Numerical Simplification and its Effect on Fragment Distributions in Genetic Programming

by

Alan 'David' Kinzett

A thesis
submitted to the Victoria University of Wellington
in partial fulfilment of the
requirements for the degree of
Master of Science
in Computer Science.

Victoria University of Wellington
2011

Abstract

In tree-based genetic programming (GP) there is a tendency for the program trees to increase in size from one generation to the next. If this increase in program size is not accompanied by an improvement in fitness then this unproductive increase is known as bloat. It is standard practice to place some form of control on program size. This can be done by limiting the number of nodes or the depth of the program trees, or by adding a component to the fitness function that rewards smaller programs (parsimony pressure) or by simplifying individual programs using algebraic methods. This thesis proposes a novel program simplification method called numerical simplification that uses only the range of values the nodes take during fitness evaluation.

The effect of online program simplification, both algebraic and numerical, on program size and resource usage is examined. This thesis also examines the distribution of program fragments within a genetic programming population and how this is changed by using simplification.

It is shown that both simplification approaches result in reductions in average program size, memory used and computation time and that numerical simplification performs at least as well as algebraic simplification, and in some cases will outperform algebraic simplification. This reduction in program size and the resources required to process the GP run come without any significant reduction in accuracy. It is also shown that although the two online simplification methods destroy some existing program fragments, they generate new fragments during evolution, which compensates for any negative effects from the disruption of existing fragments.

It is also shown that, after the first few generations, the rate new fragments are created, the rate fragments are lost from the population, and the number of distinct (different) fragments in the population remain within a very narrow range of values for the remainder of the run.

Acknowledgements

A special thank you to my supervisor Mengjie Zhang for his support, feedback and tireless work on behalf of the ECRG group. Thanks also to Mark Johnston for his unfailingly cheerful support and advice, and for preventing me from straying too far from the statistical straight and narrow.

Thank you to my employer for giving me the chance to study part-time, and especially to my wife Glynne who has had to put up with a lot these last five years.

I also wish to thank the school and faculty in general for making the task of returning to university as a mature student, after many years absence, as easy and as pain-free as possible.

My conference attendance was supported by the Marsden Fund council from the government funding (08-VUW-014), administrated by the Royal Society of New Zealand.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Genetic Programming

Genetic programming (GP) [30, 31, 33, 52] is a method for automatically generating programs to solve a given task. It is based on the Darwinian ideas of evolution and survival of the fittest. This is accomplished by taking an initial population of programs, evaluating the *fitness* (goodness) of each program, then producing the next generation by various operations where the chance of a program contributing to the next generation is dependent on that program's fitness.

There are various forms of GP, but this thesis concerns itself with the most common form, *tree-based* GP. In this form, the programs are tree structures and can be expressed as LISP[1] expressions. The nodes of the tree each take a single floating point value. The leaf nodes are either feature values from the input data, or ephemeral constants. These constants are assigned a random value when they are created and then remain unchanged. The internal nodes are operators such as multiplication or addition. GP has been successfully applied to many tasks, particularly symbolic regression [31] and both binary [55, 62] and multi-class [35, 61] classification.

[1] LISP is a symbolic programming language often used for AI

Because the tree structured program is not constrained in either shape or size, the program can evolve into whatever form is required to solve the task. This is a great strength of GP compared with many other evolutionary methods which require the form of the solution to be known before hand. This strength also contains a weakness. The programs can, and usually do, continue to grow in size as the run proceeds through the generations. If this increase in size does not bring an improvement in fitness then this unwanted growth is called *bloat* [3, 64].

Bloat causes a number of problems. It increases the memory required to store the population of programs and if the memory required becomes too large it may prevent the run completing. This limits the maximum population size that a particular hardware environment can support, and population size often has a critical influence on the success or otherwise of a GP run. Bloat increases the amount of CPU time required to calculate the program's value and therefore the time taken to evaluate the program's fitness. Fitness evaluation is usually the most time consuming part of a GP run so this is an important issue. If the amount of code in the program becomes much larger than that required for a good solution, there is a danger that the program may start to overfit the training data. This may cause misclassifications in classification tasks, and is a particular problem with symbolic regression tasks, when testing on unseen examples.

There have been a number of approaches used to control or prevent bloat. A hard limit can be placed on program size, either in number of nodes or in depth [31]. This will prevent bloat, but because the program is no longer free to take any size or shape it may inhibit or prevent the solution taking the form required for a good solution. The evolution process can be manipulated to disadvantage large programs, either by penalising fitness [31, 45, 81] (parsimony pressure), or by changing the selection process as in the *double tournament* method [36]. The *tarpeian* method [51] removes randomly selected over-large programs from the population. Another approach is to try and prevent bloat using modified genetic opera-

tors [32, 46, 3, 1]. The programs can also be simplified by removing redundant subtrees. A rules based approach [22] has been proposed. The motivation was to prevent overfitting. This objective was achieved but the approach was too expensive computationally to show any reduction in resource usage. *Algebraic simplification* [86] is a simpler approach using the standard algebraic rules and has been shown to reduce bloat and resource usage without significant reduction in accuracy, but the algorithm is fairly complex and not easy to use.

1.2 Goals

This thesis proposes a new method for online simplification of programs in tree based GP that is simpler to use than algebraic simplification with the goal of significantly reducing program sizes and computational cost without deterioration in the effectiveness[2]. This method is called *numerical simplification*.

Most datasets include a noise component. This may be the result of restricted measurement accuracy, or because of physical phenomena such as thermal noise, or the inclusion of data that does not assist in forming a good solution. The inspiration for numerical simplification came from the idea of treating two values or sets of values whose differences are within the noise level of the input data as being equal for the purpose of deciding if simplification should occur.

This thesis examines the performance and behaviour of numerical simplification and compares it to both canonical (standard) GP and the *algebraic simplification* method [86].

[2]This thesis uses the term *effectiveness* to mean the root mean square (RMS) error for a regression problem and the classification error rate on the test set for classification problems.

The specific questions examined are:

1. Can numerical simplification help control bloat, thereby reducing resource requirements without significantly reducing the effectiveness of the GP search? This question is broken down into the following sub-questions and is answered in chapter 4.

    - Will the numerical simplification method produce smaller programs than canonical (standard) GP with no simplification?
    - Will the numerical simplification method reduce memory usage and shorten computation run times compared to canonical GP with no simplification?
    - Will the numerical simplification method affect the system effectiveness?
    - How does numerical simplification perform compared to algebraic simplification?

2. Is there a relationship between the optimum value for the simplification threshold[3] in numerical simplification and the amount of noise in the input data? This question is broken down into the following sub-questions and is answered in chapter 5.

    - Is there an optimal value for the simplification threshold?
    - If there is an optimal value for the simplification threshold, is there a relationship between that optimal value and the amount of noise in the input data?
    - If this relationship exists then what is it?

[3]The simplification threshold is the maximum difference that can be treated as zero for simplification purposes.

3. Do the numerical and algebraic simplification methods change the pattern of fragments[4] within the GP population? This question is broken down into the following sub-questions:

   - How are the fragments are spread through the population in the two simplification methods and canonical GP with no simplification?
   - Do the two simplification methods destroy existing fragments?
   - Do the two simplification methods generate new fragments?
   - Do the more general fragments analysed in this thesis behave differently from the simple fragments (numerical/"constant" terminals only) analysed by Wong [78], and if so then how do they differ.

   These are answered in chapter 6 using images to show the distribution of fragments within the population, and within the search space, as the GP run progresses through the generations. These images show the distribution of fragments for single GP runs. Chapter 7 uses statistical methods to examine these questions for sets of GP runs.

1.3 Major Contributions

This thesis makes the following major contributions to Genetic Programming:

1. Numerical simplification, a new method for online program simplification is proposed. This uses numerical significance to drive decisions about program simplification during the evolutionary process. The results on a number of classification and regression tasks

[4]This thesis uses the term fragment for a subtree of fixed depth that occurs at some point within the larger tree structure that is the program.

show that this method significantly reduces program sizes and consequently memory usage and CPU run times without significant loss of accuracy.

Part of this work has been published in:

- David Kinzett, Mengjie Zhang and Mark Johnston. "Using Numerical Simplification to Control Bloat in Genetic Programming" *Proceedings of the 7th International Conference on Simulated Evolution And Learning (SEAL '08). Lecture Notes in Computer Science.*, **Vol. 5361**, Springer 2008. pp. 493–502.
- David Kinzett, Mark Johnston and Mengjie Zhang. "Numerical simplification for bloat control and analysis of building blocks in genetic programming". *(Journal of) Evolutionary Intelligence*, **Volume 2**, Number 4, pp. 151–168; DOI 10.1007/s12065-009-0029-9. Springer Berlin/Heidelberg Dec 2009

2. This thesis postulates that the optimal value for the simplification threshold will be related to and of the order of the noise amplitude in the input data. The results on two regression tasks show that while there is no clear optimum value for the threshold, there is a threshold above which the error rises sharply. The simplification threshold where this increase starts is between 3.5 and 10.0 times the maximum amplitude of the noise component in the input data.

   Part of this work has been published in:

   - David Kinzett, Mengjie Zhang and Mark Johnston. "Investigation of Simplification Threshold and Noise Level of Input Data in Numerical Simplification of Genetic Programs". *Proceedings of 2010 IEEE Congress on Evolutionary Computation.* Barcelona, Spain. 2010. IEEE. pp. 3065–3072.

3. A visualisation scheme is introduced that allows the distributions of program fragments up to three levels deep to be qualitatively assessed. This is achieved by encoding the program fragment into an integer, such that similar fragments generally encode to similar integers. These encodings are used to create images to show the distribution of fragments for individual GP runs, and how these distributions change through the evolutionary process. This thesis is able to use these images to show that *numerical simplification* and *algebraic simplification* make little or no qualitative change to the distribution of fragments.

   Part of this work has been published in:

   - David Kinzett, Mark Johnston and Mengjie Zhang. "How Online Simplification Affects Building Blocks in Genetic Programming" Proceedings of the 11th Annual conference on Genetic and Evolutionary Computation (GECCO 2009), ACM Press, 2009, pp. 979–986. ISBN:978-1-60558-325-9

4. A third encoding scheme is introduced that allows three level deep fragments to be described more precisely than the two encoding schemes used in the qualitative examination. This encoding scheme is used with statistical methods to confirm the results obtained using the visualisation scheme.

   The results also show that after the first few generations the rate at which new fragments are added to, or removed from, the population is nearly constant. The number of distinct fragments present in the population at any generation in the run is also nearly constant. *Numerical simplification* reduces the number of distinct fragments in the population thus in some sense reducing the overall diversity within the population but, at least for the datasets used in these experiments, this loss of diversity has not adversely affected the accuracy.

Part of this work has been published in:

- David Kinzett, Mengjie Zhang and Mark Johnston. "Analysis of Building Blocks with Numerical Simplification in Genetic Programming". *Proceedings of the 13th European Conference on Genetic Programming*, Istanbul, Turkey. Springer 2010. pp. 289–300.

1.4 Structure

The remainder of this document is structured as follows.

Chapter 2 provides background information for the work presented in this thesis. This chapter includes a full description of the *algebraic simplification* which is used for comparison purposes throughout much of this work. The experimental setup and the datasets used for the experiments are described in chapter 3.

A full description of *numerical simplification* is presented in chapter 4 along with experimental results comparing this method with both *algebraic simplification* and canonical GP. These results show the effect on program size and shape, resource usage, and the effectiveness of the GP run.

Chapter 5 presents an investigation of the relationship between the optimum value for the *simplification threshold* and the level of random noise in the input data.

The next two chapters present an investigation of the effect on fragment distribution. The first of these, chapter 6, presents a method for encoding the fragments that allows the distribution to be displayed as an image. Chapter 7 furthers this work by using statistical methods on a finer grained version of the encoding scheme used in chapter 6.

The conclusions plus discussion and future work are then presented in chapter 8.

Chapter 2

Literature Survey

This chapter presents the background information for the detailed chapters that follow, starting with a definition of *artificial intelligence* and proceeding through ever more specialised fields of study to *genetic programming*, which is the subject of this thesis. General principals are outlined and standard terms and practices defined. This is followed by an outline of existing work and methods for control of bloat in genetic programming, a description of the algebraic simplification method used as a comparison point in this thesis, and finally some background to the non-parametric statistical methods used in this thesis.

2.1 Artificial Intelligence

Ever since the beginning of the computer age, people have wondered and speculated about the concept of machines having intelligence. A major early worker in this area was the cryptologist Alan Turing, particularly his 1950 paper "Computing Machinery and Intelligence" [70]. At a time when computers were a primitive novelty, in a 1948 report "Intelligent Machines" [69], he proposed networks similar to neural networks, what he called a "B-type unorganised machine". In [70] Turing suggests that programming a machine to act like an adult human brain is too large a

job, instead he suggests programming a machine to be like a child's brain and then having it learn. Drawing parallels with evolution, he likened the structure of the machine to the hereditary material, changes to the machine to be mutations and natural selection being provided by the judgement of the experimenter.

There are many different definitions of artificial intelligence. They fall into two main categories.

1. Machines thinking or behaving like humans. In 1950 Alan Turing proposed his now famous test in which a human observer conducts a conversation, via a keyboard, with a machine and a human. If the observer cannot reliably distinguish which respondent is which then the machine is deemed to be displaying intelligence [70].

   In 1983, in a talk *AI, Where it has been and where is it going* [59], Arthur Samuel offered the following definition of artificial intelligence:

   > "... to get machines to exhibit behaviour which, if done by humans, would be assumed to involve the use of intelligence."

2. Machines exhibiting rational thought or behaviour. This approach emphasises pure logic and rational thought rather than mimicking natural processes. John McCarthy [39, 40] was an early proponent of this approach. In a paper [29] in Science, Kolata quotes McCarthy as writing:

   > "This is AI, so we don't care if it's psychologically real."

   a position McCarthy repeated at a conference in 2006 when he said:

   > "Artificial intelligence is not, by definition, simulation of human intelligence."

2.2 Machine Learning

The earliest work in artificial intelligence focused on reasoning and deduction, but very soon Arthur Samuel [57, 58] started investigating the idea of machines learning from experience, either their own or as supplied by knowledgeable humans. This is the field known as *machine learning* [42]. Samuel used the game of checkers[1] as his research vehicle and in 1961 his program beat an expert human player in a widely publicised match.

There are many applications for these techniques, including but not limited to:

Speech Recognition - Recognising spoken speech, usually trained using the intended operator's voice.

Computer Vision - This includes both object recognition, and object localisation/detection (finding the object(s) of interest against an often noisy background).

Time-series Prediction - This includes tasks like weather forecasting and prediction of financial markets.

Game Playing - Checkers and chess.

Natural Language Processing - Extracting meaning from written natural language, for translation or inputting question or search criteria.

Medical Diagnosis - Medical diagnosis normally requires human expertise, but machine learning techniques can be valuable for tasks like evaluating mass screening tests. In this application there are a very large number of tests to evaluate, most of which will be routine. Those that the machine learning algorithm cannot produce a clear decision on can still be referred to a human expert.

[1] also known as draughts.

Machine learning uses a set of supplied examples we call *training data* or the *training set*. The objective is to develop an algorithm, or a set of parameters for a pre-defined algorithm, that learns from the training data, allowing previously unseen data (the *test set* or *test data*) to be correctly processed. There is further explanation of this and the other optional sets of examples later in this chapter. Machine learning has three main subdivisions depending on the way in which the program learns.

2.2.1 Supervised Learning

In supervised learning [68], the training data includes the desired output value for each example. This is used to provide feedback for the learning process. The machine learning algorithm produces a function/rule set/model that allows it to predict the supplied output value from the input data. This learned function/rule set/model is then used to make predictions from previously unseen data.

2.2.2 Unsupervised Learning

Unsupervised learning uses only the training data with no supplied answers. The task is to find patterns in the training data. Examples of this are clustering, and finding and extracting tabular data in web pages.

2.2.3 Reinforced Learning

In reinforced learning the correct output values are not supplied with the training data, but there is some other method of rewarding correct behaviour. This often the case with game playing. It is not generally possible to assign a true value to intermediate games positions, but humans can supply a general indication with the final game result the ultimate validation, or not, of the choices the program may have made.

This thesis uses only supervised learning.

2.2.4 Datasets

In supervised learning the available input data is split into two or sometimes three sets as follows:

Training Set

The training set is used to induce/generate/produce or otherwise learn a hidden pattern (or classifier or model etc). It is always required.

Validation Set

This is an optional set of data. It is used in the learning process to help decide when to finish training, and to help detect overfitting. Overfitting is where the learned process follows the smallest variations in the input data (which may well be noise) too closely, which results in poor generalisation, that is poor performance on data that was not in the training set. A validation set is not used in this work.

Test Set

The test set is a set of data that has not been seen during the training process. After training has finished, the learned pattern (classifier, model etc) is then tested on the data in the test set. It is this score, (classification error rate, RMS error or other measure) that is the final measure of accuracy for the learned pattern.

n-Fold Cross Validation

When the amount of data available is not large enough to form separate training and test sets of sufficient size then this technique can be used. The most common value for n is 10 and this is the value used in this work. In 10-fold cross validation [28] the available data is split randomly into 10 sets (folds) as equal in size as possible. The experiment is then run 10

times with each of the 10 sets being used in turn as the test set, with the remaining 9 sets combined to form the training set. The final test result is the average of the 10 individual test scores.

2.2.5 Main Paradigms in Machine Learning

There have been many different approaches to machine learning. The following sections introduce the most important paradigms in the field.

Instance Based Learning

In instance based learning there is no explicit learning algorithm in the sense that most machine learning paradigms have a learning algorithm. Instead, each previously unseen item is compared directly with the training data. A common algorithm is the *nearest neighbour* algorithm [7] that uses a distance measure of some kind to determine the item in the training data that is most like (nearest) the previously unseen item. The answer is then taken from that "nearest" training item. In the k-nearest neighbour algorithm the answer is taken from the k nearest items in the training data. If it is a classification task then the result is a vote between these k items, if it is a regression task then some form of weighted average is used.

Induction Learning

The main induction learning algorithm is *decision trees* [4, 25]. The decision tree is a series of questions or conditions in a tree structure and is navigated starting from the root, with the final answers being the leaves of the tree. Decision trees have the advantage of being easy for human beings to understand the results, and they are very flexible in that they can handle categorical and Boolean data as well as numeric. They can however be prone to overfitting and finding the optimum decision tree for any given problem is known to be NP-complete [23].

Statistical Based Learning

These methods use statistical or probabilistic analysis of the training data to develop a classifier. The best known algorithm of this type is *Naive Bayes* [82].

Typically, naive bayes classifiers are boolean in nature, giving a simple true/false answer but in principal they can be multi-class. If we have a class variable C with two or more states then a classifier based on probability could be expressed as:

$$p(C|F_1, \ldots, F_n)$$

Where F_1 to F_n are the n features of the training data.

Using Bayes' theorem, this becomes:

$$p(C|F_1, \ldots, F_n) = \frac{p(C)p(F_1, \ldots, F_n|C)}{p(F_1, \ldots, F_n)}$$

$p(C)$ and $p(F_1, \ldots, F_n)$ can easily be calculated from the training data. The naive bayes algorithm makes the assumption that the feature probabilities are independent of each other. This allows the $p(C)p(F_1, \ldots, F_n|C)$ term to be expanded out into n separate terms, one for each feature, which can be easily calculated from the training data. The resulting classifier is fast to evaluate, and in spite of the assumption about feature independence, works well on many real-world problems. Many email spam detectors use naive bayes based on word counts. It functions well on this problem and it adapts well to changing patterns because the underlying word counts are easily updated based on user feedback on any mistakes the classifier may make.

Connectionist Learning

This paradigm includes *neural networks* and related algorithms. Neural networks [60] are inspired by the neurons of the human brain and are a

connected network of neurons. Each neuron (or perceptron) has one or more inputs that it combines in some fashion (in the canonical case it is by weighted sum). This result may either be used directly, or more commonly transformed using a sigmoid type function that has the properties of being contiguous and everywhere differentiable, but outside a narrow central range quickly converges to one of two values (usually either 0 and 1 or -1 and 1).

The canonical/standard neural network is a feedforward network, that is, it is an acyclic graph. More recently there has been a lot of research into recurrent neural networks [71], which include feedback loops. These can mimic short term memory (to allow time series analysis), or provide resonant behaviour for recognition, particularly in vision processing.

In the canonical feedforward network the weights on the neuron inputs are learnt using *back propagation*. This is an iterative process where the difference between network output and the desired result is propagated back through the network to adjust the weights. The weights can also be learned by many of the other machine learning paradigms, including genetic algorithms (GA), genetic programming (GP) and particle swarm optimisation (PSO).

Support Vector Machines

A support vector machine [6] is a binary classifier in which the training data is mapped on to a space with at least as many dimensions as the training data has features. A hyperplane is then constructed such that the two classes are separated either side of the hyperplane with as large a gap between them as possible, or if that is not possible, then to minimise the number of classification errors.

Evolutionary Computation

The inspiration for evolutionary computation [13] comes from the evolutionary theories of Charles Darwin. In particular his idea of *survival of the fittest*. The following section describes this paradigm in more detail, including genetic programming which is the basis of this thesis.

2.3 Evolutionary Computation

The following few sections describe the basic concepts common to all evolutionary computation methods. The details vary according to the particular algorithm and will be addressed later in this chapter.

2.3.1 Population of Individuals

Evolutionary computation uses a *population* of individuals. Most methods proceed in a series of *generations*, although some methods process one selected individual at a time with no explicit partition into generations. At each generation a new population is created using *selection* and a set of *genetic operators* on the current generation, hypothetically producing better performance on the assigned task with each new generation. The probability of a particular individual member of the population contributing in some way to the next generation depends on that individual's *fitness*.

2.3.2 Measuring Fitness

The method used to measure an individual's fitness is critical to the success of any computational evolutionary method. There are many different ways of measuring fitness, but the two used in this thesis are described below.

Error rate

In classification tasks using supervised learning a common method to assign fitness is to measure the classification error rate on the training set. The lower the error rate, the better the fitness. This is often expressed as $(1 - error)$ and called *accuracy*.

RMS error

In symbolic regression tasks a common method is *Root Mean Squared (RMS) Error*.

$$\sqrt{\frac{\sum_{i=0}^{n}(P_i - T_i)^2}{n}}$$

where P_i is the output of the program for the i^{th} instance of the training set and T_i is the target value of the same instance given by the training set. The training set contains n examples/instances.

2.3.3 Fitness Selection

There are a number of methods used for selecting individuals based on fitness [41]. The two most commonly used are described below.

Roulette Wheel

Also called *fitness proportionate selection* in genetic programming. The probability of an individual being selected is proportional to its fitness relative to the rest of the population. That is, the probability of the ith individual, with fitness F_i being selected from a population of n individuals is:

$$P_i = \frac{F_i}{\sum_{j=1}^{n} F_j}$$

Tournament

In tournament selection, a number of individuals are chosen at random. The individual with the best fitness is the tournament winner. The better an individual's fitness is, the better their chance will be of winning a tournament for which they have been selected. The larger the number of individuals chosen (the tournament size), the lower the chance of an individual with poor fitness being the tournament winner. This allows the level of selection pressure to be easily adjusted. Too low a pressure and convergence to a solution is too slow, too high a pressure and convergence is likely to be to a local optima and the globally best solution will be missed. Tournament sizes are normally in the range $4 - 11$.

Tournament selection is used throughout this work.

2.3.4 Creating a New Generation

The next generation is created by selecting individuals based on their fitness and modifying them in some way using what are known as *genetic operators*. These modified individuals, usually referred to as *children*, make up the new population.

Because this process is stochastic, there is a non-zero probability that the fittest individual in the population could be lost. For this reason, *elitism* is normally used. This involves transferring a small number of the fittest individuals unmodified directly into the new population. This is often called the *reproduction operator* in GP. All of the experiments reported in this work used an elitism rate of 5%.

The other two most commonly used genetic operators, and used in this work, are *crossover* and *mutation*.

The crossover operator takes two selected individuals, selects a portion of each, then swaps these portions into the other selected individual. This therefore produces two children for the next population. The details of how this is done depends on the evolutionary algorithm being used.

The mutation operator takes a single selected individual and replaces some portion of it with a new randomly generated portion. As with the crossover operator, the details vary according to the algorithm being used.

The details for the genetic programming method used for the experiments reported in this work are given later in this chapter.

2.3.5 Major Methods in Evolutionary Computation

There have been many evolutionary methods proposed, the most commonly used ones are detailed in the following sections. A major difference between them is the way the individuals are represented. They fall into two broad groups, ***evolutionary algorithms*** such as *genetic algorithms*, *genetic programming*, *evolution strategies*, *evolutionary programming*, *learning classifier systems*, *evolutionary multi-objective optimisation* and ***swarm intelligence*** such as *particle swarm optimisation* and *ant colony optimisation*.

Genetic Algorithms

Genetic algorithms [41] was developed by John Holland [21] in the mid 1970s. In genetic algorithms the form of the solution needs to be known, or at least a reasonable estimate, and the free parameters of that solution are encoded into a bit-string usually known as the chromosome or genome. It is this bit string that crossover and mutation are performed on.

Genetic Programming

In canonical (standard) genetic programming (GP), the individuals are tree structured programs. The interior nodes are arithmetic operators of some kind, for example addition, multiplication or cosine, and the leaf nodes are either features from the input data, or constants. In canonical GP the nodes all take a real floating point value.

Strongly-typed GP [43] functions the same as canonical GP, except that the nodes can take forms other than real numbers, and care is taken when-

ever programs are created, or modified, that all operators have the appropriate argument types, or at least ones that can be meaningfully converted. In particular, this allows the tree to be a mixture of arithmetic calculations and logical expressions. In some problem domains it can be very useful to have the nodes take complex values because of their ability to represent phase as well as magnitude.

In linear GP [2], the programs take the form of a linear sequence of programming instructions whose arguments can be features, ephemeral constants or a *register*. The output is also to one of a number of registers. The final program output is taken from a predefined register or registers. This ability to return more than one value is a strength of this method when solving multi-class classification problems.

Grammar-based GP [74, 73] uses a formal grammar to describe the allowable forms a program may take. This extends further the idea of strongly-typed GP, but allowing much more flexibility in what is permitted.

Evolution Strategies

In *evolution strategies* [54] the individuals in the population can be represented in any way that is natural to the problem. In the canonical form, only selection and mutation are used, with selection using the fitness rank rather than the absolute fitness value. Populations are usually very small with children only being added to the next generation if they have better fitness than their parent. There is no elitism.

Evolutionary Programming

Evolutionary Programming [15] was one of the early evolutionary methods. It was originally based on finite state machines but is now very similar to Evolution Strategies.

Learning Classifier Systems

In a *learning classifier system*(LCS) [21] the population is a set of rules. These are evolved using any of the evolutionary methods. The LCS also learns to make better use of the rule sets it has by using re-enforcement learning as it uses the rule sets to make decisions and interacts with its environment [20].

Evolutionary Multi-Objective Optimisation

Most evolutionary processes assume the fitness can be expressed as a single value, and that if there is more than one objective, they can be combined in some fashion. An example of this is parsimony pressure (section 2.5.2, which combines fitness and program size. Multi-objective optimisation [9, 10] uses the concept of *non-dominance*. Solution A and solution B are non-dominant if neither of them has higher fitness than the other on all objectives. Most evolutionary methods can be used, and the result is a *pareto front* of non-dominant solutions. The expectation is that some agent (typically human) will decide between the final solutions on the *pareto front*.

Particle Swarm Optimisation

Particle swarm optimisation (PSO), like the other swarm intelligence methods [12, 27], has a population of potential solutions that can be expressed in terms of a location in n-dimensional space. The solution is a point or possibly a surface within this space. The individuals in the population move through this space and cooperate in some way with the other members of the swarm.

In PSO [26], the individuals (particles) have velocity as well as position. They are updated to form the next generation using some combination of their own position and velocity, the best position they have found so far, and the best position found so far by other members of the swarm.

Ant Colony Optimisation

In *ant colony optimisation*(ACO) [11] the individuals (ants) record their position and fitness. The other ants then use this information, much like real ants use pheromone trails left by fellow ants. ACO is useful in problems where the task is to find an optimum route of some kind.

Differential Evolution

In *differential evolution* [53, 66], children are created by combining in some way the positions of two parent individuals. If the resulting child has better fitness than the parents it is accepted into the population for the next generation. It is therefore a continuous improvement algorithm, but unlike gradient descent or hill climbing it does not require the solution space to be either contiguous or differentiable. However it is not guaranteed to find a good solution.

2.4 Genetic Programming

Canonical GP, with the addition of online program simplification as described in chapter 4, is the evolutionary method used throughout this thesis. This section describes some aspects of canonical GP in more detail. The interior nodes of the tree are arithmetic operators or functions of some kind, for example addition, multiplication or cosine, and the leaf nodes are either features from the input data, or ephemeral constants. Ephemeral constants are values that are assigned a random value when created (commonly on a range of -1.0 to $+1.0$), then retain that value for the rest of the evolutionary process. In canonical GP the nodes all take a real floating point value. Figure 2.1 shows an example tree that is three levels deep, various arithmetic operators on the interior nodes and features and constants as leaves.

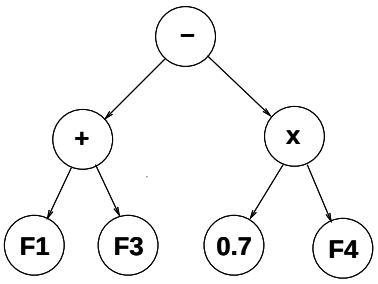


Figure 2.1: Example GP program tree.

2.4.1 Creating the Initial Population

There are two commonly used methods to generate the program trees, with a number of variations.

Full

In the the full method, all the trees have the same number of levels, and are full trees, that is, all the nodes in the bottom level are terminals (features or constants) and the rest are operators. The tree shown in figure 2.1 could have been created using the full method.

Grow

In this method each node is randomly chosen to be either an operator or a terminal. Usually there will also be a maximum depth set. Trees created using this method have a range of branch lengths within the tree, rather than every branch having the same length as in the full method. The trees in the bottom row of figure 2.2 show this sort of form.

Ramped

This method creates its trees using the full method, but with a range of tree depths.

Ramped Half-and-half

This method [31] seeks to maximise the variety of tree sizes and shapes by creating half the population using the ramped method, and half using the grow method.

2.4.2 Creating the Next Generation

Each new generation is created using a mixture of reproduction, crossover and mutation.

Elitism

To avoid the loss of the current fittest program, elitism is usually used with the best 1% to 5% of the current population being transferred unmodified directly into the next generation. In the experiments used in this work an elitism rate of 5% was used.

Crossover

There have been many different ways of performing crossover proposed in the literature. What is described here is the standard/canonical form and is what was used in all of the experiments reported in this work.

The two parents are selected using whatever fitness based selection method is being used (tournament selection in the case of the experiments in this thesis). The crossover points are selected at random. Figure 2.2 shows two example trees in the top row. The crossover points chosen are shown. The two subtrees below these crossover points are then exchanged between the two programs. The bottom row of trees in figure 2.2 show the resulting children which are then added to the population for the next generation.

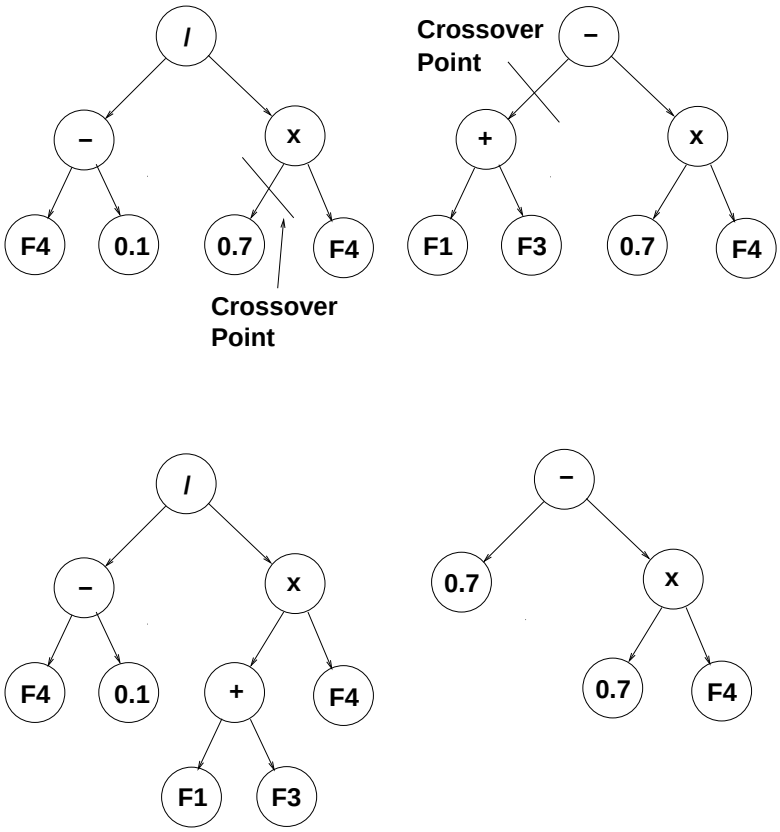


Figure 2.2: Example of crossover on program trees. The first row is the two selected programs with the crossover points indicated. The second row is the resulting children after the two subtrees have been exchanged.

Mutation

Mutation only requires one parent and this is selected using the fitness based selection method. The mutation point is chosen at random and the subtree below this point is discarded. This is then replaced by a new subtree generated by one of the methods described in section 2.4.1. The left-hand tree in figure 2.3 shows a selected program with its chosen mutation point. The right-hand tree in that figure shows the child program that results from the subtree being replaced by a newly generated subtree.

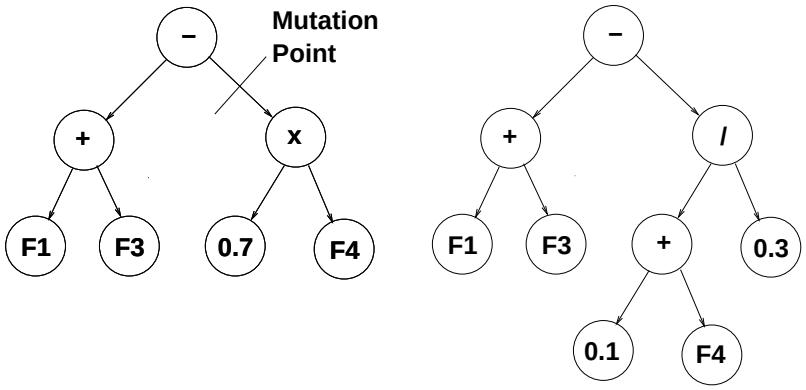


Figure 2.3: Example of tree mutation.

2.5 Background Relating to Bloat

In tree based Genetic Programming (GP), as the run proceeds through the generations the average program size usually increases. If this increase is not accompanied by an improvement in fitness then this unproductive increase is known as *bloat* [64, 65, 3, 31]. Bloat has a number of undesirable effects:

- Increased memory use because of the larger program trees.
- Increased computation time because fitness evaluation takes longer on the larger trees.
- Programs that are much larger than they need to be for their fitness may overfit the training data, reducing their ability to generalise and therefore reducing the performance on unseen data.
- Standard practice is to give all nodes some chance of being the crossover point. Hence, as the programs become deeper (in levels), the average depth of the crossover point also becomes deeper and the crossover is less likely to create a child with a significantly improved fitness, and the overall efficiency of the GP search is reduced.

2.5.1 Causes of Bloat

The cause of bloat is still an open question. There has been a lot of research done on this question but with conflicting results and conclusions.

Langdon and Poli [32] showed that without fitness pressure there is no bloat. They tested GP with both no selection based on fitness, and starting with fitness pressure and then removing it part way through the run. In both cases, when there was no fitness pressure, the mean program size remained constant although the spread of program sizes became wider.

Streeter [67] suggested that large programs are more likely to have better fitness, and fitter programs propagate through the population as evolution proceeds, therefore large programs come to dominate the population.

Nordin and Banzhaf [45] suggested that destructive crossover causes bloat because large trees are less likely to have their fitness reduced by crossover than small trees are. This provides an evolutionary pressure favouring large trees. This is similar to the *intron theory* that says that redundant code (introns) are beneficial because they protect against destructive crossover.

Soule [63, 65] also emphasises the role of introns, suggesting that there are two stages to the evolutionary run. The first stage evolves a near optimum solution, the second stage increases robustness by increasing program size by adding introns.

2.5.2 Control of Bloat

A number of strategies have been proposed to combat bloat. Some place hard limits on size, others penalise the fitness of over-large programs. Other strategies include trying to prevent, or at least delay, the creation of over-large programs or removing unproductive portions of the program tree so that the size is more appropriate for the level of fitness.

Size Limits

A limit can be placed on program size, limiting either the number of nodes or the depth of the tree [31, 2, 85]. This approach prevents bloat, but has some limitations:

- It is very difficult to set a good limit without prior knowledge. The limit needs to be small enough to control bloat, but large enough that the optimum solution can still be found. It is only bloat if the program size increases without an improvement in fitness. If the problem is not already well understood, then a series of trials will be required to establish what a good limit would be. This can be a significant extra computational cost in finding a solution.
- When performing a crossover, trimming the subtrees being exchanged to fit the limit discards genetic material that may be important [2, 77].
- While this approach limits the growth in size, the program trees may still contain a large number of nodes that make no meaningful contribution to the solution. This may still be true even if the optimum solution is evolved. This requires that the limit still be somewhat larger than that required by the optimum solution, even though the size or form of the optimum solution is not known.

Parsimony

A component can be added to the fitness function, or the selection process, that rewards smaller programs, a practice known as *parsimony pressure* [31, 45, 50, 81, 84, 37]. This approach can be successful in many problems. A high fitness program will survive in the population regardless of size, therefore parsimony pressure avoids the loss of good programs that can occur with hard limits. It can be difficult to tune the combination of fitness and program size to provide sufficient bloat control without excessive bias towards small programs. With a difficult task, the size penalty

can dominate the selection process in the early generations because it may take many generations to evolve a solution of high enough fitness to overcome the selection pressure for small size. There are a number of reports that show this approach results in a deterioration in fitness [31, 45].

Instead of combining fitness and program size in a single objective, it is possible to use multi-objective methods [16]. In can work well but there are reports of problems, particularly if high fitness individuals take too many generations to form, when it can fail badly with all programs reducing to trivial sizes [8].

Better Operators

Another approach is to try and prevent bloat from occurring. There are several schemes that focus on the genetic operators, particularly the crossover operator [3, 1, 32]. *Explicitly defined introns* [46] controls the probability of particular nodes being chosen as the crossover point in an attempt to prevent destructive crossover.

Other Indirect Mechanisms

There are other approaches that penalise large programs while trying to avoid the problems of parsimony pressure. In *double tournament selection* [36] a series of tournaments are run using program size to determine the winner, the winners of these tournaments then contest a final tournament using fitness to determine the winner. In the *tarpeian* method [51] a random subset of above average sized programs are simply eliminated from the population. In the *waiting room* method [49], newly created programs do not enter the population until after a number of generations proportional to their size. The idea being to give smaller programs a chance to spread through the population before being overwhelmed by their larger brethren. A recent article [72] reports success by placing the population on a torus, with selection defined by a Moore neighbourhood and local elitist

replacement.

Simplification

All the methods described so far *indirectly* control, or delay the production of, bloat. However, they do not directly eliminate bloat after redundant code has been created. Simplification aims to achieve this.

A program can be simplified algebraically during the run. A rules-engine approach [22] has been used to prevent overfitting. This used over 200 rules and was selectively applied during the run. It was successful in reducing bloat and overfitting but did not reduce execution time as the algorithm was computationally quite expensive. A simpler scheme [77, 78] reduced the resource usage while still reducing bloat, and it is this scheme that this thesis uses for algebraic simplification. However, evaluating whether two subtrees are equivalent still becomes computationally expensive as the size of the subtrees increases. The algorithms necessary to address this are complex, and Wong [77, 78] uses efficient hashing techniques. This approach is however not easy to implement, and the hash algorithms are not infallible as, like all hash techniques, collisions are possible. The implementation used in the experiments for this thesis is described in more detail in the next section.

Chapter 4 introduces a novel approach to program simplification called *numerical simplification*. Other researchers [44, 24] have used simplification as a post-processing technique to reduce the size of the final solution.

2.6 Algebraic Simplification

The *algebraic simplification* method is an implementation of the work of Phillip Wong [77]. It makes the kind of algebraic simplifications that a human might do based on a set of rules. It leaves the program functionally the same as before simplification, that is, the new program produced

by the simplification process will produce exactly the same result as the original program.

This method is motivated by the algebraic nature of programs using the canonical GP functions ($+, -, \times, \div$) and uses a set of *algebraic simplification rules* to remove redundancies. These rules consist of two parts, a *precondition* which represents the state of the surrounding nodes that must be present for the rule to be applied, and a *postcondition* that represents the state that the surrounding nodes are in after additions and deletions are made. These rules make up the *rule-set* for the simplification method. Table 2.1 shows the particular rule-set used in this work.

Table 2.1: Simplification rules. Lower case letters represent numerical constants, while the upper case letters represent variable/feature terminal nodes or subtrees.

Precondition		Postcondition
$a + b$	$\rightarrow$	$c, c = a + b$
$a - b$	$\rightarrow$	$c, c = a - b$
$a \times b$	$\rightarrow$	$c, c = a \times b$
$a \div b$	$\rightarrow$	$c, c = a \div b$
$A \div 1$	$\rightarrow$	A
$A \div A$	$\rightarrow$	1
$0 \div A$	$\rightarrow$	0
$0 \times A = A \times 0$	$\rightarrow$	0
$A \times 1 = 1 \times A$	$\rightarrow$	A
$A + 0 = 0 + A$	$\rightarrow$	A
$A - 0$	$\rightarrow$	A
$A - A$	$\rightarrow$	0

A hashing algorithm is used to simplify the check on subtree equality. Each operator has a hash value, and each feature has a hash value based on the feature number. Ephemeral constants have a hash value based on the value of the constant. These are combined using a "shift and xor" al-

gorithm to generate the hash for their parent node. If two subtrees have the same hash value, they are considered equivalent. In the implementation used in this thesis the hash calculations and rule evaluations are performed by the *operator* objects. This means that the hash calculation can be aware of whether the operands need to be ordered or not, and that only those simplification rules that apply to that operator need to be checked.

To simplify a program, the program tree is recursively traversed in a bottom-up fashion. For each node it checks the precondition of each rule in the rule-set. If any rule matches, then it is applied to that portion of the tree. The algorithm continues to check preconditions until none of the rules in the rule-set can be applied, at which point it moves on to the next node.

An example of the method is given in Figure 2.4. In this example, `F1` has been randomly assigned the hash value 6, while `F2` has been assigned the hash value 7. For the '+' nodes, the operator is aware that order is not important therefore the hash values calculated on the two '+' nodes will be the same. For the root node $\div$, the rule $A \div A \rightarrow 1$ is found to apply, since both left and right child nodes have the same hash value. The rule is applied and the result is a single numerical-node, with a value of `1`.

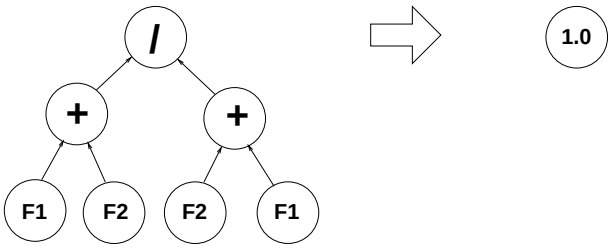


Figure 2.4: An example of a subtree matching the pre-condition for algebraic simplification using the rule: $A \div A \rightarrow 1$. It will be replaced by a single numerical node with the value 1.0

2.7 Non-Parametric Statistics

This section gives some background to the statistical techniques used in much of this thesis. Non-parametric methods are not very well known, so the following sections outline what parametric and non-parametric statistics are, and the limitations of the parametric methods that are more commonly used.

2.7.1 Parametric Statistics

The statistical measures familiar to most people are *mean* and *standard deviation*. These are the *parameters* of the *normal distribution*. They completely describe a normal distribution as defined by equation 2.1. There are other distributions that are also defined in terms of parameters in this way, such as the poisson distribution, the log-normal distribution, the pareto distribution and many others.

There are many statistical tests [18] that assume the normal distribution and are derived mathematically using the probability distribution function and defined in terms of the parameters μ (the mean) and σ (the standard deviation).

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.1}$$

Limitations of Parametric Statistics

The parametric methods are only valid if the data they are being used on conforms sufficiently closely with the distribution the method is based on. Student's t-test is a well known test for significance that is parametric and as such relies on the distribution being tested conforming to the normal distribution. If the distribution being tested is not normal then the significance test may give a misleading result.

The parametric methods are susceptible to outliers, data-points that have extreme values [5, 19], particularly if there are more than a very small number of them. This is because the large values distort the calculation of the mean and standard deviation and therefore any results that are based on them. The CPU time required for GP experiments is a distribution that almost always includes outliers. This situation becomes even worse when the distribution being tested is the difference between two sets of run-times.

The parametric methods require the data to be continuous, at least in principal. They are not valid for discrete data such as many survey results that have yes/no or graded $1-5$ type answers.

2.7.2 Non-Parametric Measures

The non-parametric methods [5, 19] were developed to handle data that cannot satisfy the requirements of the parametric methods. This can be because the data is badly skewed, has many outliers, or is discrete.

The main principal of them is that they are concerned only with the order of the data, and whether one data point is larger or smaller than another. The absolute value or magnitude of the data is not considered. This allows them to handle outliers because it does not matter how large the largest value is, it is still just larger than all the other data-points. They can handle discrete data, as long as it is possible to define an ordering, such as with $1-5$ or strongly-disagree to strongly-agree type survey data.

The methods described in the following paragraphs all involve sorting the data and assigning each data-point a *rank*. The data-point with the smallest value has rank 1 through to the largest having a rank equal to the sample size. If there is a group of two or data-points with the same value then they all take a rank that is the average rank of that group.

Measures of Central Tendency / Alternatives to the Mean

The usual alternative to the mean is the median. That is that data value that has half the sample with lower values and half the sample with higher values. If the size of the sample is even, then the median is the mean of the two middle values. In the example in table 2.2 the median is 14.5 being the mean of 13 and 16.

Measures of Dispersion / Alternatives to the Variance

The range of ranks is split into equal sized groups, usually either four or ten. The experiments in this thesis use four and the measures quoted are the 1st quartile and the 3rd quartile. The 1st quartile is larger than the smallest one quarter of the data-points and is smaller than the remaining three quarters. The 3rd quartile is larger than the smallest three quarters of the data-points and smaller than the remainder. As with the mean, if the boundary between quartiles falls between data-points then the quartile value is the mean of the two adjacent data-points. If the data is split into ten groups, they are called deciles, or percentiles in which case they are numbered 10 to 90 rather than $1-9$. Sometimes the 1st and 9th deciles are used as alternatives to the mean plus or minus two standard deviations.

In table 2.2 there are 20 data-points so the quartiles each have 5 points in them. The first quartile is 8.5 being the mean of 8 and 9 because the boundary between quartiles falls between two data-points.

Measures of Significance / Alternatives to the t-Test

Whenever experimental results suggest that a change of behaviour is occurring, or that two algorithms produce different results it is very important to know whether this is *significant* or is merely the result of random chance. In particular, what we test is that the probability of the observed behaviour being due to chance is less than some threshold. This threshold

Table 2.2: Data for non-parametric statistics examples.

Data Value	Rank	
41	20	
27	19	
25	17.5	
25	17.5	
21	16	
		3rd quartile
20	15	
18	13	
18	13	
18	13	
16	11	
		median
13	10	
12	8.5	
12	8.5	
10	7	
9	6	
		1st quartile
8	5	
7	4	
6	3	
3	2	
1	1	

is usually taken to be 0.05 or 5% and this is the standard used in this thesis. This is often expressed as a confidence level of 95%.

When two samples are being compared in this way they can either be independent or paired. In a paired test there is a correlation between the two samples. The significance tests presented in this thesis are all of the paired variety, where pairs of experiments are performed with the same training data and the same initial population, differing only in some detail of the evolutionary algorithm. The purpose of the test is then to check if any observed difference is due to the difference in algorithm or only due to chance.

The standard parametric test is Student's t-test. The most common non-parametric test for independent samples is the *Mann-Whitney U test* which is also known by the name *Wilcoxon rank-sum test*. For paired tests, the usual non-parametric test is the *Wilcoxon signed-rank test* [75, 34], and this is the test used in this thesis.

Measures of Correlation / Alternatives to Pearson's Correlation Coefficient

Another common statistical test is for correlation. This tests how closely the data can be fitted to a straight line. The parametric measure is *Pearson's correlation coefficient* and the non-parametric near-equivalent is *Spearman's rank correlation coefficient*.

2.8 Chapter Summary

This chapter has outlined the background for the detailed chapters that follow. In particular genetic programming, which is the evolutionary algorithm used in the experiments presented in the following chapters, and the concept of bloat with the existing control measures and their limitations. It is the control of bloat, or unproductive code growth, that is the

primary motivation for this work.

The existing methods, as outlined in this chapter, focus on preventing bloat. It has however been shown [32] that the combination of fitness based selection and variable length representation (such as the trees used in GP) causes bloat. Therefore the focus of this thesis is on online simplification methods that can remove redundant code from the trees after it has been created.

In particular, chapter 4 proposes a novel method (numerical simplification) of simplifying the program trees using only the values the tree nodes take during fitness evaluation. Chapter 5 then analyses the dependence on the key parameter of the method. Chapters 6 and 7 analyse the effect of online simplification on the distribution of program fragments within the population.

Chapter 3

Datasets and Experimental Configuration

3.1 Experimental Datasets

This thesis uses three classification tasks and two symbolic regression tasks in the experiments.

3.1.1 Coins Classification

This dataset [77] consists of a series of 64×64 pixel images of New Zealand five cent pieces against a random noisy background, see Fig 3.1 for example head and tail images plus an example of the background with no coin. There are 200 each of heads, tails and background only.

Figure 3.1: Example images of the coin head, tail and background with no coin.

The fourteen features used were based on a discrete cosine transform of the image. A discrete cosine transform is calculated over the whole image, using an algorithm based on a Fast Fourier Transform for square images [38] where the width is a power of 2. The resulting 64×64 matrix of spectral coefficients contains both frequency and directional information. The directional information is then removed by reducing the matrix to a one dimensional array by averaging each diagonal to give a one dimensional array of 127 non-directional frequency coefficients which are in order from the lowest frequency to the highest. If $T_{i,j}$ is a coefficient in the 64×64 matrix output by the cosine transform, then the non-directional frequency coefficients C_0 to C_{63} are given by:

$$C_i = \frac{\sum_{j=0}^{i} T_{i-j,j}}{i+1} \tag{3.1}$$

and C_{64} to C_{127} by:

$$C_i = \frac{\sum_{j=0}^{127-i} T_{63-j,i-64-j}}{128-i} \tag{3.2}$$

These are then combined into bands to form the features. If C_i is the ith coefficient and $C_{i:j}$ is the average of coefficients i through j, then the features are C_0, C_1, C_2, C_3, C_4, $C_{5:6}$, $C_{7:8}$, $C_{9:10}$, $C_{11:18}$, $C_{19:26}$, $C_{27:34}$, $C_{35:42}$, $C_{43:50}$, $C_{51:126}$.

Note that these features were chosen because they achieved reasonably good results in preliminary trials. As the goal of this work is to investigate the effect of program simplification rather than to achieve the best absolute performance on any particular task, they were used here. This consideration also applies to the other datasets described below.

3.1.2 Wine Classification

This dataset [17] is the result of a chemical analysis of wines grown in the same region in Italy but derived from three different cultivars. The analysis determined the quantic ties of thirteen constituents found in wines

from each of the three cultivars. These thirteen constituents are the features and the three classes are the cultivar from which the wine comes. This dataset was sourced from the Weka project described in [76]. There are 59 examples of class 1, 71 examples of class 2 and 48 examples of class 3, a total of 178 examples. It became apparent during testing that this dataset is quite sensitive to the split into training and test data. There appears to be some important characteristics that are represented in only a few examples. If there are too few of these in the training data then the resulting classifier performs poorly on the test set. This affects the overall classification accuracy figures from the ten-fold cross validation as in many runs one or two of the folds will have poor performance. As the goal is to compare the relative performance of different methods, this is not a big issue for this work.

3.1.3 Face Recognition/Classification

This is the ORL face data set [56], from which only four individuals were used. Therefore the set was four classes with ten examples of each. This is a rather small number of examples and makes evolving a good classifier difficult. Sample images for the first individual are shown in Figure 3.2.

Figure 3.2: Example images from the ORL face dataset.

Simple pixel statistics [83] were used to create 8 features. Figure 3.3 shows the four regions used. The mean and standard deviations were calculated on the pixel values for each of the four regions. These eight values were the features used for this dataset. The size of the face and its posi-

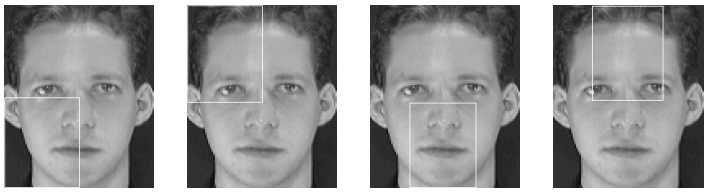


Figure 3.3: Regions used to generate pixel statistics for the faces dataset.

tion in the frame are not constant across all the images, so these regions were not chosen with a view to measuring any particular facial feature or features. As with the other datasets, it is only the relative performance between methods that is important. These features allowed at least moderate differentiation between classes and they were not optimised any further.

3.1.4 Symbolic Regression Task One

This dataset serves two roles in the thesis. The first is to provide a task that is easy enough that good results can usually be obtained under all conditions. The second is to provide a platform for investigating the relationship between the noise level in the data and the threshold used in numerical simplification (this is described to chapter 5). The noise level in the above classification tasks is unknown, and will be unknown in most other classification tasks. A regression task allows the noise level to be set when the data is generated.

The function used is:

$$f(x) = -2.0 \times x^2 + \frac{0.3}{1.05 + x} + 1.0 \tag{3.3}$$

The values for x are randomly generated from the range $[-1.0,\ 1.0]$, 200 examples for the training set with random noise added where required, and 200 examples for the test set without noise.

3.1.5 Symbolic Regression Task Two

This second regression task is intended to be much more difficult for the GP system to solve. For the experiments on the relationship between the noise amplitude and the optimum value for the simplification threshold, noise is added in the same way as with the first symbolic regression task.

The function used is:

$$f(x) = sine(x \times \pi) \tag{3.4}$$

The values for x are randomly generated of the range $[-2.0,\ 1.0]$, 200 examples for the training set and 200 examples for the test set.

This problem is much more difficult as it uses the sine function which is not included in the available operators (function set). To get a good fit to this function over one and half cycles requires a seventh order expansion of the Taylor series for the sine function. Some runs do well but success is generally more limited.

3.2 Experimental Design

This thesis directly compares the results from canonical GP and two different online simplification methods. This comparison is made easier if the experiments are all conducted with population parameters as similar as possible. Initial runs established a set of parameters that gave reasonable performance without making too great a demand on memory resources for the canonical GP (no simplification) case. This set of values for population size, initial program size and tournament size was then used through all of the runs. In general the only variation was the number of generations and the population size. The ORL face dataset required some differences to get reasonable performance. Details of this will be described in later chapters.

3.2.1 Operators

All of the experiments in this thesis use the four arithmetic functions: addition, subtraction, multiplication and protected division. No *if* function was used because it creates unnecessary complications for fragment encoding schemes used for the fragment analysis in chapters 6 and 7.

3.2.2 Generating the Initial Population

The population size was 1000 for the coin and wine datasets, 5000 for the faces dataset and 2000 for the regression tasks. Initial programs were five levels deep using the full method. Ramped or ramped-half-and-half would be more common ways to create the initial population, but because the purpose of these experiments is to compare program sizes between algorithms and between runs, the consistent starting program size of the full method was felt to be more appropriate.

3.2.3 Selection

Tournament selection was used with a tournament size of four, which is commonly used in GP. [80, 14]

3.2.4 Genetic Parameters

The experiments used 40 generations for the coin dataset, 200 generations for the wine dataset, 100 generations for the faces dataset and 30 generations for the two regression tasks.

Two sets of parameters were used for the genetic operators: one 5% reproduction, 85% crossover, 10% mutation; and one 5% reproduction, 15% crossover, 80% mutation. A high crossover rate is standard practice, but early experiments suggested the the low crossover, high mutation rate may give better results with the simplification methods so the experiments

were run with both sets of parameters to check if this observation was more generally true.

Ten-fold cross validation was used on the classification tasks. Each class was evenly distributed (as far as was possible) between the folds.

Chapter 4

Numerical Simplification

4.1 Introduction

This chapter introduces a new method of online program simplification called *numerical simplification*. Existing methods for program simplification [22, 77, 86] replicate the algebraic simplifications that a human might do. This approach can work well and is used as a comparison point in the results reported in this chapter. There are two weaknesses to this approach that *numerical simplification* aims to address.

1. An algebraic equivalence may exist that is too complicated for the simplification algorithm being used.

2. The algebraic equivalence may be very close, but one or more of the constant values may be very slightly different, or there may be an extra term that always has a very small value. Algebraic simplification can never deal with this situation.

Numerical simplification considers only the actual numeric values the tree nodes take as the fitness is evaluated. No matter how complicated the algebraic equivalence may be, the appropriate node values will be equal, or very close to it as numerical inaccuracies may introduce very small errors, and simplification can proceed as if they are equal.

4.2 Chapter Goals

This chapter addresses the first of the research questions from chapter 1, namely:

Whether numerical simplification can help control bloat, thereby reducing resource requirements without seriously reducing the effectiveness[1] of the GP search. This question is broken down into the following sub-questions:

1. Will the numerical simplification method produce smaller programs than the standard GP system with no simplification?
2. Will the numerical simplification method shorten computation run times compared with the standard GP system with no simplification?
3. Will the numerical simplification method affect the system effectiveness?
4. How does numerical simplification perform compared to algebraic simplification?

4.3 Numerical Simplification

The idea of *numerical simplification* is to consider the numerical contribution that a node or subtree makes to the output of its parent node, removing those nodes and subtrees whose impact on the result is too small to make much difference to the program result. The motivation here comes from the fact that most data includes a noise component. Instead of trying to fit to this noise, two values whose difference is smaller than the noise are considered to be equal. For efficiency reasons, this implementation addresses only the local effect of simplification at each node in the program

[1] This work uses the term *effectiveness* to mean the root mean square (RMS) error for a regression task and the classification error rate on the test set for classification tasks.

tree. There will be cases where it does affect the system performance of the whole program, but the aim is to keep this to a minimum. It may be easiest to think of numerical simplification as a kind of lossy compression (such as the jpeg algorithm for images), where the aim is to get useful reductions in program size without obvious loss in quality.

Note that numerical simplification is only applicable to tree based representations. The algebraic method used in this work as a comparison point also requires a tree based representation. Linear GP programs can be represented as a set of parallel interconnected trees and it should therefore be possible to apply numerical simplification to the trees, but this has not been attempted in this work as this is beyond the scope of this thesis.

As the fitness is evaluated across the training set, a record is kept of the minimum and maximum function values for each node in the program tree. A *simplification threshold* is chosen, which can be the result of preliminary trials, or if there is enough knowledge of the dataset then a good starting point will be the noise floor in the data. This work uses 0.001 in most of the experiments as a result of early trials. All of the features are normalised on the range [-1.0, +1.0] so this represents a noise level of about 0.1%, which seems reasonable without better information on the source data.

The simplification process is performed from the bottom up. For addition and subtraction operators, a child node or subtree whose range of values is less than the threshold times the parent's minimum absolute value is discarded. Figure 4.1(a) gives an example of such a kind. The range for Node B is $0.027 - 0.020 = 0.007$. The minimum absolute value for its parent Node A is 7.3. Since $0.007 < 0.001 \times 7.3$, the subtree headed by Node B will be discarded, and Node A will be replaced by Node C. Also, if the range of values a node takes is less than the threshold times its own minimum absolute value, the node is replaced by a constant terminal taking its average value. Figure 4.1(b) gives an example of this kind. The range for Node D is $2.0015 - 2.0000 = 0.0015$. The minimum absolute value for

Node D is 2.0000. Since $0.0015 < 0.001 \times 2.0000$, the subtree headed by Node D will be discarded, and Node D will be replaced by a constant terminal with the value 2.00075. Note that the second type of simplification takes precedence over the first.

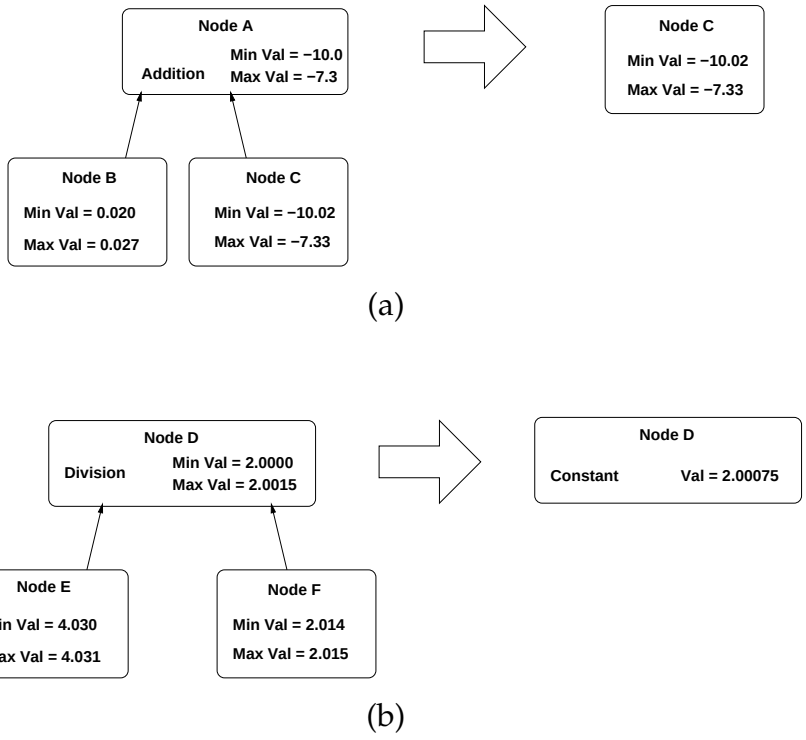


Figure 4.1: Example trees used to explain numerical simplification.

The numerical simplification method described here has the advantage that it is very simple, both in implementation and execution. The necessary information is gathered as part of the fitness evaluation and the computational cost is very small. The simplification process then requires one further traversal of each program tree after fitness evaluation is complete.

4.4 Experimental Results

Each experiment was run 200 times, in the results that follow for program size in nodes and depth they are grouped in sets of 20 runs, with the results averaged over each of these 20 runs. The intention is to show both average behaviour and also the variation in those averages. Where simplification was used, it was performed after the first generation, and every fourth generation thereafter. This results in a periodic behaviour in program sizes as will be seen on the graphs. Note that there is no limit imposed on program size as one of the goals is to investigate whether simplification is sufficient to manage program bloat.

4.4.1 Program Size - Number of Nodes

Figure 4.2 shows the number of nodes per program at each generation for the coin dataset. These graphs show the minimum, mean and maximum program sizes in nodes for each generation in the run. Each curve shows the mean value for that metric across one set of twenty runs. Note that the minimum is small and with little variation. The maximum program size however shows wide variation. Individual runs had maximum program sizes that varied from not much larger than the mean, up to $3,500$ (sic) in one run with the 85% crossover rate. Program sizes are generally larger with the 85% crossover rate, and continue to grow even with simplification albeit more slowly than without simplification. Program sizes are smaller with the lower 15% crossover rate, and level off to a steady state with simplification, allowing the possibility of longer runs without the program sizes growing out of control. This difference in performance is likely to be because the replacement subtrees used by the mutation operator are limited to a maximum depth of five levels and most are three or four levels. In contrast, the replacement subtree used by the crossover operator gets larger, on average, as the program becomes deeper. Therefore as the program depth increases, mutation is likely to produce smaller

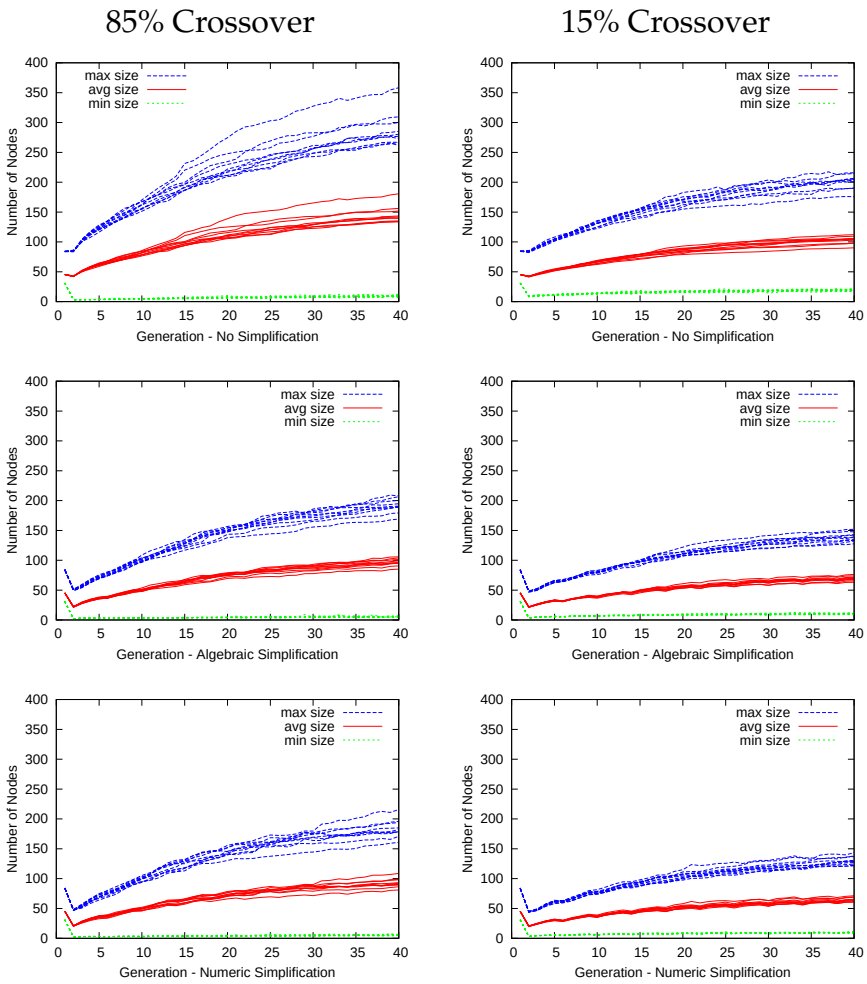


Figure 4.2: Program size (number of nodes) for the coin dataset. In each graph, the maximum, mean and minimum program size is shown. Each line is the average of 20 runs.

offspring than the crossover operator. In both cases, the program sizes are generally smaller with simplification than without. The results for the wine and faces datasets show a very similar pattern, so are not shown in detail here.

Figure 4.3 shows the mean program sizes for all three methods and for all three classification datasets. With the coin dataset the reduction in

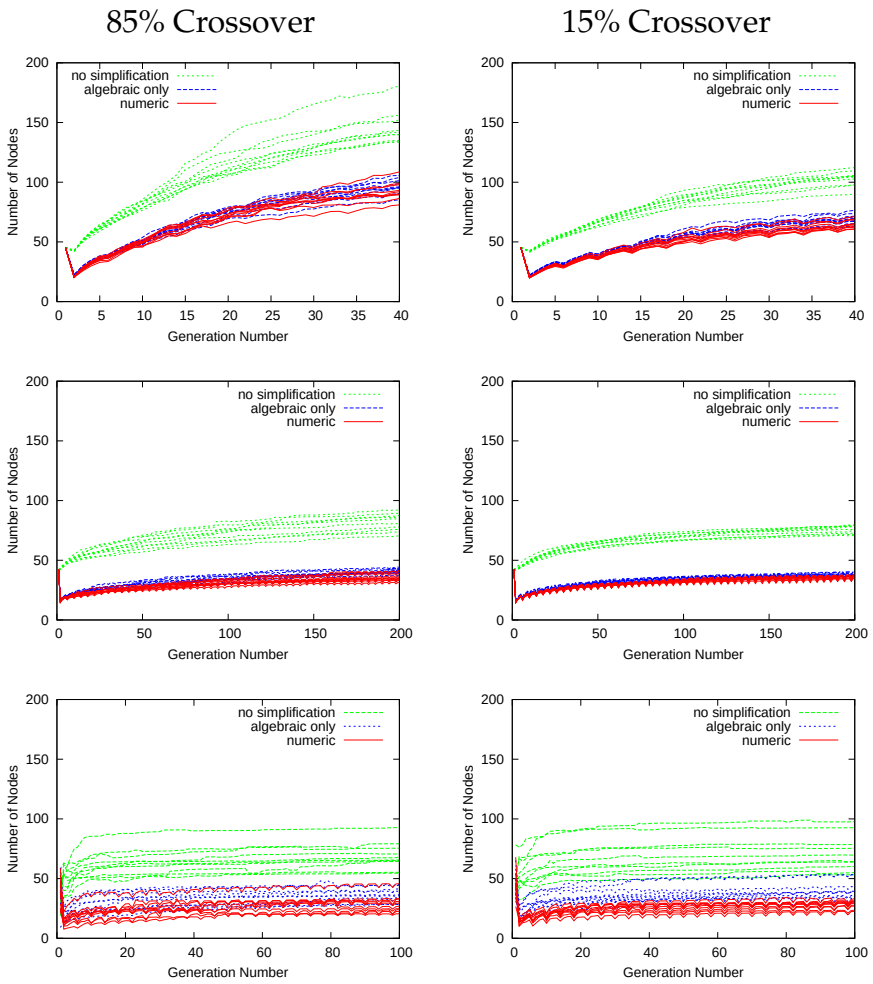


Figure 4.3: Mean program size (number of nodes) for the coin dataset (top row), wine dataset (middle row) and the faces dataset (bottom row). For each level of simplification, the mean program size is shown for 10 replications, each of which is the average of 20 runs. Note the periodic nature of the simplification lines due to simplification only being run at every fourth generation.

program sizes as a result of simplification can be clearly seen. While both simplification methods show about a 40% reduction over no simplifica-

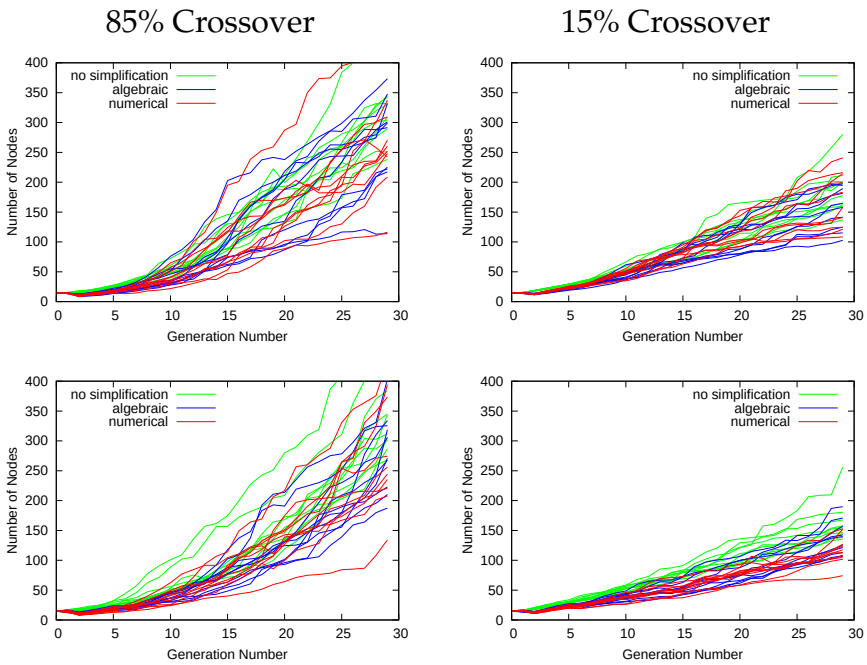


Figure 4.4: Mean program size (number of nodes) for the two regression tasks. For each level of simplification, the mean program size is shown for 10 replications, each of which is the average of 20 runs. Note the periodic nature of the simplification lines, this is due to simplification only being run every fourth generation.

tion, there are no noticeable differences between algebraic and numerical simplification. Note the periodic nature of the curves, as a result of applying simplification every fourth generation. The results for the wine dataset are very similar to those for the coin dataset. With this dataset there is a small but noticeable advantage to numerical simplification over algebraic for at least the 85% run. The faces dataset shows a lot more variation than the other two datasets but still shows a considerable reduction in program sizes with simplification. There is no clear difference between the two simplification methods in the 85% crossover case but the program sizes are noticeably smaller on average with numerical simplification than they

are with algebraic simplification when the crossover rate is 15%.

Figure 4.4 shows the program sizes for the two regression tasks. The results here are less clear-cut than with the classification tasks. As with the classification tasks the program sizes tend to be larger and there is much more variation in results with the 85% crossover rate than with the 15% rate. On the first regression task there are no clear differences between methods. The second task shows a tendency for simplification to produce smaller programs, but this is less clear cut than with the classification tasks. In the 15% crossover case there might be a slight advantage to numerical simplification over algebraic simplification but this might not be significant. Neither of these two regression tasks show the pronounced cyclic behaviour because of the simplification being performed every four generations that is so evident in the classification tasks.

The distribution of differences in program sizes is not a normal distribution as there are long tails, and in many cases the distribution is badly skewed. This rules out the use of the standard *student* T–test. Instead a non-parametric test called the Wilcoxon Signed-Rank test [75, 34] was used to test the significance of the reductions in program sizes with the two simplification methods. This test does not require that the distributions are normal and it uses only the order of the results, not the magnitude. It does require the distributions to be symmetrical and this requirement can be met by testing the medians rather than the means of the differences distributions. The results are presented in table 4.1. For the difference lines, the third column gives the Z score. Note that these are for a directional test, that is the displayed Z values indicate the confidence that the simplification method results in programs having fewer nodes than *no simplification* or that *numerical simplification* results in programs having fewer nodes than *algebraic simplification*. Where the Z score is better than the 95% confidence level, the fourth and fifth columns show the 95% confidence interval expressed as the percentage reduction in the number of nodes. The fifth column shows an * for $> 95\%$ confidence, ** for $> 99\%$,

and *** for $> 99.9\%$.
Each set of four lines gives the following:

1. The median number of nodes per program in the *no simplification* runs.

2. The reduction in node count in the *algebraic simplification* runs compared with the *no simplification* runs.

3. The reduction in node count in the *numerical simplification* runs compared to the *no simplification* runs.

4. The reduction in node count in the *numerical simplification* runs compared to the *algebraic simplification* runs. If this is negative than the numerical simplification runs had larger programs than the algebraic simplification runs.

It can be seen that for the reduction in program sizes (in nodes) is highly significant for both simplification methods on all of these tasks except for the first regression task where the result for numerical simplification of 1.56 falls short of the 95% significance level, which for a directional test such as this is 1.65. This is only just short and may be significant with more experiments.

In most cases numerical simplification resulted in smaller programs than algebraic simplification, but this was not always significant. As expected, this was not significant for the coins dataset. The apparent advantage on the wine dataset with 85% crossover turned out to be non-significant but that the small difference for the 15% crossover rate was. In both cases on the faces dataset numerical simplification produced slightly smaller programs than algebraic, but this difference was highly significant. On the regression tasks only the 15% crossover rate on the second task shows a difference that is significant to the 95% level.

Table 4.1: Significance scores and confidence intervals for the differences in the program sizes in nodes.

			% Reduction in number of nodes		
	Median	Z	Min	Max	
Coins 15% Crossover with no simplification	97.2				
Reduction with Algebraic simplification	33.0	**9.13**	28.4	41.5	***
Reduction with Numeric simplification	32.3	**9.87**	28.8	41.5	***
Difference between Algebraic and Numeric	1.8	0.63			
Coins 85% Crossover with no simplification	113.4				
Reduction with Algebraic simplification	40.4	**9.53**	36.2	55.5	***
Reduction with Numeric simplification	41.5	**8.95**	34.0	52.4	***
Difference between Algebraic and Numeric	−1.4	1.02			
Wine 15% Crossover with no simplification	66.3				
Reduction with Algebraic simplification	31.8	**15.55**	44.0	52.6	***
Reduction with Numeric simplification	33.7	**16.4**	48.2	56.3	***
Difference between Algebraic and Numeric	1.5	**2.42**	0.6	5.9	**
Wine 85% Crossover with no simplification	62.5				
Reduction with Algebraic simplification	31.5	**14.4**	51.0	64.8	***
Reduction with Numeric simplification	30.7	**14.3**	48.9	61.6	***
Difference between Algebraic and Numeric	0	0.03			
Faces 15% Crossover with no simplification	51.3				
Reduction with Algebraic simplification	19.2	**10.6**	35.9	51.7	***
Reduction with Numeric simplification	24.7	**13.9**	52.0	70.2	***
Difference between Algebraic and Numeric	6.5	**7.14**	11.7	20.7	***
Faces 85% Crossover with no simplification	56.7				
Reduction with Algebraic simplification	23.0	**12.36**	41.5	56.6	***
Reduction with Numeric simplification	28.0	**14.70**	51.9	67.5	***
Difference between Algebraic and Numeric	7.3	**6.52**	9.2	16.8	***
Regression one 15% Crossover	129.0				
Reduction with Algebraic simplification	21.9	**2.73**	5.4	32.9	**
Reduction with Numeric simplification	13.9	1.56			
Difference between Algebraic and Numeric	6.0	1.25			
Regression one 85% Crossover	154.0				
Reduction with Algebraic simplification	26.4	**2.45**	5.8	57.8	**
Reduction with Numeric simplification	27.5	**2.59**	7.0	60.3	**
Difference between Algebraic and Numeric	1.9	0.39			
Regression two 15% Crossover	122.9				
Reduction with Algebraic simplification	13.9	**2.35**	2.6	30.8	**
Reduction with Numeric simplification	33.6	**4.91**	19.0	43.3	***
Difference between Algebraic and Numeric	9.7	**2.14**	0.9	23.1	*
Regression two 85% Crossover	247.0				
Reduction with Algebraic simplification	38.3	**2.50**	5.3	51.6	**
Reduction with Numeric simplification	33.4	**2.58**	6.2	51.9	**
Difference between Algebraic and Numeric	11.4	0.71			

4.4.2 Program Size - Tree Depth

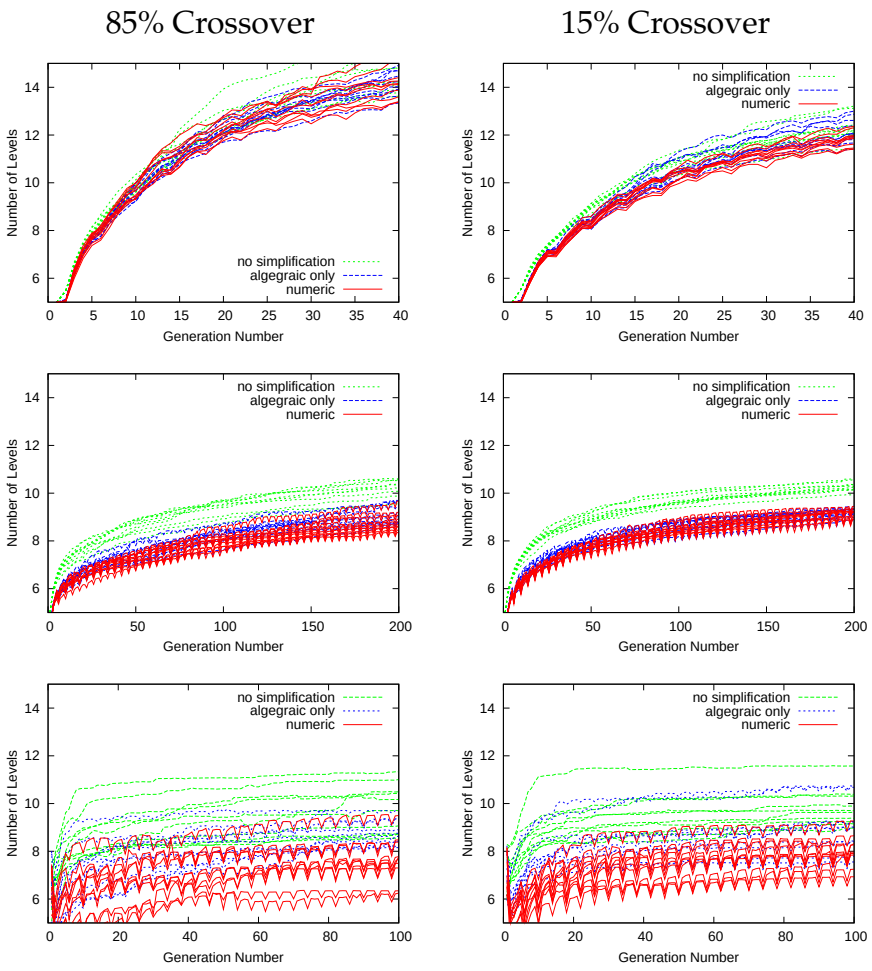


Figure 4.5: Mean tree depth for the coin dataset (top row), wine dataset (middle row) and the faces dataset (bottom row). Again these are 10 replications (curves), each of which is the average of 20 runs.

Figure 4.5 shows the mean tree depth by generation for the three classification datasets. The wine and faces datasets show the simplification methods produce shallower trees, which is as expected given the lower number of nodes. The coins dataset shows a much smaller difference with

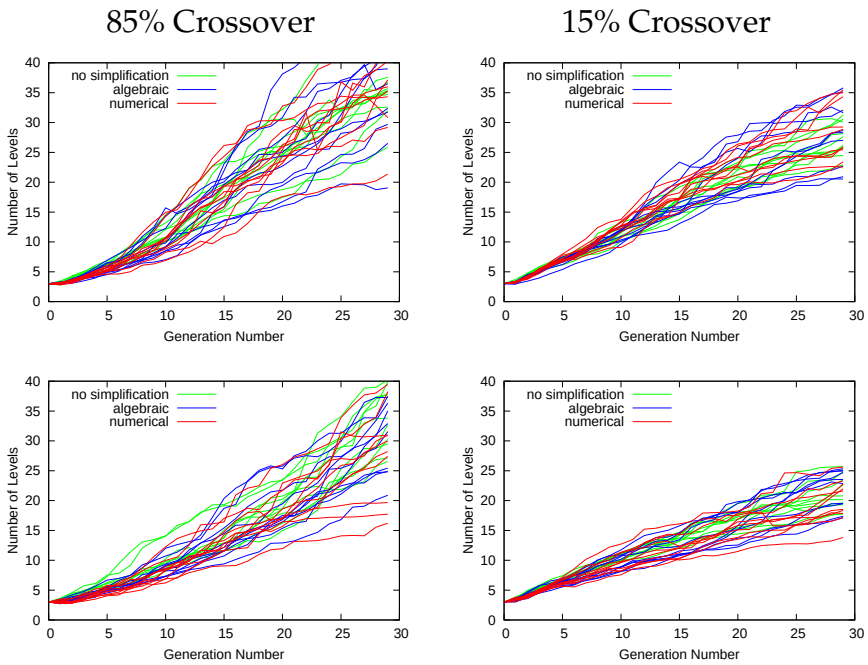


Figure 4.6: Mean tree depth for the two regression tasks. Again these are 10 replications (curves), each of which is the average of 20 runs.

a 15% crossover rate and maybe none at all with the higher 85% crossover rate. In all six cases, the difference in tree size between simplification and standard GP is more pronounced as a node count than it is in tree depth. This suggests that the shape of the tree is being changed, not just its size. This is examined in more detail in the next section.

Figure 4.6 shows the mean tree depth by generation for the two regression tasks. The results for the number of nodes showed that simplification produced a small reduction in tree size on the second task but no clear difference on the first task. The tree depth shows the same pattern, although the differences are smaller and less obvious, as they are with the classification tasks.

4.4.3 Tree shape

Figures 4.3 to 4.6 show a much larger reduction in the number of nodes than in the depth of the trees. This suggests that the reduction in the number of nodes in the tree was primarily due to thinning of the tree (reducing the number of nodes at each level) rather than reducing the depth of the tree. An additional series of experiments was performed with extra data collection to verify this. There were 50 runs for each method. The coin and faces datasets plus the second regression task were examined. Figure 4.7 shows the resulting average number of nodes at each level (depth) for the coins dataset. Figures 4.8 and 4.9 show the results for the faces dataset and the second regression task respectively. Each set of four graphs show the data at four different generations spaced through the run.

The total number of nodes in the population is proportional to the area under the curves so it is how the shape of the curves change that is of primary interest. It can be seen that the curves broaden as the run proceeds through the generations because of the increase in the average program depth; this is also true for the runs with simplification. This effect is very small for the faces dataset but more pronounced with the other two, particularly the regression task. This is as expected from figures 4.5 and 4.6 which show rate of increase in depth levelling off with the classification tasks but continuing to increase throughout the run with the regression tasks. It can also be seen that the area under the curve increases through the runs showing the increase in the number of nodes, although the effect is small for the classification tasks in the second half of the runs. on the faces dataset there is a slight tendency for simplification to move the curves to the left indicating a reduction in tree depth. With the other two tasks however the effect of simplification is to just reduce the height of the curve, rather than to narrow it or to move the peak towards a lower level. This shows that simplification is reducing the average number of nodes in the middle levels of the tree, rather than reducing the depth of the tree.

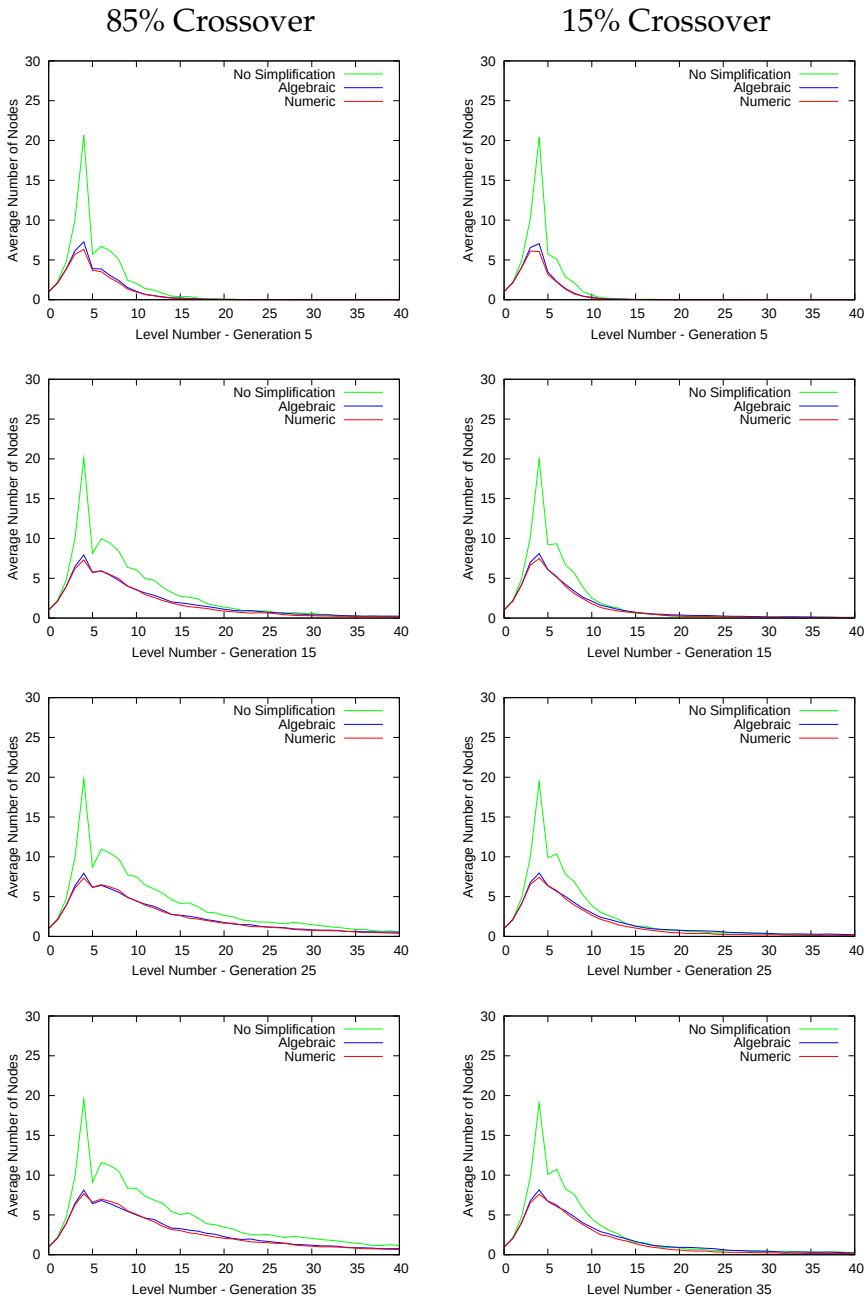


Figure 4.7: The average number of nodes at each level at four different generations on the **coins** dataset.

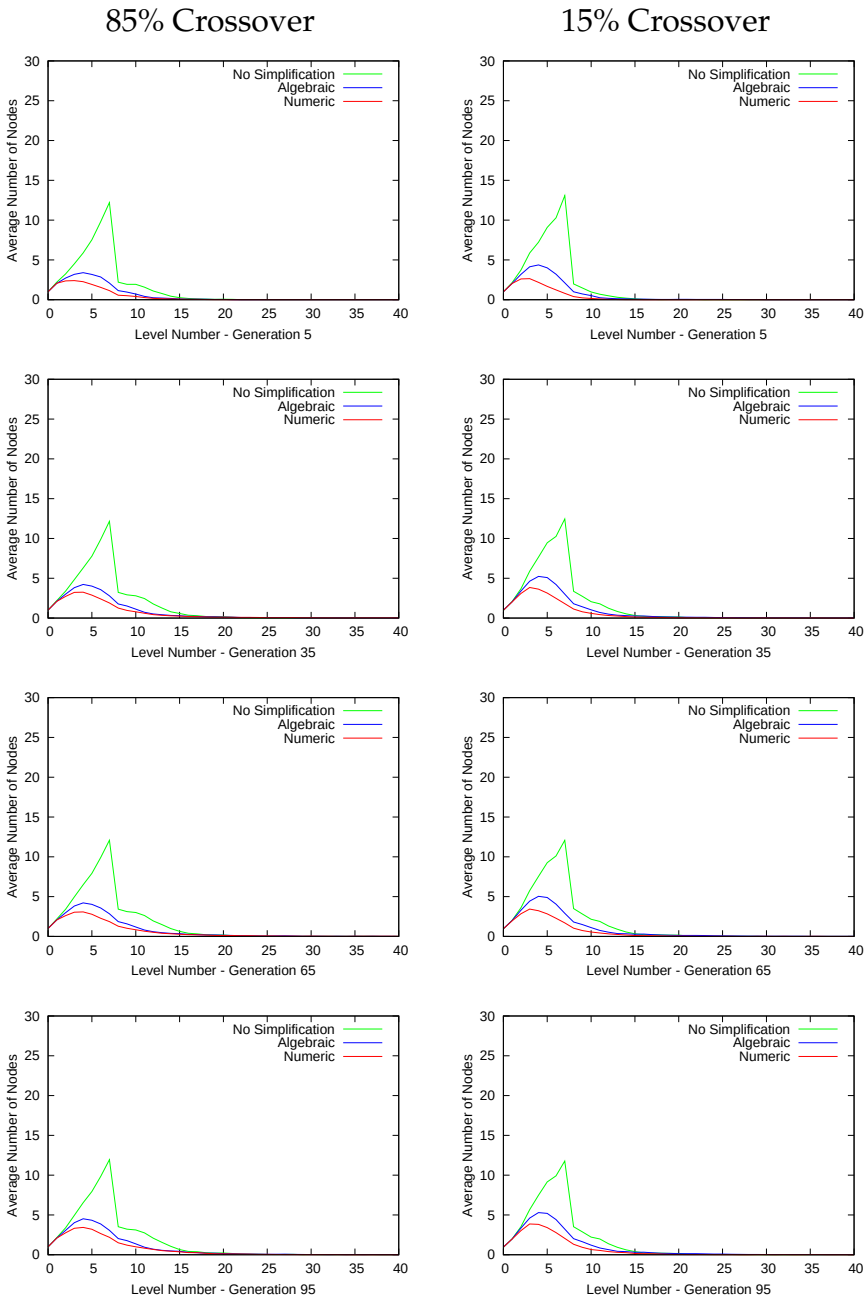


Figure 4.8: The average number of nodes at each level at four different generations on the **faces** dataset.

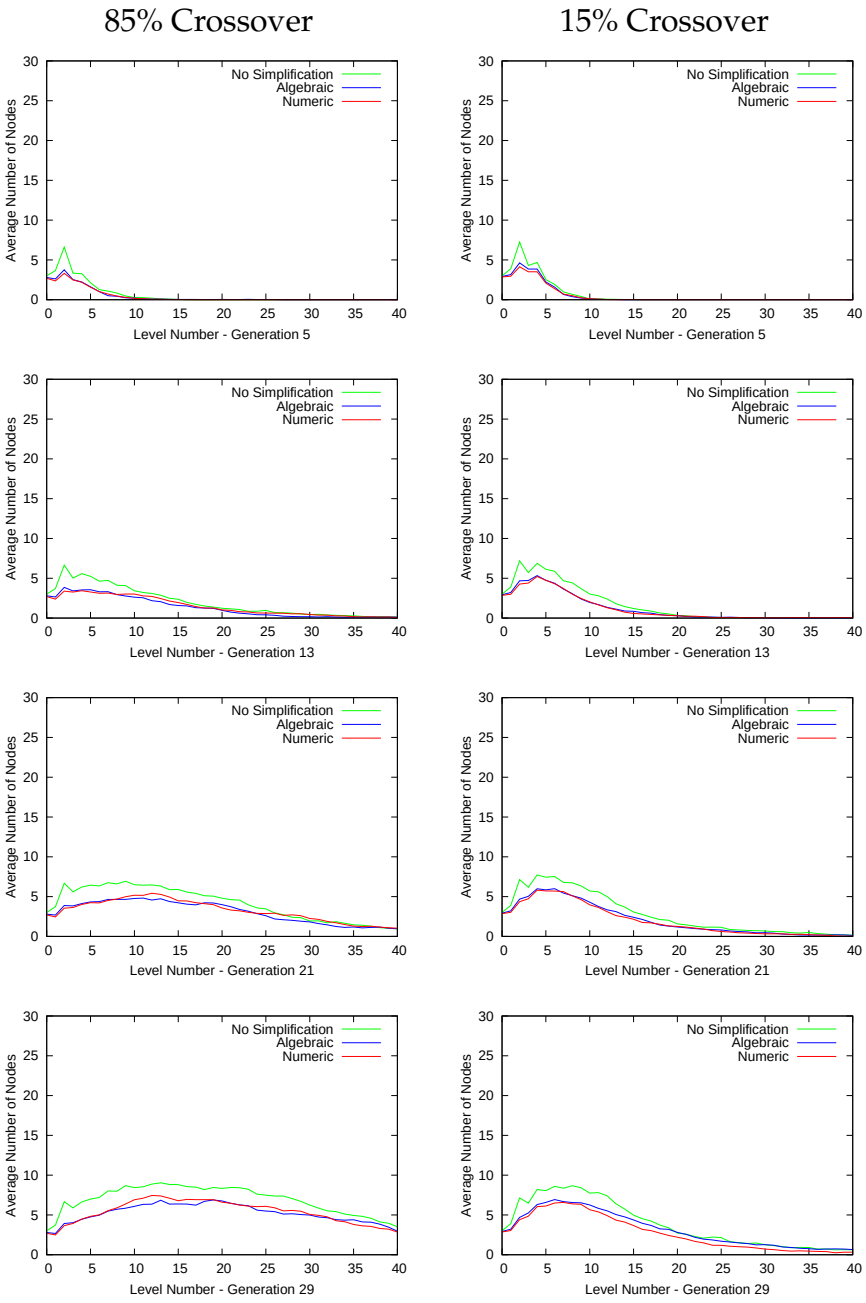


Figure 4.9: The average number of nodes at each level at four different generations on the **second regression** task.

4.4.4 Analysis of Resource Usage

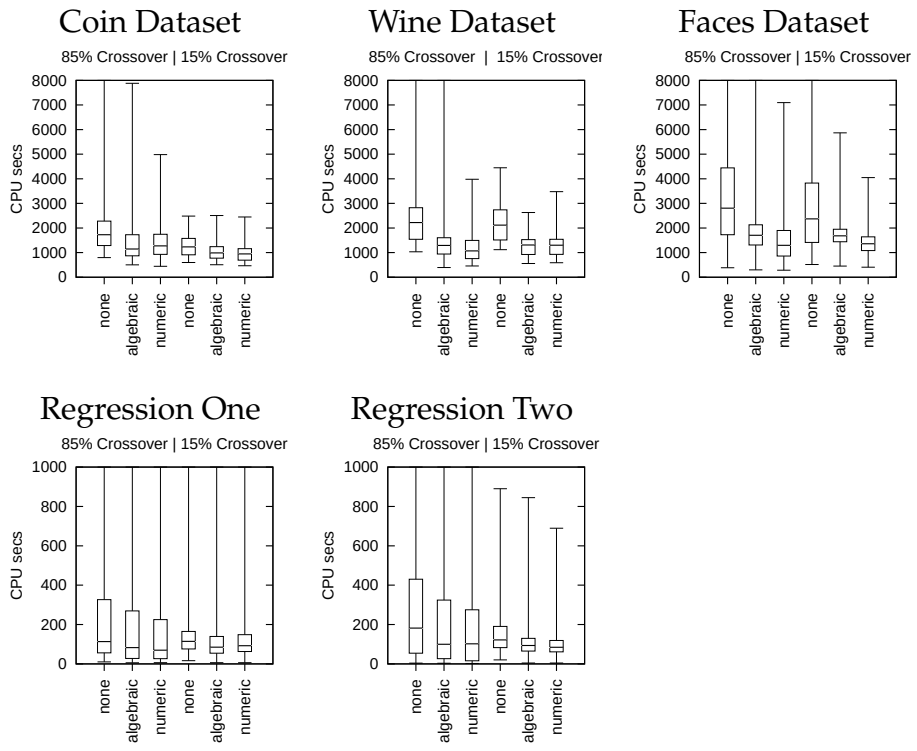


Figure 4.10: CPU used. The boxplots on each graph are (left to right) 85% crossover with no simplification, algebraic simplification, and numerical simplification, then 15% crossover with no simplification, algebraic simplification and numerical simplification.

Figure 4.10 shows boxplots for CPU time used. The plots show a wide variation in results but there is a pattern here, with simplification giving a noticeable reduction in CPU usage. The work described above shows the basic behaviour of the two simplification methods on three classification datasets and two regression tasks. It indicates that there are differences in program sizes and resource usage but does not give any indication of significance. The correlation between the starting points for the different

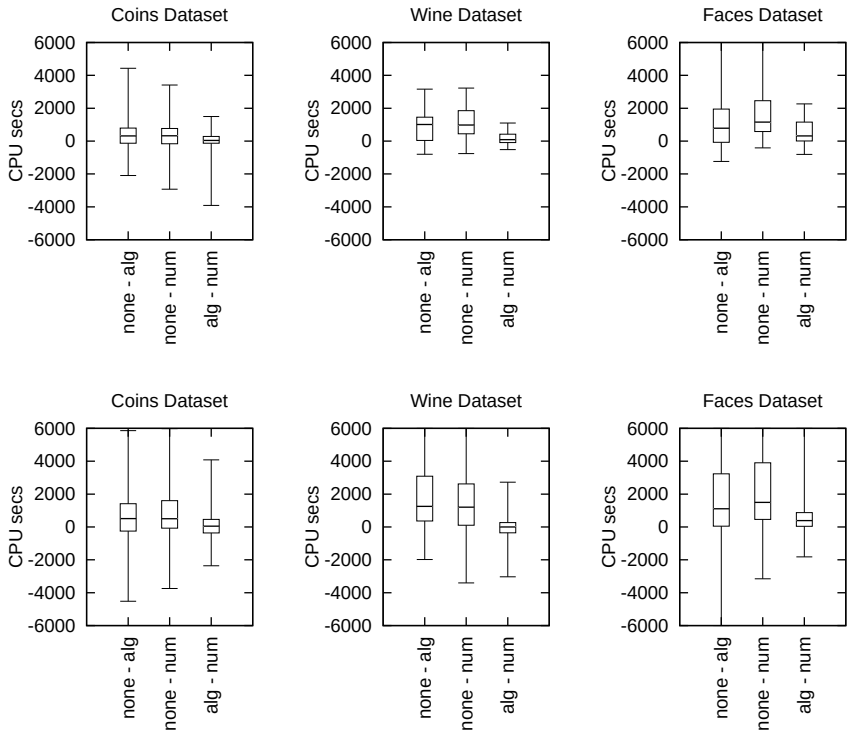


Figure 4.11: Differences in CPU resource used for the three classification datasets. The top row is for 15% crossover and the bottom row is for 85% crossover. In all graphs the first box is the difference between no simplification and algebraic simplification, the second is the difference between no simplification and numerical simplification and the third is the difference between algebraic and numerical simplification.

simplification approaches has been maximised by making each set of three experiments (one for each of *no simplification*, *algebraic simplification* and *numerical simplification*) use the same initial population, and the same split of the training set into folds for the classification tasks. Figure 4.11 shows box plots for the *differences* in CPU time for the different classification datasets and simplification methods. There is a clear advantage (differences > 0) to the two simplification methods with all datasets. The advantage is greater

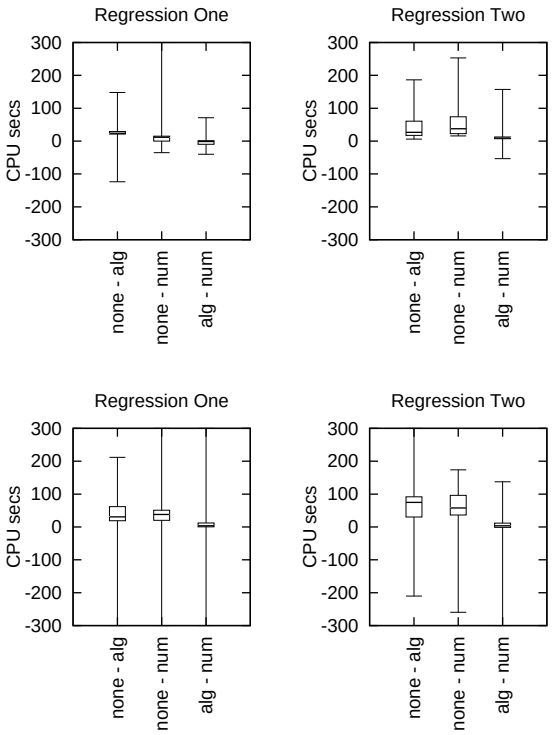


Figure 4.12: Differences in CPU resource used for the two regression tasks. The top row is for 15% crossover and the bottom row is for 85% crossover. In all graphs the first box is the difference between no simplification and algebraic simplification, the second is the difference between no simplification and numerical simplification and the third is the difference between algebraic and numerical simplification.

on the wine and faces datasets, particularly the latter. On the coins and wine datasets there is little difference between algebraic and numerical simplification, but on the faces dataset numerical simplification shows a marked reduction in CPU time compared to algebraic simplification. Figure 4.12 shows the differences for the two regression tasks. As with the classification tasks both simplification methods clearly use less time than standard GP. There is no clear difference between the two simplification

methods.

As with the program sizes, the significance of the CPU reductions was tested using the Wilcoxen signed ranks test. Table 4.2 shows the median values for the CPU time used by the various experiments. For the difference rows, the third column gives the Z score calculated using the Wilcoxon Signed-Rank test. Note that these are for a directional test. That is the displayed Z values indicate the confidence that the simplification method uses less CPU than *no simplification* or that *numerical simplification* uses less CPU than *algebraic simplification*. Where the Z score is better than the 95% confidence level, the fourth and fifth columns show the 95% confidence interval expressed as the percentage reduction in CPU time used. The fifth column shows an * for $> 95\%$ confidence, ** for $> 99\%$, and *** for $> 99.9\%$.
Each set of four lines gives the following:

1. CPU time used by the *no simplification* runs.

2. The reduction in CPU used by the *algebraic simplification* runs compared to the *no simplification* runs.

3. The reduction in CPU used by the *numerical simplification* runs compared to the *no simplification* runs.

4. The reduction in CPU used by the *numerical simplification* runs compared to the *algebraic simplification* runs.

It can be seen that for *numerical simplification* and 15% crossover on the coin data set the Z score is just under the 95% confidence level. All the other differences between *simplification* and *no simplification* are significant to at least 99% confidence. The differences between the two simplification methods show no meaningful significance for the coin dataset, and for 85% crossover on the wine dataset. The differences between them is however highly significant for 15% crossover on the wine dataset and for both cases

Table 4.2: CPU time used in seconds for the coins, wine and faces datasets, and the two regression tasks.

	Median	Z	% CPU Reduction Min	Max	
Coins 15% Crossover with no simplification	1622				
Reduction with Algebraic simplification	518	**2.58**	12	56	**
Reduction with Numeric simplification	444	1.59			
Difference between Algebraic and Numeric	34	0.15			
Coins 85% Crossover with no simplification	2420				
Reduction with Algebraic simplification	468	**2.49**	6	62	**
Reduction with Numeric simplification	798	**2.64**	7	60	**
Difference between Algebraic and Numeric	11	0.03			
Wine 15% Crossover with no simplification	2390				
Reduction with Algebraic simplification	1009	**4.75**	24	51	***
Reduction with Numeric simplification	976	**5.51**	31	57	***
Difference between Algebraic and Numeric	90	**2.15**	1	10	*
Wine 85% Crossover with no simplification	2116				
Reduction with Algebraic simplification	1254	**5.07**	43	99	***
Reduction with Numeric simplification	1207	**4.25**	34	91	***
Difference between Algebraic and Numeric	−1	0.82			
Faces 15% Crossover with no simplification	2374				
Reduction with Algebraic simplification	782	**4.03**	22	78	***
Reduction with Numeric simplification	1160	**5.93**	41	99	***
Difference between Algebraic and Numeric	315	**4.51**	11	31	***
Faces 85% Crossover with no simplification	2808				
Reduction with Algebraic simplification	1104	**3.01**	21	81	***
Reduction with Numeric simplification	1495	**4.84**	40	97	***
Difference between Algebraic and Numeric	390	**3.69**	7	24	***
Regression One 15% Crossover	114				
Reduction with Algebraic simplification	24	**11.90**	21	23	***
Reduction with Numeric simplification	21	**7.74**	6	9	***
Difference between Algebraic and Numeric	−2	1.21			
Regression One 85% Crossover	113				
Reduction with Algebraic simplification	31	**11.10**	28	38	***
Reduction with Numeric simplification	38	**9.84**	30	37	***
Difference between Algebraic and Numeric	4	**5.15**	3	6	***
Regression Two 15% Crossover	122				
Reduction with Algebraic simplification	27	**12.34**	23	34	***
Reduction with Numeric simplification	37	**12.35**	31	43	***
Difference between Algebraic and Numeric	10	**11.88**	8	9	***
Regression Two 85% Crossover	182				
Reduction with Algebraic simplification	73	**12.12**	33	40	***
Reduction with Numeric simplification	57	**11.25**	32	39	***
Difference between Algebraic and Numeric	4	**3.85**	1	3	***

on the faces dataset. On the two regression tasks the differences between the two simplification methods are small but highly significant in three of the four cases.

4.4.5 Analysis of Effect on Accuracy

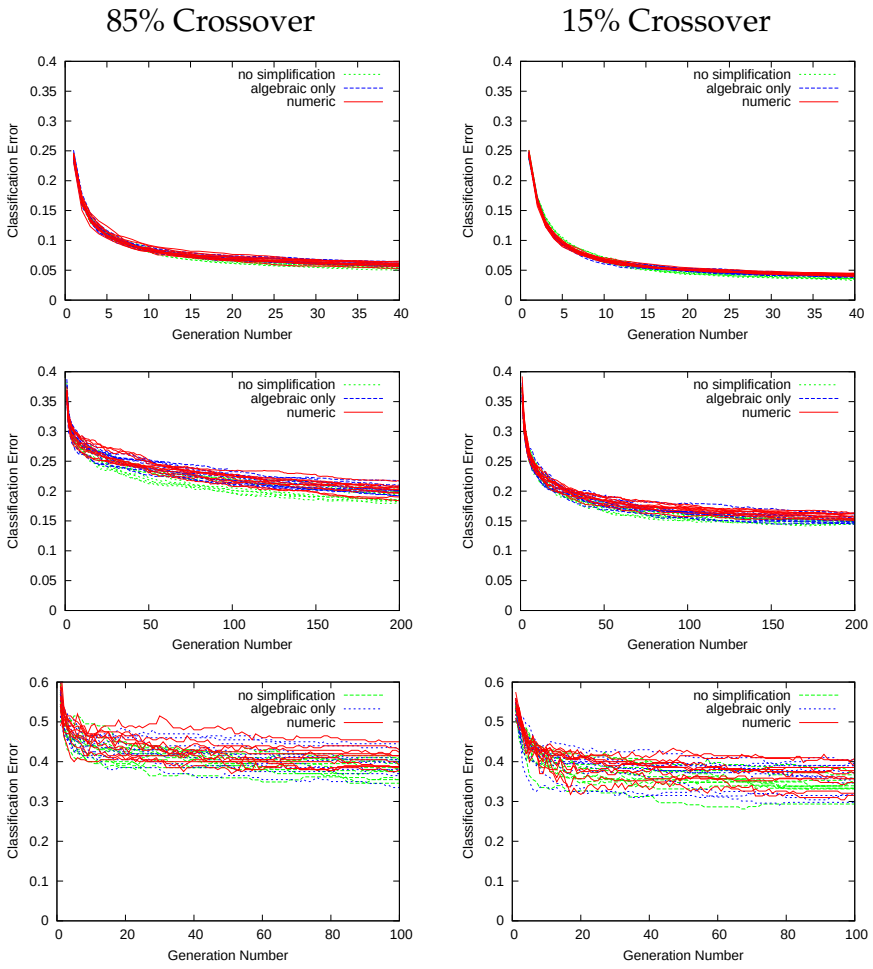


Figure 4.13: Classification performance for the coin dataset (top row), the wine dataset (middle row) and the faces dataset (bottom row).

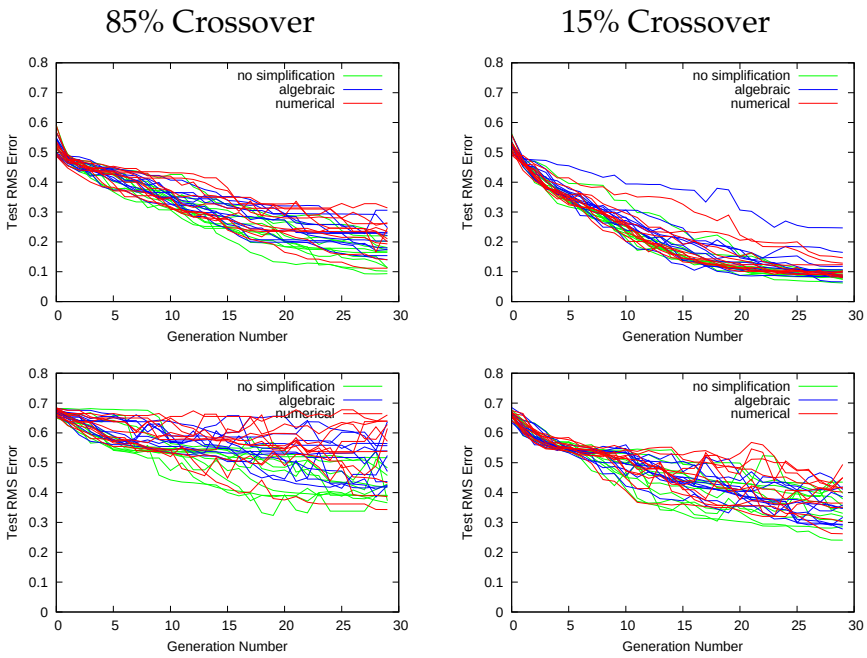


Figure 4.14: Test set performance for the two regression tasks.

Reducing the program size and computational effort is of little benefit if the accuracy suffers badly. Figure 4.13 gives the mean classification error rate on the test set for the program with the best fitness at each generation for the coin, wine and faces datasets, again showing 10 replications, each of which is the average of 20 runs. There is no noticeable difference in performance between the three levels of simplification on the coin dataset. The wine dataset shows similar results but with more variation in classification performance. The average performance for the wine dataset is a classification error rate of between 15% and 20%, but some individual runs produced classifiers with zero error rate. The classification performance also appears to be sensitive to which examples are included in the test set. The results for the faces dataset show a much wider variation in classification performance, but there appears to be very little difference between the three methods.

Note that with all three datasets, and both simplification methods, the 15% crossover case produces both smaller programs and lower classification error rates on the test set than the 85% case. This may be because of the smaller program sizes, or some other effect of the difference in genetic operators. More targeted experiments will be needed to establish this.

Figure 4.14 gives the mean RMS error for the program with the best fitness at each generation for the two regression tasks. There is considerable variation in results, particularly with the second task. This is expected because this is a much more difficult task than the first, and only some runs produced good results. There does not appear to be any difference between methods on the first task, although a few of the simplification runs have clearly produced poor results. The 15% crossover case on the second task also shows no clear difference between methods, but the number of poor performances of the simplification runs can be clearly seen with the 85% crossover rate.

These results were tested for significance and there is no statistically significant difference in effectiveness between the the three methods except for the second regression task with a 85% crossover rate, where both of the simplification methods had a marginally higher error rate than standard GP that was significant just beyond the 95% level. This regression task requires a seventh order polynomial for a good solution and it the simplification may have been a little too aggressive as while some runs performed well there was a larger number of runs that failed to find a reasonable solution than was the case in standard GP without any simplification.

4.5 Chapter Summary

The goal of this chapter was to investigate whether numerical simplification could produce smaller programs than the standard GP system with no simplification, whether it would shorten computation run times compared

with the standard GP system with no simplification, whether it would affect the system effectiveness and how numerical simplification performs compared to algebraic simplification.

Three classification and two regression tasks were used to investigate program sizes both in total number of nodes and in tree depth. The effect on the shape of the program trees was also examined. The amount of CPU used by the GP runs was examined and the classification performance/RMS error was examined. The experiments were organised in correlated sets that allowed easy comparison of differences in size, performance or resources used. Any differences observed were tested for statistical significance.

Overall, the two simplification methods almost always resulted in smaller programs and used much shorter evolutionary training times to produce similar classification performance. *Numerical simplification* performs at least as well as *algebraic simplification* and in some cases performs better than *algebraic simplification*.

The next chapter examines whether there is an optimum value for the simplification threshold, and if is, what relationship it may have to the noise level in the input data.

Chapter 5

Simplification Threshold and the Noise Level

5.1 Introduction

The motivation for numerical simplification came from the idea of treating values less than the noise component of the input data as being zero. This chapter aims to establish whether the expected relationship between the noise amplitude and the optimum value for the simplification threshold does in fact exist.

5.2 Chapter Goals

This chapter investigates the second goal of this thesis, namely:

Is there a relationship between the optimum value for the simplification threshold in numerical simplification and the amount of noise in the input data? This question is broken down into the following sub-questions:

1. Is there an optimal value for the simplification threshold?

2. If there is an optimal value for the simplification threshold, is there a relationship between that optimal value and the amount of noise in the input data?

3. If this relationship exists then what is it?

5.3 Datasets and Experimental Setup

The amount of noise in the input data for a classification task is not usually known, therefore this chapter uses the two symbolic regression tasks detailed in chapter 3. This allows complete control over the amplitude and distribution of the noise in the input data.

Each of the training and test datasets contain 200 points randomly selected from the given range. The test set contains exact values and the training set has noise randomly added up to the noise level. If the noise level is 0.01 then each function value in the training set has a random number from the range $[-0.01, 0.01]$ added to it.

There are many possible sources of noise in input data, which include (but are not limited to):

1. A feature may be a combination of real world variables for which we do not have individual values, some of which do not contribute to forming a good solution. For example, a feature $f()$ may in reality be a sum of two other lower level features that we do not have values for $f() = g()+h()$. $g()$ may be important for evolving a good solution but $h()$ is not. In this situation $h()$ is effectively just noise.

2. A feature may be a measurement of a physical quantity that includes noise due to thermal effects or background radiation such as EMF interference.

3. A feature may be the result of a physical measurement with restricted resolution or accuracy.

In the first case, the distribution of the noise component could be almost anything. If the feature is to be of value in forming a solution then the noise can not be too large in size but that is all that can be concluded. In the second case the noise distribution usually approximates a normal distribution. In the third case the noise is the measurement uncertainty, and is bounded with a distribution that is at least close to being uniform. For these experiments, having the noise amplitude bounded makes it easier to draw firm conclusions and therefore this is the distribution used in this thesis. The noise added is uniformly distributed over an interval that is a parameter of the experiment.

The fitness function is the *root mean squared error* (RMS error):

$$fitness = \sqrt{\frac{\sum_{i=0}^{n}(P_i - T_i)^2}{n}}$$

where P_i is the output of the program for the i^{th} row of the training set and T_i is the target value given by the training set. The final test score is the RMS error calculated on the test set.

The terminal set consists of the single x feature and random "constant" numbers. The function set consists of the four standard arithmetic functions (addition, subtraction, multiplication and protected division). The population size is 2000. Initial programs are four levels deep. Tournament selection is used with a tournament size of four. All experiments were run for 30 generations. Initial tests showed that there was little or no improvement past this number of generations. These parameter values were determined using heuristic guidelines and preliminary trials.

These experiments required a very large number of GP runs, therefore only the standard higher crossover rate set of evolutionary parameters was used: 5% reproduction, 85% crossover and 10% mutation. Simplification is performed after the first generation, and every fourth generation thereafter. Note that there is no maximum program size.

The runs were done in sets, each set contained a number of individual runs with different values for the simplification threshold. All of the

runs within a set used the same starting population and the same starting seed. This was done to allow significance tests to be performed between pairs of experiments within the set if this was required. Experiments were run with two different noise levels 0.001 and 0.01 for each of the two regression tasks. For each of these noise levels experiments were run with a roughly logarithmic series of values for the simplification threshold as follows: 0.0001, 0.00017, 0.00035, 0.0006, 0.001, 0.0017, 0.0035, 0.006, 0.01, 0.017, 0.035, 0.06, 0.1
Each combination of task, noise level and simplification threshold was run 100 times.

5.4 Experimental Results

The results from these experiments have many outliers and deviate substantially from the normal distribution. Therefore the results are presented using the median rather than the mean, and the first and third quartiles instead of the standard deviation. In the graphs that follow in this chapter, the solid black line shows the median value for the quantity being displayed. The error bar style vertical bars show the first and third quartiles and the red horizontal line shows the median value for the same experiment without simplification.

5.4.1 Regression Task One

This task is a fairly straight forward one, and good solutions are consistently found.

Noise Level = 0.001

The program sizes in both levels and total number of nodes are shown in figure 5.1. These figures show very little variation, either as the simplification threshold is changed or between simplification and no simplification.

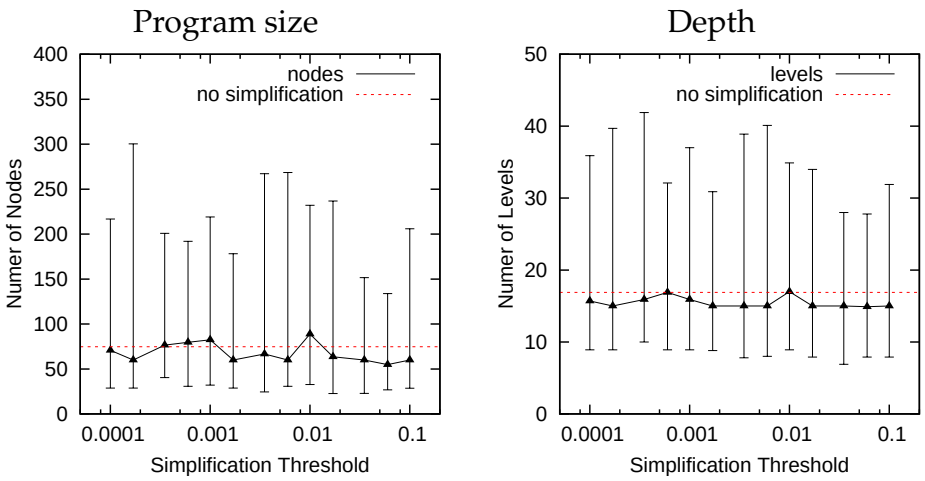


Figure 5.1: Program sizes for a noise level of 0.001. The left-hand graph shows the program size in nodes and the right-hand graph shows the depth of the program tree. The error bars show the first and third quartiles at that point.

The range of variation is very wide. Figure 5.2 shows the CPU time in seconds taken by the runs. Again there is little difference between the two methods, but in general numerical simplification takes a little less time than without simplification.

Figure 5.3 shows the fitness and test scores for this task and noise level. The fitness scores for the numerical simplification case are equal to, or better than, the scores without simplification but the variation in scores is very large. The test scores show a more interesting pattern. There is little variation in test scores until the simplification threshold reaches 0.01, at this point the test scores start to get worse, with the median and the quartiles all getting worse as the simplification threshold continues to increase. This suggests that useful information is starting to be lost in the simplification and the resulting programs are loosing accuracy.

For this task and error level there does not appear to be an optimum value for the simplification threshold, although the test scores would sug-

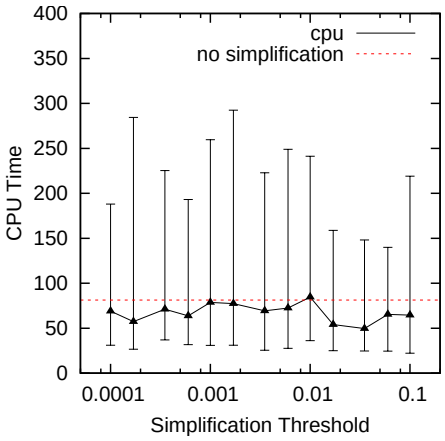


Figure 5.2: CPU usage for a noise level of 0.001. The error bars show the first and third quartiles at that point.

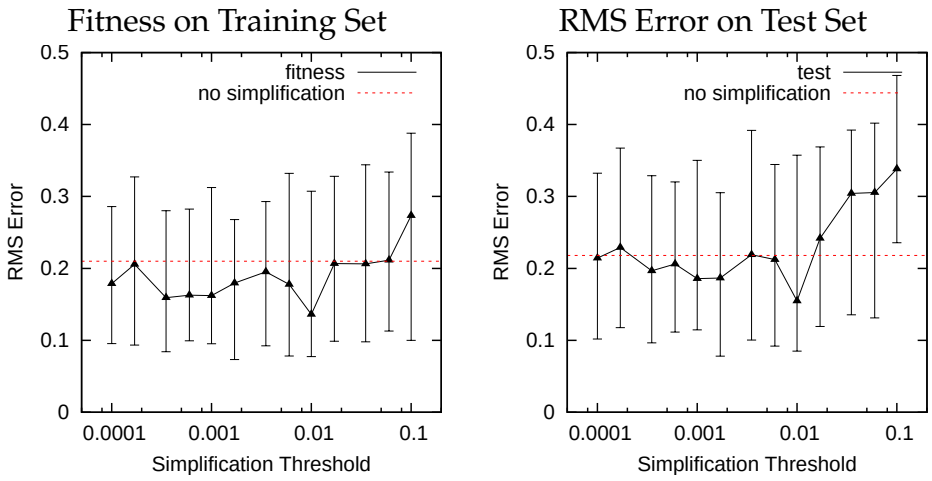


Figure 5.3: Program effectiveness for a noise level of 0.001. The left-hand graph shows the fitness values. The right-hand graph is the RMS error on the test set. The error bars show the first and third quartiles at that point.

gest that it should not be larger than 0.01.

Noise Level = 0.01

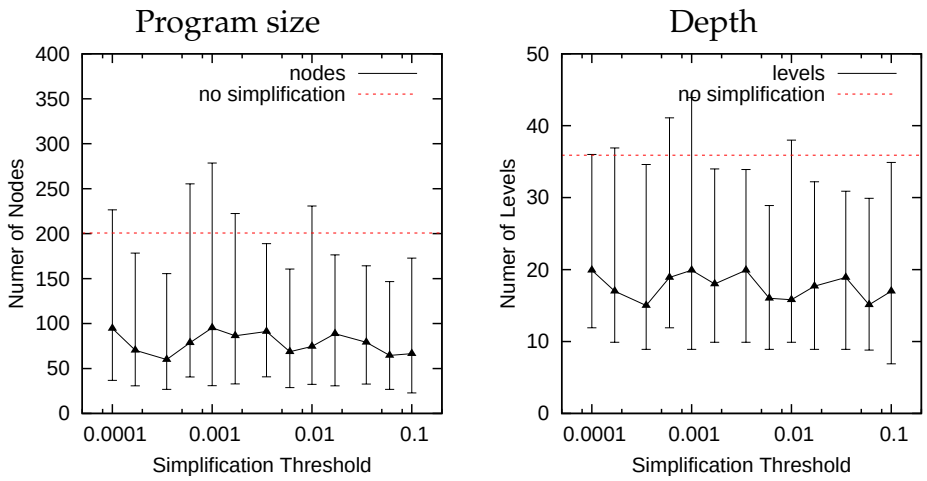


Figure 5.4: Program sizes for a noise level of 0.01. The left-hand graph shows the program size in nodes and the right-hand graph show the depth of the program tree. The error bars show the first and third quartiles at that point.

The noise level is now increased by an order of magnitude to 0.01 on the same regression task. Figure 5.4 shows the program size for this noise level. At this noise level there is a clear difference between the two methods. The program sizes for the numerical simplification method are similar to the lower noise level given earlier(figure 5.1), but the sizes without simplification have increased considerably. The CPU usage figures are given in figure 5.5. These show the same substantial increase for the standard GP case. This is as expected given the increase in program sizes. There is still no indication here of any pattern that might indicate an optimum value for the simplification threshold.

The fitness and test scores for this noise level are shown in figure 5.6. The fitness values show very similar results both with and without simplification although there appears to be a slight rising trend. The test scores

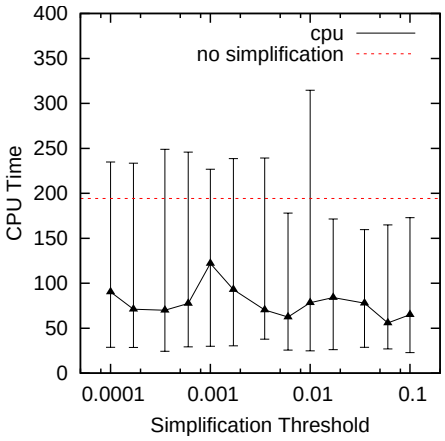


Figure 5.5: CPU usage for a noise level of 0.01. The error bars show the first and third quartiles at that point.

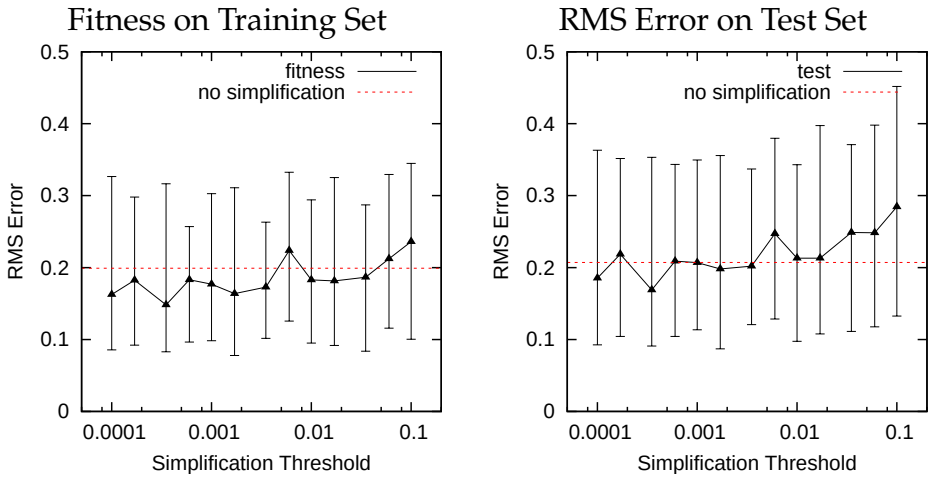


Figure 5.6: Program effectiveness for a noise level of 0.01. The left-hand graph shows the fitness values. The right-hand graph is the RMS error on the test set. The error bars show the first and third quartiles at that point.

show little variation with simplification threshold up to the 0.035 when they start to rise until at 0.1, where they are 50% higher (greater error) than the median value from the runs with no simplification. This is the same

pattern that was seen with the lower noise level. The rise is not as pronounced but it occurs at a higher value of the simplification threshold. As with the lower noise level there is no clear optimum value for the simplification threshold. At this noise level there is a clear advantage in the CPU time required and in program size and therefore memory requirements, but this advantage is already present at the smallest threshold tested. As with the lower noise level there does however appear to be an upper limit before the error on the test set starts to rise. This was 0.01 on the low noise case and 0.035 with the higher noise level.

5.4.2 Regression Task Two

This task requires a larger program than the first task for good results and is therefore more difficult to evolve a good solution. Good solutions are possible but many runs do not find one and this is reflected in the results that follow.

Noise Level = 0.001

Figure 5.7 shows the program sizes for this task with the low noise level. As expected the median program sizes are larger than they were with the first task, but with this task there is a fairly consistent downward trend in program sizes as the simplification threshold is increased, although there is a sharp increase in sizes between the first point at 0.0001 and the second point at 0.0002. Why this might be the case is unclear but it may be just chance as the first and third quartiles do not show as clear an increase. This reduction in program sizes as the simplification threshold is increased suggest that extra simplification opportunities are being found as the threshold is relaxed. This did not happen with the first regression task. The CPU run times are shown in figure 5.8. As expected these show the same downward trend as the simplification threshold increases. As with the higher noise level on the first regression task, the program sizes

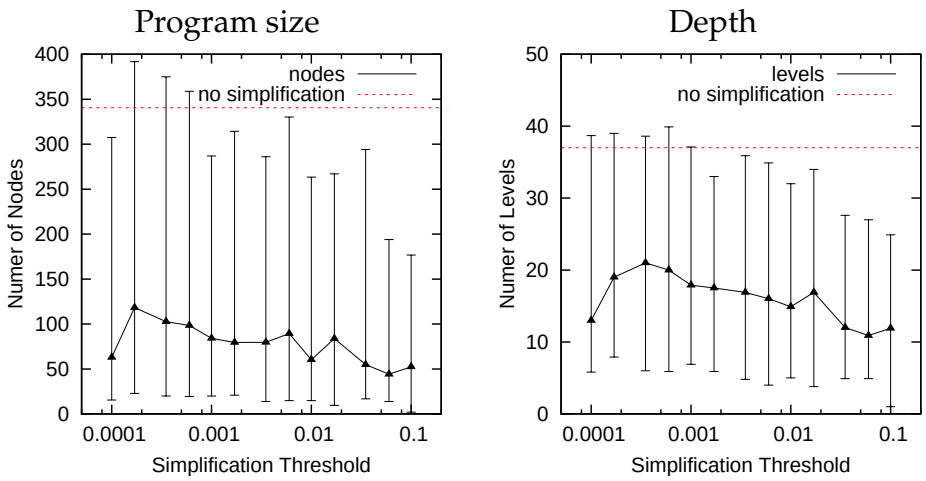


Figure 5.7: Second regression task program sizes for a noise level of 0.001. The left-hand graph shows the program size in nodes and the right-hand graph shows the depth of the program tree. The error bars show the first and third quartiles at that point.

and CPU times are much lower with numerical simplification than the median values from the runs with no simplification.

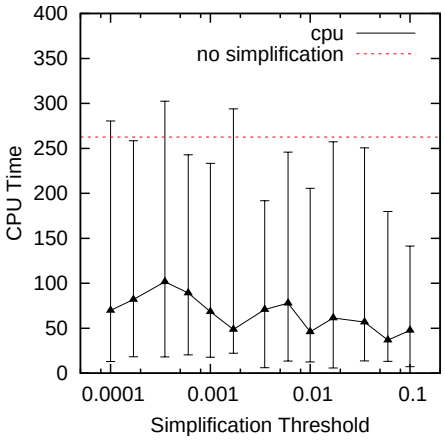


Figure 5.8: Second regression task CPU usage for a noise level of 0.001. The error bars show the first and third quartiles at that point.

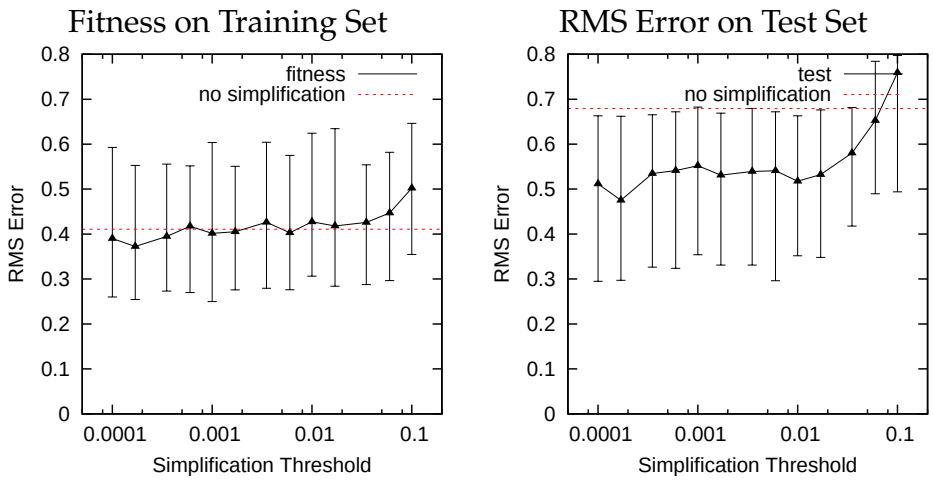


Figure 5.9: Second regression task program effectiveness for a noise level of 0.001. The left-hand graph shows the fitness values. The right-hand graph is the RMS error on the test set. The error bars show the first and third quartiles at that point.

The fitness and test scores are shown in figure 5.9. In the first regression task these stayed essentially stayed flat until about 0.01, but in this task there is a slight rising trend. This may indicate that the reduction in program size is resulting in a slight loss in accuracy. The most interesting point is the test results. There is again a sharper increase in test error from a simplification threshold of about 0.01 but the error is considerably smaller with simplification than without, even though there is little difference between the fitness scores. This suggests that without simplification there is over fitting occurring. The much smaller programs resulting from numerical simplification are avoiding that problem.

This task and noise level again shows no strong indication of an optimum value for the simplification threshold, although as with this noise level on the first regression task, 0.01 would appear to be a practical upper limit. Below this level there is a very slight advantage in resource usage at higher values and a very slight advantage in test error at the smaller end

of the range.

Noise Level = 0.01

Again the noise level is raised to 0.01, and figure 5.10 shows the program sizes. This shows a very similar pattern as the lower noise case on this task. There is the same trend of falling program sizes as the simplification threshold is increased and the sizes are much smaller than without simplification. The CPU used is shown in figure 5.11 and shows the same reducing trend as would be expected.

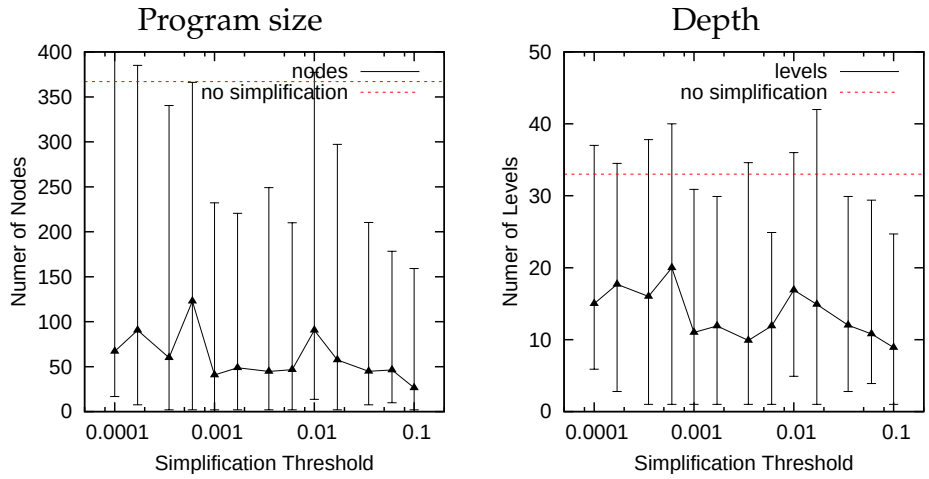


Figure 5.10: Second regression task program sizes for a noise level of 0.01. The left-hand graph shows the program size in nodes and the right-hand graph shows the depth of the program tree. The error bars show the first and third quartiles at that point.

The fitness and test scores with this noise level are shown in figure 5.12. Compared to the lower noise case the fitness scores have risen slightly, but there is still little difference between whether numerical simplification is used or not, and there is little variation across the range of values for the simplification threshold. The test scores again show over fitting in the runs

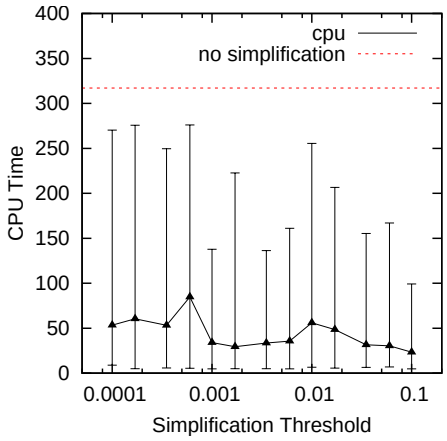


Figure 5.11: Second regression task CPU usage for a noise level of 0.01. The error bars show the first and third quartiles at that point.

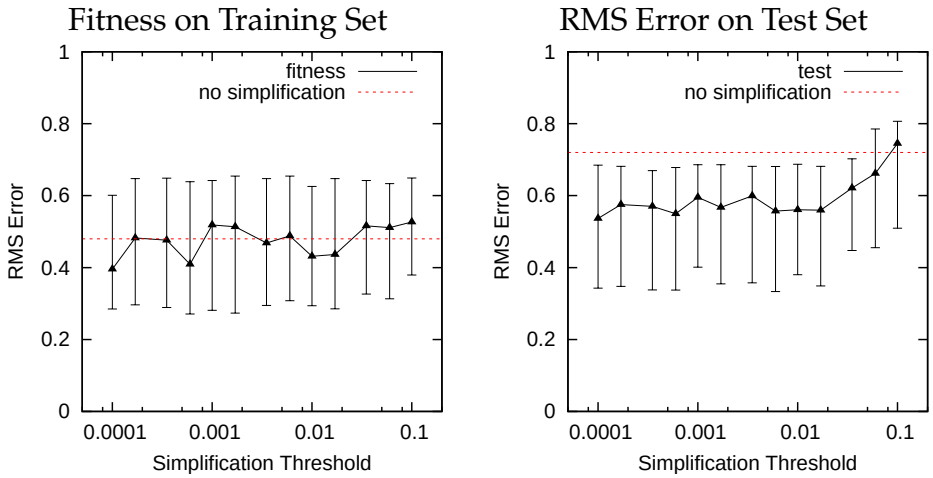


Figure 5.12: Second regression task program effectiveness for a noise level of 0.01. The left-hand graph shows the fitness values. The right-hand graph is the RMS error on the test set. The error bars show the first and third quartiles at that point.

without simplification. The numerical simplification runs show test scores much closer to the fitness scores and with little change as the simplification

threshold increases until a threshold of 0.035 when, like the first regression task, the error on the test set shows a marked increase as the threshold continues to increase.

5.5 Chapter Summary

This chapter investigated the second goal of this thesis. That is, whether there is an optimum value for the simplification threshold in numerical simplification, and if there is, what the relationship is between that optimum value and the amount of noise in the input data.

Experiments were performed on two regression tasks, with two noise levels used on each task. The results show that there is no clear optimum value for the simplification threshold. For three of the four cases examined there were clear advantages in program size and and CPU used when numerical simplification was used, and on the second regression task numerical simplification substantially reduced the over fitting that was occurring in the runs using standard GP with no simplification. These advantages were already present with the smallest simplification threshold used and did not improve in any significant way as the threshold was increased. However there does appear to be an upper limit on the threshold. This is to be expected, if the simplification threshold is too large then detail will be discarded that is required to form a good solution. In general, the program size and run times reduce very slightly as the simplification threshold is increased. Then, at a simplification threshold of somewhere between 0.01 and 0.04 the RMS error rate on the test set rises sharply, and continues to rise as the simplification threshold continues to increase. This point where error sharply increases appears to be related to the noise level in as much as it was 0.01 for the lower noise level of 0.001 and rose to about 0.035 when the noise level was 0.01. For practical purposes, a conservative choice would be 0.0001, there is little to gain from using a larger value and some risk unless the dataset is well understood.

Chapter 6

Fragment Analysis using Images

6.1 Introduction

The previous two chapters have presented the numerical simplification method and examined the effect it has on program sizes and effectiveness. This chapter and the following chapter investigate whether simplifying programs during the GP run changes the nature of the GP population and whether the simplification process has changed how the evolutionary process functions. This has been accomplished by considering the fragments that make up the programs in the population, their distribution both in frequency of occurrence and in coverage of the search space, and how these distributions change through the generations. This thesis uses the term *fragment* to mean a subtree of fixed depth, which can occur at any position in the program tree therefore the leaves of the fragment are not necessarily terminals. This chapter uses images to visualise the distribution of fragments by generation within the population. Two different encoding schemes are developed that allow the distribution of two or three level deep fragments to be displayed as images.

Wong and Zhang [79] examined the effect of algebraic simplification on fragments (they used the term building blocks) by tracking just the numeric terminals in the population. They showed that while *algebraic*

simplification does disrupt fragments (building blocks) by destroying or changing constants, it is also capable of creating new values that were not originally present and some of these new values became established in the population thus contributing to the final solution.

6.2 Chapter Goals

The goal of this chapter is to show the distribution of fragments within a population and to show if the nature of this distribution is changed by simplifying programs during the run. It examines the fragment distributions of canonical GP with no simplification and compares this to the distributions resulting from using the numerical simplification method and the algebraic simplification method. The specific research objectives are:

- How the fragments are distributed in canonical GP with no simplification and the two simplification methods;
- Whether the two simplification methods destroy existing fragments;
- Whether the two simplification methods generate new fragments;
- Whether and how the general fragments (subtrees) analysed in this chapter behave differently from the constant terminals analysed in [79].

6.3 Encoding Schemes

In any practical problem, the number of possible fragments is huge. What is needed is a way of showing the distribution of the fragments present in the population with an indication of relative frequency. The requirement is to show both the distribution relative to the other fragments in the population and also the coverage across the whole search space. This is a non-trivial problem, particularly as the size of the fragments increase.

To deal with this problem, each fragment is encoded into a bit string. The distribution of the encoded fragments is then converted into a two-dimensional image, with fragment encodings on the vertical axis and generations on the horizontal axis. The encoding is done in such a way that similar fragments generally result in similar encodings, which allows the resulting image to show the coverage of the search space. To make the presentation of the images practical, the resulting image is to be less than a page in size while still having at least one print dot per possible encoding. Even at 1200dpi there are a very limited number of pixels and therefore bits available for encoding the fragments. The encoding process will therefore simplify the description of the nodes to keep the size of the bit string manageable. The larger the fragments being examined, the larger the number of possible different fragments and the greater the simplification that will be required to keep the resulting bit string small enough to be displayed using an image.

6.3.1 Encoding Three Level Deep Fragments

A fragment that is three levels deep, with all operators having two arguments, has five nodes if one of the second level nodes is a terminal and seven nodes otherwise. At a print resolution of 1200dpi an encoding 13bits long would result in an image 6.83 inches (173.5mm) high. That would allow about two bits per node. This encoding scheme can therefore give only the most general indication of the structure of the fragments.

With two bits per node there are four available encodings for each node, which have been used as follows:

00 — *addition* or *subtraction* operator

01 — *multiplication* or *division* operator

10 — input feature

11 — numeric terminal (ephemeral constant)

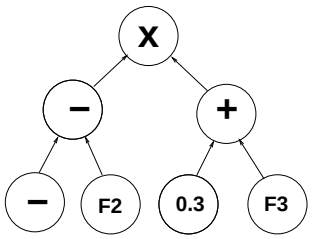


Figure 6.1: An example tree of three levels deep.

Because the root node of a fragment at least two levels deep must be an operator, the first bit will always be zero leaving 13 bits describing a three deep fragment, and an image 6.83 inches (173.5mm) high at 1200dpi which is feasible for printing on an A4 page.

At each generation all the programs in the population are traversed, encoding all fragments that are three levels deep. In each case the nodes are encoded one level at a time, starting from the root, and from left to right within the level. Any missing nodes in the five node tree are encoded as 00 without loss of generality. Figure 6.1 shows an example three-deep subtree. In this example the order the nodes would be encoded is $\times - + - F2\ 0.3\ F3$. This would be encoded as [1] [00] [00] [00] [10] [11] [10], i.e., 1000000101110.

At the end of the run an image is created with a height of $2^{13} = 8192$ pixels, one for each possible encoding and the width being a convenient multiple of the number of generations. All non-zero counts are normalised on to a range 0–191 with the highest count being assigned a pixel value of 0 (pure black) and 191 being the lowest non-zero count. All encodings with a zero count take a pixel value of 255 (pure white). This makes a clear visual greyscale distinction between a low count and zero. Examples of these images are given in figure 6.4.

Obviously this is a very coarse and simplistic representation, and there are a number of weaknesses. No account is taken of the operators $+$ and $\times$ having symmetric arguments. The communicative nature of these op-

erators mean that there may two different encodings from different fragments that are algebraically equivalent. Another problem is that this encoding scheme does not distinguish between addition and subtraction or between multiplication and division. There is no distinction between features or between different constants. It does however allow three deep fragments to be encoded and their distributions visualised.

6.3.2 Encoding Two Level Deep Fragments

To overcome the limitations of the 2 bit encoding scheme described in the previous section, while still limiting the length of the encoding to no more than 13 bits, the fragment being encoded has to be reduced in size to only two levels deep which, for binary operators, has only three nodes. There are then more bits available to describe each node. The root node is always an operator and uses two bits with the values:

00 — *addition* operator

01 — *subtraction* operator

10 — *multiplication* operator

11 — *division* operator

The two child nodes each have five bits, used as follows.

1 followed by four bits is a feature, with the four bits being the feature number.

01 followed by three bits is an ephemeral constant with the absolute value used to map the interval $[0.0, 1.0]$ onto the integer range $[0, 7]$ allowed by the three available bits. If C is the absolute value of the constant, then it is encoded as follows:

$C < 1.0 \quad \Rightarrow \quad C * 8.0$ truncated to an integer

$C \geq 1.0 \quad \Rightarrow \quad 7$

So a value of 0.32 is encoded as 2, being $0.32 \times 8.0 = 2.56$ truncated to an integer.

000 followed by two bits is an operator, enumerated as for the root node.

This encoding scheme allows more information about the fragment to be included in the encoding than was the case for the three level deep fragments, but still keeps the resulting image to a size that is feasible for printing at just $2^{12} = 4096$ pixels in height.

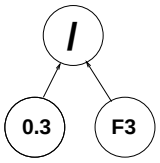


Figure 6.2: An example tree of two levels deep.

In the example of Figure 6.2, the order of encoding is $/\ 0.3\ F3$ where $F3$ is the third input feature. This then encoded as [11] [01 010] [1 0011], i.e., 110101010011.

Note that there is still some redundancy due to the symmetry of the $\times$ and $+$ operators. As long as the dataset does not have more than 16 features then the encoding is a complete description of the operators and features, with only the constant terminals having some loss of precision.

6.4 Experimental Setup

The coins, wine and faces datasets were used in these experiments. The terminal set for each dataset consisted of the features used in that dataset and random "constant"numbers. The function set for all datasets consisted of the four standard arithmetic functions (addition, subtraction, multiplication and protected division). The fitness function used the error rate

on the training set. Experiments are all conducted with the same set of population parameters. The parameters were set after some initial trial experiments and were chosen to provide reasonably good results across all the datasets and methods rather than being optimised for any one dataset or method. The population size was 200. Initial programs were five levels deep. Tournament selection was used with a tournament size of four. The coin dataset was run for 40 generations, the wine dataset for 200 generations and the faces dataset for 100 generations. As with the experiments in the earlier chapters, two sets of parameters were used when creating the next generation. One set of experiments used 5% reproduction, 85% crossover and 10% mutation; the second set of experiments used 5% reproduction, 15% crossover and 80% mutation. Where simplification was used it was performed after the first generation, and every fourth generation thereafter. Note that there was no limit on the maximum program size. The runs were done in sets of three, one for each of *no simplification*, *algebraic simplification*, and *numerical simplification*. Because fragment distributions are being compared between the three runs in each set, the three runs in each set used the same starting population. The three runs for the same set used the same starting seed, with each set using a different seed.

6.5 Results and Discussion

The particular runs and their associated images that are presented here are representative of the behaviour observed across many other similar runs. The images have been stretched in width to make them easier to see, so each generation for the coins dataset is 15 pixels wide, 3 pixels for the wine dataset and 6 pixels for the faces dataset, this gives a common size of image. Similarly the images for the two level fragments have been scaled vertically to match those for three level fragments.

6.5.1 Three Level Fragments

Figure 6.4 shows three images for the coins dataset using a crossover rate of 85%, one example individual run from each of (*no simplification*, *algebraic simplification* and *numerical simplification*). The three runs all used the same initial population. They also used the same seed for the random number generator. The horizontal axis represents the 40 generations, and the vertical axis represents each of the possible 8192 encodings. Where one of the two second level nodes is a terminal (either a feature or a constant) there is a small block of encodings that do not describe a valid subtree because these second level terminal nodes can not have child nodes. These areas are labelled with an X. In all these figures, continuous solid lines show fragments retained during evolution, while discontinuous "lines" (or scattered dots/short lines) indicate fragments were disrupted. The darkness of the lines/dots indicate the relative frequency of occurrence of the fragment within the population.

Enlarged views of a densely populated area and a sparsely populated area from one of the numerical simplification runs are shown in Figure 6.3. These two views clearly show some behaviours that are common throughout the runs that follow. In Figure 6.3(a) there are three solid dark lines that start early in the run and are then present throughout the remaining generations. These show fragments that occur frequently in the population (dark line), and remain present throughout the remaining generations without being disrupted (continuous line). There is also one grey solid line that shows a less common fragment that remains in the population without disruption.

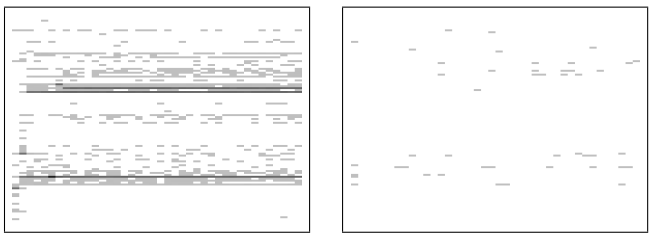
There are groups of lines close together vertically with some breaks in the lines. These show groups of similar fragments. They probably differ in only one of the third level nodes. They are less common, as shown by their grey colour, and there is occasional disruption shown by the breaks in the lines. These changes in the third level are then often reversed only one or two generations later. Elitism is being used, in these experiments the fittest

5% of the population automatically form part of the next generation, so the fact that these fragments momentarily disappear from the population indicates that they are not part of the fittest individuals in the population.

There are many fragments in the initial population that disappear after the first generation. This is common in most of the runs using simplification and indicates fragments that either can be simplified into a different fragment, or in the case of numerical simplification, provide little contribution to the result of their parent node and are therefore simplified. The first simplification occurs after the first generation. Similarly there are many other fragments that are created at the same point as a result of these simplifications. This behaviour is not seen on the wine and faces datasets when there is no simplification, but the removal of many fragments after the first generation does still occur on the coin dataset without simplification. The most likely explanation is that these fragments occurred only in low fitness individuals that did not win a tournament and therefore could not pass their fragments on to the next generation.

Figure 6.3(b) shows a sparsely populated part of the search space. There are few fragments and the light grey colour shows that they are not common. The lines are all short. This implies that the program or programs they were part of had low fitness, because they have not contributed to the next generation, and the fragments concerned have therefore disappeared from the population.

Figure 6.4(a) shows an example run on the coins dataset without simplification. Fragments that remain in the population from early in the run can be seen in areas B, C, D, E, F and G. Areas E and F show large numbers of related fragments, a few of which remain for extended periods but most only survive a few generations before being removed from the population. This is a pattern of behaviour that is typically caused by the crossover operator swapping parts of the fragments, generally at the third level. Similarly in areas B, E and F there are equally spaced groups of three or four lines that are periodically disrupted. The lines in the group differ



(a) (b)

Figure 6.3: An enlarged view of (a) a densely populated area and (b) a sparsely populated area.

in only one particular third level node.

The most important thing in these figures is the differences in behaviour between canonical GP with no simplification and the two simplification methods. The three runs shown in figure 6.4 all have the same initial population, so the differences are due to the algorithm and the stochastic nature of the GP process. Region A in all three runs shows only a very small number of scattered fragments none of which remain in the population for more than a few generations. Region B in the canonical GP run has one long lived fragment and several related fragments. These are much more fragmented in the numerical simplification run in image (c). Region C has only a few fragments, although one of them is quite persistent albeit not common, as shown by the line being light grey in colour. This fragment has remained in the population continuously in the canonical GP run but has been disrupted and re-established at different generations in the two simplification runs. Region D is similar in all three runs while region E shows a similar pattern to region B with fewer fragments in the simplification runs including the loss of the one persistent fragment in the canonical GP run. Region F has a large number of different fragments in the canonical GP run but only one is common (black line) and they do not survive for many generations at a time. The simplification runs show fewer fragments in this region but several of them remain in the popula-

tion for more generations at a time then was the case in the canonical GP run. Regions G and H have only a few, generally long lived, fragments. The two simplification runs have actually created several extra long lived fragments in these regions.

In summary, most regions show a similar pattern between the three runs and while some regions show the two simplification methods reducing significantly the number of different fragments, very few of them were either common or remained in the population for more than a few generations. Simplification has removed a couple of long lived fragments, particularly in regions B and E, but has created others in regions G and H.

Figure 6.5 shows the same three runs but using a 15% crossover rate. There are many more different fragments in these runs than with the 85% crossover runs. The high mutation rate in these runs has clearly produced a lot more diversity, but in region A the simplification methods have again removed many of them as being redundant. Again some long lived fragments have been lost due to simplification, but it has also created others, there are examples in regions D and E.

Figures 6.6 and 6.7 show the images for the wine dataset for the three methods. These show similar patterns of behaviour to the coins dataset. There are some long lived fragments in the canonical GP runs that are removed by simplification, see area E in figure 6.6(a) and area C in figure 6.7. The simplification runs also create several new long lived fragments, examples can be seen in area D in figure 6.6(b), area C in figure 6.6(c), areas B, D and H in figure 6.7(b) and areas B and H in figure 6.7(c). These new fragments contribute to solving the problem, so the classification performance is retained.

Figures 6.8 and 6.9 show the images for the faces dataset. Except for the numerical simplification run with the 85% crossover rate, these show the same pattern of behaviour as the other two datasets, with some fragments being destroyed or disrupted and other new ones being created. The 85% crossover rate numerical simplification run however has removed almost

all of the fragments in the population. Most of the fragments are removed at the first simplification, which occurs after the first generation. There are only a few left to continue the evolutionary process, but several new fragments are created mid run, most of which remain continuously in the population through to the end of the run and form a solution with at least reasonable performance.

6.5.2 Two Level Fragments

The three level deep fragments do not distinguish between addition and subtraction or between multiplication and division. They also do not differentiate between different features or different values of the constant terminals. To make these distinctions the behaviour of two-level deep fragments was checked to see whether the conclusions remain valid.

Figures 6.10 and 6.11 show the images for two level fragments for the coin dataset using the three methods. These results show a very similar pattern to those on the three-level fragments in the previous subsection with the simplification methods both removing fragments and creating new ones. There are a larger number of different fragments in the population than with the 3-level encoding scheme. This is not unexpected because the greater ability of the 2-level scheme to differentiate between similar fragments means that similar fragments may result in two or more closely spaced encoding in the 2-level scheme but be encoded the same in the 3-level encoding scheme.

Figures 6.12 and 6.13 show the images for two level fragments for the wine dataset. These show the same pattern as the previous runs. On this dataset there is a more obvious reduction in the number of the fragmented short lived fragments that remain in the population in the simplification runs, particularly with a crossover rate of 85% (figure 6.12(a) and (b)). This reduction has been particularly severe on the numerical simplification run. There is clearly a greater number of different fragments in the high muta-

tion rate 15% crossover runs than in the 85% runs. This was true across all three datasets. Across all the experiments conducted as part of this work the high mutation runs generally produced better accuracy and shorter programs than the high crossover rate runs. Further testing is required to establish whether there is any relationship between this and the higher diversity that is seen in these images for the 15% crossover runs.

Figures 6.14 and 6.15 show the images for two level fragments for the faces dataset. These show a similar pattern as the previous runs, although there appears to be less difference between the three methods. There are also a larger number of long lived fragments in all of the runs for this dataset than there was for the other two datasets. It is not clear why this might be the case. This dataset does not generally produce a really good classifier and it may be that the input data does not allow the GP process to converge on a single good solution.

While the two and three level deep fragments behave differently from the numerical constant terminals investigated by Wong and Zhang in [79], the main conclusions remain the same: the two program simplification algorithms destroy existing fragments but generate new fragments to maintain the diversity, at least for those fragments that remain in the population for many generations, and the contributions of the new fragments sufficiently compensates for the negative aspects of the disruption of existing fragments.

One observation from these new results is that many existing fragments are preserved during evolution. Wong and Zhang's analysis on numeric/constant terminals did not clearly show this. This is perhaps because the format of the numeric terminals is a single floating point number and is too simple to show this pattern.

6.6 Chapter Summary

The goal of this chapter was to show the distribution of fragments within a population and to examine if the nature of this distribution is changed by simplifying the programs during run. The results and analyses have revealed that although the two online simplification methods destroyed some existing fragments, they also generated additional new fragments during evolution, which sufficiently compensated for the negative effect from the disruption of fragments. These findings further confirmed the early hypothesis and results made by Wong and Zhang [79], where the analysis was based only on the simplest form of building blocks (fragments), being numerical constants, on two simple regression tasks. These findings are more general than [79].

The results presented in this chapter are based on individual GP runs. They were chosen as being representative. The next chapter uses statistical techniques to investigate the fragment distribution over multiple runs to check whether this chapter's conclusions remain true over a large number of runs, and whether any differences are statistically significant.

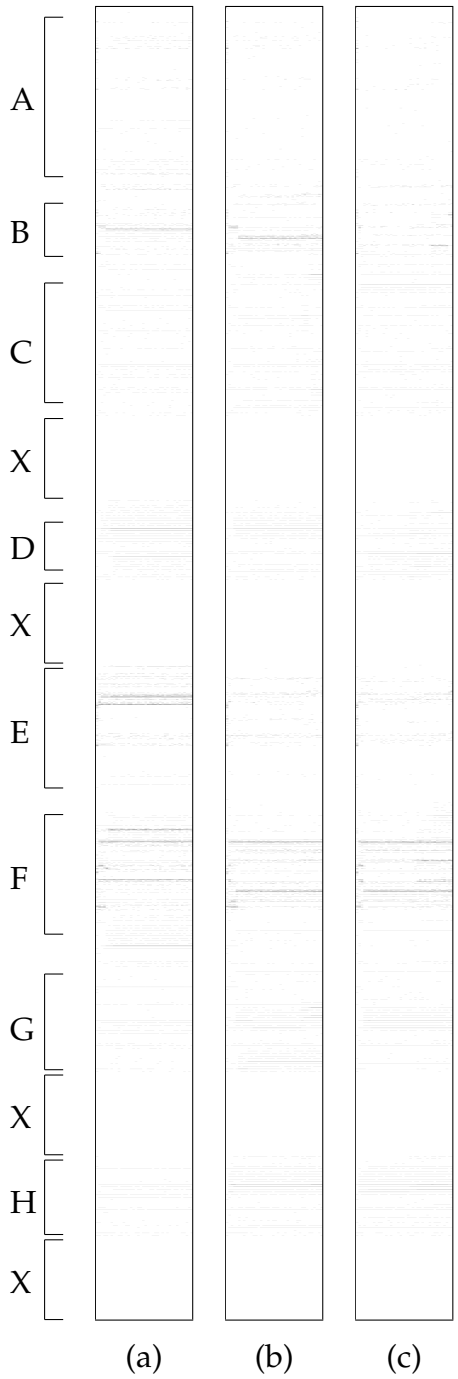


Figure 6.4: Three-level deep fragments for the coins dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

Figure 6.5: Three-level deep fragments for the coins dataset with 15% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

Figure 6.6: Three level deep fragments on the wine dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

Figure 6.7: Three level deep fragments on the wine dataset with 15% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

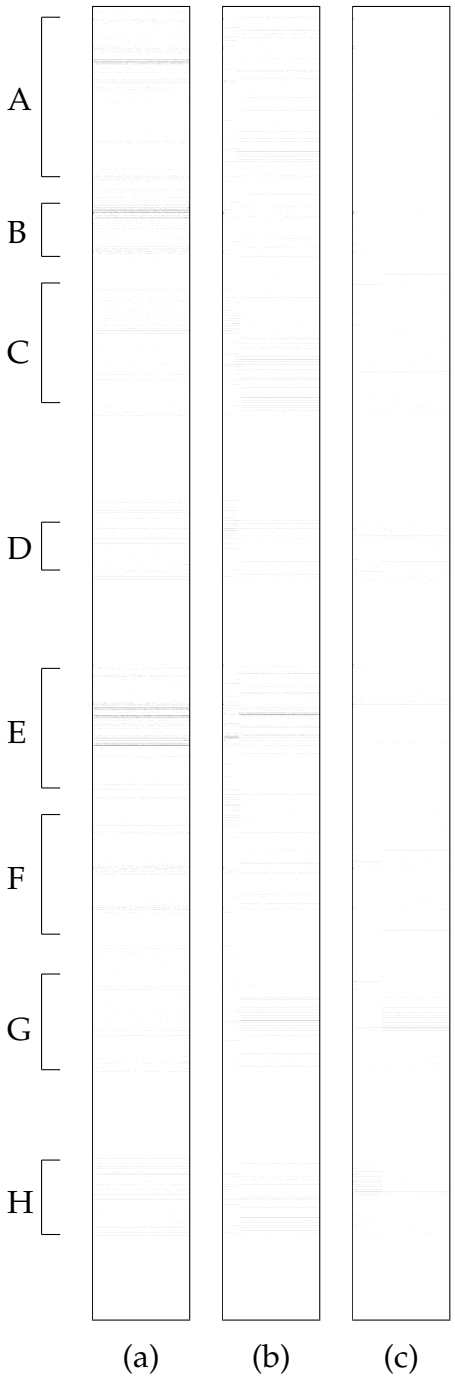


Figure 6.8: Three level deep fragments on the faces dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

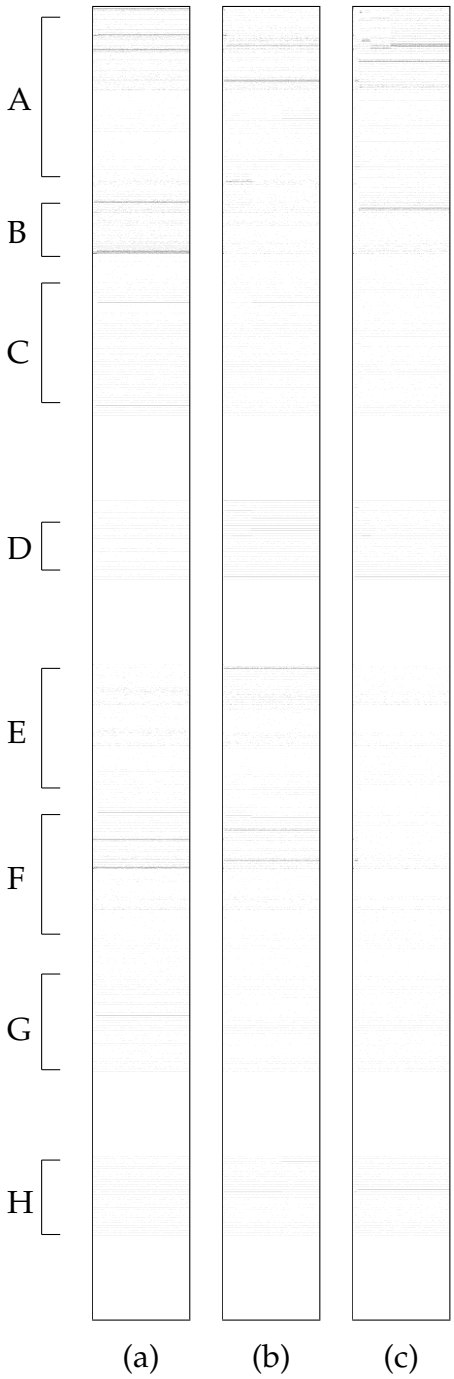


Figure 6.9: Three-level deep fragments for the faces dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

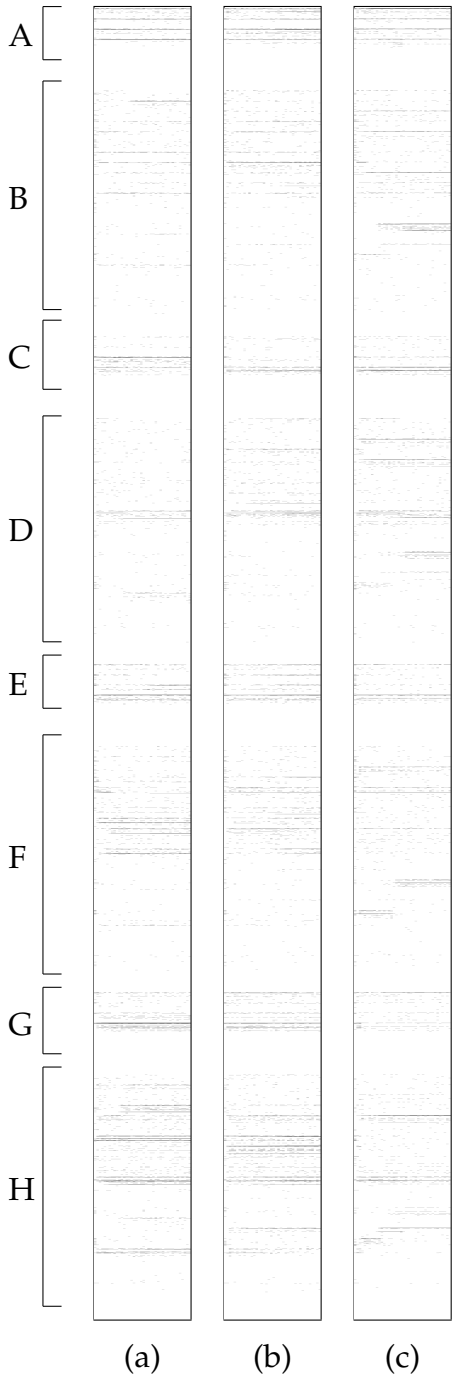


Figure 6.10: Two-level deep fragments for the coins dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

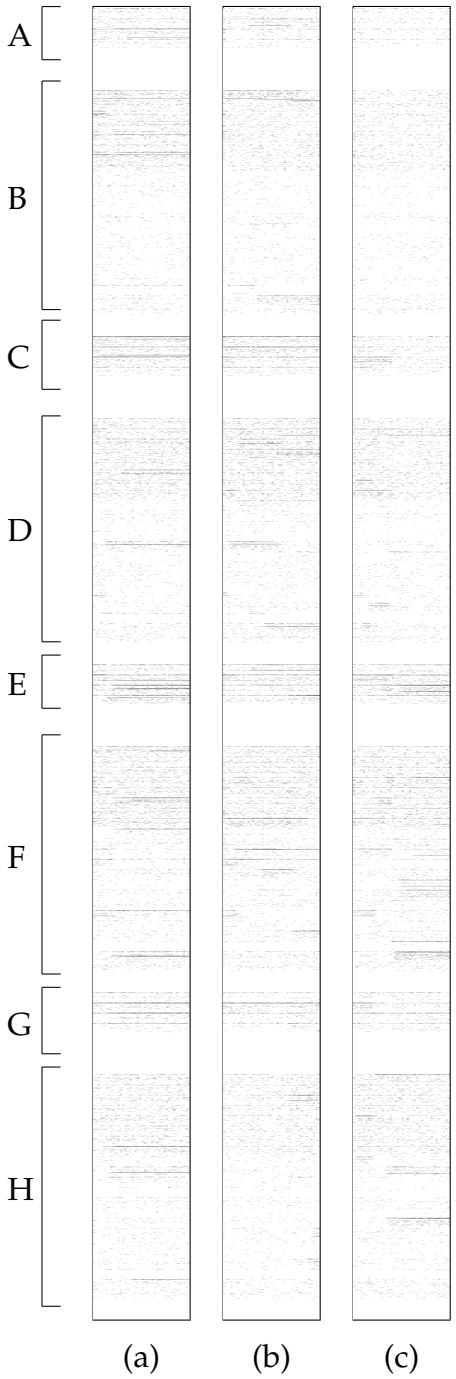


Figure 6.11: Two-level deep fragments for the coins dataset with 15% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

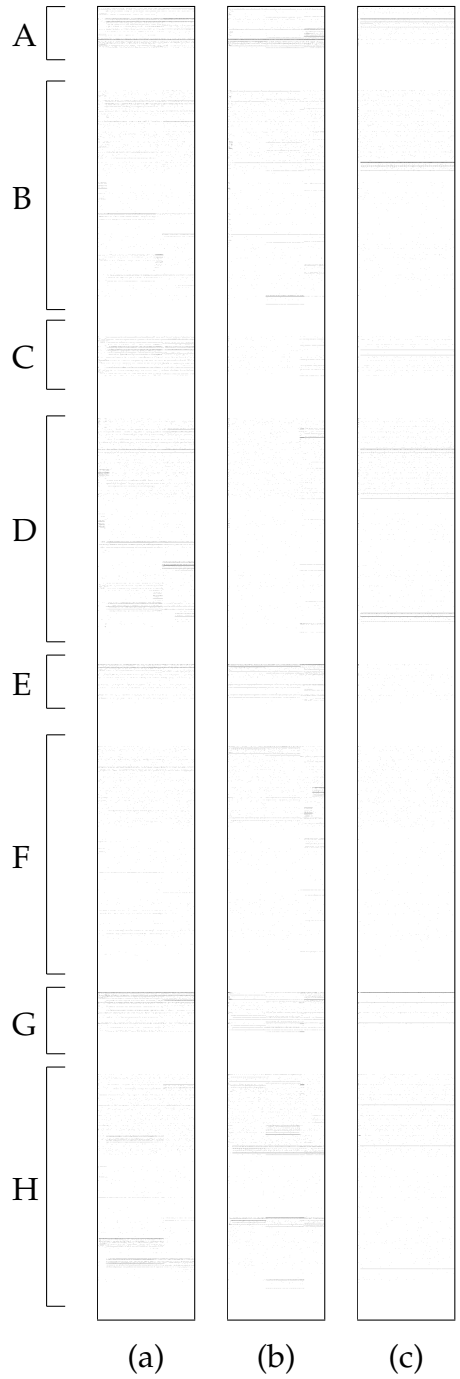


Figure 6.12: Two-level deep fragments on the wine dataset with 85% crossover (a) no simplification, (b) algebraic simplification and (c) numeric simplification.

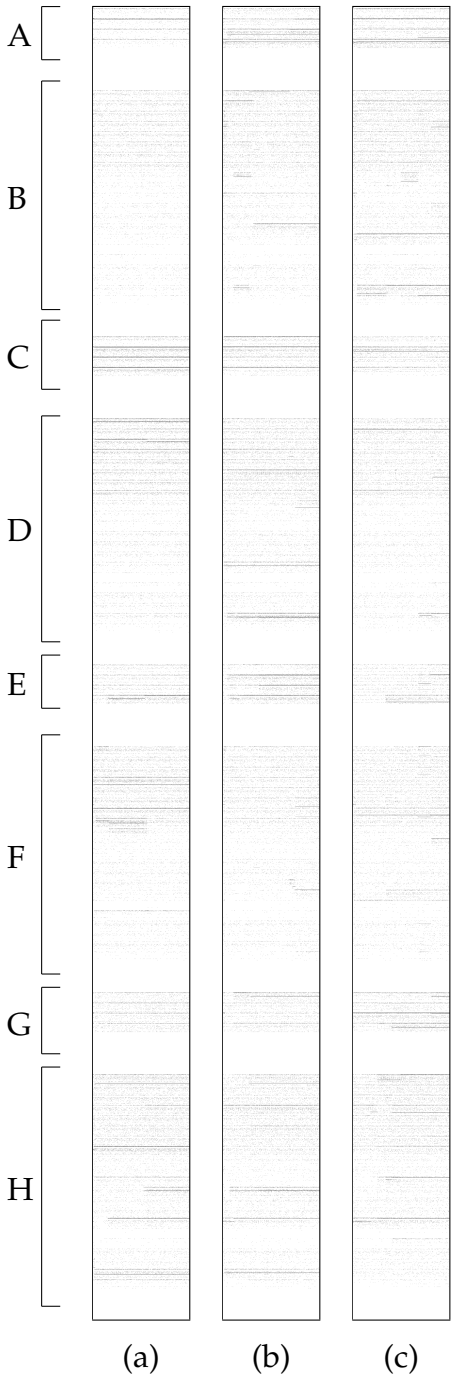


Figure 6.13: Two-level deep fragments on the wine dataset with 15% crossover (a) no simplification, (b) algebraic simplification and (c) numeric simplification.

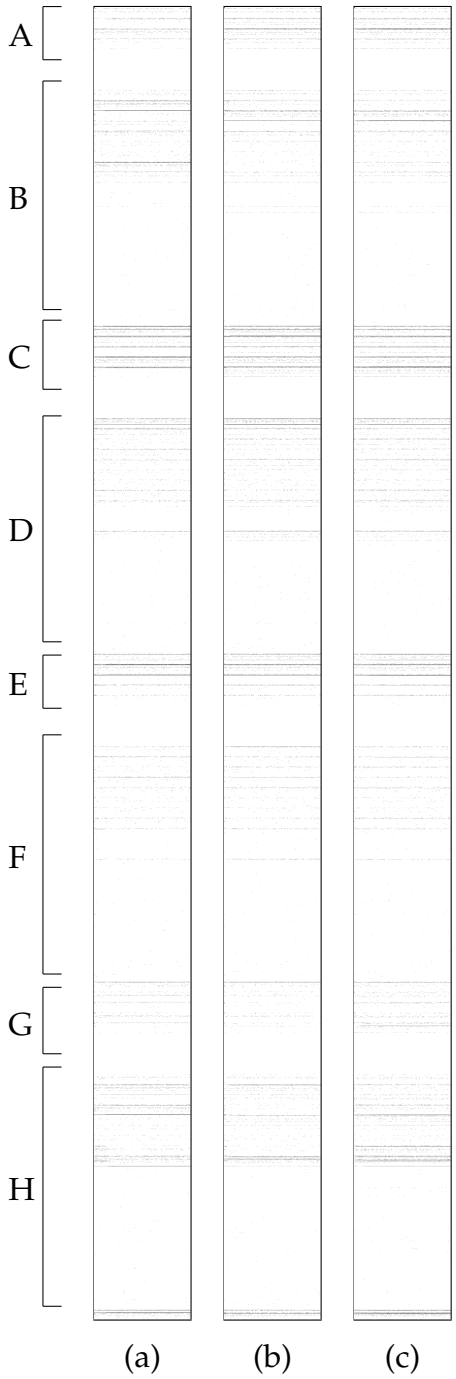


Figure 6.14: Two-level deep fragments for the faces dataset with 85% crossover. (a) no simplification, (b) algebraic simplification and (c) numerical simplification.

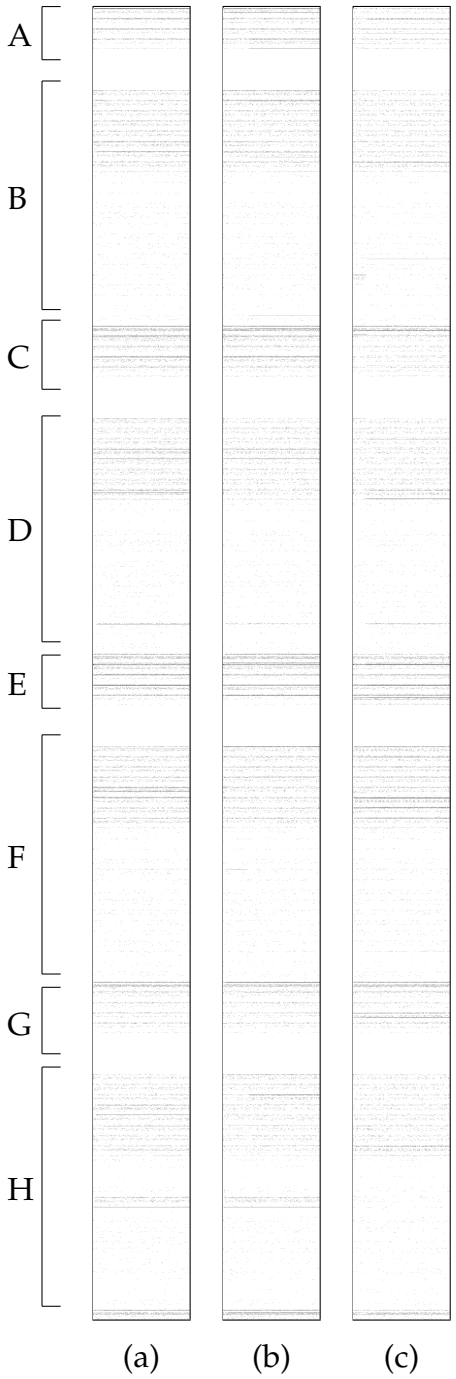


Figure 6.15: Two-level deep fragments on the faces dataset with 15% crossover (a) no simplification, (b) algebraic simplification and (c) numeric simplification.

Chapter 7

Fragment Analysis using Statistics

7.1 Introduction

The previous chapter used images to visualise the distribution of two and three level deep fragments. These images showed that a few fragments remained in the population for a substantial part of the run, while a much larger number of fragments were created, remained for a few generations and were then removed from the population. While the two online simplification methods destroyed some existing long lived fragments, they generated additional long lived fragments during evolution, which sufficiently compensated for the negative effect from the disruption of existing fragments. The requirement for the images to be printable on an A4 page restricted their size in pixels, and meant that the encoding scheme used was a very coarse representation of the three level deep fragments. Each image could only illustrate a single run and the images presented were chosen from a set of runs as being typical. While this allowed qualitative differences to be observed, no clear claim could be made about the significance of the differences observed.

7.2 Chapter Goals

This chapter aims to extend the analysis of the previous chapter by developing a more descriptive encoding scheme for the fragments and using statistical techniques across multiple runs to examine whether the behaviours observed using images are present across a much larger set of runs with the more descriptive encoding.

The numerical simplification method is used, and compared with canonical GP with no simplification. The specific research questions are similar to the previous chapter:

1. How are the fragments distributed as the evolution proceeds through the generations?
2. How does numerical simplification change this distribution?
3. Does the simplification process affect the overall diversity of fragments within the population?

7.3 Fragment Encoding Scheme using 8 Bits per Node

In this chapter, the behaviour of three level deep fragments is examined. These are large enough to be useful while keeping the encoding length within manageable bounds. In the three level deep encoding scheme used in the previous chapter there were many different fragments that produced the same encoding. This was necessary because of the restrictions imposed by using images as the presentation medium. This chapter presents a new encoding scheme with a much more precise description of the three level deep fragments. The new encoding is too long to present the results as images because even on a high definition printer the image would be much longer than any practical page size. Instead the encodings are

analysed using statistical measures. The previous chapter also examined two level deep fragments, but this was to avoid the limitations of the very coarse encoding scheme required in order to display the results as printed images. The encoding scheme developed in this chapter does not suffer from this limitation so it is not necessary to examine the simpler two level deep fragments in this chapter.

As with the encoding schemes presented in the previous chapter, the nodes are encoded one level at a time, starting from the root, and from left to right within each level. Figure 7.1 shows a three level deep subtree. In this example the order the nodes would be encoded is $\times \ - \ + \ - \ F2 \ 0.3 \ F3$.

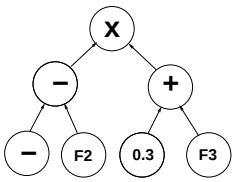


Figure 7.1: An example tree to illustrate the encoding order.

A three level deep tree with all operators having no more than two operands has a maximum size of seven nodes. If eight bits are used to describe each node then the fragment can be encoded in 56 bits. A 64 bit integer is then used to sufficiently hold the encoding.

- The operators are encoded as 111 followed by five bits identifying the operator. For these experiments this results in the following:

  1. 11100000 — *addition* operator.
  2. 11100001 — *subtraction* operator.
  3. 11100010 — *multiplication* operator.
  4. 11100011 — *division* operator.

- A feature is encoded as 10 followed by six bits identifying the feature, this allows up to 64 features, and more could be handled by using part of the encoding space allocated to operators.

- An ephemeral constant is encoded as 0 followed by seven bits which are a signed integer that maps [-1.0,1.0] on to the range [-64,63], that is:

  $$\begin{array}{lcl} C < -1.0 & \Rightarrow & -64 \\ -1.0 \leq C \leq 1.0 & \Rightarrow & C * 64 \text{ truncated to an integer} \\ C \geq 1.0 & \Rightarrow & 63 \end{array}$$

  So a value of 0.32 is encoded as 20, being $0.32 \times 64 = 20.48$ truncated to an integer.

All of the operators and the feature terminals are completely described. It is only the ephemeral constants that lose some precision. There are large gaps in the encoding space that do not correspond to a valid fragment, but that does not cause any problem for our statistical analysis. In the example of Fig. 7.1, the order of encoding is [$\times$] [$-$] [$+$] [$-$] [$F2$] [0.3] [$F3$], which is encoded as [11100010] [11100001] [11100000] [11100001] [10000010] [00010011] [10000011], the resulting encoding is then: $11100010111000011110000011100001100000100001001110000011$ or in hexadecimal $E2E1E0E1821383$.

7.4 Experimental Setup

This chapter uses the coins, wine and faces datasets as in the previous chapter. The function set for all the datasets consists of the four standard arithmetic functions (addition, subtraction, multiplication and protected division). The fitness function uses the error rate on the training set. Experiments are all conducted with the same set of parameters. The population size is 200. Initial programs are five levels deep. Tournament selection

is used with a tournament size of four. The coin dataset was run for 40 generations, the wine dataset for 200 generations, and 100 generations for the faces dataset. These parameter values were determined using heuristic guidelines and preliminary trials to obtain good results for these datasets without being optimised for any one dataset or either canonical GP or the numerical simplification method.

Two sets of parameters were used for the genetic operators: one 5% reproduction, 85% crossover, 10% mutation; and one 5% reproduction, 15% crossover, 80% mutation. Ten-fold cross validation was used and where simplification is used it is performed after the first generation, and every fourth generation thereafter. The runs are done in pairs, one for each of *no simplification* and *numerical simplification*. Because comparisons are being made between the two runs in each pair, both runs in the pair use the same starting population, the same folds, and the same starting seed. Each pair was run 50 times.

7.5 Results and Discussion

To investigate the distribution of fragments as the run proceeds, three level deep fragments are used, with the eight bits per node encoding described above in section 7.3. This section examines:

- the lifespan of fragments;
- the rate at which new fragments are added to the population;
- the rate at which fragments are removed from the population;
- the number of distinct fragments;
- the distribution of fragment counts(frequencies).

7.5.1 Fragment Lifetimes

If, at a given generation, a fragment (encoding) is present in the population, and at the previous generation it was not, then this is a *creation*. If, at a given generation, a fragment (encoding) is not present in the population, and at the previous generation it was, then this is a *destruction*.

Each time a destruction event occurs the number of generations that the fragment was in the population is recorded. This is a *lifespan*. At the end of the run the lifespan for each of the fragments still in the population is also recorded. The number of lifespans is then plotted against the lifespan in generations. The two different GP methods have different average program sizes and therefore different numbers of fragments in their populations. Therefore the plots express the lifespan counts as a percentage of the total number of lifespans in the population for that run and method.

Figure 7.2 shows the lifespan distributions for the three datasets over 50 runs. The range of lifespan counts is very large and so these graphs use a logarithmic scale for the percentages. 1 was added to each count to avoid any problem with $log(0)$. A small difference can be seen between the three methods for short lifespans. However all the lifespan percentages for lifespans of greater than 10 generations are very small. This confirms the observations from examining the images of chapter 6.

Figure 7.3 shows an expansion of just the lowest frequencies for the 85% crossover rate. In the coins dataset the small number of fragments with long lifespans can be clearly seen; they are harder to observe for the other datasets but they are present. The 15% crossover rate runs show the same pattern.

The statistical significance of the differences between the methods was tested using the Wilcoxon Signed-Rank non-parametric test [75, 34]. Figure 7.4 shows the Z scores from these tests. The red lines show the significance of the differences between algebraic simplification and no-simplification, the green lines show the significance of the differences between numeric simplification and no-simplification and the black lines show the signifi-

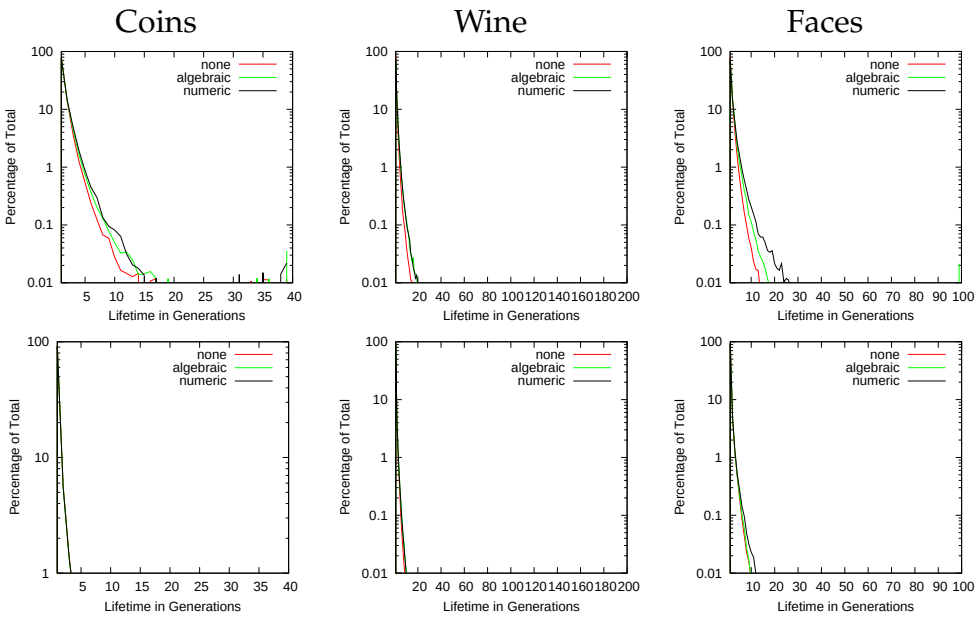


Figure 7.2: Distribution of fragment lifespans for the coin dataset (left), wine dataset (middle) and the faces dataset (right). The top row is for a crossover rate of 85% and the bottom row a rate of 15%.

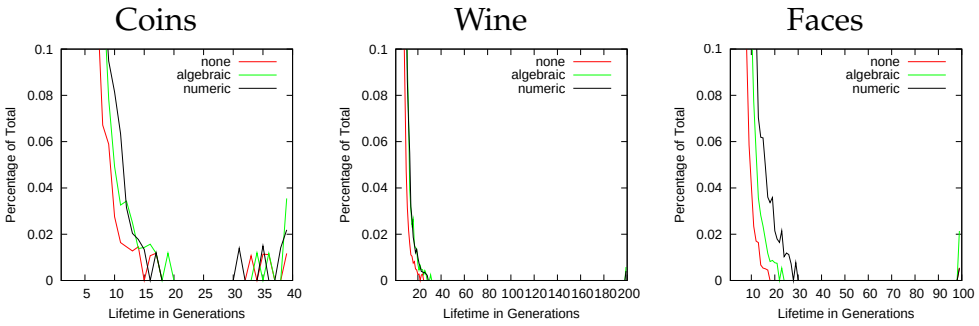


Figure 7.3: Distribution of fragment lifespans with 85% crossover for the coin dataset (left), wine dataset (middle) and the faces dataset (right). These show only the lowest frequencies to show more clearly the small number of fragments with a long lifespan.

cance of the the differences between numeric simplification and algebraic simplification. The horizontal line is for 1.96 which is the 95% confidence level. It can be clearly seen that there is no significance in the small differences seen in Fig. 7.2.

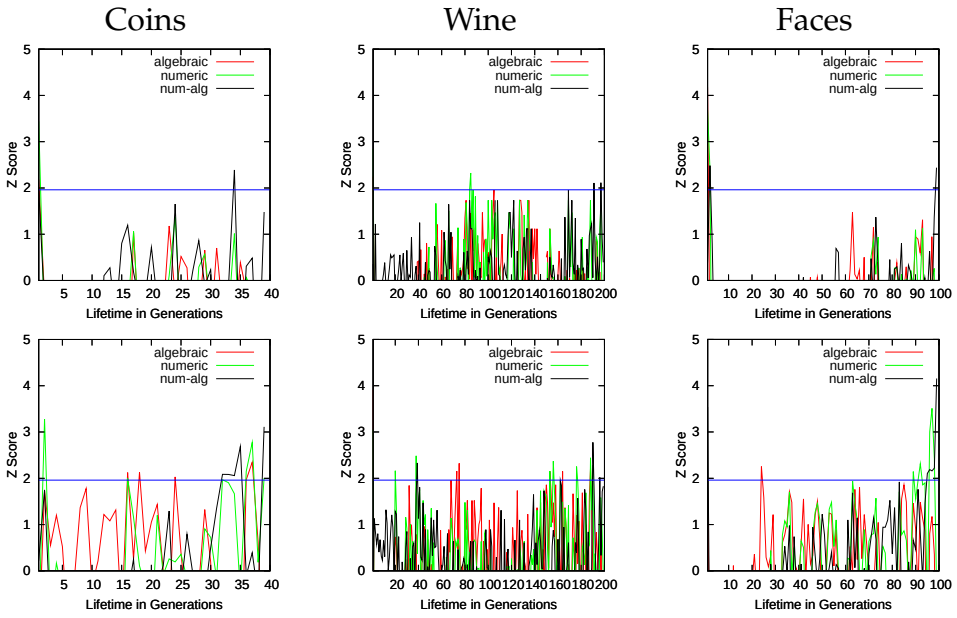


Figure 7.4: Z scores from significance tests on the differences in fragment lifetimes between no-simplification, algebraic simplification and numerical simplification for the coin dataset (left), wine dataset (middle) and the faces dataset (right). The red lines show the significance of the differences between algebraic simplification and no-simplification, the green lines show the significance of the differences between numeric simplification and no-simplification and the black lines show the significance of the the differences between numeric simplification and algebraic simplification.The top row is for the 85% crossover rate and the bottom row is for the 15% rate.

7.5.2 Creation and Destruction Rates

The rate at which new fragments are added to the population (creation) and the rate at which fragments are removed from the population (destruction) are now examined. Figure 7.5 shows the creation rates and figure 7.6 the destruction rates. Note that the periodic oscillations in the simplification lines are due to simplification being done every four generations. It can be seen that after the first one or two simplifications the numerical simplification method has consistently lower rates, for both creation and destruction, than the no-simplification case. After the first few generations the creation and destruction rates show remarkably little long-term variation.

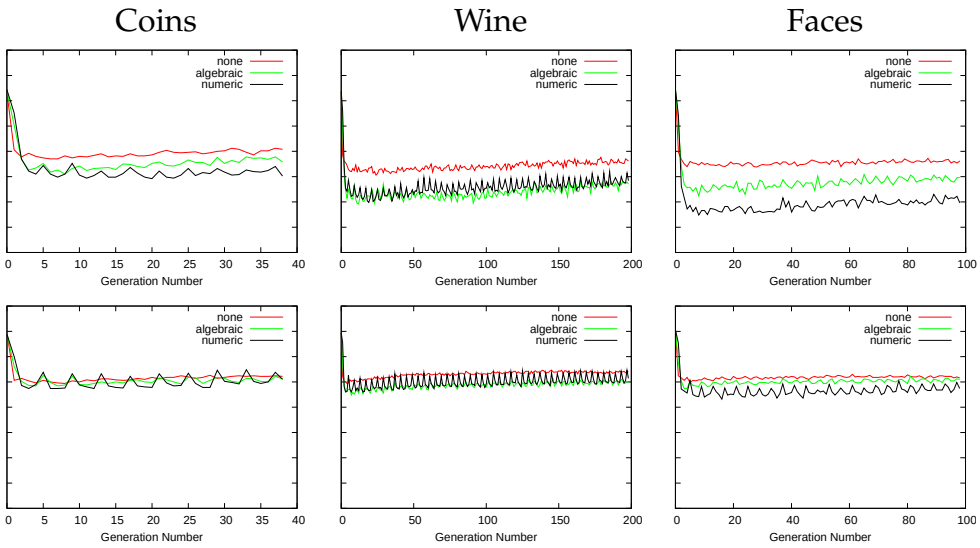


Figure 7.5: Fragment creation rates for the coin dataset (left), wine dataset (middle) and the faces dataset (right). The top row is for a crossover rate of 85% and the bottom a rate of 15%.

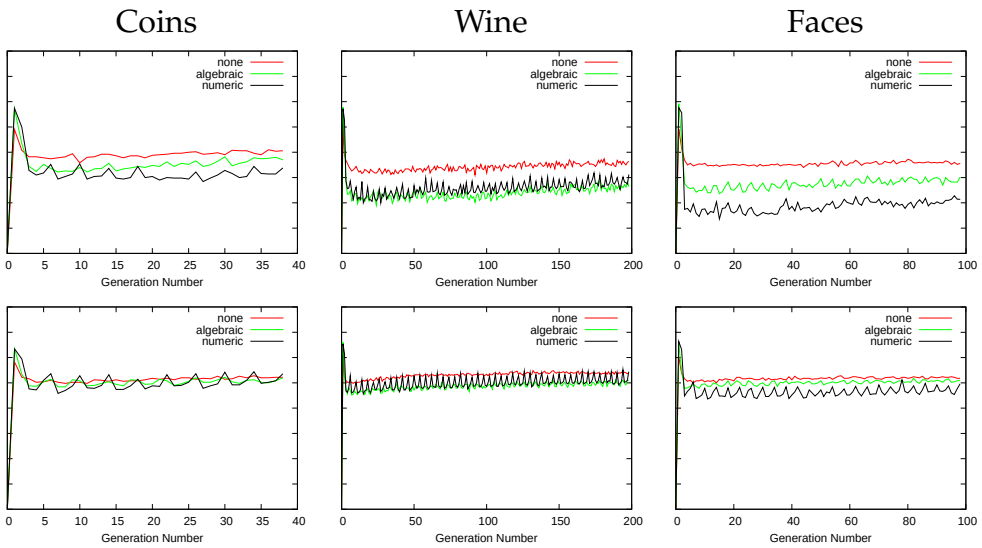


Figure 7.6: Fragment destruction rates for the coin dataset (left), wine dataset (middle) and the faces dataset (right). The top row is for a crossover rate of 85% and the bottom a rate of 15%.

7.5.3 Number of Distinct Fragments

Figure 7.7 shows the number of distinct fragments, and the total number of fragments in the population. The lines for the total number of fragments have been divided by 5 to allow them to be more easily shown on the same graph. It can be seen that in all cases the total number of fragments tend to rise through the run as the number of generations increases. There are many small variations, but most of this growth (and occasional fall) is in a number of discrete steps. It is not easily seen on these graphs but these steps are two or three generations long. Note that there is no corresponding step in the number of distinct fragments. One possible explanation for this effect is that when a genetic operation creates a program that is both larger than average and of high fitness, this program will often be the tournament winner and the fragments in this program will propagate through the population causing both an increase in the average program size and

an increase in total number of fragments. The creation of such programs only occurs every few generations resulting in this stepped behaviour.

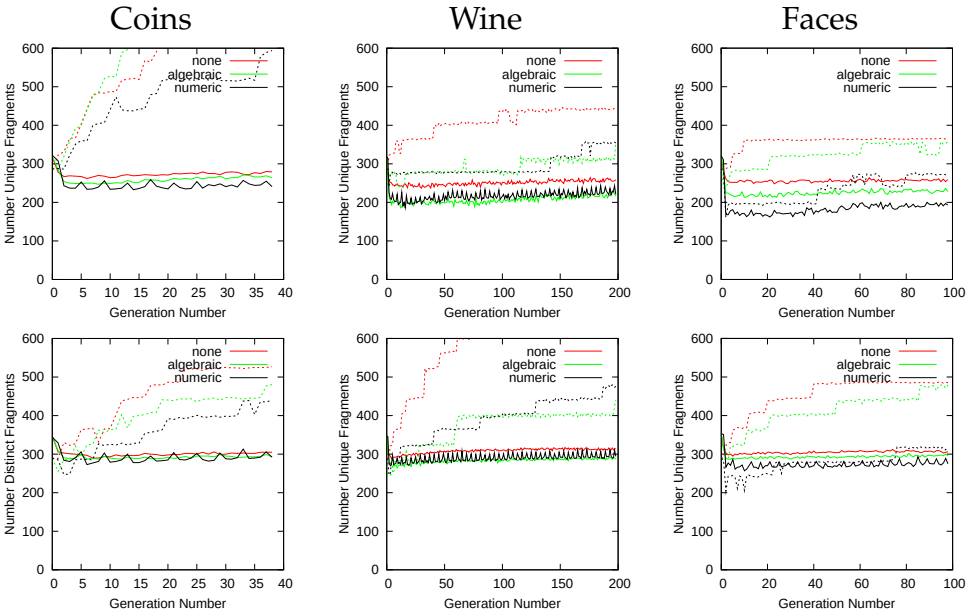


Figure 7.7: The solid lines show the number of distinct fragments in the population. The dashed lines show the total number of fragments divided by 5. The left column is the coin dataset, the middle column is the wine dataset and the right hand column is the faces dataset. The top row is the for a crossover rate of 85% and the bottom a rate of 15%. Note the periodic nature of the simplification lines, this is due to simplification only being run every fourth generation.

The first one, or sometimes two, rounds of simplification reduces the number of distinct fragments. It starts to rise again and then the next round of simplification reduces it. The long term trend remains nearly constant throughout the rest of the run. As with the creation and destruction rates, the number of distinct fragments is clearly lower for numerical simplification than for no simplification.

These results are not what might have been expected. The number

of crossover and mutation operations per generation is constant at one per program in the population. For a given number of distinct fragments already present in the population, the chance of a fragment being created that is not already in the population is likely to also be constant. Therefore if the population size is large enough the creation rate will probably also be approximately constant. What is not obvious is why the destruction rate and the number of distinct fragments should remain constant or nearly so. This will be examined further in section 7.5.5.

The consistency of the patterns shown in figures 7.5 and 7.7 would suggest that they are statistically significant. The significance of the differences between numerical simplification and no simplification was tested using the Wilcoxon Signed-Rank non-parametric test. Because of the periodic nature of the numbers, the significance testing was done using averages over four successive generations, this being the number of generations per simplification. The significance was tested at the beginning and end of the run, and at three points in between. Table 7.1 shows the Z scores. The five columns being for the four generations starting at, $0, 12, 20, 28, 36$ for the coins dataset, $0, 48, 96, 144, 196$ for the wine dataset, and $0, 24, 48, 72, 96$ for the faces dataset. Z scores of 1.96 or better indicate 95% confidence and are in bold, 2.576 indicates 99% confidence.

The Z scores show that for all three datasets, the numerical simplification method has significantly lower creation and destruction rates than the *no simplification* case. The same pattern is seen with the number of distinct fragments. Most of the scores show a significance level of higher than 99%.

The number of distinct fragments is significantly smaller for the numerical simplification method than for canonical GP. Chapter 4 showed that there was no loss of classification accuracy with simplification methods. Table 7.2 shows the means and standard deviations of the test accuracy for this chapter's experiments. The differences are very small and much smaller than the standard deviations. Tests showed that there is no

Table 7.1: Significance scores for the differences between numerical simplification and no simplification in terms of creation rate, destruction rate, and the number of distinct fragments.

	Start	1st Qtr	Mid	3rd Qtr	Final
Coins Dataset					
Creation Rate	**5.55**	**4.08**	**4.45**	**4.35**	**4.08**
Destruction Rate	**5.18**	**4.17**	**4.29**	**4.01**	**3.91**
Distinct Fragments	**5.28**	**3.83**	**4.21**	**4.37**	**3.89**
Wine Dataset					
Creation Rate	**3.33**	**3.38**	**3.04**	**2.98**	**3.16**
Destruction Rate	**3.29**	**3.43**	**2.88**	**2.75**	**3.00**
Distinct Fragments	**3.19**	**2.95**	**2.50**	**2.67**	**2.53**
Faces Dataset					
Creation Rate	**4.94**	**5.81**	**5.53**	**5.84**	**5.71**
Destruction Rate	**5.87**	**5.92**	**5.98**	**5.84**	**5.79**
Distinct Fragments	**5.43**	**5.82**	**5.40**	**5.61**	**5.44**

Table 7.2: Classification accuracy (as a percentage) on the test set of the two methods.

	No Simplification		Numerical Simplification	
	Mean	StdDev	Mean	StdDev
Coins Dataset	93.9	3.0	93.0	3.1
Wine Dataset	97.9	4.2	97.9	5.1
Faces Dataset	66.1	6.3	65.8	6.2

statistical significance to these differences.

7.5.4 Fragments Belonging to the Final Solution

The number of unique fragments is one possible measure of diversity in the population. This having a significantly lower value for the numerical simplification method poses an important question:
If the diversity is lower, how is it that the effectiveness (accuracy) is not adversely affected?

One possible answer would be that the fragments being lost are only those being deleted by the simplification process as being redundant and are not contributing to the final solution. If this is true then those fragments used by the final solution should not be being affected by simplification. Fragments that contribute usefully to high fitness individuals should not be being destroyed because of the 5% reproduction rate, so the creation and destruction rates are of little use in verifying this. As a first attempt those fragments that are part of the final solution were examined, and the generation in which they first enter the population recorded. For each run the median was calculated for the generation at which those fragments entered the population for the final time.

The results are shown in table 7.3. It shows the minimum, first quartile, median, third quartile and maximum value for these median values.

Table 7.3: Generation at which fragments that were part of the final fittest program entered the population..

	Earliest	1st Qtr	Median	3rd Qtr	Latest
Coins Dataset					
No Simplification	0	2	9	20	39
Algebraic simplification	0	2	9	21	39
Numeric simplification	0	3	9	19	39
Wine Dataset					
No Simplification	0	3	29	108	197
Algebraic simplification	0	2	18	75	198
Numeric simplification	0	5	35	86	199
Faces Dataset					
No Simplification	0	0	1	11	98
Algebraic simplification	0	1	2	18	99
Numeric simplification	0	2	21	52	99

In all cases the earliest generation was the initial population and the latest was very near the end of the run, the number of generations being 40 for the coins dataset, 200 for the wine dataset and 100 for the faces dataset. The first quartile generation was also very early. The median numbers however show much more variation, both between datasets, and between no simplification and the two simplification methods. The wine dataset in particular has at least half of the fragments in the final solution not entering the population until the $20th$ generation. There are some differences between methods, but are they significant? The significance of the differences between the median generations was tested using the Wilcoxon Signed-Rank non-parametric test.

Table 7.4 shows the Z scores for these differences. As would be expected from an examination of the figures in Table 7.3 the only differences that show significance are the median and third quartile for the faces dataset.

7.5.5 Distribution of Fragment Counts

In section 7.5.2 it was noted that the number of distinct fragments remain constant, or very nearly so, as the run proceeds through the generations. Intuitively it would be expected as the run proceeds and the fragments in the high fitness individuals start to dominate the population, that the number of distinct fragments in the population would drop. This is clearly not the case. It may be that it is only the distribution amongst the fragments that changes. To test this, the fragments were sorted in order of their frequency in the final population. The results are plotted in 3D in figure 7.8. These graphs are the average of 50 runs using a crossover rate of 85%. The quantity rank axis is the frequency order at the end of the run. 1 at the back is the fragment (encoding) with the highest number present in the final population. 2 is the second most numerous and so on. There are 100–200 different fragments present at any one time but the plots show just

Table 7.4: Z scores for the differences between methods of the median generation at which those fragments that are part of the final solution enter the population. Those scores that indicate a confidence level of 95% or better are in **bold**.

	85% Crossover
Coins Dataset	
Algebraic simplification	0.12
Numeric simplification	0.22
Difference between Algebraic and Numeric	0.39
Wine Dataset	
Algebraic simplification	1.66
Numeric simplification	0.80
Difference between Algebraic and Numeric	1.15
Faces Dataset	
Algebraic simplification	**2.05**
Numeric simplification	**4.64**
Difference between Algebraic and Numeric	**3.18**

the 20 most common in the final population as most of the fragments are present in only small numbers. In general only about twenty have more than ten copies in the population at any one generation. Each slice parallel to the generation axis is the same fragment, showing the number present in the population at each generation. Figure 7.9 shows the same data for a crossover rate of 15%.

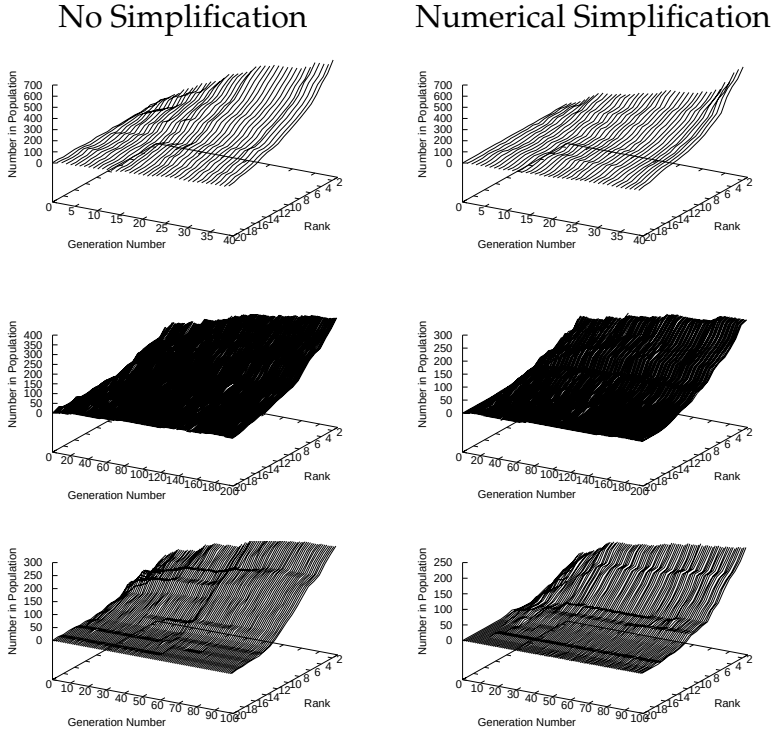


Figure 7.8: The distribution of fragments through the run for a crossover rate of 85%. The top row is the coins dataset, the middle row is wine dataset and the bottom row is the faces dataset.

The two graphs for the coins dataset show a clear tendency for the most numerous fragments to become an even larger percentage of the total population as the run proceeds. The other two datasets do not show this effect as clearly but there is still some indication of it occurring. What is clear in

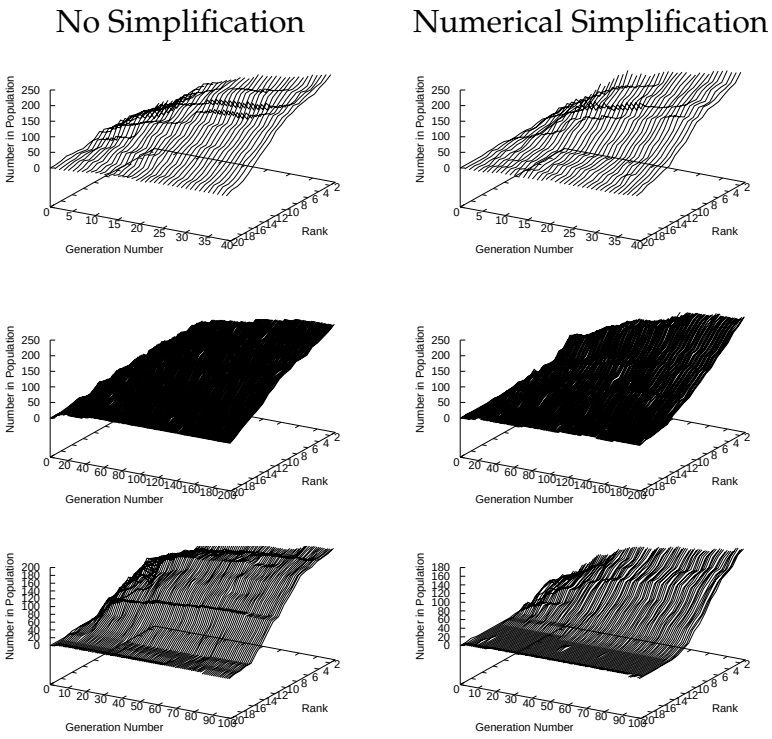


Figure 7.9: The distribution of fragments through the run for a crossover rate of 15%. The top row is the coins dataset, the middle row is wine dataset and the bottom row is the faces dataset.

all six graphs is that those fragments that are common at the end of the run were in most cases among the most common at all generations in the run. Some small ridges in the surface can be seen, particularly in the early part of the run. These are caused by fragments becoming slightly more or less common with respect to those near them, but very few fragments change their ranking by more than a few places.

During the runs the genetic operators have moved fragments between programs and they have slowly been organised into the most advantageous arrangement, but the majority of common fragments remain common. The fragments that are being removed from the population as in

Fig. 7.6 were never common. Even those fragments that are part of the final fittest program, and were created late in the run are not among the most common fragments.

7.6 Chapter Summary

This chapter has shown that there are many short lived fragments that are created in small numbers and within one or two generations have been deleted from the population. There are also a much smaller number of fragments that remain in the population for 60% or more of the generations in the run. This confirms the results from the image based encoding methods of chapter 6. *Numerical simplification* does not appear to change that pattern.

It has shown that after the first few generations the rate at which new fragments are added to, or fragments removed from, the population varies only by very small amounts with no clear trend to either increase or decrease as run proceeds through the generations. The number of distinct fragments present in the population at any generation in the run is also nearly constant. The number of distinct fragments in the population is reduced by *numerical simplification*. This was not clear from the images used in the previous chapter, although those images did show that simplification often reduced the number of the short-lived fragments. This reduction in the number of distinct fragments means that in some sense at least, the numerical simplification process reduces the overall diversity of fragments within the population. These experiments were able to show that at least for these three datasets this loss of diversity has not adversely affected the classification accuracy.

Chapter 8

Conclusions

This overall goal of this thesis was to develop a novel approach to online simplification of programs in tree based GP where the simplification decisions are based solely on the values nodes take during fitness evaluation. This goal was achieved by developing a method called *numerical simplification*. The performance and behaviour of numerical simplification was examined and compared to both canonical GP and to the *algebraic simplification* method [86].

The conclusions reported in this chapter have been split into two sections. The first section details those conclusions that answer the research questions posed in chapter 1. The second section reports other conclusions that arose out of the experiments performed and the analysis of their results but do not directly address the research questions. These are followed by a section on possible future work.

8.1 Main Conclusions

8.1.1 Effect of Simplification on Program Size and Resources Used

The numerical simplification method (chapter 4) was introduced and its performance on three classification datasets and two symbolic regression tasks was examined. The effect of simplification on program sizes, both in total number of nodes and tree depth, was examined. The experiments were organised in correlated sets that allowed easy comparison of differences in size, performance or resources used. Any differences observed were tested for statistical significance. Both numerical simplification and algebraic simplification methods produced smaller programs than canonical(standard) GP in all of the experiments. The reduction in program sizes was shown to be statistically significant in all cases except with high mutation on the first regression task. In that case the Z score fell just short of that required for a 95% confidence level. This was probably because that task was a fairly simple one and therefore the evolved solutions were already small. All of the remaining results were significantly better, at least to 95% with most at 99% or higher. The confidence intervals indicate that reductions in program size of between 30% and 40% can be expected in most cases. The percentage reduction in the depth of the program trees was smaller than the percentage reduction in the number of nodes. Targeted experiments confirmed this reduction in bushiness in the program trees with more terminals at higher levels in the tree.

The differences in program sizes between the numerical simplification method and algebraic simplification were smaller than the differences between either simplification method and canonical GP. In most cases numerical simplification produced smaller programs than algebraic simplification, but this difference was significant in only 40% of the cases. When it was significant, the reduction in program size, in nodes, was an extra

10% to 20%.

These reductions in program size were reflected in the amount of CPU time and memory required to evolve the solutions. The reductions in CPU time required were smaller than the reductions in program size because of the overhead of performing the simplification, and were not statistically significant in all cases. The results indicate a likely reduction of between 25% and 30% for the simplification methods over canonical GP and between 2% and 12% for numerical simplification over algebraic simplification.

8.1.2 Effect of Simplification on Effectiveness

These reductions in program size, CPU and memory requirements did not cause any statistically significant increase in classification error rates on the classification tasks or RMS error on the regression tasks. The second regression task showed a marginally significant increase in RMS error but this was because the task was a difficult one and the distribution of RMS error rates was skewed by a number of very poor results from unsuccessful runs. If only successful runs are considered then there is no statistically significant difference between the three methods.

8.1.3 Sensitivity of Simplification Threshold

The relationship between optimum values of the simplification threshold, if any, and the noise level in the input data was investigated (chapter 5). Experiments were performed on the two regression tasks, with two different noise levels used on each task. The results show that there is no clear optimum value for the simplification threshold. Numerical simplification reduced the program sizes and the run times, and in the case of the second, more difficult, regression problem reduced overfitting. These advantages were already present with very small simplification thresholds and did not improve in any significant way as the threshold was increased.

However there does appear to be an upper limit on the threshold. If the simplification threshold is too large then important detail is discarded that is required to form a good solution. This limit appears to be related to the noise level in as much as the upper limit was 0.01 for the lower noise level of 0.001 and the limit rose to 0.035 when the noise level was 0.01 for these problems.

8.1.4 Fragment Distributions

Whether simplification altered the distribution of fragments in the population was investigated (chapters 6 and 7). Fragments were encoded into bit strings using three different encoding schemes. Their distributions, both across the generations and within the search space, were examined in chapter 6 using images to display two of the encodings, and in chapter 7 using statistical techniques to display the third, more detailed, encoding.

The images used to display the results in chapter 6 showed that there where a few fragments that remained in the population for most of the run, and that these were generally among the most common fragments in the population. The rest of the fragments were less common and in most cases did not remain in the population for more than a few generations at a time. The simplification methods did not appear to change this pattern. While they did disrupt some of the long lived fragments from the canonical runs, they also created new fragments that then remained in the population through the remainder of the run.

Chapter 7 used statistical techniques and a more detailed encoding than was possible when using images to display the fragment distributions. This made possible quantitative results for the fragment distributions. These results echoed the results from chapter 6 in showing the small number of long-lived fragments with much larger numbers of fragments that exist in the population for only a few generations at a time. They also showed that after the first few generations, the rate at which new frag-

ments entered the population, and the rate at which fragments where lost from the population, showed very little long-term variation. In most cases these rates where lower for both simplification methods than they were for the canonical runs. In many cases these creation and destruction rates were lower for numerical simplification than they were for algebraic simplification.

The number of distinct fragments in the population also showed little long-term variation and in many cases were lower for the two simplification methods than for the canonical GP runs. This reduction in the number of distinct fragments was statistically significant for the numerical simplification method in all cases. In spite of this reduction in the diversity of fragments in the population, all differences in classification accuracy between the three methods were very small and had no statistical significance. This suggests that the fragments removed by the simplification methods were playing no useful part in the process of evolving a good solution.

The most important fragments in the population are those that make up the final fittest program. The generation at which these fragments entered the population was also examined. On the faces dataset a greater number of these entered late in the run than was the case with the canonical GP runs. This only affected about half these fragments, the rest still entered the population early in the run. On the other two datasets there was no significant difference in the pattern of entry generations between the three methods. This suggests that those fragments required by the final solution are not being adversely affected by the simplification process.

Therefore the major conclusion of this thesis is that online simplification methods produce substantial reductions in average program sizes with no adverse impact on the effectiveness of the GP search or the accuracy of the resulting fittest program. Numerical simplification performed at least as well as algebraic simplification and in some cases produced significantly smaller programs. Numerical simplification is also much easier

to implement then algebraic simplification and is not dependent in any way on the function set used.

8.2 Other Conclusions

It was noted that early trials indicated that high mutation rates often gave better performance and less bloat than high crossover rates, in spite of a high crossover rate being standard practice (chapter 3). For this reason, many of the experiments in this thesis used two different sets of evolutionary parameters, one using a high crossover rate and one using a high mutation rate. In general the high mutation rate runs produced slower growth in program sizes, i.e. less bloat, and the simplification methods often succeeded in stabilising the average program size after a number of generations.

There are few, if any, major changes to the distribution of fragments in the population as the run proceeds through the generations (Chapter 7). Those fragments that were common in the initial population remain common and few, if any, fragments that were either uncommon or not present at all in the initial population become very common by the end of the run. Even where a fragment is created part way through the run and becomes part of the final fittest program, it is preserved by the reproduction operator (elitism) but usually does not become very numerous.

After the first few generations, the rate new fragments are created, the rate fragments are lost from the population, and the number of distinct fragments in the population show very little long term variation, that is the rates remain within a narrow range of values (Chapter 7).

8.3 Discussion and Future Work

The effect of simplification on the shape of the tree as well as its size was examined (Chapter 4). It was found that the tree became more sparse (less

bushy) as a result of simplification. The experiments in this thesis all used the *full* method for creating the initial population of programs. This appeared to result in a notch (a drop in the number of programs of a particular size) in the distribution of program sizes. More experiments are needed with other methods for creating the initial population, both to check the cause of that anomaly and to verify if this pattern of reducing bushiness rather than depth is still true if the initial programs are more varied in structure.

Many of the experiments in this thesis were run with both high mutation and high crossover rates. In all cases, the high mutation rate runs produced less bloat, and in many cases better accuracy. This is only an indication, as only one type of mutation operator was used, and it may be biased to producing smaller changes to the programs than was happening on average with the crossover operator. More investigation is required into the effect of varying the crossover rate versus mutation rate when using simplification.

Langdon and Poli [32] showed that bloat is an inevitable consequence of fitness based selection with variable length representations. It is therefore not unique to tree based genetic programming. While simplification, as described in this thesis, applies only to tree structured representations, it may be possible to construct equivalent algorithms for other variable length representations. The programs in linear GP are variable in length and can be rearranged to form several parallel interconnected trees (usually one for each output register). Developing a version of the numerical simplification method for these trees is one possibility.

Feature selection is usually an explicit operation, removing features from the training data because they are redundant (highly correlated with one or more other features) or because they do not contribute to forming a good solution. The "lack of contribution" has been determined by other researchers using ideas like *information gain*, or *entropy* or directly from trial runs with various combinations of features. Some of the experiments

reported in this thesis suggest that numerical simplification may provide a degree of feature selection as a consequence of its actions. In particular, numerical simplification produced much smaller programs than algebraic simplification with the faces dataset. If the features in a particular subtree do not contribute to forming a good solution, then that subtree is likely to be making little or no contribution to its program and might therefore be removed by the numerical simplification process. Similarly, features that are highly correlated will generally be treated as equal by numerical simplification. Expressions involving differences or ratios will often be sufficiently close to being constant that the simplification process will replace them by a constant. This is certainly no substitute for more sophisticated feature selection methods but, with some datasets at least, may provide some help. More targeted experiments are needed to investigate this effect.

The *building block hypothesis* [21] was proposed by Holland in the 1970s to explain how genetic algorithms worked. The basic idea is that crossover combines small fragments of the genome that have good fitness into larger fragments with better fitness. The GP version [47] uses subtrees. While this hypothesis is widely quoted it has its critics [48] that criticise it for a lack of theoretical basis and because of experimental evidence that appears to contradict it.

The experiments in chapter 7 show that the distribution of three-level deep fragments in the population changes very little during a GP run. While these fragments are not subtrees, because they do not necessarily extend to the leaves of the tree, these results do still suggest that the crossover operator is not creating new (and fitter) subtrees in the way that the building block hypothesis would suggest. If these new, fitter, subtrees are being constructed then they certainly do not appear to dominate the population in later generations.

It is possible to create longer length encodings similar to that used in chapter 7, or to use a suitable hash function to describe larger fragments,

and in particular to describe true subtrees that are large enough to have good fitness in their own right. This would allow some more experiments to be run, with full subtrees rather than fixed depth fragments, to check whether subtrees are being built up from smaller pieces as per the building block hypothesis.

Rather than the building block approach, the experiments in chapter 7 suggest that the genetic operators are just re-arranging the existing fragments to optimise the program fitness. What would happen if the fragments from the final fittest individual were removed from the starting population and the run then repeated? Would a good solution still be found? If it is, were the removed fragments recreated, or did the evolutionary process find a different solution using the genetic material it was given? Some more experiments to investigate this may add to our knowledge of how GP actually creates solutions.

Bibliography

[1] ASHLOCK, W., AND ASHLOCK, D. Single parent genetic programming. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (Edinburgh, UK, 2-5 Sept. 2005), D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, Eds., vol. 2, IEEE Press, pp. 1172–1179.

[2] BANZHAF, W., NORDIN, P., KELLER, R. E., AND FRANCONE, F. D. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, San Francisco, CA, USA, Jan. 1998.

[3] BLICKLE, T., AND THIELE, L. Genetic programming and redundancy. In *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)* (Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994), J. Hopf, Ed., Max-Planck-Institut für Informatik (MPI-I-94-241), pp. 33–38.

[4] BREIMAN, L., FRIEDMAN, J., OLSHEN, R., AND STONE, C. *Classification and regression trees*. Cole Advanced Books & Software. Wadsworth & Brooks, Monterey, CA, 1984.

[5] CORDER, G. W., AND FOREMAN, D. I. *Nonparametric Statistics for Non-Statisticians: A Step-by-Step Approach*. Wiley, May 2009.

[6] CORTES, C., AND VAPNIK, V. Support-vector networks. *Machine Learning 20* (1995), 273–297.

[7] COVER, T., AND HART, P. Nearest neighbor pattern classification. *Information Theory, IEEE Transactions on 13*, 1 (1967), 21–27.

[8] DE JONG, E. D., AND POLLACK, J. B. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines 4*, 3 (Sept. 2003), 211–233.

[9] DEB, K. *Multi-Objective Optimization using Evolutionary Algorithms*. Wiley-Interscience Series in Systems and Optimization. John Wiley & Sons, Chichester, 2001.

[10] DEB, K., PRATAP, A., AGARWAL, S., AND MEYARIVAN, T. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Evolutionary Computation, IEEE Transactions on 6*, 2 (April 2002), 182–1917.

[11] DORIGO, M. *Optimization, Learning and Natural Algorithms*. PhD thesis, Politecnico di Milano, Italie, 1992.

[12] DORIGO, M., AND STÜTZLE, T. *Ant colony optimization:*. 2004.

[13] EIBEN, A. E., AND SMITH, J. E. *Introduction to Evolutionary Computing*. Springer, 2003.

[14] FANG, Y., AND LI, J. A review of tournament selection in genetic programming. In *Advances in Computation and Intelligence*, Z. Cai, C. Hu, Z. Kang, and Y. Liu, Eds., vol. 6382 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2011, pp. 181–192.

[15] FOGEL, L. *Intelligence through SImulated Evolution: Forty Years of Evolutionary Programming*. John Wiley, 1999.

[16] FONSECA, C., AND FLEMING, P. Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization. In *Proceedings of the fifth international conference on genetic algorithms* (1993), vol. 423, pp. 416–423.

[17] FORINA, M., LEARDI, R., ARMANINO, C., AND LANTERI, S. Parvus: an extendable package of programs for data exploration, classification and correlation. *Elsevier Scientific, Amsterdam, The Netherlands* (1988).

[18] GEISSER, S., AND JOHNSON, W. M. *Modes of Parametric Statistical Inference*. John Wiley & Sons, 2006.

[19] HETTMANSPERGER, T. P., AND MCKEAN, J. W. *Robust nonparametric statistical methods*. Kendall's Library of Statistics, 1998.

[20] HOLLAND, J., BOOKER, L., COLOMBETTI, M., DORIGO, M., GOLDBERG, D., FORREST, S., RIOLO, R., SMITH, R., LANZI, P., STOLZMANN, W., AND WILSON, S. What is a learning classifier system? In *Learning Classifier Systems*, P. Lanzi, W. Stolzmann, and S. Wilson, Eds., vol. 1813 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2000, pp. 3–32.

[21] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[22] HOOPER, D., AND FLANN, N. S. Improving the accuracy and robustness of genetic programming through expression simplification. In *Genetic Programming 1996: Proceedings of the First Annual Conference* (Stanford University, CA, USA, 28–31 July 1996), J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., MIT Press, p. 428.

[23] HYAFIL, L., AND RIVEST, R. L. Constructing optimal binary decision trees is np-complete. *Information Processing Letters 5*, 1 (1976), 15–17.

[24] KANG, M., SHIN, J., HOANG, T. H., MCKAY, R. I. B., ESSAM, D., MORI, N., AND NGUYEN, X. H. Code duplication and developmental evaluation in genetic programming. In *Proceedings of the 2006 Asia-Pacific Workshop on Intelligent and Evolutionary Systems* (Seoul, Korea, Nov. 2006), pp. 181–191.

[25] KASS, G. An exploratory technique for investigating large quantities of categorical data. *Journal of the Royal Statistical Society. Series C (Applied Statistics) 29*, 2 (1980), 119–127.

[26] KENNEDY, J., AND EBERHART, R. Particle swarm optimisation. In *Proceedings of IEEE International Conference on Neural Networks* (1995), IV, pp. 1942–1948.

[27] KENNEDY, J., AND EBERHART, R. *Swarm Intelligence*. Morgan Kaufmann, 2001.

[28] KOHAVI, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence* (1995), vol. 2, pp. 1137–1143.

[29] KOLATA, G. How can computers get common sense? *Science*, 217 (1982), 1237–1238.

[30] KOZA, J. R. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence IJCAI-89* (Detroit, MI, USA, 20-25 Aug. 1989), N. S. Sridharan, Ed., vol. 1, Morgan Kaufmann, pp. 768–774.

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

[32] LANGDON, W. B., AND POLI, R. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing* (23-27 June 1997), P. K. Chawdhry, R. Roy, and R. K. Pant, Eds., Springer-Verlag London, pp. 13–22.

[33] LANGDON, W. B., AND POLI, R. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[34] LAVANGE, L. M., AND KOCH, G. G. Rank Score Tests. *Circulation 114*, 23 (2006), 2528–2533.

[35] LOVEARD, T., AND CIESIELSKI, V. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation* (COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001), vol. 2, IEEE Press, pp. 1070–1077.

[36] LUKE, S., AND PANAIT, L. Fighting bloat with nonparametric parsimony pressure. In *Parallel Problem Solving from Nature - PPSN VII* (Granada, Spain, 7-11 Sept. 2002), J. J. Merelo-Guervos, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas, and H.-P. Schwefel, Eds., no. 2439 in Lecture Notes in Computer Science, LNCS, Springer-Verlag, pp. 411–421.

[37] LUKE, S., AND PANAIT, L. Lexicographic parsimony pressure. In *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference* (New York, 9-13 July 2002), W. B. Langdon, E. Cantú-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, Eds., Morgan Kaufmann Publishers, pp. 829–836.

[38] MARSHALL, D. The discrete cosine transform. http://www.cs.cf.ac.uk/Dave/Multimedia/node231.htm, 2001.

[39] MCCARTHY, J. Programs with common sense. In *Proceedings of the Teddington Conference on the Mechanization of Thought Processes* (1959), Her Majesty's Stationery Office, pp. 756–791.

[40] MCCARTHY, J., AND HAYES, P. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4 (1969), 463–502.

[41] MITCHELL, M. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[42] MITCHELL, T. M. *Machine Learning*. McGraw-Hill, 1997.

[43] MONTANA, D. J. Strongly typed genetic programming. BBN Technical Report #7866, Bolt Beranek and Newman, Inc., 10 Moulton Street, Cambridge, MA 02138, USA, 7 May 1993.

[44] MORI, N. A novel diversity measure of genetic programming. In *Randomness and Computation: Joint Workshop "New Horizons in Computing" and "Statistical Mechanical Approach to Probabilistic Information Processing"* (Sendai International Center, Sendai, Japan, 18-21 July 2005). Extended Abstract.

[45] NORDIN, P., AND BANZHAF, W. Complexity compression and evolution. In *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)* (Pittsburgh, PA, USA, 15-19 July 1995), L. Eshelman, Ed., Morgan Kaufmann, pp. 310–317.

[46] NORDIN, P., FRANCONE, F., AND BANZHAF, W. Explicitly defined introns and destructive crossover in genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (Tahoe City, California, USA, 9 July 1995), J. P. Rosca, Ed., pp. 6–22.

[47] O'REILLY, U. M. The trouble aspects of a building block hypothesis for genetic programming. Tech. rep., Santa Fe Institute, Feb 1994.

[48] O'REILLY, U. M., AND OPPACHER, F. The troubling aspects of a building block hypothesis for genetic programming. Working Paper 94-02-001, Santa Fe Institute, 1399 Hyde Park Road Santa Fe, New Mexico 87501-8943 USA, 1992.

[49] PANAIT, L., AND LUKE, S. Alternative bloat control methods. In *Genetic and Evolutionary Computation – GECCO-2004, Part II* (Seattle, WA, USA, 26-30 June 2004), K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, Eds., vol. 3103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 630–641.

[50] PARROTT, D., LI, X., AND CIESIELSKI, V. Multi-objective techniques in genetic programming for evolving classifiers. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation* (Edinburgh, UK, 2-5 Sept. 2005), D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, Eds., vol. 2, IEEE Press, pp. 1141–1148.

[51] POLI, R. A simple but theoretically-motivated method to control bloat in genetic programming. In *Genetic Programming, Proceedings of EuroGP'2003* (Essex, 14-16 Apr. 2003), C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of *LNCS*, Springer-Verlag, pp. 204–217.

[52] POLI, R., LANGDON, W. B., AND MCPHEE, N. F. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008. (With contributions by J. R. Koza).

[53] PRICE, K., STORN, R., AND LAMPINEN, J. *Differential evolution: a practical approach to global optimization*. Springer Verlag, 2005.

[54] RECHENBERG, I. *Evolutionsstrategie - Optimierung technischer System nach Prinzipien der biologischen Evolution*. PhD thesis, Technical University of Berlin, 1971.

[55] ROSS, B. J., GUALTIERI, A. G., FUETEN, F., AND BUDKEWITSCH, P. Hyperspectral image analysis using genetic programming. *Applied Soft Computing 5*, 2 (Jan. 2005), 147–156.

[56] SAMARIA, F., AND HARTER, A. Parameterisation of a stocastic model for human face identification. In *Proceedings of the Second IEEE Workshop on Applications of Computer Vision* (1994).

[57] SAMUEL, A. L. Programming computers to play games. *Advances in Computers 1* (1960), 165–192.

[58] SAMUEL, A. L. Symposium on pattern recognition. In *IFIP Congress* (1962), pp. 467–470.

[59] SAMUEL, A. L. Ai, where it has been and where is it going. *IJCAI* (1983), 1152–1157.

[60] SIEGELMANN, H. T., AND SONTAG, E. D. Turing computability with neural nets. *Applied Mathematics Letters 4*, 6 (1991), 77–80.

[61] SMART, W. R., AND ZHANG, M. Classification strategies for image classification in genetic programming. In *Proceeding of Image and Vision Computing NZ International Conference* (Palmerston North, New Zealand, Nov. 2003), D. Bailey, Ed., Massey University, pp. 402–407.

[62] SONG, A., AND CIESIELSKI, V. Texture analysis by genetic programming. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation* (Portland, Oregon, 20-23 June 2004), IEEE Press, pp. 2092–2099.

[63] SOULE, T. *Code Growth in Genetic Programming*. PhD thesis, University of Idaho, Moscow, Idaho, USA, 15 May 1998.

[64] SOULE, T., FOSTER, J. A., AND DICKINSON, J. Code growth in genetic programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference* (Stanford University, CA, USA, 28–31 July 1996), J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., MIT Press, pp. 215–223.

[65] SOULE, T., AND HECKENDORN, R. B. An analysis of the causes of code growth in genetic programming. *Genetic Programming and Evolvable Machines 3*, 3 (Sept. 2002), 283–309.

[66] STORN, R., AND PRICE, K. Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces. *Journal of Global Optimization 11* (1997), 341–359.

[67] STREETER, M. J. The root causes of code growth in genetic programming. In *Genetic Programming, Proceedings of EuroGP'2003* (Essex, 14-16 Apr. 2003), C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, Eds., vol. 2610 of *LNCS*, Springer-Verlag, pp. 443–454.

[68] SUTTON, R., AND BARTO, A. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. MIT Press, 1998.

[69] TURING, A. M. *Intelligent Machinery*. Report for National Physical Laboratory, 1948.

[70] TURING, A. M. Computing machinery and intelligence. *Mind* (1950).

[71] TUTSCHKU, K. Recurrent multilayer perceptrons for identification and control: The road to applications, 1995.

[72] WHIGHAM, P., AND DICK, G. Implicitly controlling bloat in genetic programming. *Evolutionary Computation, IEEE Transactions on 14*, 2 (April 2010), 173–190.

[73] WHIGHAM, P. A. Grammatical genetic learning and schemata: Restated. Technical Report CS13/95, Department of Computer Science, University College, University of New South Wales, Australia, 1995.

[74] WHIGHAM, P. A. Grammatically-based genetic programming. In *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications* (Tahoe City, California, USA, 9 July 1995), J. P. Rosca, Ed., pp. 33–41.

[75] WILCOXON, F. Individual comparisons by ranking methods. *Biometrics Bulletin 1*, 6 (1945), 80–83.

[76] WITTEN, I., AND FRANK, E. *Data Mining: Practical machine learning tools and techniques.*, 2nd ed. Morgan Kaufmann, San Francisco, 2005.

[77] WONG, P., AND ZHANG, M. Algebraic simplification of GP programs during evolution. In *GECCO 2006: Proceedings of the 8th annual conference on Genetic and evolutionary computation* (Seattle, Washington, USA, 8-12 July 2006), M. Keijzer, M. Cattolico, D. Arnold, V. Babovic, C. Blum, P. Bosman, M. V. Butz, C. Coello Coello, D. Dasgupta, S. G. Ficici, J. Foster, A. Hernandez-Aguirre, G. Hornby, H. Lipson, P. McMinn, J. Moore, G. Raidl, F. Rothlauf, C. Ryan, and D. Thierens, Eds., vol. 1, ACM Press, pp. 927–934.

[78] WONG, P., AND ZHANG, M. Effects of program simplification on simple building blocks in genetic programming. In *2007 IEEE Congress on Evolutionary Computation* (Singapore, 25-28 Sept. 2007), D. Srinivasan and L. Wang, Eds., IEEE Computational Intelligence Society, IEEE Press, pp. 1570–1577.

[79] WONG, P., AND ZHANG, M. Numerical-node building block analysis of genetic programming with simplification. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation* (London, 7-11 July 2007), D. Thierens, H.-G. Beyer, J. Bongard,

J. Branke, J. A. Clark, D. Cliff, C. B. Congdon, K. Deb, B. Doerr, T. Kovacs, S. Kumar, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, R. Poli, K. Sastry, K. O. Stanley, T. Stutzle, R. A. Watson, and I. Wegener, Eds., vol. 2, ACM Press, pp. 1761–1761.

[80] XIE, H. *An Analysis of Selection in Genetic Programming*. PhD thesis, Computer Science, Victoria University of Wellington, New Zealand, 2008.

[81] ZHANG, B.-T., AND MÜHLENBEIN, H. Balancing accuracy and parsimony in genetic programming. *Evolutionary Computation 3*, 1 (1995), 17–38.

[82] ZHANG, H. The Optimality of Naive Bayes. In *FLAIRS Conference* (2004), V. Barr and Z. Markov, Eds., AAAI Press.

[83] ZHANG, M., ANDREAE, P., AND PRITCHARD, M. Pixel statistics and false alarm area in genetic programming for object detection. In *Applications of Evolutionary Computing, EvoWorkshops2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM* (University of Essex, UK, 14-16 Apr. 2003), G. R. Raidl, S. Cagnoni, J. J. R. Cardalda, D. W. Corne, J. Gottlieb, A. Guillot, E. Hart, C. G. Johnson, E. Marchiori, J.-A. Meyer, and M. Middendorf, Eds., vol. 2611 of *LNCS*, Springer-Verlag, pp. 455–466.

[84] ZHANG, M., AND BHOWAN, U. Program size and pixel statistics in genetic programming for object detection. In *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC* (Coimbra, Portugal, 5-7 Apr. 2004), G. R. Raidl, S. Cagnoni, J. Branke, D. W. Corne, R. Drechsler, Y. Jin, C. R. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, Eds., vol. 3005 of *LNCS*, Springer Verlag, pp. 379–388.

[85] ZHANG, M., AND SMART, W. Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. *Pattern Recognition Letters 27*, 11 (Aug. 2006), 1266–1274. Evolutionary Computer Vision and Image Understanding.

[86] ZHANG, M., WONG, P., AND QIAN, D. Online program simplification in genetic programming. In *Simulated Evolution and Learning, Proceedings 6th International Conference, SEAL 2006* (Hefei, China, Oct. 15-18 2006), T.-D. Wang, X. Li, S.-H. Chen, X. Wang, H. A. Abbass, H. Iba, G. Chen, and X. Yao, Eds., vol. 4247 of *Lecture Notes in Computer Science*, Springer, pp. 592–600.