossover will contain large numbers of relatively small programs. In both the Artificial Ant and Parity problems these short programs are associated with low fitness [Langdon and Poli, 2002]. Crossover has, therefore, created a large proportion of smaller programs with relatively poor fitness values, whilst FULL originally produced programs above

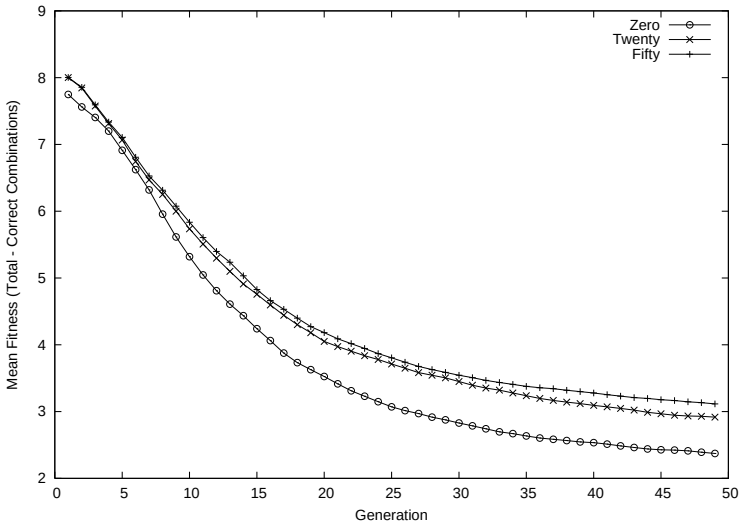


Figure 4.4: Comparison of mean fitness values for populations with zero, twenty and fifty prior generations of crossover without fitness for the 4 Bit Even Parity problem. Note, values for prior generations where the fitness function returns a constant value are not shown.

a reasonable threshold.[3]

We can also apply these findings to the problem of understanding the origins of bloat, as we discuss in the next section.

4.5 Crossover-Bias Bloat Theory

As discussed in Chapter 2, bloat, the growth of program size during a GP run without a significant return in terms of program fitness, is seen in many GP experiments. Figures 4.9 and 4.10 show graphs of program growth for the two problems described in the previous section. As we can see there is a noticeable increase in program growth for the populations with 'Lagrange-like' initialisation, the same populations that have a relatively lower rate of program fitness improvement.

[3]It should be noted, however, that in problems where initial relatively high fitness is associated with very small programs, the opposite may be true. This is investigated further in Chapter 6.

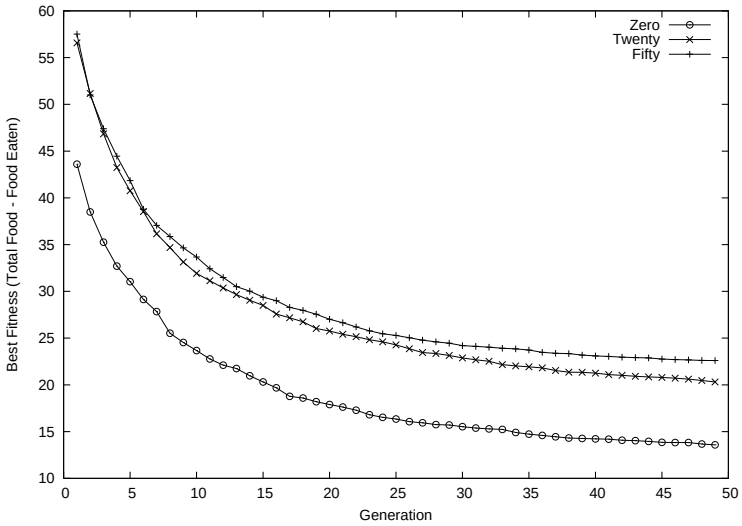


Figure 4.5: Comparison best fitness values for populations with zero, twenty and fifty prior generations of crossover without fitness for the Artificial Ant problem.

At first one might find this result very surprising. How is it that initialisation has such a big effect on bloat, which is typically associated with later generations of a GP run, when effectively the population starts stagnating? An explanation for this lies in the combination of the crossover sampling distribution and fitness. This process can be described simply as:

I In each generation selection will populate the mating pool with relatively fit programs

II The sub-tree swapping crossover operator will then produce children with a length distribution biased towards smaller programs irrespective of their fitness

III If smaller programs cannot obtain a relatively high fitness they will be ignored by selection in the next generation

IV Hence, average program size will increase as ever larger programs are

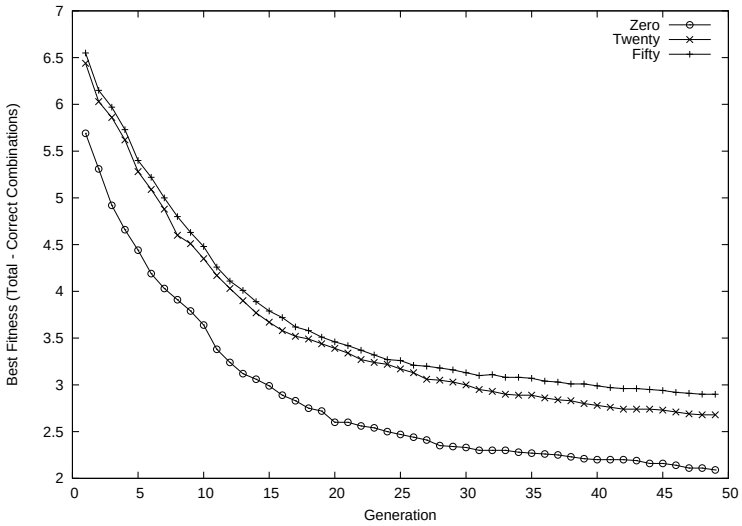


Figure 4.6: Comparison best fitness values for populations with zero, twenty and fifty prior generations of crossover without fitness for the 4 Bit Even Parity problem.

placed into the mating pool

It is important to note that there is no change in the average size of programs found in the mating pool from those produced in the resulting child population, i.e., the next generation. However, the distribution has a sampling bias towards smaller programs, with relatively few larger programs.

Although it is unlikely that within a single generation our population will reach the limiting length distributions described in chapter 3, we know that crossover will begin to distribute, in terms of length, the programs that have been presented to it (i.e., in the mating pool) in a similar fashion.

This explanation fits completely within the mathematical analysis of the dynamics of mean program size provided in [Poli, 2003], where the following

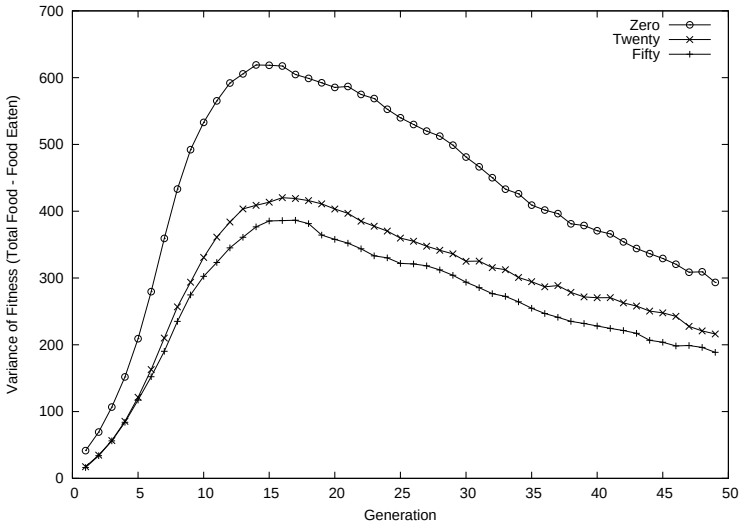


Figure 4.7: Comparison fitness variance values for populations with zero, twenty and fifty prior generations of crossover without fitness for the Artificial Ant problem

size evolution equation was derived[4]

$$E[\mu(t+1) - \mu(t)] = \sum_{l} N(G_l)(p(G_l, t) - \Phi(G_l, t)) \tag{4.2}$$

where E is the expectation operator, $\mu(t)$ is mean program size at generation t, G_l represents all programs of a particular shape, $N(G_l)$ represents the size of programs of shape G_l, $p(G_l, t)$ represents the selection probability for programs of shape G_l and $\Phi(G_l, t)$ represents their frequency in the population. As length classes are a super-set of shape classes we can rewrite Equation (4.2) as:

$$E[\mu(t+1) - \mu(t)] = \sum_{l} l \times (p(l, t) - \Phi(l, t)) \tag{4.3}$$

[4]A major finding of this paper is that for symmetric (where probability of node selection points is independent of order of parent presentation) sub-tree swapping crossover operators, "The mean program size evolves as if selection only was acting on the population". The derivation of this equation is explained in Section 5.4 of [Poli, 2003].

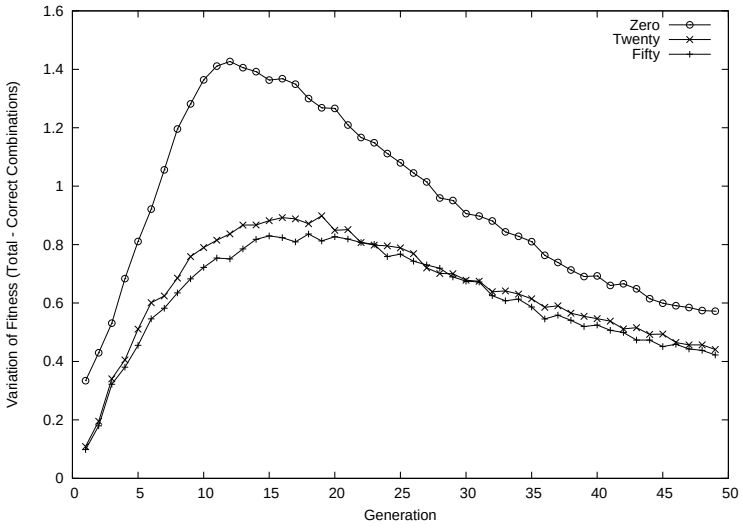


Figure 4.8: Comparison fitness variance values for populations with zero, twenty and fifty prior generations of crossover without fitness for the 4 Bit Even Parity problem

where $p(l,t)$ is the selection probability for programs of length l and $\phi(l,t)$ is their current proportion.

If, as is the case for the Artificial Ant and Parity problems, the selection probability for short programs is consistently lower than their frequency, then, everything else being equal, one must expect $E[\mu(t+1) - \mu(t)] > 0$, and hence bloat will occur.

For problems where initially relatively high fitness can be obtained with small programs, e.g., some symbolic regression problems, it will take a number of generations for fitter larger programs to be produced (see Chapter 6). However, after this point bloat will then start to occur.

Recent work [Poli et al., 2008b] has analysed the Crossover-Bias theory presented here further and found that population size can have a significant effect with regard to code growth, particularly that individuals in smaller populations will grow at a slower rate than those in larger ones. The reason for this is that populations need to be significantly large for smaller programs

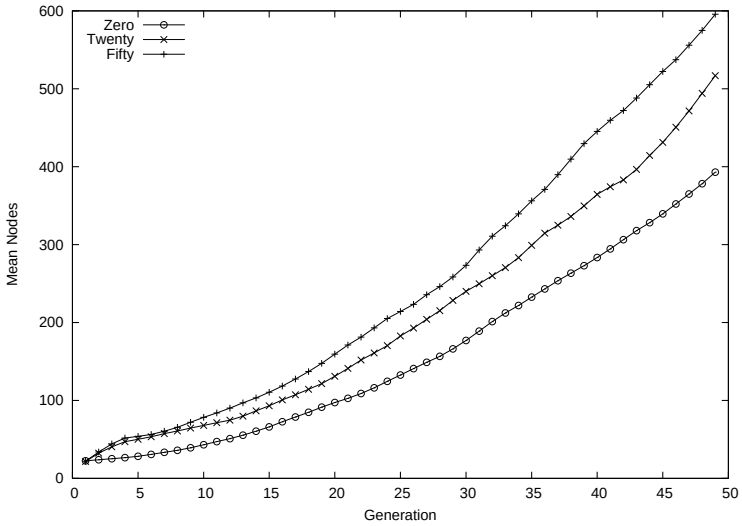


Figure 4.9: Comparison mean number of nodes for populations with zero, twenty and fifty prior generations of crossover without fitness for the Artificial Ant problem

to be sampled consistently. The authors have termed this work a *Refined Crossover-Bias Theory*.

4.6 Effects of Size Limits

The standard technique to control bloat, namely the application of a depth or length limit, is known to have significant effects on GP dynamics (see, for example, [Crane and McPhee, 2005]). Unfortunately, we don't have a mathematical model for the limit distribution of sizes (neither in terms of internal nodes nor in terms of lengths) in the presence of length limits. However, we can conduct experimentation to study their effects on such a distribution. Figures 4.11 and 4.12 show the affect of applying length limits of 50, 75 and 100 nodes, using ECJ, set-up as in Section 4.4[5] to the Artificial Ant and 4 Bit Even Parity problems. The effect of the length limit is that

[5]Similar results were also reported for a different set of experimental parameters in [Dignum and Poli, 2008] which have been modified here to ensure consistency.

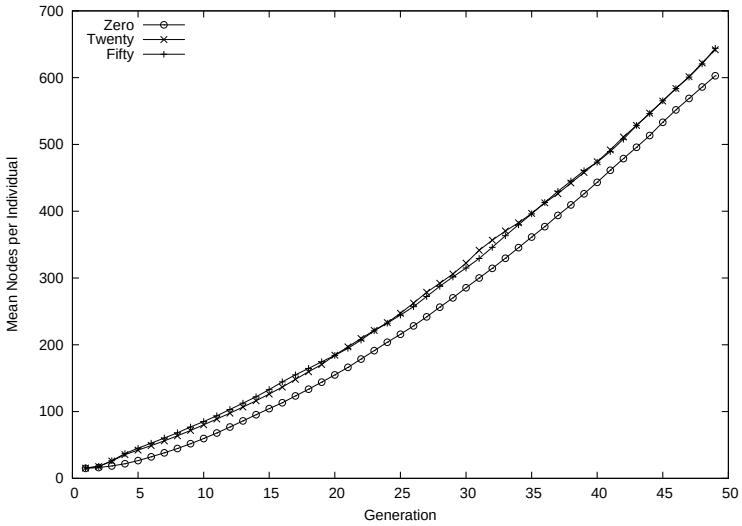


Figure 4.10: Comparison mean number of nodes for populations with zero, twenty and fifty prior generations of crossover without fitness for the 4 Bit Even Parity problem

programs become more frequent in the smaller length classes.

In the presence of fitness, this effect can be counteracted but not cancelled by selection. So, one should expect more sampling and resampling of short programs. However, following the line of reasoning of the *Crossover-Bias* bloat theory (see Section 4.5) we know that for most problems these programs cannot be solutions, and in fact are typically very unfit, and, so, longer programs will even more be preferentially selected, leading to more bloat. Thus, *size limits effectively increase the tendency to bloat since they induce more sampling of short programs, and, so, in the presence of non-flat fitness landscapes, GP populations rush towards the limit even more quickly than in the absence of the size limit!* This effect is particularly clear if one looks at the peak (i.e., the mode) of the program length distribution with and without length limits. Figures 4.13 and 4.14 show how the mode (averaged over 100 independent runs) changes generation by generation for different limits in the case of the Artificial Ant and 4 Bit Even Parity prob-

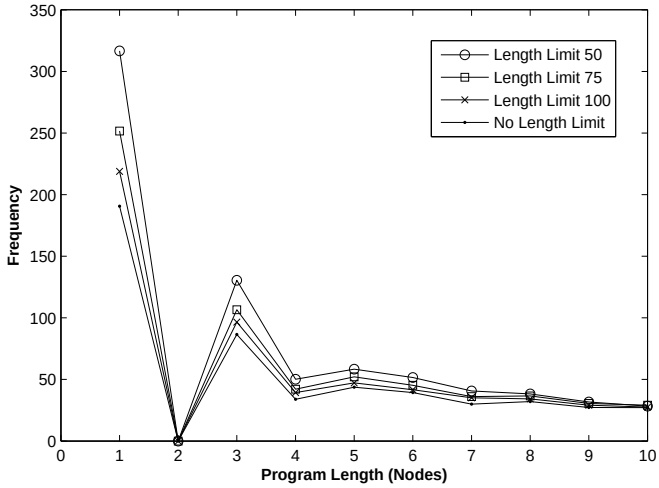


Figure 4.11: Comparison of sampling frequencies, at generation 100, associated with length limits for the Artificial Ant Problem applied to a flat fitness landscape.

lems (this time with selection). We can see that smaller size limits encourage GP to sample larger programs in the early generations before the size limit is reached.

These results suggest that, if size limits are imposed to combat bloat, then these should not be applied from generation 1, but much later and on demand, for example, if the average program size exceeds some prefixed threshold. This would avoid speeding up program growth in the early generations of a run.

Naturally, virtually all methods to combat bloat give more selective preference to shorter program than to longer ones. If in so doing they cause an oversampling of the short programs w.r.t. the base case (i.e., in the absence of the anti-bloat method). Therefore, anti-bloat methods may change the program size distribution in a somehow similar way to size limits. As a result, we should expect this phenomenon to still take place also with other bloat-control mechanisms, although perhaps with a lesser degree.

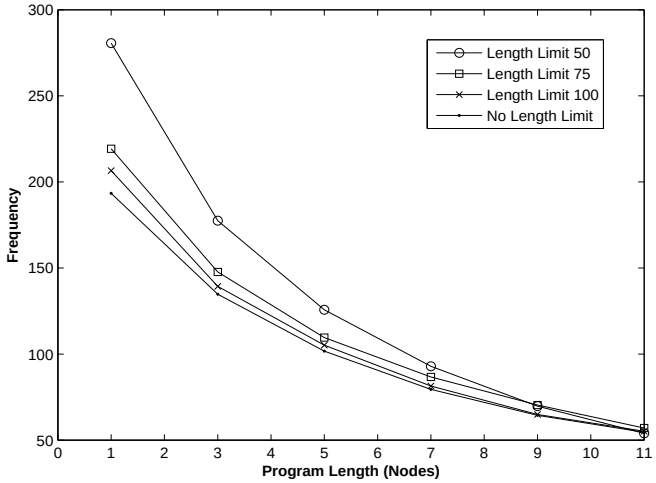


Figure 4.12: Comparison of sampling frequencies, at generation 100, associated with length limits for the 4 Bit Even Parity Problem applied to a flat fitness landscape.

4.7 Conclusions

In this chapter it has been shown that the bias for sub-tree swapping crossover to sample smaller programs has a significant effect on GPs ability to search the program space. Although one would expect larger unique programs to be sampled infrequently simply due to their number, we can see this is exasperated by crossover. All is not lost, however, as we can substantially reduce the bias by deliberately increasing the average size of individuals in the population.

We next turned our attention to initialisation and crossover. One would expect by initialising the population to match the size distribution desired by crossover we would improve the performance of our GP run. For our test problems the opposite is true: we encounter reduced fitness w.r.t. to that seen using our original FULL initialisation. The reasons for this are straight forward: both problems rely on a particular length threshold before fitness

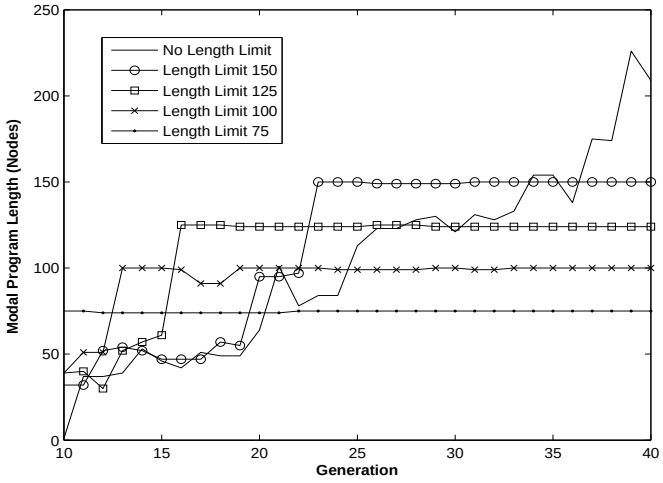


Figure 4.13: Comparison of modal (peak) classes associated with length limits for the Artificial Ant Problem with selection. RAMPED initialisation has been used with a maximum depth of 6 and minimum depth of 2.

can improve, and our bias towards smaller programs has produced many more relatively unfit programs.

Increased code growth was also encountered for our 'Lagrange-like' initialisation. Again the roots of this lie within sub-tree swapping crossover's bias to sample smaller programs and can be used to describe a new bloat theory, *Crossover-Bias*. Put simply crossover in itself does not alter average program size, but it does produce a distribution skewed towards smaller programs. If the smaller programs are ignored by selection, average program size will increase with each generation. For problems such as the Artificial Ant and 4 Bit Even Parity where a size threshold exists for fitter programs, bloat can be induced from generation zero via a 'Lagrange like' initialisation. For other problems bloat will begin to occur in later generations as larger fitter programs are discovered.

Finally, we looked at the effect of applying length limits to our test problems and found that this increased the sampling of the smallest programs,

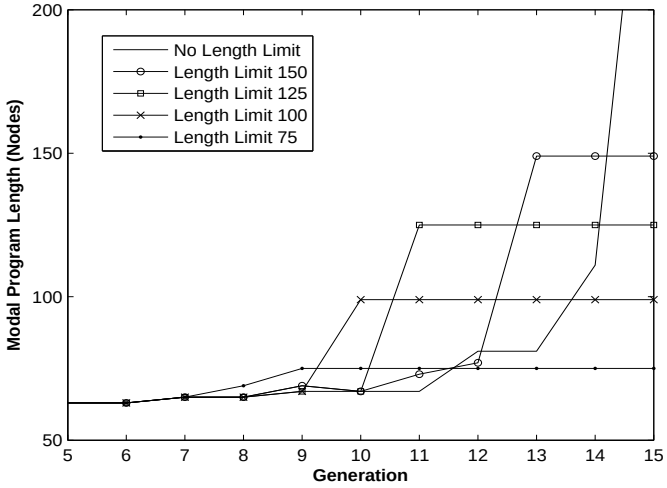


Figure 4.14: Comparison of modal (peak) classes associated with length limits for the 4 Bit Even Parity Problem with selection. RAMPED initialisation has been used with a maximum depth of 6 and minimum depth of 2.

hence speeding bloat (to the length limit) during earlier generations, thereby, defeating their original purpose of combating bloat during this stage of the GP run.

Chapter 5

Sampling Parsimony

5.1 Introduction

In the last chapter, we identified a major characteristic of sub-tree swapping crossover: the resampling of smaller programs. In this chapter, we examine the effects of resampling and how they might be controlled.

In Section 5.2, we find that on a flat fitness landscape, i.e., isolating our bias, sub-tree swapping crossover will progressively resample programs from smaller classes whilst only creating new unique programs in larger classes. In Section 5.3 a novel method to control resampling is introduced, *sampling parsimony*, that applies fitness penalties to programs that have been resampled. From the application of this method it is found that the rate of program growth can be reduced or increased through appropriate choice of parameters, in effect providing a program growth control mechanism, the mechanics of which can be explained using the crossover-bias bloat theory.

As resampling penalties have a direct effect on program growth, practitioners need to be aware of this when implementing methods that ensure unique programs are sampled. It is shown, however, that this effect can be

controlled when an *allowance* for resamples is implemented, the appropriate value of which is found to be problem specific.

5.2 Sampling and Resampling

Our first step is to see how sub-tree swapping crossover will sample the search space on a flat fitness landscape. Our primary purpose for doing this is simply to isolate the resampling bias for crossover. In particular, we are interested in finding out how much resampling goes on. This gives us an idea of the efficiency or otherwise of the search. It should be noted that whilst in the presence of fitness gradients, selection will help to counteract crossover bias (this is analysed further in conjunction with selection in Section 5.3), there are situations where the crossover bias may become the prominent search bias. This may happen, for example when GP search reaches an area of neutrality, e.g., when GP operators, during an experimental run, are unable to escape areas of similar fitness.

To empirically analyse crossover sampling we took two out-of-the-box problems from the ECJ evolutionary toolkit [Luke, 2008]: 4 Bit Even Parity and the Artificial Ant. As the Parity problem uses Boolean operators only we know that, in the absence of selection, the limit program length distribution to be that of a 2-ary tree similar to that shown in Figure 3.9, whilst, as previously discussed, the Artificial Ant will tend to follow a generalised distribution similar to the one in Figure 3.11.

Adjustments were made to ECJ to return a constant fitness value, remove mutation, ensure uniform selection of crossover points, and to prevent a depth or length limit being applied. A population size of 1,000 individuals was used and the results for 200 generations were averaged over one hundred independent runs. All experiments were initialised using the RAMPED

method [Koza, 1992] with a maximum depth of 6 and minimum depth of 2.

The total number of programs for each length was recorded at each generation along with the number of programs for each length that had been sampled in a previous generation. Taking the Artificial Ant problem, as we can see in Figure 5.1, at generation 200 the number of new unique programs is extremely small compared to the total for that generation. The majority of all programs sampled under these conditions are in the smaller length classes.

As shown by the ratio between the number of new programs and the total number of programs, plotted in Figure 5.2, it is clear that newly sampled programs are only being generated at the larger length classes and that subtree swapping crossover is progressively resampling more and more programs in the smaller length classes.

This bias to resample can impact the efficiency of GP search in a number of ways. Firstly, resampling is costly in terms of re-evaluations (although thankfully we are only re-evaluating the smaller programs). Secondly, we would like our search to address as many different potential solutions during an experimental run i.e., to improve our chances of success in finding a solution.

One could argue that, combined with selection, the re-presentation of smaller program components to crossover may in fact be intrinsic to the success of the GP search process i.e., to perform a more local search.[1] However, from our previous analysis it seems unlikely that smaller programs would even be presented to crossover (see Section 4.5) and resampling may in effect speed bloat. This is analysed further in the following section.

[1]Without knowledge of the program spaces for particular problems this could equally be a disadvantage in that we may prefer a broader search.

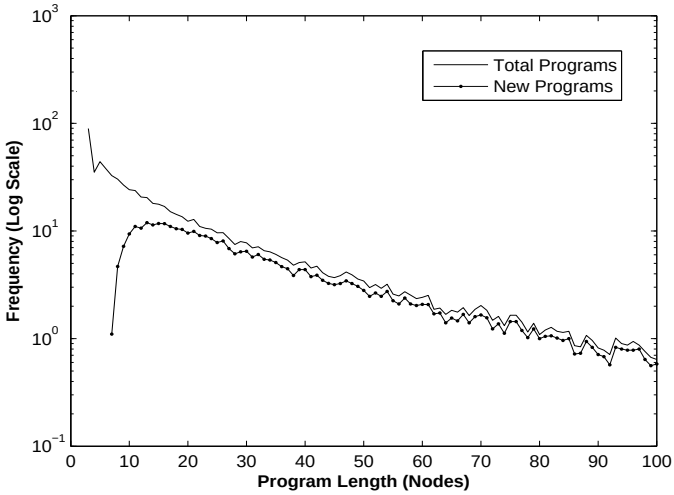


Figure 5.1: Frequencies of new unique programs not sampled previously compared to all programs generated at generation 200, for the Artificial Ant problem applied to a flat fitness landscape. Invalid length 2 has been removed.

5.3 Sampling Parsimony and Bloat

In section 5.2 we looked at how sub-tree swapping crossover likes to sample smaller programs and the progressive resampling of programs (with associated inefficiencies) that result from this. In this section we look at the prevention of resampling and its effect on program length.

To understand the effect of resampling and to control it, a novel technique has been employed called *sampling parsimony*. This has two parameters, a resampling penalty to be applied, which is implemented as a percentage reduction of fitness,[2] and a count of the number of times a unique program can be sampled before that penalty is applied or removed.

Our first application is to look at how average program length will be affected by the application of a super penalty (10,000%) ensuring that a

[2]As we are looking to minimise the fitness function value returned for the problems used in this chapter, this penalty is actually a multiplier.

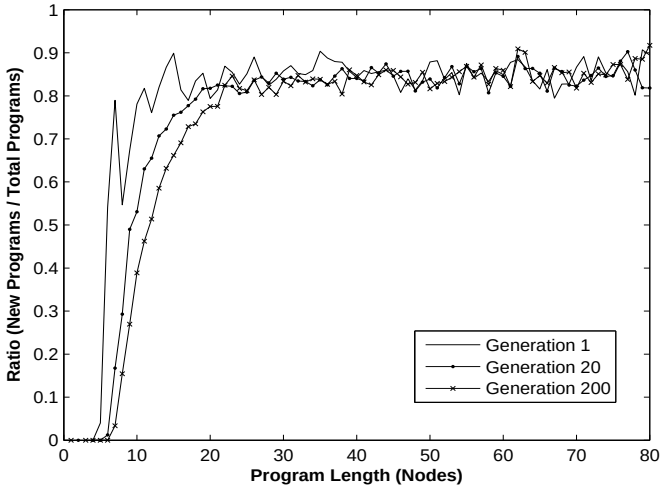


Figure 5.2: Ratio of new unique programs not sampled previously compared to all programs generated at generations 1, 20 and 200, for the Artificial Ant problem applied to a flat fitness landscape.

resampled program will not be selected in the next generation. Using our Artificial Ant and Parity ECJ problems with parameters as described in Section 5.2 from Figures 5.3 and 5.4 we can see that, as we progressively prevent resampling by lowering our resampling limit (the number of resamples allowed before a penalty is applied), we increase the average size of the programs in our population. We have in effect created an effective fitness landscape [Langdon and Poli, 2002] where the ability for a child to exist in the next generation is solely determined by whether that program has previously been sampled.

From our earlier analysis, it is unsurprising that we see that by depressing the fitness of resamples, we will increase the sampling of larger programs, thereby increasing the average program size as we are in effect penalising smaller programs. What is more interesting is that we have managed to isolate the crossover-bias bloating effect as described in Section 4.5. In essence, our method only penalises children and prevents them from being

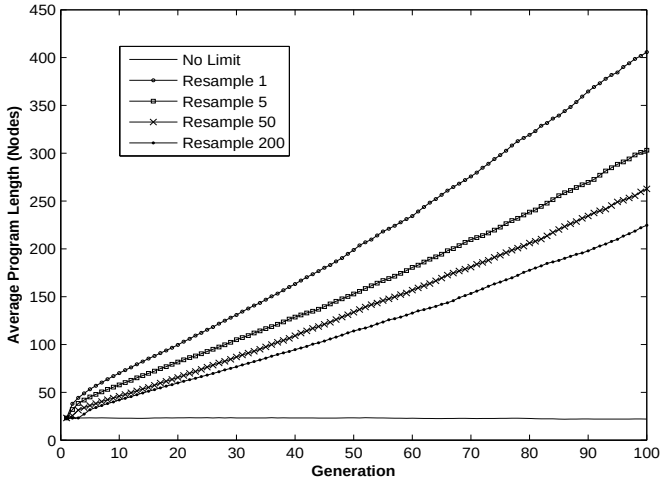


Figure 5.3: Comparison of average program size applying resampling limits to the Artificial Ant problem with a flat fitness landscape.

parents rather than preventing their creation. GP, therefore, uses larger programs as parents (see Figure 5.2), hence, increasing the average size of children and thereby increasing the average program size in the next generation. As smaller children are still created by crossover but have no chance of being chosen by selection, this process will continue. With even a relatively large resampling allowance of 200 on our flat landscape, the penalty will greatly increase program size.

We apply our resampling penalty method to our problems with selection in Figures 5.5 and 5.6. We can see that our penalty increases program growth within 100 generations. This is because we have, effectively, accelerated the crossover bias effect (crossover creating small programs that selection then ignores) already present in the 'No Limit' distribution. Practically, we can see that this acceleration only happens beyond a problem specific value of the number of resamples allowed, suggesting that experimental resampling restrictions may not attract significant additional program growth once an

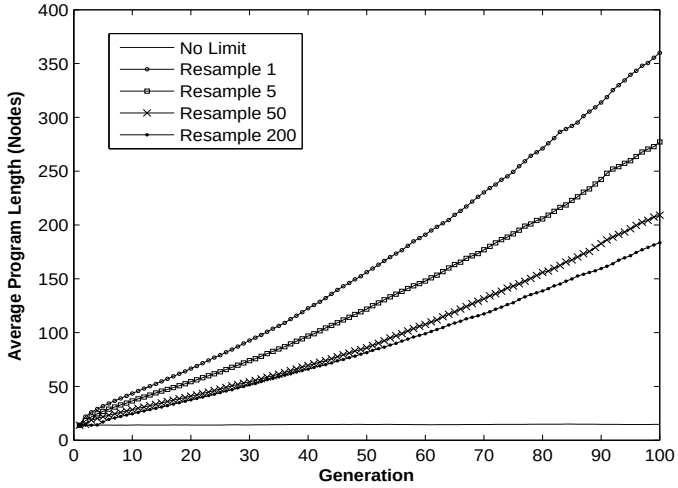


Figure 5.4: Comparison of average program size applying resampling limits to the 4 Bit Even Parity problem with a flat fitness landscape.

acceptable limit has been determined. Experimentation showed that no changes are observed beyond 5 resamples for the 4 Bit Even Parity problem, and approximately 15 for the Artificial Ant.

Finally, we reverse our method to apply a penalty to all programs from the beginning. We only remove the penalty after a specific number of resamples have been achieved. From Figures 5.7 and 5.8 we can see that program growth is significantly reduced by applying a single sample penalty. Progressively increasing the sampling threshold before normal fitness is applied will reduce program growth towards a limit of approximately 50 samples for the Artificial Ant problem and exactly 2 for the 4 Bit Even Parity Problem.[3] Again the threshold is problem dependent.

Although its effect on bloat is self evident, it remains to be seen whether the sampling parsimony method can be successfully applied to improving

[3]Any value above 2 gives single node programs such an evolutionary advantage that the population is soon dominated by programs of that length and crossover becomes unable to produce programs of a larger size.

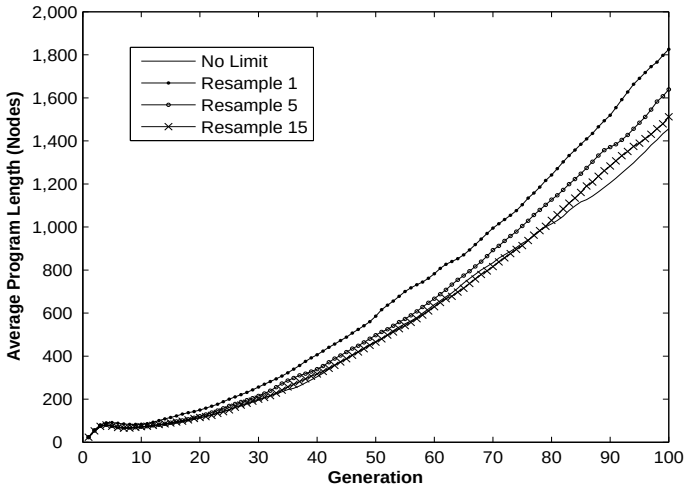


Figure 5.5: Comparison of average program size applying resampling limits to the Artificial Ant problem with selection.

overall program fitness over an entire run. We leave this for future work. The current 'blanket' method is of course very unsophisticated in that we prevent entire search spaces from being investigated without regard to program fitness. However, this remains an interesting technique that is worth exploring in greater depth and which might find application in a variety of areas, including, for example, escaping experimental stagnation under various conditions.

5.4 Conclusions

Both theory and experimentation show that the application of sub-tree swapping crossover will quickly encourage a population to converge to a distribution that will exponentially sample smaller programs more frequently than longer ones. As there are exponentially fewer unique smaller programs than larger ones, the sampling of new programs becomes less likely during a GP run if only crossover is applied. We also know from previous chapters that

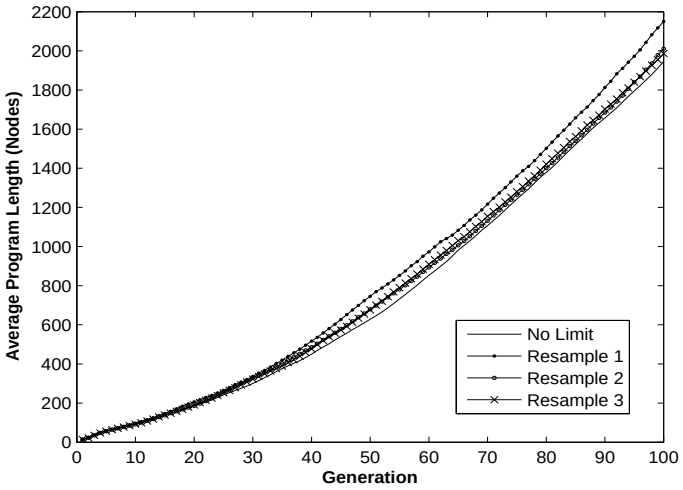


Figure 5.6: Comparison of average program size applying resampling limits to the 4 Bit Even Parity problem with selection.

this bias becomes more acute with smaller average population sizes (see Section 4.3) and with the application of length limits (see Section 4.6).

Although the application of selection will work against the crossover bias, smaller programs will always be created by crossover. As it is unlikely that these programs will be able to obtain a reasonable fitness, particularly during later stages of a GP run, they will be ignored by selection for the next generation and only larger parents will be selected. The continuing application of selection and crossover, therefore, causes the mean program size to increase, thereby creating bloat.

To explore what happens if one directly addresses this sampling-related cause for bloat, we have introduced a novel technique called sampling parsimony to influence program growth. This can be used to accelerate growth as well as to reduce it. This effect has practical implications, i.e., if one was to impose a unique program restriction on the creation of programs. We have shown experimentally, however, that there is a problem specific limit

to this effect based upon a resampling allowance, a parameter that could be easily applied to such a mechanism.

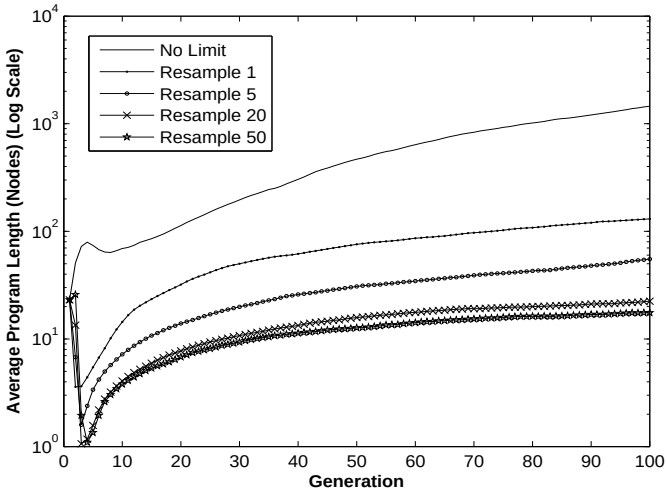


Figure 5.7: Comparison of average program size applying sampling penalties to the Artificial Ant problem with selection. Penalty is only lifted after a predetermined number of resamples have been reached.

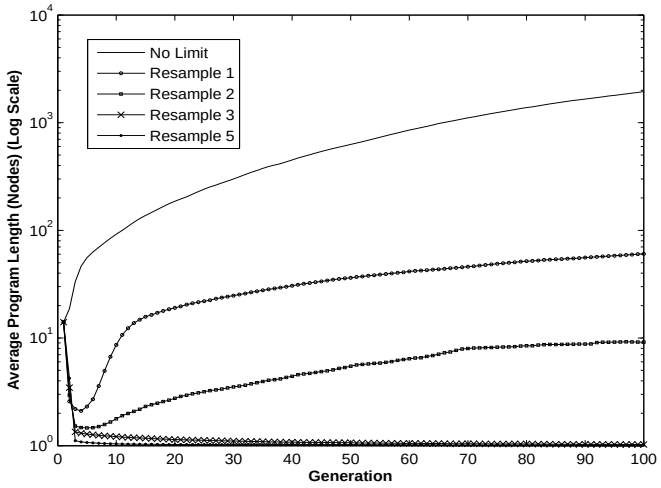


Figure 5.8: Comparison of average program size applying sampling penalties to the 4 Bit Even Parity problem with selection. Penalty is only lifted after a predetermined number of resamples have been reached.

Chapter 6

Operator Equalisation

6.1 Introduction

An intrinsic feature of traditional Genetic Programming (GP) is its variable length representation. Although, this can be considered one of the method's strengths, researchers have struggled with the phenomenon of bloat,[1] the growth of program size during a GP run without a significant return in terms of program fitness, since GP's inception.

Research has shown (see section 2.6), that beyond a certain minimum program length the distributions of program functionality and, therefore, fitness converge to a limit. Before that limit, however, there may be program-length classes with a higher or lower average fitness than that achieved beyond the limit. Ideally, therefore, GP search should be limited to program lengths that are within the limit and that can achieve optimum fitness. We might want, for example, to restrict our search fixing program sizes at the point where our smallest optimal or near optimal solutions can be found thereby avoiding the need to search much larger spaces with the additional

[1] Bloat, and a number of techniques to control it, are discussed in section 2.5 whilst a new bloat theory is described in section 4.5.

computational effort that entails.[2] For most applications simpler solutions are also much more desirable than larger solutions.

This chapter describes a simple method, *operator equalisation*, that can be applied easily to existing GP systems. This method forces GP to search specific length classes using pre-determined frequencies so that we can control the sampling rates of specific program lengths.

This technique has several advantages. For example, whenever the length distribution and the corresponding sampling bias provided by standard operators is not suitable for a specific program space, we can change such a bias freely making it possible to sample or oversample certain length classes we believe can benefit our search. The user is given complete control over the program length distribution, and bloat can be entirely and naturally suppressed by simply asking operator equalisation to produce a static length distribution.

The mechanics of operator equalisation are explained in Section 6.2. Two test problems which are known to bloat under unconstrained conditions are then described in Section 6.3. Next we look at how different, static target length distributions can influence performance regarding these problems in Section 6.4. In addition to controlling bloat, operator equalisation also enables us to automatically sample and exploit the best fitness values associated with particular length classes. This is examined in Section 6.5, where we look at how coarsely grained distributions can be used to determine potential high fitness length classes.

[2] If there is little understanding of the program space i.e., likely ratios of solutions to number of programs at length classes, such a method would certainly be justified in terms of resources as a first attempt to discover an acceptable solution.

6.2 Operator Equalisation

Investigations into the properties of program length have often used the tool of histogram representation in order to compare frequencies of programs sampled at particular lengths during a GP run (e.g., see [Poli and McPhee, 2001, Poli et al., 2007, Dignum and Poli, 2007]). The operator equalisation method aims at controlling the shape of length histograms during a run. The method is loosely inspired by both the gray-level histogram equalisation method [Rosenfeld and Kak, 1982] used in image processing and digital photography to correct underexposed or overexposed pictures and the Tarpeian bloat control method [Poli, 2003] which, with a certain probability, sets to zero the fitness of newly created programs of above average length effectively suppressing their insertion in the population. We have taken these ideas forward to see if by filtering which programs are allowed to be inserted in the population we can directly manipulate those frequencies in order to force GP to sample programs of particular lengths at pre-specified rates.

The method requires users to specify the desired length distribution (which we will call `target`) that they wish the GP system to first achieve and then continue to use when sampling a program space. This allows one to specify both simple well known probability distributions (Section 6.4) and also coarser grained models (Section 6.5). During the initialisation of the GP system a `histogram` object is created. This needs only to be primed with the maximum size allowed, number of bins (the size of the bins being calculated from these) and of course the `target` distribution. Then the method requires wrapping the existing code for offspring generation with code that simply accepts or disallows the creation of a child based on its

length. The wrapper is extremely simple:

```
repeat {
  <create a child using standard genetic operators>
} until( histogram.acceptLength( child.length ) )
```

Internally, the `histogram` object maintains a set of numbers, one for each length class, which act as acceptance probabilities. The `acceptLength` method simply generates a uniform random number between 0 and 1 and compares it against the acceptance probability associated with the length class associated to `child`, returning `true` if the random number is less than the acceptance probability, and `false` otherwise.

At the end of each generation the `histogram` object updates the acceptance probabilities for each class using the following formula:

```
newProbability = currentProbability + ( normalisedDiff * rate )
```

where `normalisedDiff` is the difference between the desired frequency in `target` and the current frequency divided by the desired frequency. Small discrepancies for large classes are, therefore, largely ignored. The user defined parameter `rate` determines how quickly the distributions should converge. After some experimentation the setting `rate=0.1` was found to work well and has been used in all experimentation presented in this chapter.

As one can see the method can easily be applied to existing GP applications with minimal changes: users need to modify only very few lines of code in their existing GP systems.

6.3 Test Problems

Two GP problems of differing natures have been deliberately chosen, a parity problem and a symbolic regression problem, to show the benefits and

limitations of this approach. As will be shown, the first requires a relatively large program size before fitness will significantly improve whilst the second is able to achieve relatively high, though far from optimal, fitness values with small program sizes.

The Even Parity problem attempts to build a function that evaluates to 1 if an even number of boolean inputs provided evaluate to 1, 0 otherwise. We have chosen a relatively large input set of size 10. However, it is possible to evaluate all possible fitness cases (1024) for each potential solution within a reasonable time given the short length limit imposed.

Our second problem is a 10-variate symbolic regression problem: $x_1x_2 + x_3x_4 + x_5x_6 + x_1x_7x_9 + x_3x_6x_{10}$ as described in [Poli, 2003][3], which we have called Poly-10. 500 test cases are used each comprising of a (uniform) randomly generated value for each variable ranging between -1 and 1 and the resulting value of the equation.

As with the Even-10 problem only functions with arity 2 are used: ADD, SUBTRACT, MULTIPLY and a protected division function called PDIV which returns the numerator if the magnitude of the denominator is less than 0.001. No Ephemeral Random Constants (ERCs) were used.

Both problems have been sourced from [Poli, 2003] with minor alterations[4] to enable comparison and analysis. Each problem is expected to bloat under non-constrained conditions the reasons for which are described in the original paper.

[3]This problem can be simplified to $x_1(x_2 + (x_7x_9)) + x_3(x_4 + (x_6x_{10})) + x_5x_6$ to give a smallest GP tree size of 19 nodes.

[4]Our Even-10 problem has no NOT function. So all functions have an arity of 2. Also, Poly-10 here uses 500 fitness cases, where originally 50 were used.

6.4 Equalising to Simple Program Length Distributions

All experiments were initialised using the GROW method [Koza, 1992] with a maximum depth of 6. Sub-tree swapping crossover with uniform selection of crossover points was applied without mutation or replication. Elitism was not applied. Tournament selection was used with a tournament size of 2 in all experimentation. The algorithm was generational. All experiments used a population of 10,000 and ran for 100 generations. Results were averaged over 100 runs. It should be noted that due to the wrapper-like implementation there is no reason why mutation, replication or other forms of crossover could not be applied in isolation or combination. In fact it is hard to imagine any form of standard GP experimental set-up which could not be used easily.

In order to satisfy our stated desire of bloat free GP a strict, deliberately small, length limit of 100 nodes has been chosen. This has the added benefit of allowing us to evaluate a large set of fitness cases for each potential solution within acceptable experimental run times.

6.4.1 Does Operator Equalisation Work?

Initial investigation using our parity and regression problems showed that using a fairly unforgiving initialisation method (GROW), i.e., that in no way matched to our desired length distributions, we could equalise program lengths within approximately 20 generations. This is shown in Figure 6.1 for the Poly-10 problem equalised for a uniform length distribution.

With both problems there was a small dip for some of the early length classes. This is due to the fact that when the population has a uniform length distribution, crossover is less likely to produce short programs than

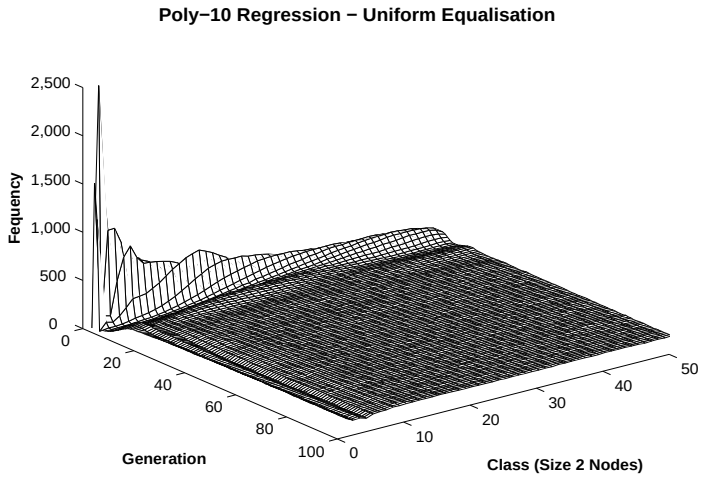


Figure 6.1: Length histogram for Poly-10 regression problem with uniform equalisation of program length classes.

is ordinarily the case in the absence of equalisation. This is illustrated in Figure 6.2 where we look at the number of programs rejected by the wrapper at generation 100. As we can see the number of programs rejected for these length classes is extremely small. Our equaliser is, therefore, doing the best with what it has been presented by the underlying GP system[5]. We can see that the very smallest length classes are still reasonably well represented.

Although it is possible to imagine extreme conditions where infinite loops could be encountered, for the experimentation detailed in the following sections, all runs were completed successfully and no unusually large run-times were recorded. It is of course possible to add a simple retry limit to the wrapper code to escape such loops.

[5]In other experiments (not reported) it was found that the dip is slightly worsened by the use of larger tournament sizes since this increases the ability of selection to repeatedly present certain program sizes. The effect is, by contrast, reduced when using a steady state model, as GP can select newly created programs, i.e., those accepted by our equaliser, immediately without having to wait for a generation to be completed.

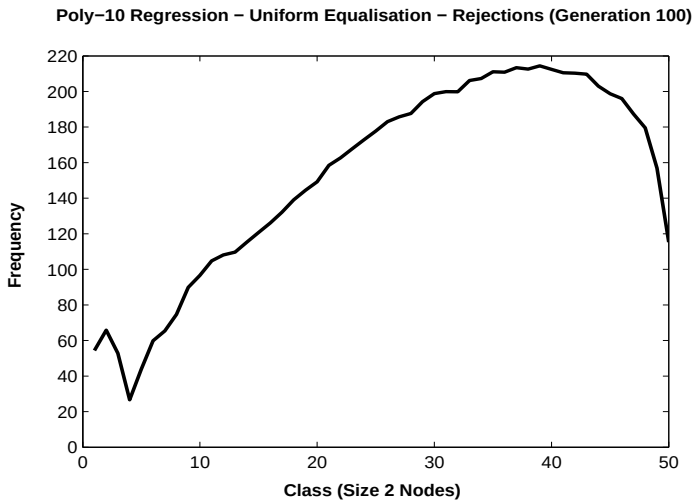


Figure 6.2: Number of equaliser rejections, at generation 100, for Poly-10 regression problem with uniform equalisation of program length classes.

6.4.2 Efficiency of Different Length Distributions

Having established that the operator equalisation algorithm works for the test problems, our next step is to apply the method to see how the use of elementary, easily recognisable, target probability distributions could affect our search. In this thesis, we only consider static distributions, although operator equalisation works also with dynamic targets (see Section 9.2 for further discussion regarding this issue).

In Figure 6.3 we see the final length distributions for the parity problem, i.e., at generation 100, for different target distributions. Each length class is 2 nodes in size. Given that all the functions in our function set (AND, OR, NOR, NAND, XOR and EQ) have an arity of 2, we have an individual class for every possible length up to our size limit.

A number of distributions have been analysed. A uniform distribution where each length is sampled with the same frequency, a triangular distribution which has a linearly increasing bias towards sampling larger programs,

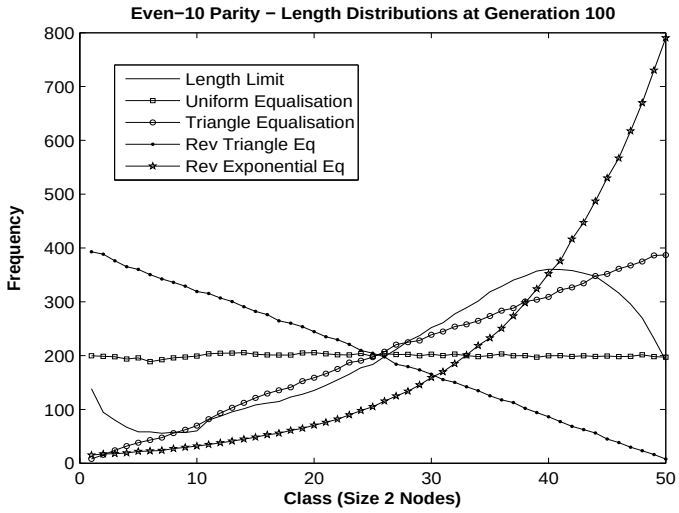


Figure 6.3: Length distributions, at generation 100, for the Even-10 parity problem using a length limit of 100 nodes with different equalisation targets.

a 'reverse' triangle where smaller programs are sampled more often and a reverse exponential distribution where we sample larger programs exponentially more frequently than shorter ones. Note, the distribution for the length limit with no equalisation is also shown. In all cases the target distribution was reached very quickly. After some initial fluctuations, as we can see in Figure 6.4, the average size for each of the experiments settles to a fixed value.

If we compare the best fitness values recorded for different target distributions (Figure 6.5), we can see that the push towards sampling larger programs has had a beneficial effect compared to using the simple length limit.

The exponential distribution has a sharper upwards gradient for generations 20 to 60 than that of the triangular distribution although both eventually converge to the same value. The bias towards the sampling of smaller programs has had the most negative effect. Selection does, however,

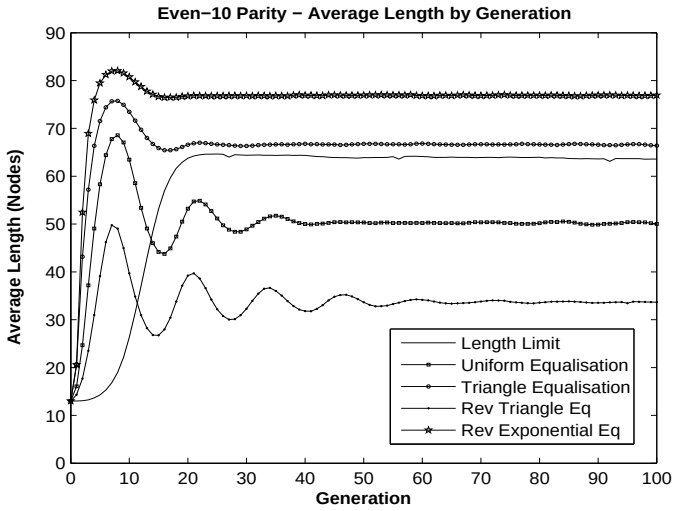


Figure 6.4: Average length (nodes) for Even-10 parity problem using a length limit of 100 nodes with different equalisation targets.

manage to improve fitness in all of our experiments. Perhaps surprisingly, all equalisation methods improve the best fitness value compared to the simple length limit during the early generations. The value of exploring certain length classes during early generations is discussed further below.

Unlike the Even-10 problem we can see in Figure 6.6 that the imposition of target length distributions has a negative effect on all forms of equalisation for best fitness compared to our simple length limit for the Poly-10 problem, any undersampling of smaller programs during the early generations having the most marked effect. It has long been known that in symbolic regression problems smaller programs can obtain relatively high fitness. In fact, the reverse triangle distribution performs as well as the simple length limit up to generation 15 and outperforms most other methods most of the time. This indicates that in this problem the dynamics of the length distribution is important, and GP benefits from exploring short programs for 10 or 15 generations and then progressively moves towards sampling longer programs,

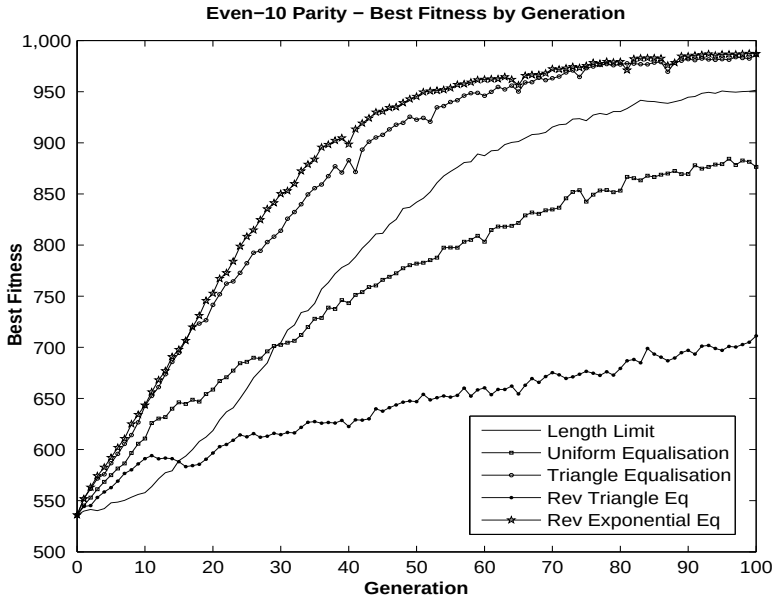


Figure 6.5: Best fitness (number of test cases matched) for Even-10 parity problem using a length limit of 100 nodes and different equalisation targets.

as GP with a simple length threshold does. So, this suggests that there could be benefits in using dynamic target distributions.

Methods to detect this bias are discussed in the next section.

6.5 Length Class Sampling

As we have a method to directly influence the sampling of particular length classes, we can now look at two sampling techniques that can help us gain an insight into the program space that we wish to search. These techniques are presented in Sections 6.5.1 and 6.5.2.

Note that for the experiments described below the same GP system as in Section 6.4 was used, but with two small, yet important, differences. Firstly, in order to remove any initial length bias the GROW initialisation method has been replaced with the RAND_TREE method described in [Iba, 1996]. Secondly, to show that useful insights into the program space of a problem

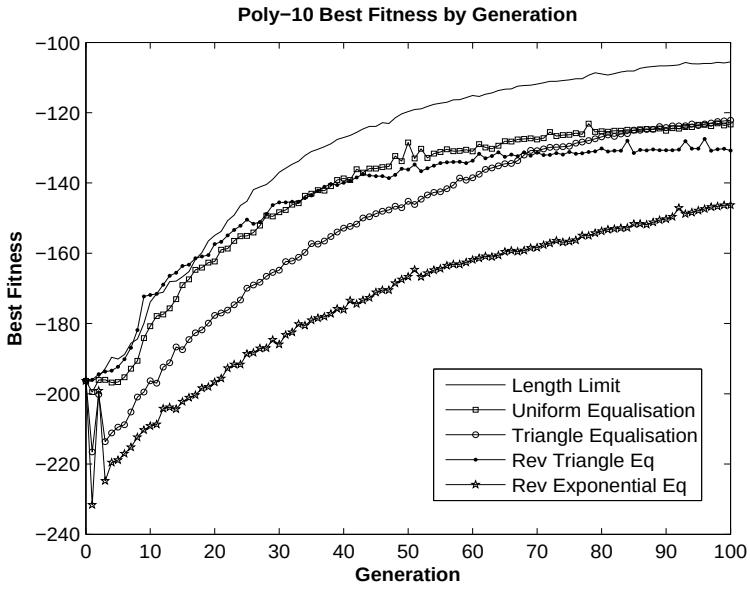


Figure 6.6: Best fitness (minus mean squared error) for Poly-10 symbolic regression problem using a length limit of 100 nodes and different equalisation methods.

can be achieved without undue computer resources, smaller length limits of 60 and 80 nodes and a much reduced population size of 1,000[6] have been used.

6.5.1 Single Length Classes

Using the RAND_TREE method we can sample without bias specific length classes. We can, therefore, look at the sampling of individual classes in isolation. For our experimentation the search space was divided into 15 equal length classes with each class sampling two distinct program lengths e.g. 1 and 3 for the first class, 5 and 7 for the second etc[7]. The objective was to find out which area (length class) of the search space would appear preferable to a GP system in the early generations of a run.

[6] As we have used a smaller population size we cannot directly compare the best fitness results reported in this section with those reported in the previous section.

[7] Even sized programs are not possible for 2-ary trees.

For the Even-10 problem (Figure 6.7) we can see quite clearly that there is a small threshold where potential solutions cannot achieve anything better than 512 correct classifications, exactly half the total possible. However, as we move to larger program sizes we can see a distinct improvement in fitness. Selection will, therefore, quickly guide GP to larger programs in the early stages of a GP run.

Figure 6.8 shows that, for the Poly-10 problem, when we initially sample the program space, we find that the smallest programs do indeed have relatively better fitness than their larger counterparts. This explains GP's concentration in this area during earlier generations in the experimentation reported in Section 6.4. Of course, these areas do not contain optimal solutions: we need at least 19 nodes to achieve that. However, to an initial random sampling these areas display a higher proportion of relatively fit programs than those of the larger program size search spaces sampled. This explains why without operator equalisation GP first samples the short programs but then quickly moves towards the longer programs, where, upon sufficiently sampling, better solutions can be found. This also explains why equalisation with a reverse triangular distribution does well initially, but cannot compete with standard GP later on (see Figure 6.6). Finally, it also explains why equalisation with distributions that sample the longer programs more frequently, such as the reverse exponential distribution, produce much worse fitness than standard GP and reverse triangular equalisation, initially.[8]

[8]The fitness plot for the reverse exponential distribution in Figure 6.6, however, remains parallel to the plot of standard GP, suggesting that given enough generations operator equalisation with this distribution would eventually catch up.

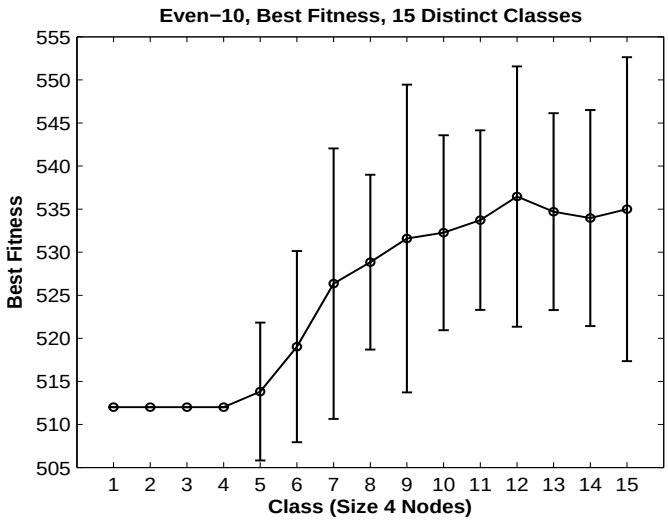


Figure 6.7: Best fitness for Even-10 problem sampling 15 distinct size classes using the RAND_TREE method. Results are averages over 100 samples of 1,000 individuals each (1,000=GP population size).

6.5.2 Multiple Length Classes

Of course the picture may change significantly if we sample two or more classes, perhaps with differing proportions. Also, what may look like a good sampling histogram initially (upon the random sampling produced by initialisation) may later turn out to be sub-optimal after many generations of GP exploration. So, in this section we look at how the picture changes when using multiple length classes in combination and when comparing the initial to the final generation of runs.

To this end, the length distribution was divided into 4 bins of size 20 nodes with each combination of bins sampled using frequencies that were multiples of 20%. For example, bins 2 and 3 might have frequencies of 40% each, while bin 1 might have a frequency of 20% and bin 4 a frequency of 0%. Every combination, including those with multiple empty bins, was sampled. There were 56 combinations in total. For each the resulting best

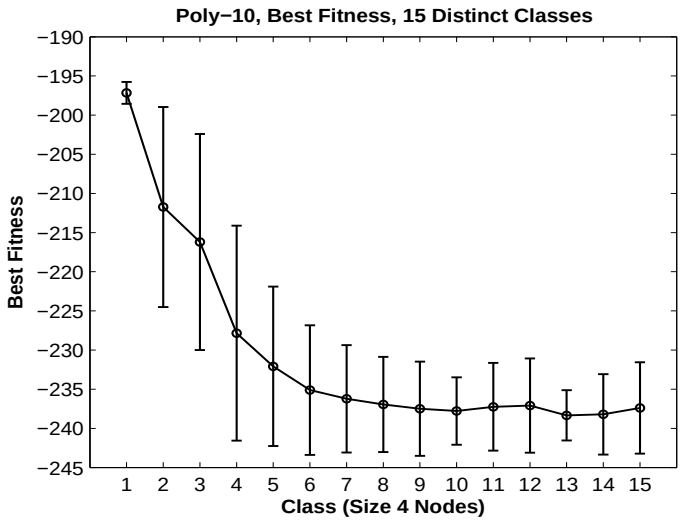


Figure 6.8: Best fitness for Poly-10 problem sampling 15 distinct size classes using the RAND_TREE method. Results are averages over 100 samples of 1,000 individuals each (1,000=GP population size).

fitness values at each generation were tabulated. This produced a large dataset which can be summarised using multiple linear regression formulas resulting from fitting the data at generations 0 and 100. The formulae have the following form:

$$bestFitness = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \beta_3 X_3 + \beta_4 X_4 \tag{6.1}$$

β_0 is the constant term and β_i being the coefficient of each of the length classes X_i, X_1 being the smallest class.

After the multiple linear regression was applied to the Even-10 problem the following formula was found for our initial generation:

$$bestFitness = 424.897 + 96.256 X_1 + 103.899 X_2 + 110.831 X_3 + 113.911 X_4 \tag{6.2}$$

As we can see there is a small improvement in best fitness as we search the

larger classes. After applying GP search, at generation 100 the improvement is more distinct as shown by the regression formula:

$$bestFitness = 509.914 - 36.000X_1 - 12.000X_2 + 216.276X_3 + 342.748X_4 \tag{6.3}$$

For the Poly-10 problem for the first generation we obtain:

$$bestFitness = -179.595 - 15.509X_1 - 54.645X_2 - 56.678X_3 - 52.763X_4 \tag{6.4}$$

while after 100 generations the picture is somewhat different:

$$bestFitness = -147.146 - 33.712X_1 - 25.438X_2 - 46.835X_3 - 41.161X_4 \tag{6.5}$$

We can clearly see that for Poly-10 different parts of the search space yield different results for our initial generation and later stages of GP search. As one would expect from these results the best and worst combinations for our 100th generation showed a strong dislike for the third class. A 100% sampling of which, was indeed our worst result of -215.265, whilst more interestingly a broader sampling of the surrounding classes yielded the best results all of which were below -170.

6.6 Conclusions

In this chapter we have introduced operator equalisation, a programatically simple method that can be easily applied to current experimental environments that allows us to finely bias GP search to specific program lengths. In particular, the method can force GP to sample the search space using static and arbitrary length distributions. This completely and naturally

suppresses bloat.

This method has been applied to first seeing how simple bias can influence the results of two different but potentially bloating problems. The Even-10 parity problem was shown to have a simple positive bias towards longer programs within the 'experimentally-friendly' 100 node limit specified, whilst the Poly-10 regression problem was shown to have a positive bias towards the sampling of shorter programs during early generations.

Using simple statistical techniques it has also been shown that we can use the method to quickly gain information about the search space and the best way to sample it with GP (with and without equalisation).

The primary aim of bloat free GP is to sample program spaces in such a way that we allow GP to discover optimal or acceptable near-optimal solutions without wasting resources searching ever larger spaces with little return with regard to fitness. Here we have made some strong steps in this direction. An automatic method of defining the appropriate search space for a GP problem may not be so far off. For example, there is no reason why the method introduced in this chapter cannot be applied to the initial setting of size limits (either maximum or minimum), or even to define a dynamic schedule for biasing the sampling of programs to certain sizes over the entire run or during different stages of a GP run.

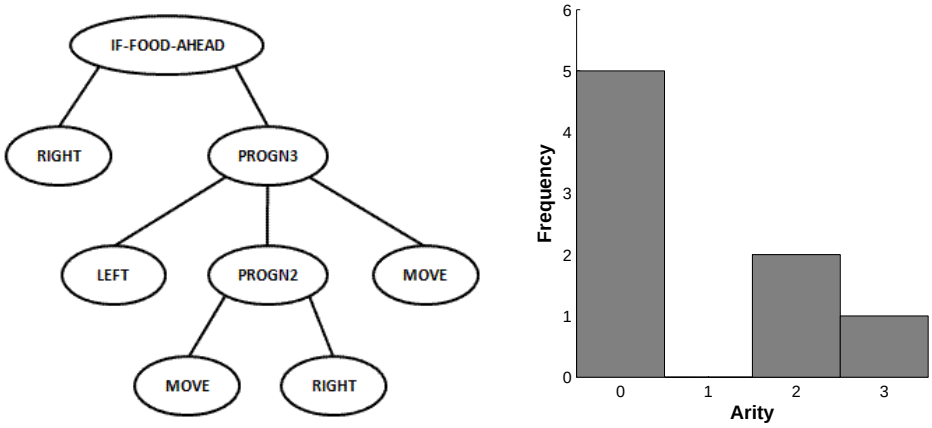
Chapter 7

Allele Diffusion and Structural Convergence

7.1 Introduction

In Chapter 3, a hypothetical model was provided to show that sub-tree swapping crossover will sample exponentially more shorter programs for a-ary trees when applied to a flat fitness landscape in the absence of mutation, i.e., when its bias is isolated. This was extended by generalisation to mixed-arity trees and then to true length-classes (from internal node counts). Strong empirical support has been provided for the original model and each generalisation. In this chapter we want to understand what other biases sub-tree crossover presents beyond its length biases.

One can divide the space of all possible programs into subsets in a number of ways. As discussed in the previous chapters, one way is to group programs by the number of nodes in the tree representing them. We will call each such set a *length class*. A finer classification would be to divide the programs by their tree shape. This is what we will call a *shape class*.



(a) (b)

Figure 7.1: A proposed solution for the Artificial Ant problem (a) and its associated arity histogram (b).

Each program shape is characterised by the number of primitives/nodes of each arity it contains. This can provide a (non-unique) signature for the shape, which we will call an *arity histogram*, see Figure 7.1 for an example. Of course, all shapes with a particular arity histogram also have an identical number of nodes. So, if we group programs by their arity histograms, we obtain a sub-division of the program space, which is between the length class and the program shape, in that many shapes (but only one program size) can correspond to an arity histogram.[1]

An assumption of the original hypothesis in Chapter 3, indirectly corroborated numerically in [Poli et al., 2007], was that all tree shapes within a particular length class for a-ary trees would be equally likely, as all correlations present within the shapes would be removed by the crossover operator. This implies a diffusive process where any node is equally likely to be in any position within the tree shape. If this diffusion process occurs, we can as-

[1]Naturally, the distinction between length-class and arity histogram disappears for a-ary trees. Also, in both the single and the mixed-arity cases, the number of terminals is always determined by the rest of the arity histogram.

sume that sub-tree swapping crossover is unbiased in its exploration of the search space within each length class, i.e., it will explore all programs with equal probability within each length.

The appropriateness of bias (or lack of it) is problem dependent (see No Free Lunch Theorems [Wolpert and Macready, 1997] discussed in Section 2.1). However, characterising the bias allows us to understand why GP has been successful in solving certain problems or classes of problems and unsuccessful with others. Understanding such bias also allows us to explain how GP searches, when areas of neutrality are reached or when selection reduces fitness variance in the population during the later stages of a GP run. It also provides a starting point in the analysis of the effects of combinations of GP operators.

In Section 7.2, a Cartesian node reference system is used to identify all possible positions within a tree. Using this, we can, in this chapter, provide evidence of a diffusion process showing that all correlations between nodes are broken by repeated application of sub-tree swapping crossover in the absence of selection and other reproduction operators.

We turn our attention to unique shapes within length classes in Section 7.3. As predicted, shape classes are shown, empirically, to have equal occurrence within each length class for a-ary trees, although as predicted in Chapter 3, shapes with smaller lengths are more widely sampled than those of larger lengths. Shapes within length classes for mixed-arity trees, however, are not sampled equally. Empirical evidence is found to show that only those within each distinct arity histogram class are sampled in such a way. This extends previous research showing that the repeated application of crossover is likely to distribute trees according to their arity histogram. Earlier results for a-ary representations are a special case of this more gen-

eral result.

From this characterisation of crossover's biases, we are in a position to explain the lack of structural convergence of GP solutions during experimentation [Banzhaf et al., 1998, page 278]. Structural convergence is an effect seen in other forms of evolutionary search, notably GAs, where it is often used as a stopping criterion for experimental runs. This is discussed in Section 7.4 along with potential broader convergence detection measures, while in Section 7.5 we summarise our findings.

7.2 Allele Diffusion

Our first task is to test the assumption that crossover will remove any correlations between nodes ensuring that all node labels are equally likely to be found at any position within trees created purely from the application of crossover.

Earlier work provided theoretical and empirical evidence to support this claim for linear GP [Poli et al., 2002], where only internal nodes of arity 1 were used. This, of course, is a specific case of the a-ary assertion in Chapter 3.

We have chosen to implement the technique used in [Poli et al., 2002] where a node marker or *dye* is applied at specific positions within trees during initialisation. The amount of dye is then recorded for each node position in subsequent generations.

With linear GP it is possible to compare directly node positions within length classes. This is not true, however, for a-ary trees or those with mixed arities. We have chosen, therefore, to implement a Cartesian node reference system to assign unique node positions for all possible trees based upon the maximum arity that may be used. The exact method is described

in [Poli and McPhee, 2003a]. However, it can simply be described as producing a template based on a maximal tree, i.e., one where only the largest arity is used without terminals up until a maximum depth. Each node is assigned a unique integer number in the order of left-to-right breadth-first traversal, 1 being the position of the root node.

For each set of experiments, a population of 100,000 individuals was used. Dye was placed either at reference 1 (the root node) or at reference 5. These positions have been chosen carefully to ensure dye was applied once to every tree during initialisation for all chosen arity mixes, hence, simplifying theoretical calculations. For all experimentation, a flat fitness landscape was used and sub-tree swapping crossover with uniform selection of crossover points was applied with no mutation or reproduction. All programs were initialised using the FULL method with a depth of 3 (depth 0 being the root node) and all results have been averaged over 20 independent runs.

In Figure 7.2a we can see that for the proportion of internal nodes with dye, for 2-ary trees of length 11, we move rapidly to our expected value at each of the first fifteen possible node references.[2] For 2-ary trees initialised with the FULL method with depth 3, each tree will have only one dye node for each of the possible seven internal nodes, hence, after diffusion has taken place we expect all positions to have a dye proportion of 1/7 for internal nodes. Consistently similar results, i.e., convergence to predetermined predicted proportions are seen in additional experiments for 3-ary trees, and mixed arity trees of 1, 2, 3, 4 & 5 and 2, 2 & 3 arity nodes.[3] These are shown in Figures 7.2b-d respectively.

[2]Note, it is possible for internal nodes to reach a position of 31 using our reference system for 2-ary trees of length 11. A limit of 15 was chosen for consistency across experimentation.

[3]All experimentation shown was subjected to a $\chi^2_{10\%}$ test which showed support for the assertion that the first 15 positions, at generation 100, would each contain a number of nodes determined by initial population proportions.

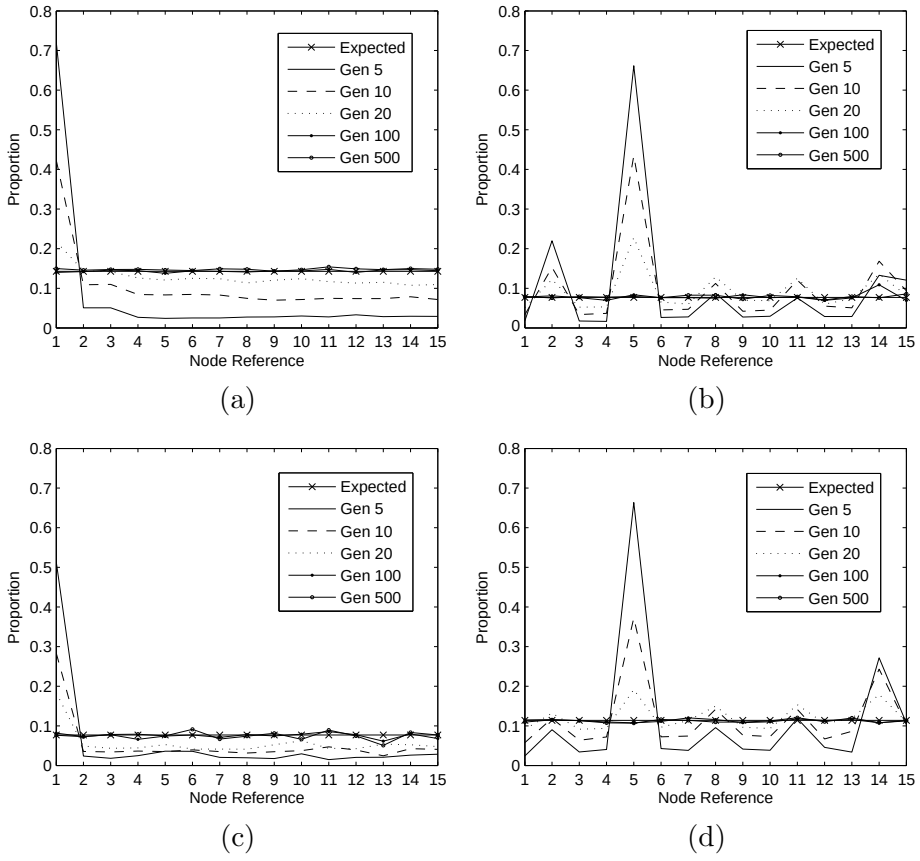
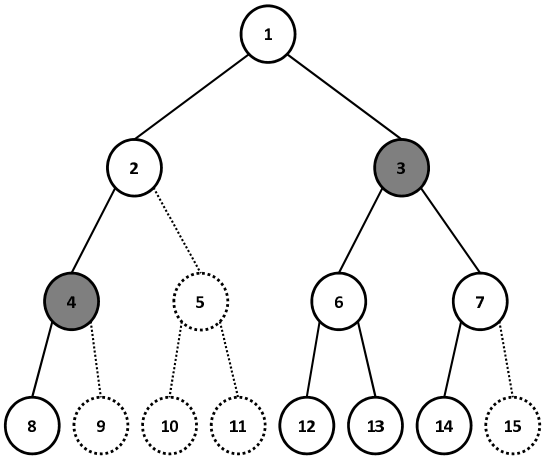


Figure 7.2: Plots of the relative proportion of non-terminal dye alleles vs node references for: (a) 2-ary programs of length 11, initial dye reference 1, expected value: 1/7, (b) 3-ary programs of length 13, initial dye reference 5, expected value 1/13, (c) mixed arity 1, 2, 3, 4 & 5 programs of length 11, initial dye reference 1, expected value: 100,000/1,297,856.85, (d) mixed arity 2, 2 & 3 programs of length 13, initial dye reference 5, expected value: 100,000/877,648.25. Note, selected tree lengths are smaller than the smallest trees created by the initialisation method hence data is not recorded for generation 0. Expected values for mixed arities are calculated from initial internal node counts.

We next turn our attention to co-occurrence of pairs of non-terminals, i.e., whether we can consistently see any correlation between dye positions. In order to do this for each generation a 15 by 15 matrix is produced. Each row and column records the first 15 positions using the Cartesian

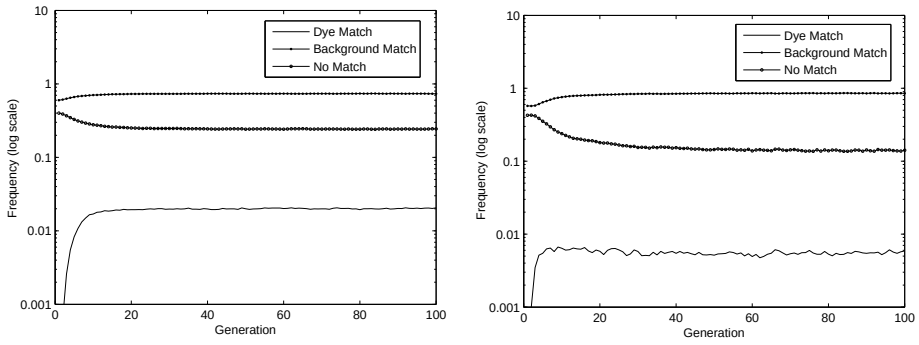


	1	2	3	4	5	6	7
1		B	N	N		B	B
2	B		N	N		B	B
3	N	N		D		N	N
4	N	N	D			N	N
5							
6	B	B	N	N			B
7	B	B	N	N		B	

(a) (b)

Figure 7.3: (a) describes a tree made up of mixed 1 & 2 arity internal nodes with cartesian node references shown as node labels. Dotted lines indicate node references not sampled by the tree, grey nodes indictate dye positions, white nodes indicate background. (b) shows the first 7 rows and columns of the co-occurrence matrix for this tree, D indicates a dye match, B a background match and N, no match.

node reference system. For the first row in the matrix we determine if the first node in the tree, using our reference system, has a dye or background value. Then, for each associated matrix column, we determine whether this matches (both have dye or background values) for any of the other positions in the tree and record the match, or lack of, in the corresponding position in our matrix, i.e., row is determined by node under investigation, column for nodes to be matched. We repeat the process for each remaining row. A matrix element (r, c) will record if position r in the tree has the same value as that of position c. One of three values will be stored. A dye match (dye



(a) (b)

Figure 7.4: Plots of the mean relative frequency of co-occurrence of pairs of non-terminal alleles vs. generation for 2-ary (a) and mixed arity 1, 2, 3, 4 & 5 (b) programs of length 11. Population initialised as Figure 7.2.

is present in both positions), a background match (no dye was present in either position), or no match (dye was present in only one position). Note, diagonals in the matrix are ignored as we will always obtain a match. See Figure 7.3 for an example of co-occurrence matrix construction.

In Figure 7.4a we can see that for 2-ary trees initialised with dye at the root position we quickly move to values predicted by a diffusive process. Dye sits on the diagonal for the initial generation and hence is not recorded but then we apply crossover and after approximately 20 generations we have reached our theoretical proportions: $(1/7)^2 \approx 0.020408$ for dye matching, $(6/7)^2 \approx 0.73469$ for background matching, and $2(1/7)(6/7) \approx 0.24490$ for no match. The same is true for our mixed arity trees. For example, in Figure 7.4b, our population of 100,000 individuals was initialised with an average of 1,297,856.85 internal nodes, 100,000 of which where marked with dye, our theoretical value for dye co-occurrence is $(100,000/1,297,856.85)^2 \approx (0.07705)^2 \approx 0.00594$. Background matching is, therefore, $(1 - 0.07705)^2 \approx (0.92295)^2 \approx 0.85184$ and finally our no match value will be $2(0.07705)(0.92295) \approx 0.14223$. Each of these values is also

obtained within 10 to 20 generations. Similar results were also found for our 3-ary and 2, 2 & 3 mixed arity experiments.[4] See [Poli et al., 2002] for similar results for linear GP, i.e., 1-ary trees.

From both sets of experiments we can see a diffusive process is at work. Firstly, our expectation of an allele appearing at any node position in a tree will be determined solely by the proportion found in the overall population at the initial generation. Secondly, sub-tree swapping crossover will remove any correlations between alleles for node positions within a tree.

7.3 Shape Bias

There is one final aspect of sub-tree swapping crossover that we should analyse before we complete our picture: how we sample shapes within length classes. The length distribution described in Chapter 3 is derived from an expectation that all shapes will be sampled uniformly within length classes for a-ary trees. In Figure 7.5, we can indeed provide experimental evidence for 2-ary trees for our length classes chosen. However, looking at mixed arities, we can see that there is a distinct bias to sample certain shape classes more often than others within each length. It was found, however, (see Table 7.1 as an example) that shapes with the same arity histogram are sampled uniformly.

This bias for mixed arities is easily explained if we look at the dynamics of the proportion of primitives of each arity in the population. On average, sub-tree swapping crossover will replace as much as it removes; this also holds true for node arities. To illustrate, see Figure 7.6 as an example of how the proportion of primitives of each arity stays constant in a population when

[4]For all experiments tree lengths up to a maximum of 40 nodes were analysed, each showed similar results.

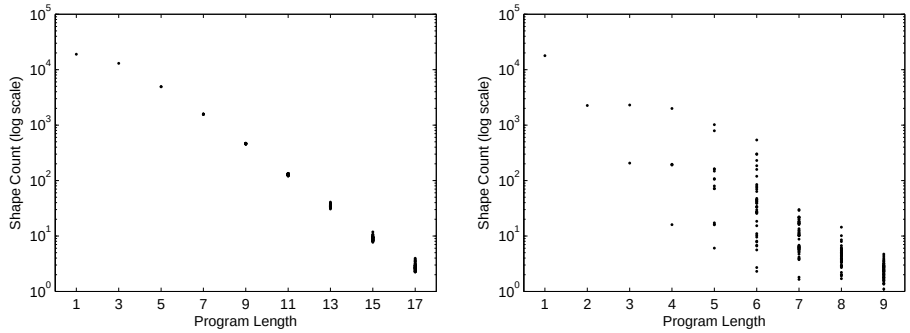
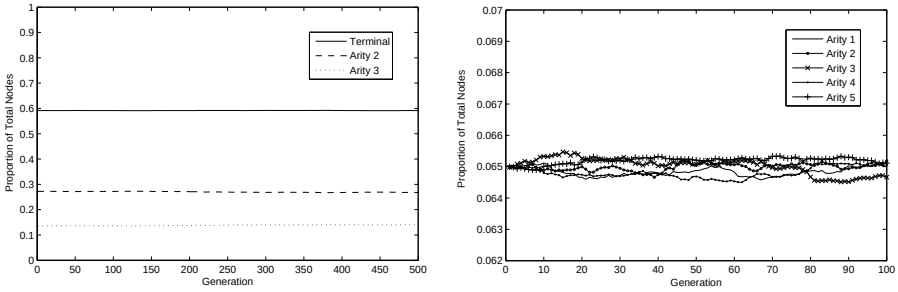


Figure 7.5: Scatter plots of (unique) shape counts by length for 2-ary (a) and mixed arity 1, 2, 3, 4 & 5 (b) programs, for the first 9 possible lengths at generation 500. Population initialised as Figure 7.2. Note, there are far more possible shapes for larger length classes. Also, these classes are sampled far less often than those of smaller lengths, hence the greater sampling noise.

sub-tree swapping crossover only is applied for our mixed arity experiments described earlier. There is, therefore, no bias to remove or resample certain higher or lower arities. So, not only does average size remain constant under repeated application of crossover, but also the proportions of each arity will remain constant within the population. Therefore, any (note, highly sampled) smaller shapes with an unequal proportion of arities, or those that can be produced using only a single arity, will modify those node arities available for other classes. We return to this in Chapter 8 where a hypothetical model is produced to predict exact length distributions from arity histograms.

7.4 Convergence

First suggested in [Poli et al., 2002], we can now provide strong evidence that GP's inability to structurally converge is caused primarily through sub-tree swapping crossover's bias to first distribute a population in terms



(a) (b)

Figure 7.6: Plots of the proportions of arities for each generation. (a) shows the first 500 generations for a population initialised with 2, 2 & 3 arities. (b) shows the first 100 generations of a population initialised with 1, 2, 3, 4 & 5 arities, note the highly reduced scale in this example. Due to the reduced scaling, terminals are not shown in (b) but follow a consistent proportion as shown in (a), in this case centering tightly around a proportion of 0.675. Populations initialised as in Figure 7.2.

of length and arity histogram and then to diffuse node labels within those classes. As fitness converges during the later stages of a run, crossover will make its program sampling bias emerge, hence, the processes described in this chapter and Chapter 3 will prevent any structural convergence taking place. No matter how strong the selection scheme, e.g., even if the mating pool was populated solely by copies of a single individual (say by using a tournament size equal to that of the population), the resulting child popula-

Table 7.1: Averaged counts at generation 500 for all program shapes for 2, 2 & 3 arity programs of length 7. Population initialised as in Figure 7.2.

S-Expression	Count
(2 0 (2 0 (2 0 0)))	407.10
(2 0 (2 (2 0 0) 0))	407.95
(2 (2 0 0) (2 0 0))	401.05
(2 (2 0 (2 0 0) 0))	404.25
(2 (2 (2 0 0) 0) 0)	410.40
(3 0 0 (3 0 0 0))	258.75
(3 0 (3 0 0 0) 0)	258.05
(3 (3 0 0 0) 0 0)	258.75

tion created by sub-tree swapping crossover would first contain individuals of differing lengths and secondly, node labels would be dispersed within those individuals.[5] This would not be true in a GA system using n-point crossover acting on traditional fixed length vector representations, as there is no opportunity to alter individual lengths or to move alleles to different locations.

Although GP using sub-tree swapping crossover will prevent convergence to a single syntactic structure, it will start to search within ever tighter bounds and begin to resample heavily smaller classes (see Chapter 5). With this in mind, we can suggest new stopping criteria for GP runs based on structural convergence. A very simple method would be to determine the undue influence of crossover by detecting if the ratio of smaller programs is higher than some threshold. An inexpensive resampling measure based on simple program hashes could also be used, possibly causing run termination when a program has been resampled a pre-specified number of times. Additionally, more sophisticated methods may look at the length distribution as a whole, i.e., a convergence to the model distribution, or in conjunction with fitness measures, such as a corresponding reduction in fitness variance.

7.5 Conclusions

This chapter has analysed the biases presented by GP sub-tree swapping crossover in relation to the primitives forming GP trees. Strong evidence has been presented to show that there is a diffusive process that takes place within length classes when sub-tree swapping crossover is repeatedly applied to a flat fitness landscape in the absence of selection. All node labels (alleles) are equally likely to be found within any possible node position for each

[5]Barring the unlikely situation where the same crossover points are chosen in all cases.

length class.

From our evidence, we now know that program shapes are highly likely to be uniformly sampled within arity histogram classes. a-ary trees are a special case in that there is only one arity histogram per length class. Hence, programs will be sampled uniformly within each length. This, however, is not true for mixed arities, where a more sophisticated process is taking place. The reasons for this lie within the constant population proportions of each arity during each generation and the highly sampled smaller programs with unequal arity proportions.

Although we now know that GP using sub-tree swapping crossover is highly unlikely to converge in terms of individual program structure, we do have an understanding of a broader, population based, form of structural convergence. This allows us to propose a set of convergence measures that, in the future, might be used for stopping conditions similar to those found in GA experimentation. Further research is required to establish the effectiveness of such measures.

In light of the empirical analysis provided here, in the next chapter a mathematical model similar to those described in Chapter 3 is presented. As with those models, this provides the probability of GP individual occurrence on a flat fitness landscape with the application of sub-tree swapping crossover. However, this model has been extended to consider exact proportions of internal node arities, i.e., arity histograms.

Chapter 8

Arity Histogram Distributions

8.1 Introduction

In Chapter 3, a number of models were proposed to predict a limiting distribution of GP tree sizes when sub-tree swapping crossover was applied on a flat fitness landscape. The limiting distribution of internal nodes for a-ary trees was shown to be a Lagrange distribution of the second kind. Strong empirical support was found for this model and a generalisation, still in terms of internal node counts, was obtained for mixed arity trees. A further generalisation to true length classes, i.e., to also include external nodes (leaves), was found to be successful for a-ary trees. The generalisation to true length classes for mixed-arity trees was found to be less successful, however, at smaller length classes.

In the previous chapter, we provided empirical evidence to suggest that the probability of the occurrence of an individual in a GP population after repeated application of sub-tree swapping crossover on a flat fitness land-

scape would be determined by the individual's arity histogram.[1] If we are to predict true length distributions for mixed arity cases, we will, therefore, have to take into account differing internal node arities in our models. For a-ary trees the arity histogram is, of course, simply the associated internal and external node counts, which explains our earlier success with the a-ary models.

In Section 8.2, we extend the reasoning from Chapter 3 and provide a model to predict individual occurrence using arity histograms. This is then extended to predict length class frequencies exactly. Strong empirical evidence is provided in Section 8.3 to support this model; in particular we show how the model can be successfully fitted to shorter length classes for mixed arity cases. In Section 8.4, we discuss the sampling implications of the model and its relationship to the work presented previously in this thesis. Finally, we summarise our findings in Section 8.5.

8.2 Arity Histogram Model

From the work in the previous chapter, we know that we wish to predict the occurrence of an individual with a particular arity histogram. If we choose n_a to represent a count of arity a nodes, we can define a particular arity histogram of an individual, as the tuple $(n_0, \ldots, n_{a_{max}})$. Note, n_0, is the number of leaves, i.e., nodes with an arity of zero. Using our new notation we can term our target probability, $\Pr\{n_0, \ldots, n_{a_{max}}\}$.

Below, we will attempt to identify this function by means of generalisation from previous results and intuition. The 'acid test' for the result of our generalisation will be whether or not it fits the empirical data in a variety of conditions.

[1] The set of counts for each arity.

Let us start by reviewing Equation 3.22, the original model for a-ary representations, shown below for reference:

$$\Pr\{n\} = (1 - ap_a)\binom{an+1}{n}(1-p_a)^{(a-1)n+1}\ p_a^n$$

We can see that in order to generalise it, we need to introduce the concept of multiple arities, particularly the associated p_a and n_a values.

First, we postulate that we now have a set of p_a values each associated with a single arity. If we interpret these as forming a probability distribution, we can then imagine that product ap_a in the first term of the equation, actually represents an 'expectation' of a.[2] If this is correct, then the first term $(1 - ap_a)$ should be changed to $(1 - \sum_{a\geq 1} ap_a)$.

The original binomial coefficient term represents the number of ways of choosing internal nodes of the same arity, a, from the length of the resulting tree, $an + 1$. We need to alter this by selecting each arity count, n_a, from the tree length that can be built with this collection of arities, $\sum_{a\geq 1} an_a + 1$. Our binomial coefficient term, therefore, becomes the multinomial coefficient $\binom{\sum_{a\geq 1} an_a+1}{n_0,\ldots,n_{amax}}$, where n_0 is the count of leaves, n_1 is the count of the functions with arity 1, etc.

The third term, $(1 - p_a)^{(a-1)n+1}$, can be broken into two parts. The superscript is simply the number of terminals for the tree, which we know to be n_0. As with the first term we alter $(1-p_a)$ to a mixed arity equivalent, which we postulate to be $(1 - \sum_{a\geq 1} p_a)$.

Continuing this analogy, the final term, p_a^n, represents the value, p_a, to the power of the number of nodes, n. We need to now split out the term so that each value of p_a is associated with the appropriate n_a value. The most natural way to do this is to turn the final term into the product $\prod_{a\geq 1} p_a^{n_a}$.

[2] In our a-ary model: $E[a] = 0 * (1 - p_a) + a * p_a = ap_a$.

Putting this altogether, our new, mixed arity model becomes:

$$\Pr\{n_0, \ldots, n_{a_{max}}\} = (1 - \sum_{a \geq 1} a p_a) \binom{\sum_{a \geq 1} a n_a + 1}{n_0, \ldots, n_{a_{max}}} (1 - \sum_{a \geq 1} p_a)^{n_0} \prod_{a \geq 1} p_a^{n_a} \tag{8.1}$$

Note, the introduction of counts for program leaves will only affect the second term and third terms. On closer inspection we can also see that there is in fact no need to calculate p_0.[3]

Next, we need to create a model that will turn arity histogram probabilities into those of length classes. The set of arity histograms that represent a particular program length ℓ can be defined as:

$$\left\{ n_0, \ldots, n_{a_{max}} : \sum_{a \geq 1} a n_a + 1 = \ell \right\} \tag{8.2}$$

We can, therefore, calculate the probability of a particular program length by summing the probabilities for each of the set of associated arity histograms, i.e.,

$$\Pr\{\ell\} = \sum_{n_0, \ldots, n_{a_{max}} : \sum_{a \geq 1} a n_a + 1 = \ell} \Pr\{n_0, \ldots, n_{a_{max}}\} \tag{8.3}$$

8.3 Empirical Validation

As in Sections 3.3 to 3.6, in order to verify empirically the distribution proposed, a number of runs of a GP system in Java was performed. A relatively large population of 100,000 individuals was used in order to reduce drift of average program size and to ensure that enough programs of each length class were available. The FULL initialisation method was used with

[3] If we define p_0 to be $1 - \sum_{a \geq 1} p_a$ and allow the fourth term to run from $a = 0$, we could also omit the third term.

non-terminals being chosen with uniform probability. Each run consisted of 500 generations. All results were averaged over 20 runs.

In order to confirm that our model still accurately predicts a-ary distributions, Figures 8.1 and 8.2 show the model and observed data from the final generation for 1-ary and 2-ary trees. p_a values for the model have been calculated using Equation 3.23.

It has not been possible to produce formulae to pre-determine each value of p_a for mixed arity representations. However, we can still fit the model to our experimental data. This fit was achieved using a hill climber search program (see Section 2.1) that reduced the mean squared error from that observed in the final generation and that predicted by the distribution, by altering the p_a values.

As we can see in Figure 8.3 and Figure 8.4 we can now fit our true length classes for mixed arities at earlier lengths. The model now not only captures both the smooth descending values for a-ary length distributions but also the fluctuating early values for mixed arity representations.

In essence, we now have evidence that we have isolated the fundamental components of the limiting length distribution for sub-tree swapping crossover. Further work is required to make this a predictive model, i.e., we need a formula to determine p_a values for mixed arity representations. However, we can now place the findings from earlier in this thesis into further context. This is discussed in the next section.

8.4 Sampling Implications

From our analysis we can now be confident in the assertion that the limiting distribution of program lengths for a GP population after the repeated application of sub-tree swapping crossover, is determined solely by the mix

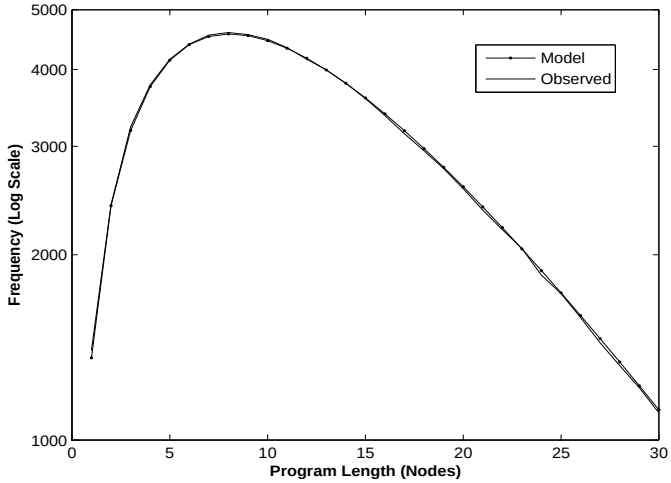


Figure 8.1: Comparison between empirical size distributions and an arity histogram model created with arity 1 functions and terminals only, initialised with FULL method (depth = 15, initial mean size $\mu_0 = 16.00$, mean size after 500 generations $\mu_{500} = 16.15$). Population size = 100,000.

of node arities in the initial population.

From the work provided in Chapter 7, we can see that there is no bias for sub-tree swapping crossover to place a particular node label at any position in a tree. All programs with a particular arity histogram are, therefore, equally likely to be sampled by the application of sub-tree swapping crossover in the absence of other variation operators. By extension, we can also say that all programs of a certain length are equally likely to be sampled for a-ary trees; this is not true, however, for mixed arity representations. If one wishes to ensure uniform sampling within length classes, alternative variation operators will need to be devised when mixed arity representations are employed. Using the operator equalisation method described in Chapter 6 does not guarantee uniform sampling within the length classes desired.[4] We could of course extend the method to sample uniformly within

[4]Although, interestingly, as we have chosen to use 2-ary representations we can be reasonably confident that the sampling was unbiased within length classes for the results

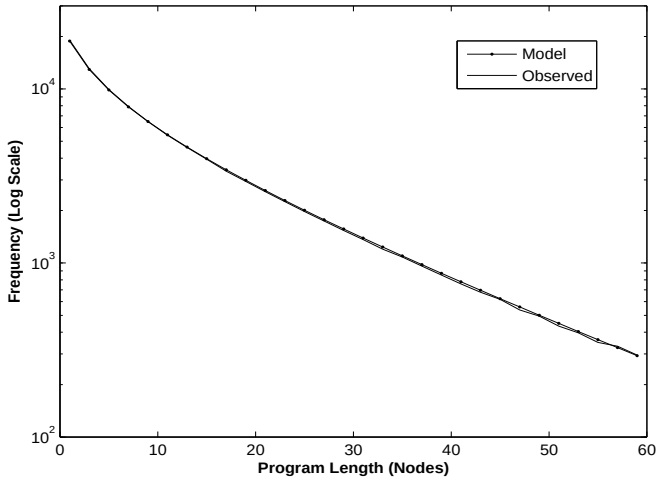


Figure 8.2: Comparison between empirical size distributions and an arity histogram model created with arity 2 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 15.00$, mean size after 500 generations $\mu_{500} = 14.19$). Invalid even lengths are ignored. Population size = 100,000.

length classes by storing a histogram of arity histograms, possibly using a hashing function as lengths increase.

Looking more closely at Equation 8.1, we can see that the first term will remain constant for all arity histograms whilst the second term, the multinomial coefficient, will increase the probability for arity histograms that can produce more shapes. The third and final terms decrease rapidly with increasing values of the n_a's producing the eventual smooth curve. Therefore, arity histograms presented to Equation 8.3, that can produce more shapes than other arity histograms in a particular length class, will have a higher probability of being sampled within that class.

Disregarding the fluctuations shown in earlier length classes for mixed arity classes, Equation 8.3 is decreasing. The crossover bias theory presented in Section 4.5 was originally proposed based upon evidence presented by the

presented in Chapter 6.

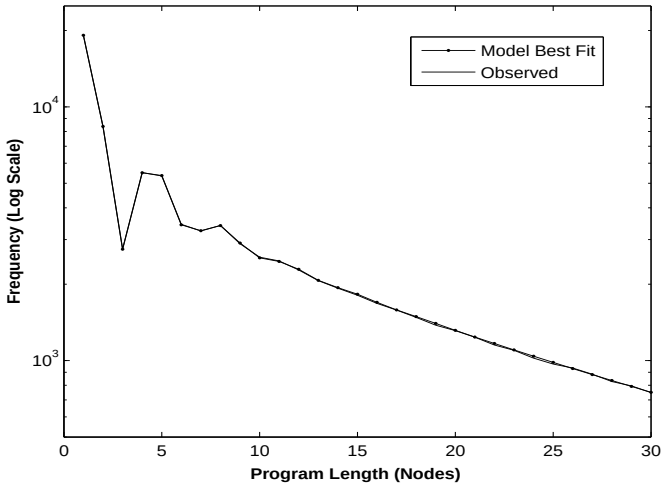


Figure 8.3: Comparison between empirical size distributions and an arity histogram model obtained by best fit for trees created with arity 1 and 3 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 15.00$, mean size after 500 generations $\mu_{500} = 15.75$). Population size = 100,000.

internal node count models ([Dignum and Poli, 2007] and Chapter 3) and their decreasing nature. Equation 8.3 and the empirical work provided in Section 8.3 provides extra evidence to support this theory in that our more pertinent true length model varies only slightly from the smooth descent described for the internal node models presented in Chapter 3.

The internal node and true length a-ary models presented in Chapter 3 can be used as predictive models without modification. The true length model for mixed arity trees, presented in Section 3.6 remains a strong model for approximation if an exact fit for earlier length classes is not required. One could use this to implement broad convergence measures suggested in Section 7.4, for example. If a more exact model was required, a fit to internal node counts could be used, albeit with a slight programming overhead.

The generalised mixed arity internal node model (Equation 3.27) is also

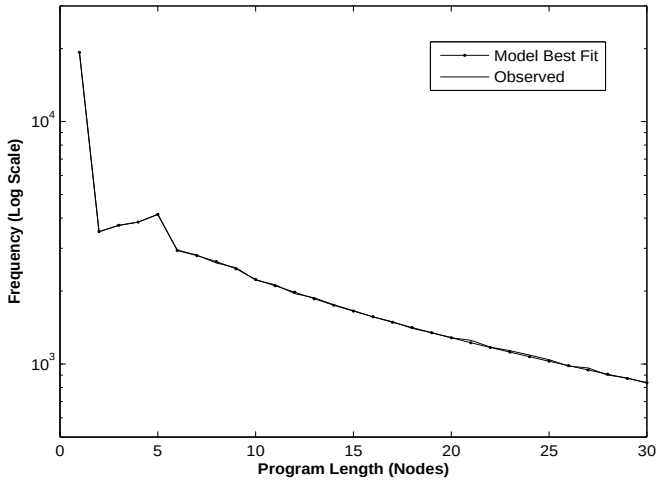


Figure 8.4: Comparison between empirical size distributions and an arity histogram model obtained by best fit for trees created with arities 1, 2, 3 and 4 functions and terminals only, initialised with FULL method (depth = 3, initial mean size $\mu_0 = 25.38$, mean size after 500 generations $\mu_{500} = 23.72$). Population size = 100,000.

an interesting starting point to further analyse the arity histogram model proposed here. We can ask how was such a generalised model so successful when only leaves were removed from the investigation? For example, would Equation 8.3 collapse to Equation 3.27 with further analysis? This is left to future work.

8.5 Conclusions

In light of the findings in Chapter 7, in this chapter we have extended our work from Chapter 3 to now model limiting length distributions for sub-tree swapping crossover using arity histograms as opposed to internal node counts and average arities. This extension has allowed us to accurately model not just the smooth descending curves of the internal node models but also those of the more rugged true length distributions, i.e., those that

also include leaves.

Although the model does not yet have the predictive power of the models presented in our earlier chapters, as we do not have formulae to determine appropriate p_a values for mixed arity representations, this model has isolated the fundamental components of sub-tree swapping crossover. From this, we can now place our earlier findings into further context and have more confidence in the assertions made.

Chapter 9

Conclusions

9.1 Contributions Made by this Thesis

In this section, we list and describe the contributions made by this thesis.

In Chapter 2, the concept of automatic programming was introduced, i.e., to enable a computer to build computer programs to a desired quality with minimal human intervention. Care was taken to portray GP as only one method that could be used to achieve such an aim. GP is an AI search technique that uses the analogy of Darwinian evolution to iteratively improve programs using a population of solutions, a selection method and a number of variation operators.

To understand the likelihood of success for GP, or any other search method, in discovering acceptable solutions to specific problems, or classes of problems, we need to understand the biases of their associated operators.

As part of this process, this thesis takes one operator, GP sub-tree swapping crossover, and analyses the biases inherent in its application. In light of *nature of program spaces* theories, particular attention is paid to the effect of its application on program length.

Chapter 3 provides a number of theoretical models (with strong empirical validation) that determine the limiting distribution of program sizes for populations under the repeated application of sub-tree swapping crossover. In effect this, therefore, reveals the bias of this operator in terms of program length sampling. This distribution was found to be a Lagrange distribution of the second kind. The distribution has two parameters: an average of internal node arities and the mean program size of the initial population. Under typical GP initialisations, these parameters produce a distribution that will sample ever decreasing program sizes.

In Chapter 4, we analysed how these models help us understand typical GP search. The bias to sample smaller programs combined with the combinatorial explosion of possible programs as length increases, makes it increasingly difficult for GP to sample unique programs at larger lengths. By increasing average program length in initial GP populations, hence altering one of the parameters of the Lagrange distribution, it is shown that we can reduce this bias. This chapter also looked at how initialisation can affect program growth in light of sub-tree swapping crossover bias. From this work, a new bloat theory has been produced called crossover-bias. This chapter also contains work regarding the effects of length limits on program sampling, showing that such limits can increase the bias to sample smaller programs, hence accelerating bloat (up to the limit).

On the basis of the program resampling results presented in Chapter 4, Chapter 5 introduced a novel technique to penalise resampling called sampling parsimony. This works by altering program fitness, after a pre-defined number of resamples, during a GP run. This was shown to have a direct affect on program growth which could be explained using the crossover-bias bloat theory.

To directly control the sampling of program lengths by GP, bypassing the biases exhibited by sub-tree swapping crossover or any other variation operator, Chapter 6 introduced another novel technique called operator equalisation. This took the form of a wrapper that can be placed around existing reproduction code which enables GP to sample program lengths according to pre-defined distributions. In a number of experiments, this was shown to provide important information regarding the nature of GP search in relation to the problems to which it was applied. It was also shown, that for certain problems, GP search could be improved by simple alteration of length bias.

In Chapter 7, we looked at sampling by sub-tree swapping crossover within length classes. Empirical evidence was provided to show that there was no bias for node labels to be placed at particular tree locations and that a diffusive process was taking place. For the a-ary tree representations we could, therefore, conclude that programs would be sampled by sub-tree swapping crossover uniformly within length classes. However, further evidence was provided to show that for mixed arity tree representations, programs with certain counts of node arities were found to be more likely to be sampled within length classes than others. Such arity counts were termed arity histograms.

In Chapter 8 we extended our theoretical length models to incorporate arity histograms. Strong empirical evidence was provided to support the model provided, particularly for the early length classes of mixed arity representations, an area that had previously been found difficult to model.

Finally, from the analyses provided in Chapters 3, 7 and 8 we can conclude that GP is highly unlikely to converge, structurally, to a single solution when sub-tree swapping crossover is applied. However, if fitness values can converge, a broader form of structural convergence will take place.

9.2 Future Work

Within this thesis, we have looked at proposing and validating theoretical limiting length distributions by comparison to observed data. Later observed generations have been used, typically 500, to ensure we have indeed reached a limit. As we now have confidence in these models, a further area of research is to determine how quickly a population will converge to such a distribution, i.e., to analyse conditions that may affect the strength of our biases. This has particular relevance to the alternative structural convergence measures described in Section 7.4, i.e., to allow us to determine how long we would need to wait to see evidence of such convergence. Whether such methods can improve the performance of GP runs, compared to existing GP stopping methods, remains an interesting area of research.

In [Galvan-Lopez et al., 2008] initial attempts have been made to look at length distributions when an artificial form of neutrality is applied. The ability for neutrality to help solve certain problems in GP may lie within its effect on the sampling of certain program sizes. Continuation of this work may shed further light onto the effects of neutrality.

Daida and others have looked at *structural difficulty* problems in GP [Daida and Hilss, 2003, Daida et al., 2003, Hoai et al., 2006]. In effect, certain tree structures are difficult to obtain using sub-tree swapping crossover, particularly an inability to produce very full or narrow trees. This thesis concludes that there is no shape bias for sub-tree swapping crossover, only a bias to sample arity histograms, their structures being sampled equally within histogram classes. A process involving both crossover and selection must be involved in such structural bias. The bloat theory presented in Section 4.5 was produced looking at the combined effects of GP operators; similar research, therefore, seems likely to produce an explanation of this

problem.

The operator equalisation method described in Chapter 6 has only currently been tested against static length distributions. There is no reason why the method cannot be enhanced to follow a schedule, i.e., to change at each generation the distribution desired. Another idea is to alter the desired distribution dynamically to explore current high fitness peaks. We may also wish to extend our method to explore length classes uniformly or to penalise resampling associated with smaller classes.

Finally, this work aims to provide an understanding of the biases of sub-tree swapping crossover. In order to achieve the wider aim of automatically matching search operators to problems we would need to develop a form of language to match their associated characteristics. Rather than simply matching an existing operator, one can foresee a generative language that could create search operators with required biases on demand.

9.3 Summary

This thesis has analysed the biases of sub-tree swapping crossover as used within typical GP experimentation. It has provided a number of models to show how, with repeated application, the operator will sample the program space. Particular attention has been given to the sampling of program lengths although the sampling of programs within those classes has also been addressed. From this work, important results have been provided regarding GP search in the areas of resampling, structural convergence and bloat. Practical experimental advice has also been produced notably relating to initilisation and the setting of size limits. In light of this research, the thesis has also introduced two novel additions to typical GP implementation, sampling parsimony and operator equalisation, both of which have shed fur-

ther light on GP search. The work also provides a starting point for further analysis of GP search and the categorisation of search operator biases.

Bibliography

[Angeline, 1996] Angeline, P. J. (1996). An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. In Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L., editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 21–29, Stanford University, CA, USA. MIT Press.

[Angeline and Pollack, 1992] Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Bloomington, Indiana, USA. Lawrence Erlbaum.

[Avery et al., 1944] Avery, O. T., MacLeod, C. M., and McCarty, M. (1944). Studies on the chemical nature of the substance inducing transformation of pneumococcal types. *Journal of Experimental Medicine*, (79):137–159.

[Bäck and Schwefel, 1993] Bäck, T. and Schwefel, H. (1993). An overview of evolutionary algorithms for parameter optimization. *Evolutionary Computing*, 1(1):1–23.

[Banzhaf et al., 2006] Banzhaf, W., Beslon, G., Christensen, S., Foster, J. A., Kps, F., Lefort, V., and Miller, J. F. (2006). From artificial evolution to computational evolution: A research agenda. *Nature Reviews Genetics*, (7):729–735.

[Banzhaf et al., 1998] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA.

[Beasley et al., 1993] Beasley, D., Bull, D., and Martin, R. (1993). An overview of genetic algorithms: Part 2, research topics. *University Computing*, 15:170–181.

[Callan, 2003] Callan, R. (2003). *Artificial Intelligence*. Palgrave Macmillan, Basingstoke, England.

[Cawsey, 1997] Cawsey, A. (1997). *The Essence of Artificial Intelligence*. Prentice Hall Europe, Harlow, England.

[Consul and Shenton, 1972] Consul, P. C. and Shenton, L. R. (1972). Use of lagrange expansion for generating discrete generalized probability distributions. *SIAM Journal on Applied Mathematics*, 23(2):239–248.

[Crane and McPhee, 2005] Crane, E. F. and McPhee, N. F. (2005). The effects of size and depth limits on tree based genetic programming. In Yu, T., Riolo, R. L., and Worzel, B., editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 15, pages 223–240. Springer, Ann Arbor.

[Crawford-Marks and Spector, 2002] Crawford-Marks, R. and Spector, L. (2002). Size control via size fair genetic operators in the PushGP genetic programming system. In Langdon, W. B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 733–739, New York. Morgan Kaufmann Publishers.

[Daida and Hilss, 2003] Daida, J. M. and Hilss, A. M. (2003). Identifying structural mechanisms in standard genetic programming. In Cantú-Paz, E., Foster, J. A., Deb, K., Davis, D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M. A., Schultz, A. C., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1639–1651, Chicago. Springer-Verlag.

[Daida et al., 2003] Daida, J. M., Li, H., Tang, R., and Hilss, A. M. (2003). What makes a problem GP-hard? validating a hypothesis of structural causes. In Cantú-Paz, E., Foster, J. A., Deb, K., Davis, D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M. A., Schultz, A. C., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1665–1677, Chicago. Springer-Verlag.

[Darwin, 1859] Darwin, C. (1859). *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. John Murray, London.

[Davis et al., 1993] Davis, R., Shrobe, H. E., and Szolovits, P. (1993). What is a knowledge representation? *AI Magazine*, 14(1):17–33.

[Dawkins, 2006a] Dawkins, R. (2006a). *The Blind Watchmaker*. Penguin Books, London, England, fourth edition.

[Dawkins, 2006b] Dawkins, R. (2006b). *The Selfish Gene*. Oxford University Press, Oxford, England, 30th anniversary edition.

[D'haeseleer, 1994] D'haeseleer, P. (1994). Context preserving crossover in genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 256–261, Orlando, Florida, USA. IEEE Press.

[Dignum and Poli, 2007] Dignum, S. and Poli, R. (2007). Generalisation of the limiting distribution of program sizes in tree-based genetic programming and analysis of its effects on bloat. In Thierens, D., Beyer, H.-G., Bongard, J., Branke, J., Clark, J. A., Cliff, D., Congdon, C. B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K. O., Stutzle, T., Watson, R. A., and Wegener, I., editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1588–1595, London. ACM Press.

[Dignum and Poli, 2008] Dignum, S. and Poli, R. (2008). Crossover, sampling, bloat and the harmful effects of size limits. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E., editors, *Proceedings of the 11th European Conference on Genetic Programming*, volume 4971 of *LNCS*, pages 158–169, Naples. Springer.

[Eiben and Smith, 2003] Eiben, A. E. and Smith, J. E. (2003). *Introduction to Evolutionary Computing*. Springer.

[Felsenstein, 1974] Felsenstein, J. (1974). The evolutionary advantage of recombination. *Genetics*, (78):737–756.

[Fogel et al., 1966] Fogel, L. J., Owens, A. J., and Walsh, M. J. (1966). *Artificial intelligence through simulated evolution*. John Wiley and Sons, New York.

[Freitas, 2002] Freitas, A. (2002). *Data Mining and Knowledge Discovery with Evolutionary Algorithms*. Springer-Verlag.

[Galvan-Lopez et al., 2008] Galvan-Lopez, E., Dignum, S., and Poli, R. (2008). The effects of constant neutrality on performance and problem hardness in GP. In O'Neill, M., Vanneschi, L., Gustafson, S., Esparcia Alcazar, A. I., De Falco, I., Della Cioppa, A., and Tarantino, E., editors, *Proceedings of the 11th European Conference on Genetic Programming*, volume 4971 of *LNCS*, pages 312–324, Naples. Springer.

[Ghahramani, 1996] Ghahramani, S. (1996). *Fundamentals of Probability*. Prentice-Hall Inc, Upper Saddle River, NJ 07458.

[Good, 1975] Good, I. J. (1975). The lagrange distributions and branching processes. *SIAM Journal on Applied Mathematics*, 28(2):270–275.

[Haccou et al., 2005] Haccou, P., Jagers, P., and Vatutin, V. A. (2005). *Branching Processes: Variation, Growth, and Extinction of Populations*. Cambridge University Press, United Kingdom.

[Hilton and Pederson, 1991] Hilton, P. and Pederson, J. (1991). Catalan numbers, their generalization, and their uses. *Mathematical Intelligencer*, 13:64–75.

[Hoai et al., 2006] Hoai, N. X., McKay, R. I. B., and Essam, D. (2006). Representation and structural difficulty in genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):157–166.

[Holland, 1975] Holland, J. H. (1975). *Adaptation in Natural and Artificial Systems*. The University of Michigan Press, Ann Arbor.

[Iba, 1996] Iba, H. (1996). Random tree generation for genetic programming. In Voigt, H.-M., Ebeling, W., Rechenberg, I., and Schwefel, H.-P., editors, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 144–153, Berlin, Germany. Springer Verlag.

[Jackson, 1999] Jackson, P. (1999). *Introduction to Expert Systems*. Addison Wesley, Harlow, England.

[Janardan, 1987] Janardan, K. (1987). Weighted lagrange distributions and their characterizations. *SIAM Journal on Applied Mathematics*, 47(2):411–415.

[Janardan and Rao, 1983] Janardan, K. and Rao, B. (1983). Lagrange distributions of the second kind and weighted distributions. *SIAM Journal on Applied Mathematics*, 43(2):302–313.

[Jones, 2000] Jones, S. (2000). *The Language of the Genes*. Flamingo, Harper Collins, London, England, second revised edition.

[Jurafsky and Martin, 2000] Jurafsky, D. and Martin, J. H. (2000). *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition*. Pearson Education, Harlow, England.

[Keijzer and Foster, 2007] Keijzer, M. and Foster, J. (2007). Crossover bias in genetic programming. In Ebner, M., O'Neill, M., Ekárt, A., Vanneschi,

L., and Esparcia-Alcázar, A. I., editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 33–43, Valencia, Spain. Springer.

[Kirkpatrick et al., 1983] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598):671–680.

[Koza, 1992] Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

[Koza, 1994] Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts.

[Koza et al., 2003] Koza, J. R., Keane, M. A., Streeter, M. J., Mydlowec, W., Yu, J., and Lanza, G. (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

[Langdon, 2000] Langdon, W. B. (2000). Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119.

[Langdon, 2002a] Langdon, W. B. (2002a). Convergence rates for the distribution of program outputs. In Langdon, W. B., Cantú-Paz, E., Mathias, K., Roy, R., Davis, D., Poli, R., Balakrishnan, K., Honavar, V., Rudolph, G., Wegener, J., Bull, L., Potter, M. A., Schultz, A. C., Miller, J. F., Burke, E., and Jonoska, N., editors, *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 812–819, New York. Morgan Kaufmann Publishers.

[Langdon, 2002b] Langdon, W. B. (2002b). How many good programs are there? How long are they? In De Jong, K. A., Poli, R., and Rowe, J. E., editors, *Foundations of Genetic Algorithms VII*, pages 183–202, Torremolinos, Spain. Morgan Kaufmann. Published 2003.

[Langdon, 2003] Langdon, W. B. (2003). Convergence of program fitness landscapes. In Cantú-Paz, E., Foster, J. A., Deb, K., Davis, D., Roy, R., O'Reilly, U.-M., Beyer, H.-G., Standish, R., Kendall, G., Wilson, S., Harman, M., Wegener, J., Dasgupta, D., Potter, M. A., Schultz, A. C., Dowsland, K., Jonoska, N., and Miller, J., editors, *Genetic and Evolutionary Computation – GECCO-2003*, volume 2724 of *LNCS*, pages 1702–1714, Chicago. Springer-Verlag.

[Langdon and Poli, 2002] Langdon, W. B. and Poli, R. (2002). *Foundations of Genetic Programming*. Springer-Verlag.

[Langdon and Poli, 2006] Langdon, W. B. and Poli, R. (2006). On turing complete T7 and MISC F–4 program fitness landscapes. In Arnold, D. V., Jansen, T., Vose, M. D., and Rowe, J. E., editors, *Theory of Evolutionary Algorithms*, number 06061 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. <http://drops.dagstuhl.de/opus/volltexte/2006/595> [date of citation: 2006-01-01].

[Langdon et al., 1999] Langdon, W. B., Soule, T., Poli, R., and Foster, J. A. (1999). The evolution of size and shape. In Spector, L., Langdon, W. B., O'Reilly, U.-M., and Angeline, P. J., editors, *Advances in Genetic Programming 3*, chapter 8, pages 163–190. MIT Press, Cambridge, MA, USA.

[Luke, 2000] Luke, S. (2000). Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283.

[Luke, 2008] Luke, S. (2008). ECJ: A Java-based Evolutionary Computation Research System. http://cs.gmu.edu/ eclab/projects/ecj/.

[Luke and Panait, 2006] Luke, S. and Panait, L. (2006). A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344.

[Majeed and Ryan, 2007] Majeed, H. and Ryan, C. (2007). On the constructiveness of context-aware crossover. In Thierens, D., Beyer, H.-G., Bongard, J., Branke, J., Clark, J. A., Cliff, D., Congdon, C. B., Deb, K., Doerr, B., Kovacs, T., Kumar, S., Miller, J. F., Moore, J., Neumann, F., Pelikan, M., Poli, R., Sastry, K., Stanley, K. O., Stutzle, T., Watson, R. A., and Wegener, I., editors, *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, volume 2, pages 1659–1666, London. ACM Press.

[McPhee and Miller, 1995] McPhee, N. F. and Miller, J. D. (1995). Accurate replication in genetic programming. In Eshelman, L., editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 303–309, Pittsburgh, PA, USA. Morgan Kaufmann.

[McPhee et al., 2001] McPhee, N. F., Poli, R., and Rowe, J. E. (2001). A schema theory analysis of mutation size biases in genetic programming with linear representations. In *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, pages 1078–1085, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea. IEEE Press.

[Michel, 2001] Michel, O. (2001). Evolutionary neurogenesis applied to mobile robotics. In M. Patel, V. Honavar, K. B., editor, *Advances in the*

Evolutionary Synthesis of Intelligent Agents, chapter 7, pages 185–213. MIT Press, Cambridge, MA, USA.

[Miller, 1999] Miller, J. F. (1999). An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In Banzhaf, W., Daida, J., Eiben, A. E., Garzon, M. H., Honavar, V., Jakiela, M., and Smith, R. E., editors, *Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1135–1142, Orlando, Florida, USA. Morgan Kaufmann.

[Mitchell, 1997] Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill International Editions, Singapore.

[Montana, 1995] Montana, D. J. (1995). Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230.

[Nordin, 1994] Nordin, P. (1994). A compiling genetic programming system that directly manipulates the machine code. In Kinnear, Jr., K. E., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press.

[O'Reilly and Oppacher, 1994] O'Reilly, U.-M. and Oppacher, F. (1994). Program search with a hierarchical variable length representation: Genetic programming, simulated annealing and hill climbing. In Davidor, Y., Schwefel, H.-P., and Manner, R., editors, *Parallel Problem Solving from Nature – PPSN III*, number 866 in Lecture Notes in Computer Science, pages 397–406, Jerusalem. Springer-Verlag.

[O'Reilly and Oppacher, 1995] O'Reilly, U.-M. and Oppacher, F. (1995). Hybridized crossover-based search techniques for program discovery. In *Proceedings of the 1995 World Conference on Evolutionary Computation*, volume 2, pages 573–578, Perth, Australia. IEEE Press.

[Poli, 1999] Poli, R. (1999). Parallel distributed genetic programming. In Corne, D., Dorigo, M., and Glover, F., editors, *New Ideas in Optimization*, Advanced Topics in Computer Science, chapter 27, pages 403–431. McGraw-Hill, Maidenhead, Berkshire, England.

[Poli, 2003] Poli, R. (2003). A simple but theoretically-motivated method to control bloat in genetic programming. In Ryan, C., Soule, T., Keijzer, M., Tsang, E., Poli, R., and Costa, E., editors, *Genetic Programming, Proceedings of EuroGP'2003*, volume 2610 of *LNCS*, pages 204–217, Essex. Springer-Verlag.

[Poli and Langdon, 1998a] Poli, R. and Langdon, W. B. (1998a). On the search properties of different crossover operators in genetic programming. In Koza, J. R., Banzhaf, W., Chellapilla, K., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M. H., Goldberg, D. E., Iba, H., and Riolo, R., editors,

Genetic Programming 1998: Proceedings of the Third Annual Conference, pages 293–301, University of Wisconsin, Madison, Wisconsin, USA. Morgan Kaufmann.

[Poli and Langdon, 1998b] Poli, R. and Langdon, W. B. (1998b). Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252.

[Poli et al., 2007] Poli, R., Langdon, W. B., and Dignum, S. (2007). On the limiting distribution of program sizes in tree-based genetic programming. In Ebner, M., O'Neill, M., Ekárt, A., Vanneschi, L., and Esparcia-Alcázar, A. I., editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 193–204, Valencia, Spain. Springer.

[Poli et al., 2008a] Poli, R., Langdon, W. B., and McPhee, N. F. (2008a). *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`. (With contributions by J. R. Koza).

[Poli and McPhee, 2008] Poli, R. and McPhee, N. (2008). Parsimony pressure made easy. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA. ACM Press. forthcoming.

[Poli and McPhee, 2001] Poli, R. and McPhee, N. F. (2001). Exact schema theorems for GP with one-point and standard crossover operating on linear structures and their application to the study of the evolution of size. In Miller, J. F., Tomassini, M., Lanzi, P. L., Ryan, C., Tettamanzi, A. G. B., and Langdon, W. B., editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 126–142, Lake Como, Italy. Springer-Verlag.

[Poli and McPhee, 2003a] Poli, R. and McPhee, N. F. (2003a). General schema theory for genetic programming with subtree-swapping crossover: Part I. *Evolutionary Computation*, 11(1):53–66.

[Poli and McPhee, 2003b] Poli, R. and McPhee, N. F. (2003b). General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206.

[Poli et al., 2008b] Poli, R., McPhee, N. F., and Vanneschi, L. (2008b). The impact of population size on code growth in GP: Analysis and empirical validation. In *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Atlanta, Georgia, USA. ACM Press. forthcoming.

[Poli et al., 2002] Poli, R., Rowe, J. E., Stephens, C. R., and Wright, A. H. (2002). Allele diffusion in linear genetic programming and variable-length genetic algorithms with subtree crossover. In Foster, J. A., Lutton, E., Miller, J., Ryan, C., and Tettamanzi, A. G. B., editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 212–227, Kinsale, Ireland. Springer-Verlag.

[Price, 1970] Price, G. R. (1970). Selection and covariance. *Nature*, (227):520–521.

[Rechenberg, 1973] Rechenberg, I. (1973). *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Evolution (PhD Thesis)*. Fromman-Holzboog, Stuttgart.

[Ridley, 1993] Ridley, M. (1993). *Evolution*. Blackwell Scientific Publications, Boston.

[Ridley, 1994] Ridley, M. (1994). *The Red Queen: Sex and the Evolution of Human Nature*. Penguin Books, London, England.

[Rosca and Ballard, 1996] Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming. In Angeline, P. J. and Kinnear, Jr., K. E., editors, *Advances in Genetic Programming 2*, chapter 9, pages 177–202. MIT Press, Cambridge, MA, USA.

[Rosenfeld and Kak, 1982] Rosenfeld, A. and Kak, A. C. (1982). *Digital Picture Processing, Vol. 1 and 2*. Academic Press, New York.

[Rowe and McPhee, 2001] Rowe, J. E. and McPhee, N. F. (2001). The effects of crossover and mutation operators on variable length linear structures. Technical Report CSRP-01-7, University of Birmingham, School of Computer Science.

[Russell and Norvig, 2003] Russell, S. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 2nd edition.

[Schwefel, 2000] Schwefel, H.-P. (2000). Advantages (and disadvantages) of evolutionary computation over other approaches. In Baeck, T., Fogel, D. B., and Michalewicz, Z., editors, *Evolutionary Computation 1 Basic Algorithms and Operators*, chapter 3, pages 20–22. Institute of Physics Publishing, Bristol.

[Silver, 2007] Silver, L. (October, 7, 2007). The year of miracles: Biology reborn. *Newsweek International*.

[Smart et al., 2007] Smart, W., Andreae, P., and Zhang, M. (2007). Empirical analysis of GP tree-fragments. In Ebner, M., O'Neill, M., Ekárt, A.,

Vanneschi, L., and Esparcia-Alcázar, A. I., editors, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *Lecture Notes in Computer Science*, pages 55–67, Valencia, Spain. Springer.

[Snyder and Qi, 2004] Snyder, W. E. and Qi, H. (2004). *Machine Vision*. Cambridge University Press, Cambridge, England.

[Soule and Foster, 1997] Soule, T. and Foster, J. A. (1997). Code size and depth flows in genetic programming. In Koza, J. R., Deb, K., Dorigo, M., Fogel, D. B., Garzon, M., Iba, H., and Riolo, R. L., editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 313–320, Stanford University, CA, USA. Morgan Kaufmann.

[Soule and Foster, 1998a] Soule, T. and Foster, J. A. (1998a). Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation*, 6(4):293–309.

[Soule and Foster, 1998b] Soule, T. and Foster, J. A. (1998b). Removal bias: a new cause of code growth in tree based evolutionary programming. In *1998 IEEE International Conference on Evolutionary Computation*, pages 781–186, Anchorage, Alaska, USA. IEEE Press.

[Tackett, 1995] Tackett, W. A. (1995). Mining the genetic program. *IEEE Expert*, 10(3):28–38.

[Teller and Veloso, 1996] Teller, A. and Veloso, M. (1996). PADO: A new learning architecture for object recognition. In Ikeuchi, K. and Veloso, M., editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press.

[Truss, 1999] Truss, J. K. (1999). *Discrete Mathematics for Computer Scientists*. Addison Wesley, Harlow, England.

[Watson and Galton, 1875] Watson, H. W. and Galton, F. (1875). On the probability of the extinction of families. *The Journal of the Anthropological Institute of Great Britain and Ireland*, 4:138–144.

[Watson and Crick, 1953] Watson, J. D. and Crick, F. H. C. (1953). A structure for deoxyribose nucleic acid. *Nature*, (171):737–738.

[Wolpert and Macready, 1997] Wolpert, D. H. and Macready, W. G. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

[XMLCoordinationGroup, 2008] XMLCoordinationGroup (2008). Extensible Markup Language (XML). http://www.w3.org/XML/.

[Zhang and Mühlenbein, 1993] Zhang, B.-T. and Mühlenbein, H. (1993). Evolving optimal neural networks using genetic algorithms with Occam's razor. *Complex Systems*, 7:199–220.

[Zhang and Mühlenbein, 1995] Zhang, B.-T. and Mühlenbein, H. (1995). Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation*, 3(1):17–38.

[Zhang et al., 1997] Zhang, B.-T., Ohm, P., and Mühlenbein, H. (1997). Evolutionary induction of sparse neural trees. *Evolutionary Computation*, 5(2):213–236.