VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui

Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Simplification of Genetic Programs During Evolution

Phillip Wong

Supervisor: Dr. Mengjie Zhang

Submitted in partial fulfilment of the requirements for
Bachelor of Science with Honours in Computer Science.

Abstract

Program Bloat/Redundancy is a critical problem in the area of Genetic Programming (GP). Redundancy leads to slow downs in the GP process, particularly in complex tasks, and causes unnecessary exploration of irrelevant parts of the genetic search space.

This project investigates the application of simplification to genetic programs during the course of the evolutionary process to combat program bloating. This system is then applied to several regression and multi-class classification tasks of varying difficulty.

Two simplification methods are tested in this project. The first is new algebraic method which acts directly on tree-based programs and uses an algebraic equivalency component to identify equivalent expressions. The second is an improved version of PRES, an existing simplification system which involves a prime number representation. Both methods were found to normally improve performance on the experimental tasks.

Variation in how often simplification is applied and to what proportion of programs is investigated. Three selection methods for selecting programs to be simplified are tested and compared: Random Selection, Fitness Based Selection and Fatness Based Selection. Fatness Based Selection is found to provide superior performance to the other two methods.

Analyses of building blocks within GP systems both with and without simplification are also performed. Two different perspectives of building blocks are used, and analyses using these perspectives strongly suggest that simplification removes building blocks in the system.

Acknowledgments

I would like to acknowledge the following:

- My supervisor Dr. Mengjie Zhang, for pushing me to do project work throughout the year. The weekly meetings and constant reminder proved invaluable to the project.
- The 4^{th} floor ″BIT″ lab CO437, where I resided most of the year. The other denizens of the lab were a *constant* source of entertainment.
- Melissa, who has supported me throughout this project when things were not going as planned.

Contents

Figures

List of Tables

Chapter 1

Introduction

Genetic programming (GP) [15, *Koza 1992*], is a method of automatically creating programs for solving specified tasks by using the concept of *evolution*. Firstly, an initial group of randomly generated programs (usually represented as parse trees such as LISP-S trees) is created. The process of selection (selecting programs based on fitness) is carried out to provide a basis for the next program generation. Fitness is determined by running the programs and evaluating them on a set of criteria. The genetic operators of *crossover* (swapping of sections of programs), *mutation* (random alterations to a program) and *reproduction* (making exact copies of a program) are applied to the selected programs to create a new population of programs. The process of creating new generations is repeated until termination criteria is met. The *"best"* program in this last generation is returned as the resulting solution. GP can be seen as a genetic beam search through the space of possible solutions to the task.

Genetic programming is an emerging field in evolutionary computing and machine-learning and has already been applied to many tasks, including image analysis [25, *Poli 1996*], object detection [36, *Winkeler & Manjunath 1997*], classification, regression problems [15, *Koza 1992*] and even control programs for walking robots [4, *Busch et al.*]. Genetic programming has been very successful in solving or performing these tasks and "now routinely delivers high-return human-competitive machine intelligence." [17, *Koza 2004*].

1.1 Motivation

1.1.1 Program Bloat

One of the problems in GP is that the process of genetic programming will inevitably introduce some redundancy into the evolved programs ([15, *Koza 1992*], [32, *Soule 1996*], [2, Blickle 1994]). This redundancy is regarded as a fundamental problem of genetic programming as it slows down the search process by consuming large amounts of memory and causes exploration of large unnecessary parts of the search space. The search process continues to slow as the programs become larger until the programs become too large for the system's memory to hold, halting the system before a "good" solution can be found. Redundancy can also result in an unnecessarily complex program, which is inefficient in its execution and difficult to interpret and comprehend.

As an example, the algebraic expression '$\alpha^2 \times 1 - \alpha^2 + \beta^3$' can be simplified to the expression 'β^3'. The subexpression '$\alpha^2 \times 1 - \alpha^2$' is thus considered a redundancy, as when evaluated, the result of this subexpression is 0. In other words, it contributes nothing to the fitness of the program, nor to accomplishing the task. Another example of redundancy, is '$\gamma + 4.1234 + 9.7530$'. In this case, the constants can be brought together to form '$\gamma + 13.8764$'.

On the other hand, this redundancy may aid the effectiveness of the evolutionary process by providing a more diverse selection of program fragments for the process to use. For example, consider the expression '$cos(\theta) - cos(\theta)$'. This too evaluates to 0, but if this is the only instance of '$cos(\theta)$' in the system, and simplification removes this expression, it would remove '$cos(\theta)$' from the available pool of program fragments. Redundancy may also help *preserve* good building blocks (a subexpression which makes a large contribution to solving the task) within the program, by reducing the chances of a good building block being destroyed by the crossover operator (as crossover may randomly choose a crossover-point within that block) [2, *Blickle 1994*].

1.1.2 Simplification

Simplification can be implemented in various ways, including using simple algebraic techniques, translation into canonical forms or numeric hashing techniques. A few of these have been applied to GP ([29, *Smart, 2003*],[40, *Zhang, 2004*], [5, *Ekart, 1999*]) and each approach has various strengths and weaknesses that could possibly be combined together and improved upon.

Typically simplification is applied at the end of the evolutionary process to remove some of the complexity of the program, reducing the resource usage and improving comprehensibility, enabling it to run faster and to be easier to interpret (the *editing operation* proposed by Koza [15] is an example of this). But as program redundancy is a problem which also occurs *during* the evolutionary process, simplification during the evolutionary process to improve performance in the whole system needs to be investigated, in particular, whether the removal of this redundancy breaks up the good building blocks, and reduces the effectiveness of the system.

1.2 Project Goals

This project addresses the problem mentioned above of redundancy in genetic programs and investigates the effects of simplifying the genetic programs during the evolutionary process. It focuses on whether the effects from the removal of redundancy and reduction in complexity outweigh the benefits of leaving the redundancy in the program, and also whether a balance can be found by varying the frequency/proportion of applying simplification. Specifically, this project addresses the following goals/questions:

- How can simplification of the genetic programs can be achieved using simple algebraic techniques?
- How can simplification of the genetic programs can be done using a numeric hashing method?
- Does simplification in genetic programming destroys good buildings blocks in the system?
- Can a system using simplification outperform the standard genetic programming approach in terms of system effectiveness, efficiency and comprehensibility?
- How does the frequency to which the simplification algorithm is applied to the evolutionary process affect the effectiveness/efficiency of the GP system? For example, several generations of normal GP process followed by a single generation where simplification is applied.
- How does the proportion of the programs in each generation to apply the simplification algorithm affect the effectiveness/efficiency of the GP system? For example, applying the simplification to only 10% of programs in a generation.

1.3 Contributions

This project puts forward the following contributions:

1. This project describes another method using algebraic techniques to simplify the genetic programs in GP *during* evolution.

   This method uses a simple set of simplification rules, along with an algebraic equivalence component to directly simplify tree-based programs without the need to convert them to another form. The set of rules are designed to target simple forms of algebraic redundancy (e.g. `x -`

`x = 0`) in order to remain small and fast. They are applied to the program in a bottom-up recursive manner.

   Using this component to simplify genetic program during evolution, it is found that the performance of GP systems can normally be improved, with no loss in effectiveness (and in many cases an improvement in effectiveness was achieved). Systems using simplification are generally more efficient and yield more comprehendable solutions.

2. This project outlines improvements to PRES (an existing numeric hashing simplification system developed by Zhang[40]) by integrating an algebraic equivalence component among other improvements.

   The PRES algorithm encodes tree-based programs into another representation consisting of products of primes. By using an algebraic equivalence component, identical subtrees can be properly identified and encoded with the same prime number, allowing the algorithm to simplify more types of expressions.

   The addition of a *Monte Carlo* form of reconstruction (where simplified programs may radically change from the original, while still being functionally equivalent) harnesses the properties of *unstable* simplification to create new building blocks as a byproduct of simplification.

   The inclusion of an arbitrary precision math library prevents arithmetic overflows and allows the application of the PRES algorithm to larger and more complex programs.

   Using the improvements made to this method, it was found that this method improved the performance of GP for relatively difficult tasks.

3. This project develops and tests three different methods of selecting programs for simplification: *Random Selection*, *Fitness Based Selection*, *Fatness (Size) Based Selection*. Experiments show that while yielding very similar results to each other, *fatness based selection* can obtain marginally better performance than the other two methods.

   Using this selection method for further experiments, guidelines are deduced for determining at what frequency simplification should be performed and to what proportion of programs, in order to achieve the best system effectiveness and the best system efficiency.

4. This project presents initial building block analysis of GP systems using simplification, using two different perspectives of building blocks. Using results from both of these definitions, it is found, by tracking the number and size of building blocks contained in GP systems running on a simple task, that simplification does indeed remove building blocks from GP systems.

1.4 Structure of this Document

This chapter has introduced the main ideas and motivations behind this project. The rest of the project report is organised as follows.

- Chapter 2 presents background material and related work in more detail to familiarise the reader with the concepts of GP, simplification, multi-class classification and related work.
- Chapter 3 describes the datasets and experimental setup (GP and simplification parameters) used in this project to gather results.
- Chapter 4 describes a simplification system based on applying a set of algebraic rules and the effects of using it on programs in a GP system during evolution.
- Chapter 5 outlines improvements to PRES, an existing simplification system that uses hashing involving prime numbers to simplify programs to arbitrary depth. Most significant is the integration of an Algebraic Equivalence Hashing method.

- Chapter 6 investigates the effects of varying the frequency and proportion of simplification and whether a balance of effectiveness/efficiency can be found through this variation. Three different proportion selection schemes are developed and tested.
- Chapter 7 looks at the building blocks in a GP system to determine how simplification affects these building blocks. Two forms of building block analysis is used to track building block propagation from generation to generation in a GP system evolving a solution for a simple task.

Chapter 2

Background

2.1 Overview of Genetic Programming

Genetic programming (GP) [15, *Koza, 1992*] is an exciting new area of evolutionary computing and machine learning, in which programs are automatically generated using concepts from biological evolution to solve various tasks. Initially, a *population* of programs is randomly generated. These programs are then executed to perform the specified task the system is designed to solve and evaluated based on a *fitness criteria* (e.g. best accuracy for classification, traveling the furthest distance for a walking robot) by use of a *fitness function*. A new population of programs is derived from the previous population by applying *genetic operators* to the programs and placing the results in the new population.

This process of generating new populations and evaluating them is repeated until a desired stopping condition is met (e.g. a solution is ″good enough″ and meets a fitness requirement, the system has run for a predetermined number of generations). The ″best″ program in the final generation is regarded as the system's *solution* for the task. In essence, GP uses a genetic *beam search* through the space of candidate solutions (programs) to a task.

A flow chart of the GP process is shown in figure 2.1.

2.1.1 Program Representation

Programs in GP systems can be represented in many different forms. The standard approach is to represent programs in a *tree-based* form (e.g. LISP-S), although several other approaches have been explored, including *graph* representations ([13, *Keijzer 1996*]), *linear* representations ([1, *Banzhaf et al.*], machine code representations ([22, *Nordin 1997*]) and grammar based representations [35, *Whigham 1995*]).

In the standard tree-based approach (which is used in this project), the leaf nodes represent the program inputs or *terminals*, and the internal nodes represent the *functions*, which form the working part of the program. The functions take their child nodes as inputs and output to their parent node, building up and outputting a *single* numerical result from the root of the tree. The sets of functions and terminals for a GP system are usually highly task dependent and often have to be carefully crafted to suit the task if one wants to achieve good results.

Figure 2.2 shows an example of a typical GP program and represents the LISP expression `(+ (* 0.12 (- 0.66 x))y)` or in more ″conventional″ mathematical notation `(0.12 * (0.66 - x)) + y`. If given the input values `x = 2` and `y = 7`, the result of the program evaluates to `6.8392`.

2.1.2 Initial Program Generation

There are several approaches to generating the initial program generation ([15, Koza 1992]).

- *Full Method* - Functions are selected as the nodes of the program tree until a specified depth is reached. Then a layer of terminals nodes are added to form the rest of the program. This makes

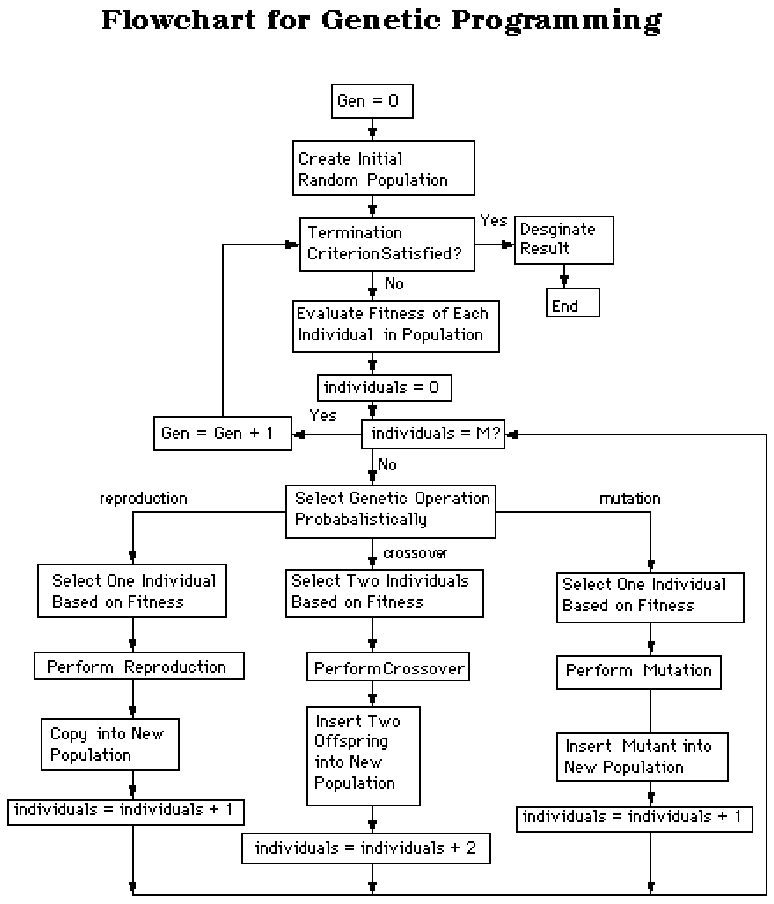
Flowchart for Genetic Programming
Gen = 0
Create Initial Random Population
Termination Criterion Satisfied?
Yes
Desginate Result
End
No
Evaluate Fitness of Each Individual in Population
individuals = 0
Yes
Gen = Gen + 1
individuals = M?
No
reproduction
Select Genetic Operation Probabalistically
mutation
crossover
Select One Individual Based on Fitness
Select Two Individuals Based on Fitness
Select One Individual Based on Fitness
Perform Reproduction
Perform Crossover
Perform Mutation
Copy into New Population
Insert Two Offspring into New Population
Insert Mutant into New Population
individuals = individuals + 1
individuals = individuals + 2
individuals = individuals + 1

Figure 2.1: Flowchart of the GP system process, from The GP Tutorial [7]

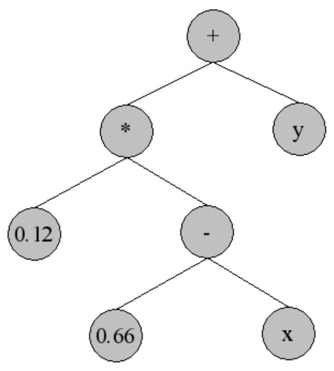

Figure 2.2: Example of a standard GP program

all programs in the initial generation the same depth.

- *Grow Method* - Either functions *or* terminals are selected to be the nodes of the program tree. If a terminal is selected, then that branch of the program is not "grown" any further. This varies the depth and size of the programs generated for the initial generation.

- *Ramped Half-and-Half Method* - This is a mixture of the other two methods. Half of the programs in the generation are created using the *grow* method, and the other half are created using the *full* method.

For this project, the *full* method of creating the initial generation of programs was used.

2.1.3 Genetic Operators

As mentioned above, the genetic operators are used to alter the programs in the current population in order to create a new population. In a standard GP system, the following three genetic operators are usually present: *Reproduction*, *Mutation* and *Crossover*.

Reproduction/Elitism

Reproduction (or elitism), is the operation of copying a number of the *fittest* programs from one generation to the next. This process preserves these programs and ensures that the fittest program in the next generation is *at least as fit* as the fittest program in the current generation. An example of elitism is shown in figure 2.3.

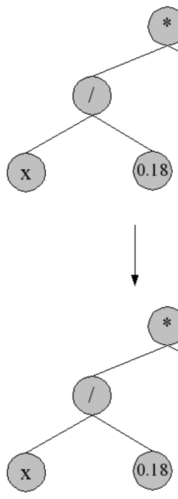

Figure 2.3: Simple example of elitism

Mutation

Mutation is akin to biological mutation, wherein a random DNA sequence in an organism is altered, possibly affecting the organism in a positive, negative or neutral manner. In standard GP, mutation of a program takes a random subtree of that program and replaces it with an entirely *new* randomly generated subtree. This process ensures the introduction of new functions and terminals into the population to help retain diversity of the "genetic materials" and prevents *stagnation* and *homogeneity*, which results in all the programs in a population becoming identical. An example of mutation is shown in figure 2.4.

Crossover

Crossover is akin to biological sexual reproduction, wherein offspring inherit a *combination* of genes from its two parents. Standard 2-point crossover achieves this by first selecting two programs in the current population. A single "crossover point" is randomly chosen within *each* of these programs and the subtrees below these points are swapped, producing two offspring programs with differing combinations of the parents structure. An example is shown in figure 2.5.

2.1.4 Terminal and Function Sets

These sets create the pool of terminals (leaf nodes) and functions (internal nodes) from which the initial programs will be created from. Mutation also generates its random subtrees from this pool. It is important for these sets to be *sufficient* and *closed* ([15, *Koza 1992*]).

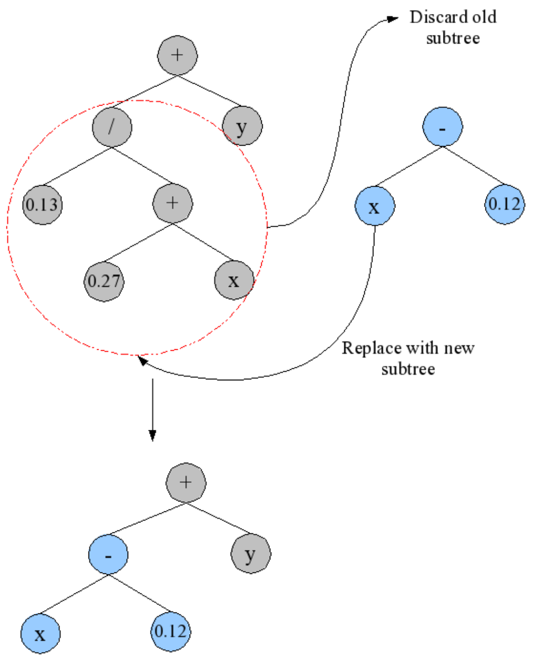

Figure 2.4: Simple example of mutation

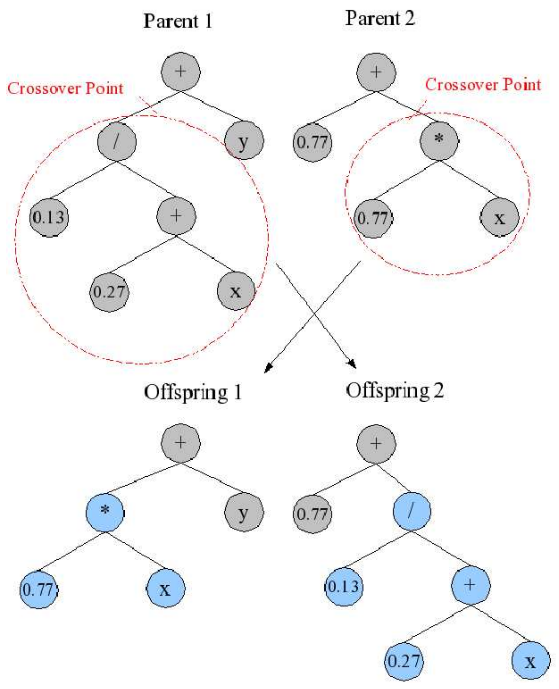

Figure 2.5: Simple example of crossover

In standard GP, *Sufficiency* is a requirement that the specified set of functions and terminals given is actually capable of generating a solution to the given problem. Obviously, if the given functions and terminals are not sufficient, a solution cannot be found. *Closure* requires that all of the functions and terminals return the same type, so that every function can accept the output of any other function or terminal. This ensures that any generated or recombined tree combination is a valid program.

2.1.5 Fitness Function

In GP, the *fitness function* is a measure of how "good" a genetic program is at performing a task. Since a GP system favours programs which are "fitter" (e.g. through elitism), a fitness function can largely affect the way the evolutionary process behaves. A carefully crafted fitness function help preserves programs that aid the evolution process in obtaining the best solution.

2.1.6 Other GP Parameters

Other GP parameters exist which also affect the way a GP system behaves. These include:

- *Population Size*: This is the number of programs that exist in a population. If this is two small, the amount of "genetic" diversity is lessened. If this is too large, each generation will take a long time to process.
- *Crossover, Reproduction and Mutation Rates*: These dictate the proportion of a population that is subjected to these genetic operators. Balancing these rates is task dependent, as some tasks may be easy to get to a "decent" solution, but require a higher mutation rate to obtain a higher fitness.
- *Minimum/Maximum Tree Depth*: Standard GP trees grow very quickly, and so are usually limited in depth to keep resource usage down. If the maximum tree depth allowed is too low (i.e. solutions only exist with larger program trees), then the system will perform poorly.
- *Maximum Generations*: This is used as a *Termination Criterion* (explained in the next subsection). If this number is too low (especially when combined with a small population size), then the system will be prematurely stopped before a "good" solution can be evolved.

These parameters have to be considered when designing the experimentation, and a list of the parameters used in this project is shown in section 3.4.

2.1.7 Termination Criteria

The termination criteria determine when the system should stop executing, and a final "solution" should be outputted. These are split up into several categories including:

- *User Control* - The GP keeps running until the user manually terminates the system.
- *Fitness Control* - If the fitness measure passes a user defined value (e.g. 100% accuracy for the training set), then the GP system is stopped.
- *Early Stopping* - Can be achieved through a number of methods (e.g. validation sets). This criterion is designed to stop the system before *overfitting* occurs (see next subsection for an overview of overfitting).
- *Generation Control* - The system stops after a user defined number of generations has been executed. A common value for the number of generations is 50.

Usually a combination of these termination criteria are used in a single GP system.

Over-fitting

Over-fitting is a situation where the learning system (whether it be GP, Neural Networks, Bayesian or any other system) begins to generate solutions that are too specific to the training data provided. This means that at a point in the learning process, while the training fitness is still becoming better, the fitness of the programs on the testing data has started to worsen. Figure 2.6 shows an example

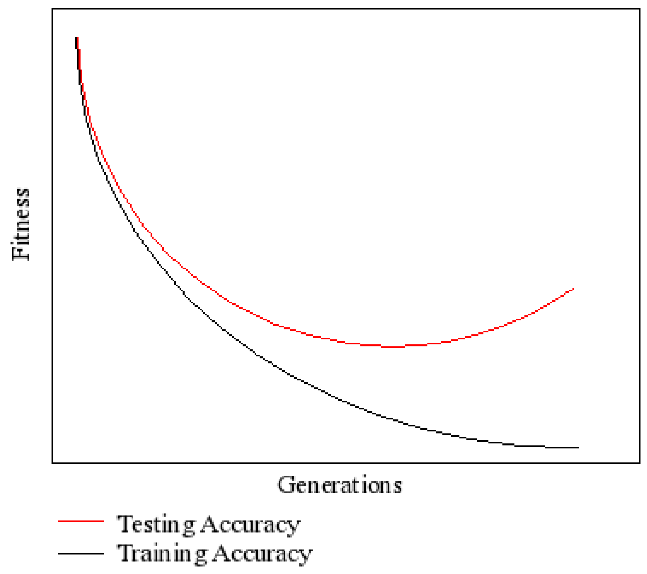

Figure 2.6: Graph example of 'over-fitting'

of an error measure (i.e. less error is better) being used as a fitness function, and an example of the system overfitting training data.

While over-fitting is not an issue when *all* possible instances are given in the training dataset, usually only a portion of possible instances are available for the system to train on. To help prevent over-fitting, the dataset is usually split into *two* sets, a training set and a *validation* set. The learning system is trained using the training set and tested on the validation set. If the fitness of the system is continually improving for the training set, but worsening for the validation set, over-fitting is assumed to have begun and the system is prematurely halted.

2.2 Multi-class object Classification

Classification is an area of Data Mining/Knowledge Discovery [6, *Fayyad et al.*] and Pattern Recognition ([24, *Paulus, Hornegger 1998*], [28, *Schuermann 1996*]) which concerns identifying and organising different types of data into a set of coherent classes. Object classification is a subset of this, in which the data is related to specific objects (e.g. computer images or descriptions). This task arises in many different applications, including object detection in images [36, *Winkeler 1997*], including automatic tracking systems [30], face recognition ([33, *Teller 1995*]) and handwriting recognition ([19, *LeCun, 1995*]).

In most object classification cases, object classifiers have limited themselves to distinguishing between two types of classes (e.g. A boat, or *not* a boat, a square or a circle). Multi-class object classifiers deal with tasks which involve three or more distinct classes of objects (e.g. square, circle and triangle classes, multiple faces).

Not surprisingly, multi-class object detection ([37, *Zhang, 1999*]) and classification is generally a more difficult task for a GP system (or indeed, any system) to solve. It is especially hard in a standard GP system as there is only a *single* output from any of the programs. This output must be translated into a class identifier in order to perform the classification.

The most basic method is a *static-range selection* method ([38, *Zhang, 2003*], [39, *Smart, Zhang*]). Each class is designated a window of fixed size in the output value to represent that class. This method is not the best performing when compared to other, dynamic methods (e.g. *centered dynamic range selection*, *slotted dynamic range selection* [29]), but this approach is usually sufficient for some tasks. As an example of how the static boundary partitions the output. Figure 2.7 shows a partition of a four class problem (with a fixed window size of 0.5 for each class).

$$class = \begin{cases} class1 & output < -0.5 \\ class2 & -0.5 \leq output < 0 \\ class3 & 0 \leq output < 0.5 \\ class4 & 0.5 \geq output \end{cases}$$

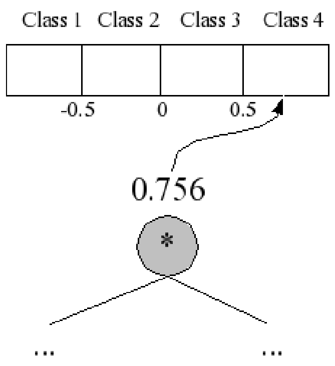

Figure 2.7: Diagram of static-range selection multi-class classification

As the goal of this project is not to develop or investigate multi-class translation methods, the experiments in this project simply uses the *static-range* method outlined above. This method is also one of the easiest to translate (as it is not dynamic) and so will make understanding how programs work (comprehensibility) much easier.

2.3 Program Redundancy and Simplification

2.3.1 Redundancy

As mentioned in the introduction, program redundancy is a section of a program that does not contribute at all to solving the task. Figure 2.8 shows an example of program redundancy, where the `(- x x)` subtree does not contribute (as `0 + a = a`).

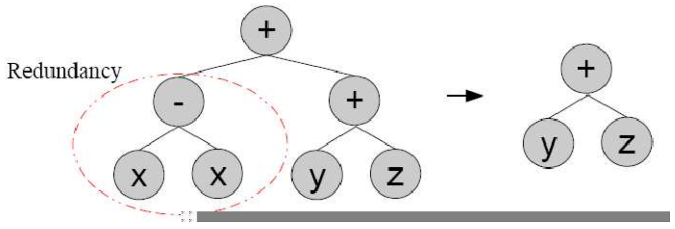

Figure 2.8: Example of redundancy in GP programs

Redundancy is not limited to mathematical expressions. Consider a control program for a robot, where the function set includes { turn left, turn right, move forward }. It is easy to see that a subprogram `turn left, turn left, turn left` is equivalent to a single `turn right`, and that complex combinations of these may be equivalent to much shorter programs.

In the example in figure 2.9, the task is to construct a program that moves diagonally north-west two squares (a *very* basic program). Both path A and path B represent two different "solutions" to the task. Although they both are valid, it is obvious that the path A is more optimal, and that path B contains some redundancy (moving down only to move up later on).

This shows that for many tasks (even with specialised, non-standard function/terminal sets), redundancy can exist, and so simplification to remove that redundancy affects tasks across the GP spectrum.

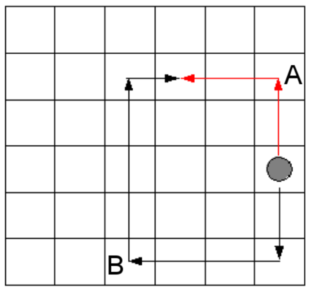

Figure 2.9: Example of redundancy in non-arithmetic task

2.3.2 Simplification

Simplification systems to automatically simplify algebraic expressions have existed for many years, simply because computer systems are a very useful tool for performing menial mathematical procedures. While not necessary for a computer system, simplification is a very useful process to make programs more understandable for human interpretation.

Algebraic simplification: a guide for the perplexed [21, Moses 1971], describes the different types of algebraic simplification systems. Moses outline four basic types of systems, and named them in a political sense:

- *Radical*: Works with well defined expression types (e.g. polynomials) and relies of keeping a canonical internal representation of all expressions. Any simplification rules rely on this canonical structure in order to work.
- *Liberal*: Works with more general forms of expressions and uses simple rules that cater for most situations. This is most akin to how people simplify expressions using pen and paper.
- *Conservative*: This type of system works on the principle that "there is no set of rules that can handle *all* situations". It contains very few rules of its own and allows the user to define their own set of simplification rules.
- *Catholic*: This type of system uses more than one type of approach to simplification, so that if one approach does not work, it tries another.

These four categories encompass the majority of simplification systems available today.

2.4 Schema Theory

Schema theory is an attempt to find a general theory as to why genetic programming works. Such a theory exists in Genetic Algorithms (GA) in the form of the *Building Block Hypothesis*: That the GA process combines short, relatively fit schemata ('building blocks') together to form more and more complete solutions to a task. This is generally used to explain how GA systems work. In GP however, there exists no equivalent hypothesis and many works have been made in attempting to form one ([26], [15], [23]).

Most of these attempts have been purely theoretical, although Poli and Langdon [26] provided both theoretical and *experimental* results to investigate the propagation and destruction of schema in a GP system. They specifically investigated the crossover operator to determine its effects on the schema in the system.

2.5 Related Work

2.5.1 Program Bloat

Several approaches have been developed and investigated to combat the program bloating problem.

One of these is *parsimony pressure*, which uses a *multi-objective fitness function* (in this case, a fitness function that would measure the program fitness *and* the program size) to penalise larger sized programs during the evolutionary process. It is hence possible for a "fit", large program to be given a lower fitness than smaller programs that may not be as good at solving the task. This method has been shown to worsen the effectiveness of the evolved solutions [31, *Soule*] and has been largely discarded as a viable solution to program bloat.

Another approach is the addition of *reusable code*, subtrees which are stored once and can be linked to by multiple programs. An example of this is the *automatically defined function* (ADF) [16, Koza 1994]. An ADF, like other programs, is also evolved during the GP run and may be linked to by programs in the population. This approach benefits from tasks which are highly regular and involve many repetitive actions/routines, as subtrees which perform these actions/routines need only be stored once, resulting in a reduction in memory usage.

It should be noted that program simplification methods are different from the above two methods, although they all aspire to solve the same problem.

2.5.2 Program Simplification

In previous work, [5, *Ekart 1999*], [29, *Smart 2003*] and [40, *Zhang 2004*] applied varying simplification methods to different GP tasks.

Ekart used an algebraic rule system in the form of a set of `prolog` statements. This required programs to be converted into string format, a form which `prolog` can parse and use. This system was tested on symbolic regression tasks only, with little variation in frequency and proportion of simplification.

Smart also used an algebraic rule system based on *string matching*, where a program again needed to be translated into string format. A set of simplification rules (using pattern matching) was enacted on this string, the result transformed back into a program and inserted back into the GP system. Using this system on two multi-class classification tasks, it was found that overall, small improvements in effectiveness were achieved, as well as shorter training times. In this project, an algebraic simplification method that works directly on the tree structure itself is developed and applied to the programs in a GP system. The standard and modified GP systems will be applied to several symbolic regression and multi-class classification tasks.

As simplification may destroy good building blocks in a GP system, frequency and proportion will be varied, allowing some programs to avoid simplification and possibly "sparing" good building blocks.

Zhang used a system (dubbed *PRES*) which hashed a program by utilising prime numbers. Programs were encoded into a "prime product quartuple" representation. This is achieved by mapping nodes and subtrees of programs to prime numbers. The prime product quartuple stored these nodes and subtrees as a product of the primes they are mapped to.

By representing programs in this form, cancelling of redundancies becomes a simple task of factoring out the *greatest common divisor* (gcd) from the prime product quartuple.

After simplification has been performed, the nodes and subtrees were then reconstituted from the leftover primes to form a simplified form of the original program. A more detailed explanation of the PRES algorithm is explained in chapter 5.

This form of system is essentially a "radical" system as it converts the program into an internal representation structure in order to perform simplification. Using this simplification method, again, very small improvements were concluded to be possible. The algorithm in its current form is limited in the size of programs it can be performed on. In this project, changes are made to improve the effectiveness of PRES, allowing the algorithm to support more varied types of expressions and be applied to larger and more complex tasks.

2.5.3 Building Block Analysis

Much work has been undertaken in previous years to explain why GP works. In genetic algorithms, there is a widely accepted (although not by all) hypothesis that "short, low-order, highly fit" schema are recombined to form even more highly fit, higher-order schema. This "ability to produce fitter and fitter partial solutions by combining blocks is believed to be the primary source of the GA's search power" [9, *Forrest, Mitchell 1996*]. [12, *Holland 1975*] provided further support for this hypothesis, by using schema analysis to gain experimental results.

The schema concept provides a different perspective on GA and GP systems. Rather than using the view of evolutionary processing of programs, one can think of the system as "schema processing". A schema's fitness is determined by averaging the instances (programs) that a schema matches.

Many attempts have been made to bring a similar theory to the "Building Block Hypothesis" to the realm of GP. Koza [15, *Koza 1992*] first addressed this issue in 1992, defining a schema as a set of subtrees formed from a special schema defining set (e.g. if H_1 is defined as {`(- 2 y), (+ 2 3)`}, then H_1 is a schema representing all programs that contain `(- 2 y)` or `(+ 2 3)`). Koza concluded that GP crossover preserves good schema and that smaller, good schema are combined into larger schema.

Other schema work ([23, *O'Reilly 1995*], [26, *Poli, Langdon 1997*]) has attempted to bring schema definitions closer to those in GA. O'Reilly introduced a "don't care symbol" (#) to Koza's definition, which could take the value of any valid subtree.

Conclusions from these works find that probability of disruption is dependent on the size of the program, and that GP may actually be a two-phase system.

All of these works however have concentrated on explaining the effects of crossover GP, and no work has been done to investigate simplification's effect on a GP system. In this project, similar methods to these works are used to investigate simplification. Whether simplification disrupts building blocks, or whether simplification can even construct new building blocks.

Chapter 3

Datasets and Experimentation Setup

In order to investigate the goals set out by this project, several tasks were chosen. These tasks were selected to test both GP systems with simplification and the standard GP system over different types of problems, with different difficulties. The following tasks were chosen for to achieve this.

3.1 Task 1: Symbolic Regression

Symbolic regression [15, *Koza, 1992*] is the task of "regressing" a mathematical expression/formula which best fits a given set of data points. This method of regression works differently to other methods of regression, such as using *neural networks*. Depending on the quality of the set of data points given (i.e. the amount of noise) or the complexity of the formula, this expression may be very easy or very difficult to obtain.

For this project, two symbolic regression tasks were chosen of which one is a relatively "easy" expression to regress and the other is relatively "hard".

3.1.1 Easy Regression Problem

$$EasyProblem : f(x) = x^2 + x + 4.5$$

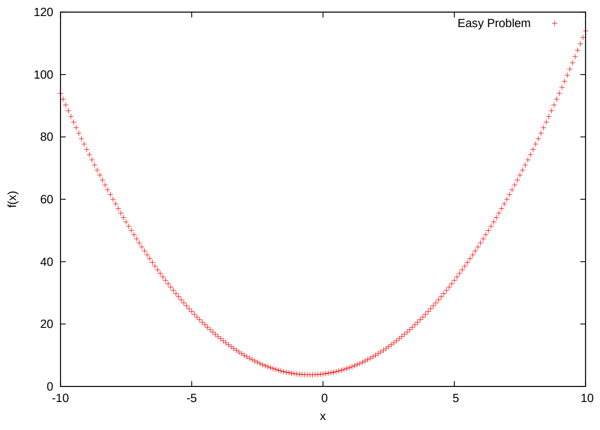

Figure 3.1: A plot of the Easy Problem equation

This first problem is a relatively straight-forward problem to solve, as it only consists of a couple of functions ($+$ and $*$) and only a single, small constant. The constant being small is important, as it can therefore be produced easily by combining the small constants initially generated by the GP system. The dataset for this problem consists of 200 data points taken at 0.1 intervals between to values of -10 and 10. Sample data points from this dataset are shown in table 3.1.

x	...	-0.4	-0.3	-0.2	-0.1	0.0	0.1	0.2	0.3	0.4	...
$f(x)$		3.76	3.79	3.84	3.91	4	4.11	4.24	4.39	4.56	

Table 3.1: Sample 'Easy Regression' data points

3.1.2 Hard Regression Problem

$$HardProblem : g(x) = \begin{cases} x^3 - 5.8x + 3 & x < 0 \\ \frac{x-12}{x^2} + 1 & x \geq 0 \end{cases}$$

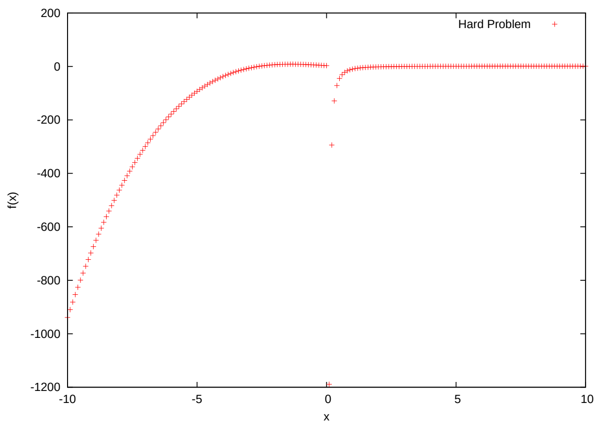

Figure 3.2: A plot of the Hard Problem equation

This second regression problem is a much more difficult task for the GP system, as the function which is used to generate the data points is much more complex. It uses all of the standard arithmetic operators, the $if < 0$ function, and several constants. Similarly to the "easy" problem above, the dataset for this problem consists of 200 data points taken at 0.1 intervals between to values of -10 and 10. Sample data points from this dataset are shown in table 3.2.

x	...	-0.4	-0.3	-0.2	-0.1	0.0	0.1	0.2	0.3	0.4	...
$g(x)$	5.256	4.713	4.152	3.579	3	-1189	-294	-129	-71.5		

Table 3.2: Sample 'Hard Regression' data points

3.1.3 Features

The expressions used in both symbolic regression tasks consist of a single variable (x). Therefore there is only one feature input for the GP system, which corresponds to that x.

3.1.4 Terminal and Function Sets

The terminal set consists of randomly generated *numerically valued terminals* as well as *feature terminals*. The numeric features are simply n floating point numbers generated in the range of [-1, 1] using a uniform distribution random number generator ($r_0, r_1, ..., r_n$). In the standard GP system, these numeric terminals will remain unchanged in the course of the evolutionary process, while in the modified system, simplification will bring some of these terminals together to form new constants. There is only a single feature terminal (f_0) which represents the single variable in the expression.

$$\textit{Terminal Set} = \{r_0, r_1, ..., r_n, f_0\}$$

The function set used for this tasks consists of the four basic arithmetic operators, as well as a conditional *if* operator. The division used is the commonly used "protected division" where a divide by zero results in zero, removing the *undefined* case. The conditional *if* operator takes three parameters, a *condition* which will be evaluated, a *true branch* if the condition evaluates to $<$ zero and a *false branch* if the condition evaluates to $\geq$ zero.

$$\textit{Function Set} = \{+, -, *, /, \texttt{if<0}\}$$

3.1.5 Fitness Function

The fitness of each program is determined by evaluating the program over 200 consecutive points, between the range [-10, 10] using 0.1 increments. The SSE (sum squared error) is calculated from the differences between the program output and the target expression output to give the program fitness. A lower fitness is therefore desirable in this case with 0 (no difference between the program output and the "true" output) being the optimal fitness.

$$SSE = \sum(output_{actual} - output_{desired})^2 \tag{3.1}$$

3.1.6 Termination Criteria

Two basic termination criteria were used, stopping the system when the fitness of the best program in a generation was 0 (i.e. the optimal fitness) or when the 50 generation limit was surpassed.

3.2 Task 2: Coin Classification

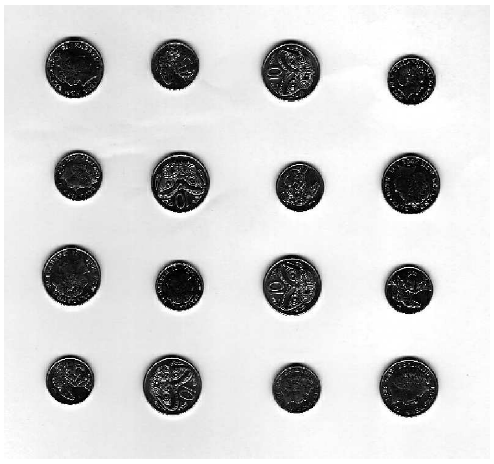

Figure 3.3: A sample of coin images in the dataset

This task is an example of a *multi-class object classification* task, in which objects depicted in images are classified by a program and the accuracy measured. The dataset used in this case is a set of images which contain 5 and 10 cent New Zealand coins with various orientations on a relatively uniform (white) background. The objects in this dataset consist of 480 70x70pixel cutouts of these coins from the image, and are split into four distinct classes:

Class	Example
10c Tail	
10c Head	
5c Tail	
5c Head	

Table 3.3: The four coin classes

3.2.1 Features

The features for this task consist of 8 *pixel statistic* features extracted from the object images in the dataset. The 8 features used are the mean and standard deviation of four concentric square regions of increasing size in each object image, which have been normalised to the range between -1 and 1.

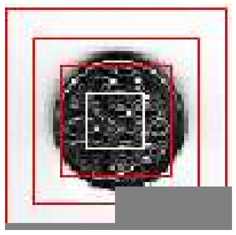

Figure 3.4: The regions from which the coin features are extracted

3.2.2 Terminal and Function Sets

The terminal set consists of randomly generated *numerically valued terminals* as well as *feature terminals*. The numeric features are simply n floating point numbers generated in the range of [-1, 1] using a uniform distribution random number generator ($r_0, r_1, ..., r_n$). In this classification task, the feature terminals ($f_0, ..., f_7$) represent the 8 domain independent, *pixel statistic* features that were extracted from the images.

$$\textit{Terminal Set} = \{r_0, r_1, ..., r_n, f_0, ..., f_7\}$$

Again, the function set used for this tasks consists of the four basic arithmetic operators, as well as a conditional *if* operator. The division used is the commonly used "protected division" where a divide by zero results in zero, removing the *undefined* case. The conditional *if* operator takes three parameters, a *condition* which will be evaluated, a *true branch* if the condition evaluates to $<$ zero and a *false branch* if the condition evaluates to $\geq$ zero.

$$\textit{Function Set} = \{+, -, *, /, \texttt{if<0}\}$$

3.2.3 Fitness Function

The fitness of each program is determined by the classification accuracy of the program on the given *training* set of images (although final evaluation for results of evolved solutions are performed on a *test* set). Obviously, a higher fitness is desirable in this task with the maximum being 100% accuracy where all of the object patterns are correctly classified.

$$fitness = \frac{no.\ objects\ correctly\ classified}{total\ no.\ objects} \tag{3.2}$$

3.2.4 Classification Strategy

This task consists of 4 classes, and so using the static-range selection method (with a slot size of 0.25, which was determined to give better results than larger slot sizes), the class boundaries are defined as follows:

$$class = \begin{cases} 5chead & output < -0.25 \\ 5ctail & -0.25 \leq output < 0 \\ 10chead & 0 \leq output < 0.25 \\ 10ctail & 0.25 \geq output \end{cases}$$

3.2.5 Termination Criteria

For this task, three termination criteria were used to stop the GP system, whichever criterion occurs first. They were:

- The fitness of the program is optimal (i.e. 1.00), which denotes that the training accuracy is 100%.
- The accuracy of the validation set "starts to fall", which is used as an indication of overfitting.
- The 50 generation limit is surpassed.

3.3 Task 3: Face Recognition

This task is another example of a *multi-class object classification* task. The dataset uses a subset of the *Yale Face Database B* [11], a database of images of different peoples faces, with different lighting conditions.

This task is intended to be a more difficult task than the coin classification task.

The *full* database "contains 5760 single light source images of 10 subjects each seen under 576 viewing conditions (9 poses x 64 illumination conditions)". For this project, only a subset of this database was used for experimentation, consisting of 5 subjects and only a single pose (directly front on) for each subject. This constructed a dataset of 5x64 = 320 images, all similar to the image shown in figure 3.5.

For each of these images, the face was located (using coordinate information, also provided with the database) and cutout of the image. This resulted in 320 cutout images, similar to those in figure 3.1. Notice the vast range of lighting variation present in the dataset, even in the selection of cutouts presented in the figure.

Figure 3.5: An example raw image from the dataset

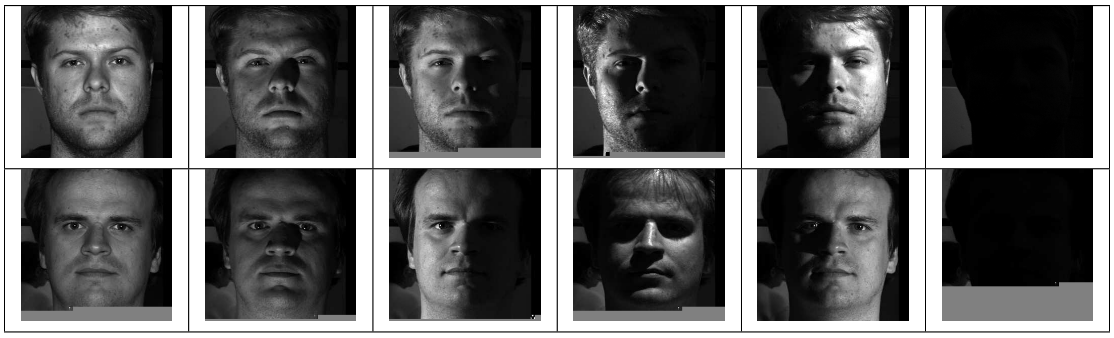

Figure 3.1: Example 300x300 cutout images in the dataset

3.3.1 Features

As in the coin classification task, several simple pixel statistic features (mean and standard deviation) were extracted from various regions of the face image. For this dataset, the regions are (also depicted in figure 3.6):

1. The image was split into quarters, and each quartile used as a region. These regions together represent the image as a whole.
2. A 300x300 square which encompasses the entire face.
3. A 50x20 rectangular region for each eye.
4. A 80x40 rectangular region for the mouth.
5. A 60x60 square region for the nose.

This results in 18 different *pixel statistic* features ($2 \times (4 + 1 + 2 + 1 + 1)$). These 18 features, extracted from the 320 images, make up the first face dataset *"Face Dataset #1"*. All features are normalised to be between the range of -1 and 1.

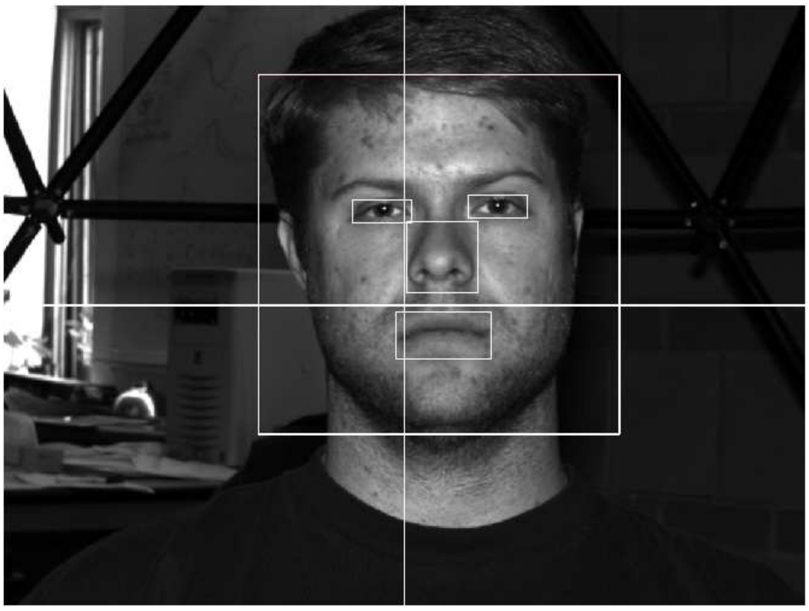

Figure 3.6: The regions from which the face features are extracted

The *Yale Face Database B* also contains coordinate positions of the eyes and mouth of the subject in the image. These positions are used to accurately obtain the rectangular regions around the eyes and mouth. The position of the nose is calculated as the centre of the triangle formed by the eyes and mouth, and the square region is taken around that point.

The inclusion of these coordinate positions brings another set of possible features to light. Those of the *distance ratios* between the eyes and mouth. All of the features used thus far are simple pixel statistic features and are easily affected by the lighting variance in the dataset of images. By including distance ratios, the features should be more robust and resistant to lighting variance.

To investigate this conjecture, another set of features is extracted from the images consisting of the 18 features in the first dataset, and including 2 additional features:

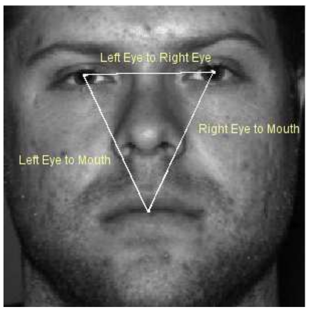

Figure 3.7: The distances from which the ratio features are extracted

- The ratio of: distance of left eye to right eye, and distance of left eye to mouth.
- The ratio of: distance of left eye to right eye, and distance of right eye to mouth.

This makes a new total of 20 features. These features, extracted from the 320 images, make up the second face dataset used in this project *"Face Dataset #2"*. Again all features are normalised to be between the range of -1 and 1.

Feature	Face Dataset #1	Face Dataset #2
Mean/SD of First Quartile	×	×
Mean/SD of Second Quartile	×	×
Mean/SD of Third Quartile	×	×
Mean/SD of Fourth Quartile	×	×
Mean/SD of Face	×	×
Mean/SD of Left Eye	×	×
Mean/SD of Right Eye	×	×
Mean/SD of Mouth Eye	×	×
Mean/SD of Nose Eye	×	×
Ratio of L/Eye-R/Eye to L/Eye-Mouth		×
Ratio of L/Eye-R/Eye to R/Eye-Mouth		×

Table 3.4: Summary of features for both face datasets

3.3.2 Terminal and Function Sets

The terminal set consists of randomly generated *numerically valued terminals* as well as *feature terminals*. The numeric features are simply n floating point numbers generated in the range of [-1, 1] using a uniform distribution random number generator ($r_0, r_1, ..., r_n$). In this classification task, the feature terminals ($f_0, ..., f_{19}$) represent the 20 *pixel statistic* features that were extracted from the face images.

$$\textit{Terminal Set} = \{r_0, r_1, ..., r_n, f_0, ..., f_{19}\}$$

$$\textit{Function Set} = \{+, -, *, /, \texttt{if<0}\}$$

3.3.3 Fitness Function

The fitness of each program is determined by the classification accuracy of the program on the given *training* set of images. Obviously, a higher fitness is desirable in this task with the maximum being 100% accuracy where all of the object patterns are correctly classified.

$$\textit{fitness} = \frac{\textit{no. objects correctly classified}}{\textit{total no. objects}} \tag{3.3}$$

3.3.4 Classification Strategy

This task consists of 5 different classes, and so using the static-range selection method (again, with a slot size of 0.25), the class boundaries are defined as follows:

$$class = \begin{cases} subject1 & output < -0.75 \\ subject2 & -0.75 \leq output < -0.50 \\ subject3 & -0.25 \leq output < 0 \\ subject4 & 0 \leq output < 0.25 \\ subject5 & 0.25 \geq output \end{cases}$$

3.3.5 Termination Criteria

Unlike in the previous task, only two termination criteria were used to stop the GP system for this task, they were:

- The fitness of the program is optimal (i.e. 1.00), which denotes that the training accuracy is 100%.
- The 50 generation limit is surpassed.

The *early stopping* criterion using a validation set was not used as experiments for these two datasets used a *10-fold validation* method (explained in a later section in more detail).

3.4 GP System Parameters

Table 3.5 details the system parameters used for each task experiment:

Task	Gens	Pop. Size	Mutation	Elitism	Crossover	Min. Depth	Max. Depth
Easy	50	500	30%	10%	60%	3	6
Hard	50	500	30%	10%	60%	3	8
Coins	50	500	30%	10%	60%	3	6
Face	50	500	30%	10%	60%	3	8

Table 3.5: Genetic Programming System Parameters

Because of the relative difficulty of both the *Hard regression* and *Face recognition* datasets, the maximum tree depth parameter is increased to 8. This allows for more complex programs to evolve in the system and so more accurate solutions can be evolved as well.

All of the results gathered are averages and standard deviations of *30* system runs. This is to provide more accurate results without having to run the systems for extremely long times. In the case of the faces datasets, because of the relatively low number of patterns available for each class (64 per 5 classes), a *10-fold cross validation* [14, *Kohavi 1995*] technique was used instead of the normal 30 runs. This is a validation technique that involves splitting the dataset into 10 subsets, and running the systems 10 times. Each time a different subset is used as the testing set, while the remaining 9 are used for training. This rotating of which subset is the test set provides a better accuracy in results when dealing with a low number of patterns in the dataset.

In order to draw conclusions about the results, they were analysed in terms of *Effectiveness*, *Efficiency* and in some cases *Comprehensibility*.

Effectiveness is the quality of solutions that the GP systems yield (how well they perform the specified task). In terms of GP, the fitness of the program outputted by the system as the "best program". *Efficiency* is measured in terms of *program size* (which directly affects memory usage), *generational time* (the number of generations it takes to find a suitable solution) and *execution time* (the wall-clock time it takes for the system to finish execution and output its solution). *Comprehensibility* is a subjective measure, and is an opinion on how "readable" and "understandable" the solutions from a GP system are. This is a difficult aspect of GP systems to investigate.

3.5 Other Notes

- All experiments were performed on systems with the following specifications.

System Details	
Computer:	Dell Optiplex GX280
CPU:	Pentium IV 2.8GHz
Memory:	1024Mbytes
Disk:	80.00Gbytes

- The VGP package written by Will Smart (developed at the *School of Mathematics, Statistics & Computer Science, Victoria University of Wellington*) was used to implement and test each experiment task.

- For all of the classification datasets, it was found to be *very* important that the set of instances in the dataset had been randomised in order before splitting up the data into their respective training/validation/testing sets. Overlooking this resulted in extremely poor test set accuracies, as classes of objects present in the test set were not present *at all* in the training set after the dataset was split. This led to programs largely overfitting only a subset of the classes.

  In figure 3.8, the original un-randomised dataset has been split into a training set (3/5 proportion) and testing set (2/5 proportion). It is easy to see that since the training set does not include *any* of the classes in the test set (and vice versa), that resulting test accuracies will be very poor.

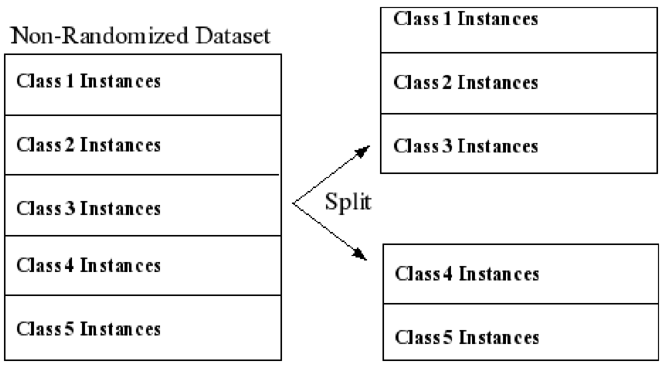

Figure 3.8: Poor dataset preparation

Chapter 4

Algebraic Simplification

In standard GP, the programs are represented as a LISP-S (or similar language) expression, which is stored in a tree representation. The standard operators and terminals used are the four arithmetic operators ($+, -, \times, \div$), an *if* operator and numeric and variable terminals. In other words, the standard GP program looks a lot like an algebraic expression. It is a straightforward idea that algebraic simplification rules can be applied to these programs in order to obtain a smaller program of equal fitness (i.e. provides the same output given the same set of inputs). This section looks into the use of such a system.

4.1 Simplification Functions

The algebraic simplification system in this project draws inspiration from STRIPS operators ([8, *Fikes, Nilsson 1971*]). STRIPS is a formal language for defining planning problems, and the operators represent the set of possible operators which include preconditions (what must be satisfied in order to do the action) and postconditions (what must be satisfied after the action is executed). One can imagine simplification as a planning problem of getting from an unsimplified expression (initial state) to a simplified expression (goal state). The only difference is that in simplification, one does not know when an optimal goal state has been reached (a form of "blind" search).

Hence the simplification rules are described in the system by a *precondition* and an *effect*. The precondition represents the state of the local nodes that must be present in order to be able to simplify, and the effect represents the changes made to the tree to obtain the simplified form. For example:

Simple Constants Operator
Precondition: All child nodes are constant terminals
Effect: Apply operator and replace subtree with single constant terminal

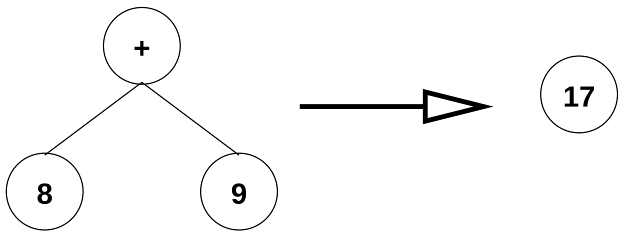

Figure 4.1: Example simplification rule definition

Multiple rules of this form make up the simplification system, along with a basic "greedy-search" engine that applies these rules. A complete list of the rules used in the simplification system is available at the end of this document (appendix A). This list does not cover all possible algebraic simplification rules, but it does cover the major sources of redundancy and is sufficiently effective for use in this project.

The ″greedy″ engine is a simple recursive algorithm. It recursively travels through the tree from the bottom-up checking the precondition for each simplification rule. If a rule matches, it is applied (without regarding the outcome). If *none* of the rules can be applied at a node the algorithm moves to the parent node.

```
procedure AlgSimplify (Node root)
  for each Child Node C
    AlgSimplify(C)

  for each Rule R
    if precondition of R is true
      apply R to C
```

Figure 4.2: Pseudo-code of the simplification algorithm

This means that each node is only visited once, and all simplification rules only look at a static, limited area, which means deeper levels of simplification (simplification of terms which are not neighbouring) are not supported in this algorithm.

4.2 Algebraic Equivalence and A Hashing Approach

A major component of any simplification system is that of expression equivalency or algebraic equivalency. Knowing that two non-identical expressions are in fact representative of the same expression is very useful and increases the number of simplifiable expressions for the system. There are several approaches to determining equivalence, including: reduction of expressions to their *canonical* forms for straight comparison, and straight-forward evaluation and comparison at multiple arbitrary points (e.g. evaluating both expressions for a range of points between -10 and 10 and comparing values).

In this project, hashing is used to address algebraic equivalence. The hashing function is used to extend the algebraic system outlined in the previous section (to be capable of simplifying more expressions). [20, *Martin*] outlines a method for achieving algebraic equivalence using hashing methods, and the algebraic equivalence method used here is a modified/extended version of that work (in order to cope with *all* common GP system functions and terminals).

A hashing function can be viewed as a mapping from a source set to a target set. In this case it is a mapping of LISP-S expressions to a set of sequential integers. In order to be an effective hash function, it should be able to guarantee the following condition:

If two expressions are equivalent, they hash to the same value.
That is, for a hashing function $f : Expression \to HashValue$, $x = y \to f(x) = f(y)$

Notice that this does not guarantee that if two expressions hash to the same value, that they are equivalent. This is of course unavoidable using this approach, as the hash function maps an infinite set of expressions $\mathcal{S}$ to a finite set of integer values $\mathbb{Z}_p$. Each hash value then, represents one (or many) *equivalence classes* of LISP-S expressions:

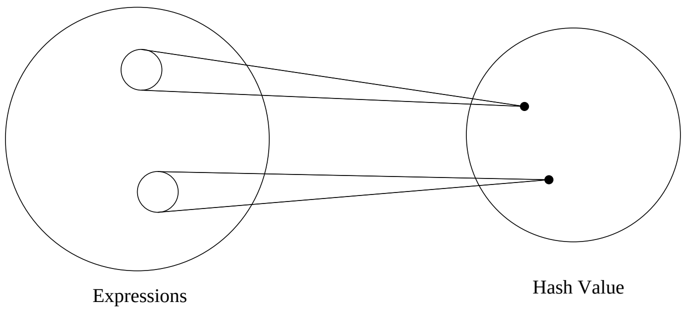

Figure 4.3: Diagram of an expression hash function

The inclusion of a heuristic to determine algebraic equivalence adds in the risk of two *non-equivalent* subexpressions being determined as equal and one or both being discarded (depending on the simplification rule). In this project, the hashing function uses properties of a finite field (as defined in group/ring theory) in order to accomplish the above condition. It uses a very large number of distinct hash values (i.e. field *order*) to ensure that the number of *collisions* is kept minimal, and is probabilistically minute. This ensures that programs which are simplified (probabilistically) yield the same outputs as the unsimplified program, given the same inputs. In this report, *p* is used to denote the *order* of the field used by the hash function (i.e. the total number of possible hash values). In order to qualify as a field, *p* must be a prime number.

4.3 Hashing of Basic GP Functions and Terminals

Each of the following sections details each type of terminal/function in a basic GP system and how they are handled.

4.3.1 Constants

In a GP system, constants can be any numeric type: integers, rationals, floating point, etc. Therefore, the hash function needs to be designed to handle all these types, of which the most difficult is floating point. [20, *Martin*] does not describe a solution to this in his paper. In this project, we handle this by approximating the floating point with a rational number, thus converting it to a division of two integers.

Calculating accurate and minimal rationals can be very time consuming, so a quick approximation is used. The numerator is formed by multiplying the floating point by a predefined precision constant (e.g. 1000000) and truncating the leftover fractional part. By using the same precision constant as a denominator, a rational representation can be very quickly found.

$$Hash(A) = \frac{A \times precision}{precision} \bmod p$$

4.3.2 Variables/Features

In a GP system, a variable/feature can represent *anything*, may it be image features, pixel statistics or even other LISP-S expressions. The *important* attribute is that in any GP system, a variable/feature always represents the same value (i.e. if, for a single particular object pattern $f0 = 7$, then for any program acting on that same pattern, $f0 = 7$). Keeping this in mind, variables/features are assigned random hash values at the beginning of the GP run and are kept consistent through till termination.

$$Hash(Feature_n) = \text{a random value in } \mathbb{Z}_p$$

4.3.3 The Arithmetic Operators

Because the hashing method takes place in a finite field, all of the standard arithmetic methods are easily handled using *modulo arithmetic*. Hashing of these operators is equivalent to evaluating them within the field:

$$\begin{aligned} Hash(A+B) &= (A+B) \bmod p \\ Hash(A-B) &= (A-B) \bmod p \\ Hash(A \times B) &= (A \times B) \bmod p \\ Hash(A \div B) &= (A \div B) \bmod p \end{aligned}$$

4.3.4 The 'if' operator

The *IF* function is a more difficult case, as it is *not* an arithmetic function and so cannot simply be converted to its *modulo arithmetic* equivalent. Additionally, it consists of *three* parameters: a condition, a true branch and a false branch. All three of these parameters must be considered when hashing this function as well as the order in which they appear. The following approach was formulated to handle this function:

$$Hash(IF(A,B,C)) = (A \times \tfrac{C}{B}) \bmod p$$

This uses division to take into account the position of the three parameters (condition, true branch, false branch). Interestingly, it also can handle the following two equivalent expressions:

```
if<0(A, B, if<0(C, B, D)) and
if<0(C, B, if<0(A, B, D))
```

This can be shown by assigning the variables random values as the hashing method describes and evaluating: Let $A = 4, B = 8, C = 9, D = 2$ and let the hash values be in $\mathbb{Z}_{11}$:

```
if<0(A, B, if<0(C, B, D)) = 4 * ((9 * (8 / 2) / 8)) = 7
if<0(C, B, if<0(A, B, D)) = 9 * ((4 * (8 / 2) / 8)) = 7
```

The two expressions used above are the same as those used in [40, *Zhang, 2004*] as an example of the difficulty in hashing the *IF* function.

4.3.5 Operator Closure

All of the functions supported are closed, meaning that for any of the functions $\diamond \in \{+, -, \times, \div, IF\}$, $Hash(A) \diamond Hash(B) = Hash(A \diamond B)$. More specifically:

$$\begin{aligned}
&Hash(A + B) = Hash(A) + Hash(B)\\
&Hash(A - B) = Hash(A) - Hash(B)\\
&Hash(A \times B) = Hash(A) \times Hash(B)\\
&Hash(A \div B) = Hash(A) \div Hash(B)\\
&Hash(IF(A + B, C, D)) = Hash(A + B) \times \tfrac{Hash(D)}{Hash(C)}
\end{aligned}$$

This means that by storing already calculated hash values within the tree node structure, one does not need to recalculate the hash values of subtrees each time a tree is to be hashed, as hash values of subtrees can be combined to give correct hash values of the whole tree. This property reduces the time complexity for using this approach to a linear one ($O(n)$).

4.3.6 Examples

Here are a couple of examples of how the algebraic equivalence hashing works. The first is a diagram of a simple program being hashed. The second is an example of hashing a more complicated ″real world″ evolved program.

Example 1

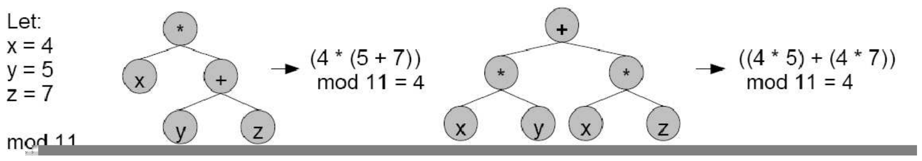

Figure 4.1: Example 1 of algebraic equivalence

Example 2

Original Program

```
(+ (- (* (+ f0 (* -0.018733 f0)) -5.068436) (if<0 f0 f0 (% 0.988933 f0)))
  (* (* (if<0 f0 f0 0.578400) f0) f0))
```

Program with Hash Value Substituted

```
= (+ (- (* (+ 4 (* 735 4)) 1008) (* 4 (% (% 61 4) 4))) (* (* (* 4 (%
    2 4)) 4) 4))
= (+ (- (* 1167 1008) (* 4 16)) (* (* (* 4 889) 4) 4))
= (+ (- 1739 64) 42)
= (+ 1675 42)
= 1717
```

Let `f0 = 4` (randomly chosen), field order $p = 1777$ (semi-large prime number) and constant precision = 1000000. The constants (`-0.018733`, `-5.068436`, `0.988933`, `0.578400`) are each converted to rational numbers ($\frac{18733}{1000000}$, $\frac{5068436}{1000000}$, $\frac{988933}{1000000}$, $\frac{578400}{1000000}$) and evaluated *mod* 1777 to obtain their hash value representations (`-0.018733 = 735`, `-5.068436 = 1008`, `0.988933 = 61`, `0.578400 = 2`). The `if<0` operator is also changed into its hashing function.

This process yields the following program (with hash values substituted in), which is evaluated to the final hash value representing the entire program.

As the program hashes to `1717`, this program is deemed *algebraically equivalent* to any other program that hashes to `1717`, even to the numerical constant `1717` itself. The key to this approach is to use a large field order, so that collisions are kept minimal.

4.4 Testing and Results

In this section, the results for each experimentation task is presented followed by a discussion of the results of *all* the tasks. Also as part of the results is a list of the 3 ″best/fittest″ programs yielded by each of the GP systems. This is a large list and so has been relegated to a section in the appendix. Any program in that list that is referred to in the discussion is restated.

In addition to the GP system parameters, the following additional ″simplification parameters″ need to be specified:

Parameter	Value
Field Order (for Alg. Simp.)	1000077157
Constant Precision	10000000
Proportion	100%
Frequency	Varied: Every [0, 1, 2, 4, 6] Gens.

Table 4.1: Algebraic Simplification: Simplification Parameters

Field Order defines the number of distinct hash values that programs can be hashed to. For reasons explained earlier, this has been given a large value. *Constant Precision* is the number of decimal places that are considered when hashing a floating point number (i.e. the precision that is kept). This is used when converting floating points to a rational number ″equivalent″. *Proportion* is simple the percentage of programs which are simplified when the simplification system is invoked. In this set of experiments, this is set to *all* programs. *Frequency* decides *when* simplification should take place. In this set of experiments, this was varied to help investigate the effect of frequency on simplification results. Five different values for frequency were tested, obtaining results for systems: *without* simplification, simplification every generation, every 2nd generation, every 4th generation, and every 6th generation.

These parameters govern the behaviour and reliability of the algebraic simplification systems.

The results table contains averages and standard deviations of the following:

- *Final Generation* - The generation at which the system terminates with its final solution.
- *Final Best Fitness* - The fitness of the final solution that the system outputs.
- *Time* - The time (in seconds) that the system takes to train/evolve its final solution.
- *Average Program Size* - The average size (number of nodes) of the programs in the system, including *all* programs from *all* generations.
- *Average Program Fitness* - The average fitness (as determined by the fitness function) of the programs in the system, including *all* programs from *all* generations.

The best case for each of the above is also highlighted in each column. Graphs of the ″Average Program Size″ and ″Final Best Fitness″ over generational time are also presented.

Note: The ″Final Best Fitness″ graphs have had their y-range adjusted to make the separations between lines more clear.

4.4.1 Easy Regression

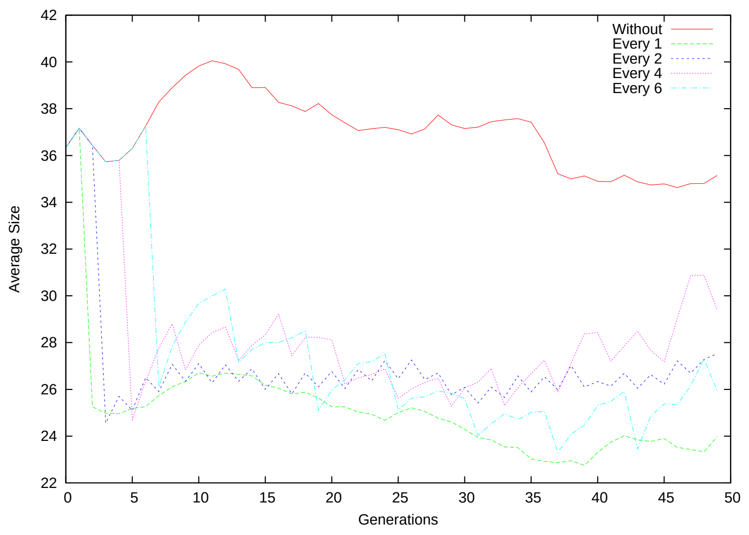

Figure 4.4: Easy Symbolic Regression - Average Program Size per Generation

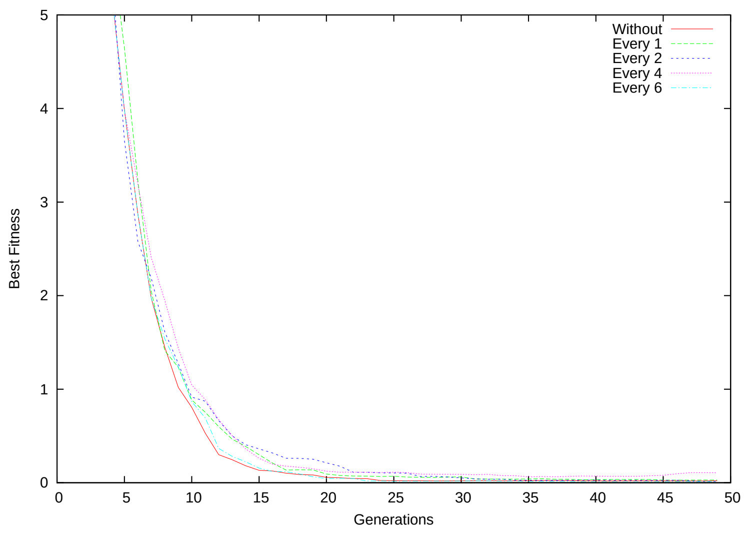

Figure 4.5: Easy Symbolic Regression - Fitness of Best Program per Generation

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	**28.781 ± 13.427**	0.005 ± 0.013	1.221 ± 0.501	37.611 ± 5.634	421807.87 ± 207935.21
Every 1	32.438 ± 13.119	0.011 ± 0.042	1.232 ± 0.464	**25.606 ± 2.937**	373212.90 ± 147767.37
Every 2	31.562 ± 14.291	0.005 ± 0.013	1.109 ± 0.486	27.232 ± 3.667	401389.21 ± 188242.12
Every 4	31.094 ± 13.359	0.027 ± 0.119	1.071 ± 0.494	28.412 ± 5.074	397186.03 ± 182768.07
Every 6	31.688 ± 12.228	**0.003 ± 0.009**	**1.070 ± 0.359**	29.200 ± 4.581	**367574.34 ± 127265.58**

Table 4.2: Easy Symbolic Regression - Results

For this particular task, the *final generation* for each of the systems using simplification was higher than the standard GP system. But, the *execution times* for the systems with simplification were lower, with only simplification at *every generation* having a higher execution time.

The average program size (over all generations) was significantly reduced when using simplification, with the largest reduction when using simplification at every generation. This is also evidenced in the first graph, which shows the average program size during the GP system runs. The first time simplification is invoked, a large drop in program size can be seen.

Best fitness showed a comparable fitness at every 2nd generation and an *improvement* in fitness when applied every 6 generations.

As shown in the best fitness graph, none of the systems were continuously fitter than other other systems. So which system yields the "best" result seems to be dependent on when the system stops executing.

4.4.2 Hard Regression

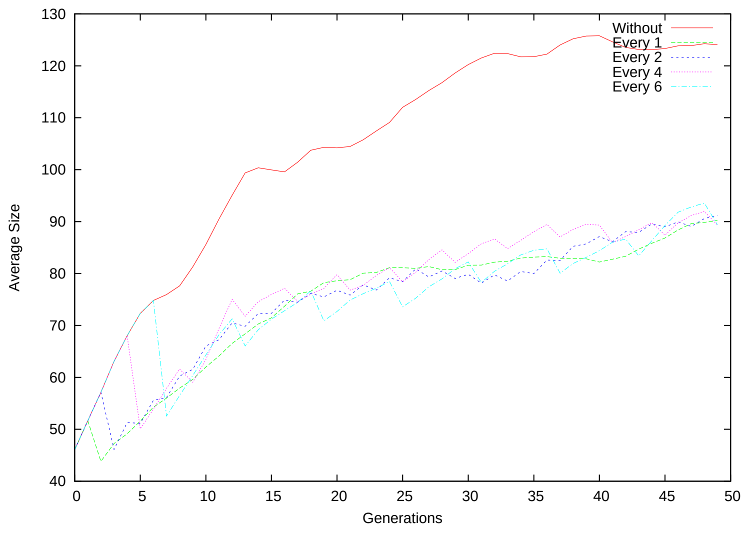

Figure 4.6: Hard Symbolic Regression - Average Program Size per Generation

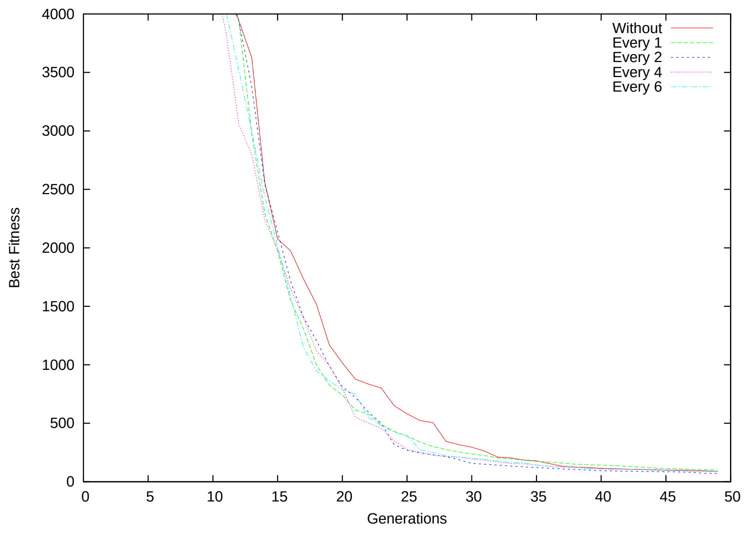

Figure 4.7: Hard Symbolic Regression - Fitness of Best Program per Generation

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	44.875 ± 4.756	83.774 ± 75.283	5.141 ± 1.019	104.436 ± 22.171	**414357.68** $\pm$ **87175.63**
Every 1	44.875 ± 4.756	92.884 ± 80.624	5.206 ± 0.861	**74.362** $\pm$ **13.642**	465193.18 ± 76491.16
Every 2	44.875 ± 4.756	**67.346** $\pm$ **59.315**	4.270 ± 0.759	74.841 ± 13.886	439192.59 ± 76727.60
Every 4	44.875 ± 4.756	82.471 ± 85.606	4.152 ± 1.069	77.337 ± 21.487	467364.06 ± 114495.28
Every 6	44.875 ± 4.756	85.301 ± 93.883	**3.989** $\pm$ **0.627**	75.549 ± 12.840	477070.71 ± 108510.25

Table 4.3: Hard Symbolic Regression - Results

In this task, the *final generation* for each of the systems was identical. The *execution times* for the systems with simplification were all lower, again with only simplification at *every generation* having a higher execution time.

The average program size was again significantly reduced when using simplification.

Best fitness showed a comparable fitness at every 4 generations and an *improvement* in fitness is apparent when applied every 2 generations.

4.4.3 Coin Dataset

For classification tasks, *Final Best Fitness* in the results table is replaced by *Final Best Accuracy*, which is the final solutions accuracy in classifying instances in the *test* set. This differs from regression as, in regression the training and test sets are identical. Whereas in classification, training and test sets are kept dissimilar in order to prevent *overfitting*.

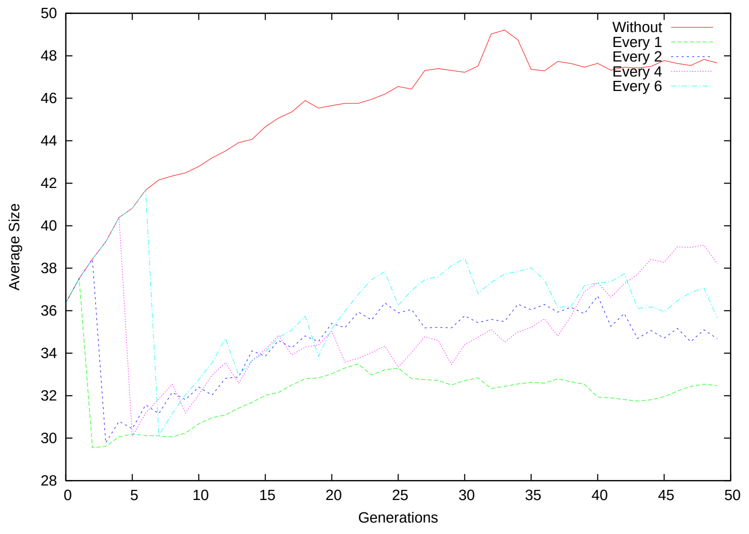

Figure 4.8: Coins Dataset - Average Program Size per Generation

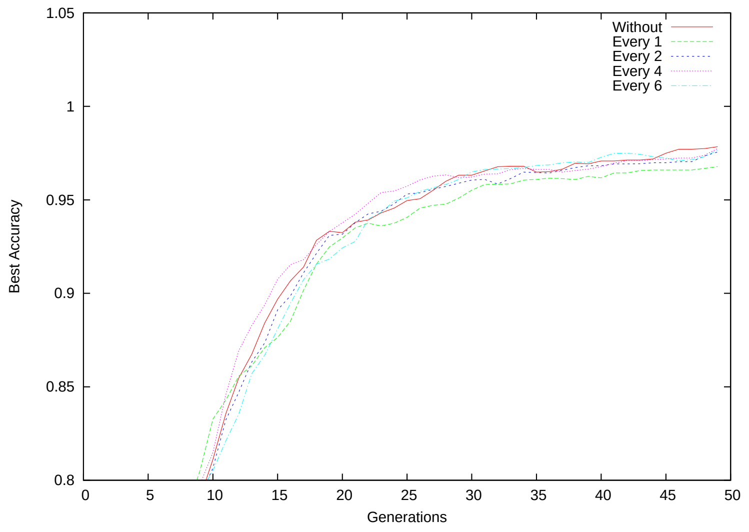

Figure 4.9: Coins Dataset - Fitness of Best Program per Generation

	Final Gen	Final Best Acc	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	35.750 ± 11.200	0.973 ± 0.025	1.657 ± 0.532	44.476 ± 7.302	**0.530** ± **0.066**
Every 1	37.469 ± 10.992	0.964 ± 0.039	1.700 ± 0.452	**32.539** ± **5.622**	0.515 ± 0.063
Every 2	**35.031** ± **11.290**	**0.974** ± **0.028**	1.492 ± 0.407	34.720 ± 4.253	0.516 ± 0.055
Every 4	36.656 ± 10.527	0.974 ± 0.032	**1.477** ± **0.411**	34.884 ± 5.264	0.521 ± 0.055
Every 6	37.250 ± 10.336	0.954 ± 0.054	1.522 ± 0.355	36.566 ± 3.919	0.529 ± 0.052

Table 4.4: Coin Dataset - Results

In this task, the *final generation* was optimal when simplification was applied at every 2 generations, but at other simplification frequencies was slightly higher. The *execution times* for the systems again exhibited the same behaviour as the above two tasks, with all simplification frequencies apart from 'every generation' having lower times.

The average program size was again significantly reduced when using simplification.

Best fitness was slightly improved when simplification was at every 4 generations *and* when simplification was performed at every 2 generations.

4.4.4 Face Datasets

Because of the relatively *low* number of patterns available for each class (64 for 5 classes), the experiments on this dataset used a *10-fold cross validation* method to obtain more accurate results.

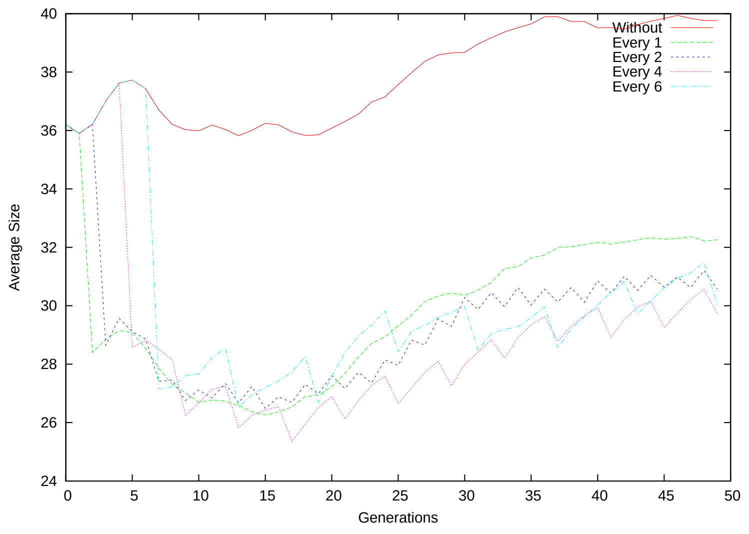

Figure 4.10: Faces Dataset #1 - Average Program Size per Generation

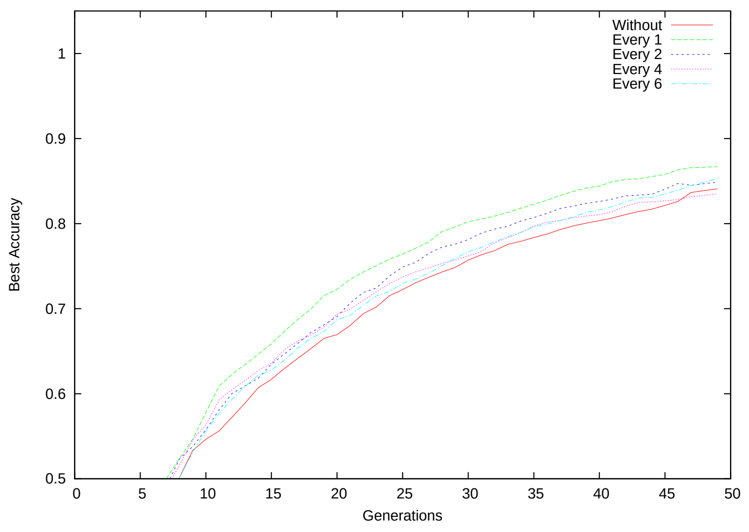

Figure 4.11: Faces Dataset #1 - Fitness of Best Program per Generation

	Final Gen	Final Best Acc	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	46.077 ± 3.578	0.855 ± 0.117	2.646 ± 0.578	37.861 ± 8.755	**0.427 ± 0.052**
Every 1	**45.712 ± 4.415**	**0.876 ± 0.104**	2.622 ± 0.583	29.798 ± 6.571	0.427 ± 0.057
Every 2	45.885 ± 3.717	0.867 ± 0.117	2.367 ± 0.460	29.364 ± 5.966	0.423 ± 0.050
Every 4	46.077 ± 3.578	0.851 ± 0.105	**2.251 ± 0.441**	**28.917 ± 5.757**	0.413 ± 0.054
Every 6	46.077 ± 3.578	0.866 ± 0.075	2.288 ± 0.464	30.081 ± 6.274	0.421 ± 0.051

Table 4.5: Faces Dataset #1 - Results

In this task, the *final generation* was optimal when simplification was applied at every generation. The *execution times* slightly deviated from the trend in the other tasks, with lower times for all frequencies used.

The average program size was again significantly reduced when using simplification.

Best fitness was generally higher when using simplification, with higher fitnesses for all frequencies except every 4 generations.

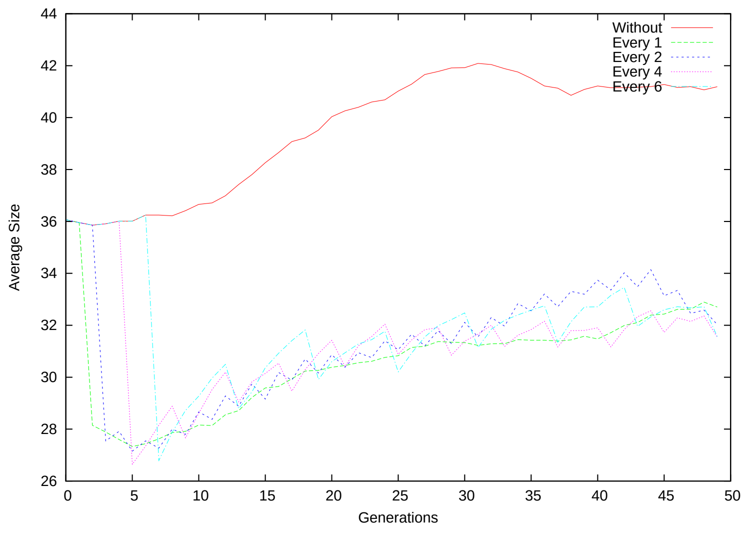

Figure 4.12: Faces Dataset #2 - Average Program Size per Generation

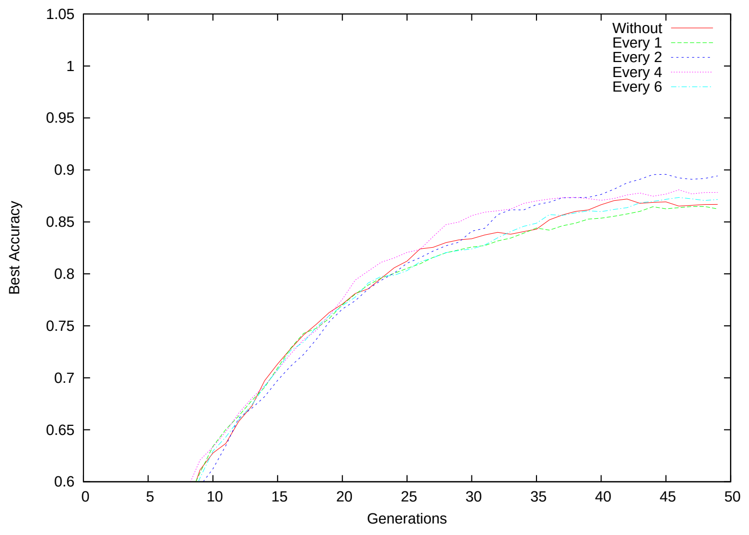

Figure 4.13: Faces Dataset #2 - Fitness of Best Program per Generation

	Final Gen	Final Best Acc	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	44.481 ± 5.442	0.897 ± 0.097	2.681 ± 0.451	39.566 ± 5.905	$\mathbf{0.430 \pm 0.056}$
Every 1	44.365 ± 6.235	0.889 ± 0.093	2.714 ± 0.548	$\mathbf{30.653 \pm 5.734}$	0.421 ± 0.053
Every 2	$\mathbf{42.865 \pm 7.905}$	$\mathbf{0.928 \pm 0.089}$	2.422 ± 0.487	31.230 ± 4.416	0.425 ± 0.050
Every 4	43.692 ± 6.244	0.917 ± 0.086	$\mathbf{2.332 \pm 0.419}$	31.356 ± 4.738	0.428 ± 0.056
Every 6	44.635 ± 5.356	0.917 ± 0.088	2.371 ± 0.363	31.992 ± 4.252	0.427 ± 0.044

Table 4.6: Faces Dataset #2 - Results

In this final task, the *final generation* was optimal when simplification was applied at every 2 generations. The *execution times* were back in line with the first 3 tasks, with lower times for all frequencies apart from every generation.

The average program size was again significantly reduced when using simplification.

Best fitness was generally higher when using simplification, with higher fitnesses for all frequencies but every generation.

4.5 Discussion of Results

4.5.1 Effectiveness

The results gathered show that using simplification on these tasks can obtain comparable or even superior results to the system without simplification. In all but one dataset (faces dataset #1), applying simplification at every generation led to a small loss in fitness. This suggests that in general, simplification should not be done at every generation. Each of the results gathered (even without simplification) fall within a standard deviation of one another. Due to the unpredictable nature of genetic programming, a substantial variance/standard deviation is to be expected. This makes comparisons between system results hard to derive in some cases without resorting to using only direct *mean* comparisons.

GP systems using simplification generally performed better than the standard GP system on both of the face datasets. This suggests that this simplification method is particularly effective on relatively difficult tasks.

The frequency that yielded, on average, the fittest solutions varied across the experimentation tasks. But simplifying every 2 generations shows itself on more than one occasion to be a good starting point when determining the frequency parameter (every 2 was optimal for: hard regression, coins and faces #2). Frequency presents itself, like most GP parameters, to be *task dependent* and requires multiple experiments to find an optimal value for a specific task.

4.5.2 Efficiency

In all five of the tasks, the average size of each program (in terms of the number of nodes each program contains) is significantly reduced. This can be seen in the results tables as well as in the graphs. The graphs also show the *periodic effect* of simplification on the average program size. Program growth occurs in-between the generations where simplification takes place before simplification induces a steep reduction in size. Not surprisingly, performing simplification at lower frequencies results in a higher average program size (than performing at every generation). But this increase in size is very small, showing that simplification need not be performed at every generation in order to have a big effect of the average size of programs.

As the number of nodes used in the system is reduced (correlated to the average program size), this result also means that the required *memory usage* for a system with simplification is less than that of a standard GP system.

In terms of *real/wall-clock* time, the systems using simplification generally take less time due to the fact that the system has to process *smaller* programs in each generation. The exception to this is simplification at every generation, which in all cases but the *Face Dataset #1* led to a very small increase in execution time. This can be attributed to the overhead introduced into the system by the simplification component. This overhead, when occurring at every generation, outweighs the time saved from processing smaller programs and so overall execution time increases.

In terms of *generational* time (i.e. the number of generations that the system goes through before terminating with a solution), all of the GP systems were relatively the same, with only savings of 1 or 2 generations possible (if at all).

4.5.3 Comprehensibility

In a symbolic regression task, where the goal is the simply extract the original mathematical equation from the set of given data-points. A simplified expression is usually the easiest to comprehend. For the *easy* symbolic regression task, the difference in the solutions obtained is quite apparent.

To give contrast on the difference in the solutions given by the system, here is a program from the system *without simplification*:

```
Every 0
(+ (if<0 (+ (- f0 (+ 0.409400 f0)) -0.553533) 0.762133 (* (+ (% -0.274267 f0)
 (+ f0 f0)) (if<0 (- f0 f0) (+ 0.691800 0.587400) (+ f0 -0.794400)))) (+ (if<0
 (* f0 -0.553533) (if<0 (+ 0.533533 0.807933) (- 0.587400 -0.601733) f0) (- f0
 0.056600)) (+ (* f0 f0) (% -0.362467 (if<0 0.845267 f0 -0.108933))))),0.00013
```

It is difficult to compare this to the original equation ('$x^2 + x + 4.5$') and realise that they are indeed representative of the same set of data. Inspection reveals two `(+ f0 f0)` sub-expressions and a single `(- f0 f0)` sub-expression, which accounts for the 'x^2' in the original equation. The presence of multiple unnecessary conditional `if<0` statements adds to the possible confusion when interpreting the information, and suggest that the original equation was a piece-wise one (which it is not).

On the other hand, this is a program with similar fitness (`0.00013` vs. `0.000112`) from the system with simplification every 4 generations:

```
Every 4
(+ 0.291074 (+ (+ 0.536400 f0) (- (* f0 f0) -3.161922))),0.000112
```

While this is not the *shortest* program from any of the systems, it shows a solution with similar fitness which is indeed much shorter and easier to comprehend. Comparing it to the original equation ($x^2 + x + 4.5$), there is clearly a x^2 term (`(* f0 f0)`) and an x term (`f0`) present. It is also easy to see that the constants combine to be close to the 4.5 in the original equation. This program is much clearer for humans to interpret, as well as being more efficient for a computer system to interpret. Clearly this is program is a "solution" to the easy regression problem, and so is easy to comprehend.

For a classification task, understanding why a program performs well is less trivial. It requires knowledge of the features the program is using as well as the effect of performing operations on those features.

Firstly, take a program from the *top 3* programs in not applying simplification. This program in its entirety is:

```
Every 0
(- (% (if<0 (if<0 f2 (+ 0.069133 0.188200) (if<0 f5 0.261400 0.316067))
 0.755667 (* -0.180333 (- f1 0.261400))) 0.904267) (if<0 (* (- (* f6
 0.224133) (if<0 -0.356267 0.076467 -0.383400)) 0.224133) -0.180333
 0.418067)),0.992188
```

This program relies on 4 features from the dataset: f1, f2, f5, f6, which encompass a single feature from each concentric square region. The expression can be inspectively split into two sections: The first being (% (if<0 (if<0 f2 (+ 0.069133 0.188200) (if<0 f5 0.261400 0.316067)) 0.755667 (* -0.180333 (- f1 0.261400))) 0.904267) and the second being (if<0 (* (- (* f6 0.224133) (if<0 -0.356267 0.076467 -0.383400)) 0.224133) -0.180333 0.418067).

The f6 feature (which corresponds to the mean of the outer most square) used in the second section appears to distinguish between *10c* and *5c* objects. The (- ... at the very beginning will either take away -0.180333 or -0.418067 depending on f6. From the raw dataset (before normalisation), the value of f6 for 5c is usually around 90 and for 10c is usually around 50, with an average or around 70. This means that when normalised to [-1,1], f6 will carry negative values for 10c and positive values for 5c. If negative, the program will minus -0.180333, which is actually adding 0.180333, pushing the final output towards the positive numbers (which carry the two 10c classes). If positive, the program will take minus 0.418067, pushing the final output towards the negative numbers (which carry the two 5c classes).

The other three features in the first section, appear to then determine when the coin is a heads or a tails object. Firstly the f2 (mean of 2nd inner square) feature is evaluated, which is usually negative for heads and positive for tails, but borderline cases do exist which makes this feature not entirely reliable. The other two features used (f1 (standard deviation of inner most square) and f5 (mean of 3rd square)) are used to handle these borderline instances. Bringing the output value into a region where subtracting the evaluated value of the second section will bring the output into the correct classification class boundary.

Now we look at a program with the same fitness from a GP system using simplification:

```
Every 6
(- (% (* -0.100933 (- f1 0.575133)) 0.571400) (if<0 (* (if<0 (if<0 f7
 f0 0.803933) -0.654600 (% f5 -0.698933)) 0.224133) -0.180333 0.418067))
,0.992188
```

The first thing one notices is that this expression is much smaller, with no redundant conditionals or operations of only constants. This reduces the amount of information needed to be processed by a human reader. Understanding this problem though still requires some thought.

Again, this program appears to split the problem into two sections. The first (% (* -0.100933 (- f1 0.575133)) 0.571400), which uses the standard deviation of the innermost square, appears to determine whether the image is of a head or a tail. The other section is much like the first section. It uses a three features from the same square regions as the first program (although not exactly the same ones) and appears to determine whether a given pattern is of a 10c or a 5c coin.

4.6 Summary

In this chapter, an algebraic simplification method was developed, which acted directly on tree-based programs. This method used basic rules similar to STRIPS operators, and covered forms of redundancy for each of the arithmetic operators ($+, -, /, *$), as well as the conditional operator (if<0). An algebraic equivalency component was also added to expand the set of expressions that could be simplified.

When applying this method to a GP system, an improvement in the performance of the system was achieved. In all tasks, comparable or better fitness solutions were able to be obtained when using simplification. Simplification also reduced program size and execution time, improving the overall efficiency of the system.

Chapter 5

Integration of Prime Number Simplification and Algebraic Equivalence

One of the problems with a rule based algebraic simplification system is that defining rules that reach beyond neighbouring nodes to an *arbitrary* depth in the tree is infeasible (one would have to cover *all* possible combinations and depths). This means that the system is restricted to only local simplifications and cannot perform deeper leveled simplification (e.g. `(- x (+ y (+ x 2))` is equivalent to `(+ y 2)`, but a localised rule would only see the `y` and + nodes and not simplify). To solve this problem, an algorithm which *can* perform deeper level simplification should be added. This comes in the form of the *PRES* algorithm.

The original PRES algorithm [40] featured a few problems in its original form. In this chapter, each problem is explained and the approach to solving it detailed. Then the entirety of the altered algorithm is outlined.

The four main problems identified were:

- *Hash Function* - The hash function used in PRES is a two-phase modulus function over the primes `47`, and `13`. This allows for a total of 611 different hash values which is considered low when one considers the number of possible expressions in a basic GP system, especially in larger and more complex programs.
- *Simplification of `if`* - `if` is not easily represented in a structure with only 2 "layers", as it is a function which takes three parameters.
- *Arithmetic Overflow* - Because the prime product quartuple is made up of the mathematical product of (possibly) many subtrees, it is easy to see that it grow exponentially, quickly outgrowing the normal `32-bit` integer space. A program `(+ (+ (+ (+ (+ (+ (+ (+ x x) x) x) x) x) x) x) x)`, which has 9 nodes to combine together into the prime product quartuple now has to store the number p^9, where p is the prime number allocated to `x`.
- *Monte Carlo Simplification* - PRES had the side effect of causing an "unstable" simplification effect (the program could be reconstructed in a different order, e.g. `(x + y)` may restructure to `(y + x)`). This could allow for new building blocks to be formed during the simplification process, and so this property should be harnessed in the algorithm. PRES originally deterministically chose the subtree, not using the "unstable" property of the algorithm.

5.1 Integrating Algebraic Equivalence

In chapter 4 a method for determining algebraic equivalence of two expressions was described. One of the advantages of PRES is its ability to equate expressions which only differ in the order the terminals appear in (e.g. `x + y` and `y + x`).

However for more complicated equivalences, where two seemingly dissimilar expressions can actually represent the same expression (e.g. $sin(x)^2 + cos(x)^2 = 1$, $5(x+1) = 5x+5$), PRES treats them as separate equations. Using the addition of algebraic equivalence hashing, these types of expressions can be simplified.

The use of a algebraic equivalence also relieves PRES from having to deal with the `if` function, which has no "inverse" but is a simplifiable expression. Recall that `if` is handled in the algebraic equivalence method, allowing for two equivalent `if` statements to be identified correctly.

5.2 Prime Product Overflow

One of the problems with building the prime product quartuple is that it tends to grow very quickly. In the original PRES algorithm, subtrees were allocated these primes by hashing the prime product over the numbers `47` and `13`, making the maximum prime number possible the 611^{th} which is `4513`.

This kept the prime factors relatively small, so overflow was not that large of an issue. With the inclusion of *algebraic equivalence* to determine the prime factor (and thus a requirement for a larger number of hash values), the overflow issue is a much larger problem.

In order to remedy this issue, an arbitrary precision integer library (The GNU Multiple Precision Arithmetic Library [10]) was used in the implementation of PRES. This allows (limited by the amount of system memory) the prime product to be built for larger and more complex programs without the risk of overflow preventing the GP system from finishing execution.

5.3 Monte Carlo Reconstruction

As PRES associates *equivalent* expressions with the same prime number, these expressions can be assumed to be equivalent expressions with the only difference being the way they are expressed. This means when reconstructing a program from the simplified prime product quartuple, if there is more than one choice of subtree in the hashtable that matches a prime number, it should not matter which subtree is chosen. This gives light to several possible ways of choosing a subtree:

- *Choosing the smallest subtree*: This is the most intuitive solution, as the goal of simplification is to create the smallest program. But as it always favours the smallest subtrees, this may aid the extinction of some building blocks.
- *Choosing a subtree purely at random*: As every program is functionally equivalent, then choosing a subtree should be as good as choosing any other subtree. This may aid in keeping a better distribution of building blocks.
- *Choosing a subtree at random, where the chances of being chosen is proportional to the size of the subtree*: This is a middle ground between the previous two. This results in all subtrees having a chance of being chosen, but through probability will favour the smaller subtrees. This is the method used in the implementation in this project.

The original PRES algorithm deterministically chose the first available subtree in the hashtable, which doesn't use the potential of *unstable* restructuring. For example: if `( - (+ a b) c)` restructures to `(+ (- a c) b))`, then a new `(- a c)` building block is introduced which may be useful for the particular task.

5.4 The Algorithm

The modified algorithm is detailed in this section. It makes use of a relatively simple problem to help illustrate the stages in the algorithm.

The following program will be used as an example to help explain the algorithm:

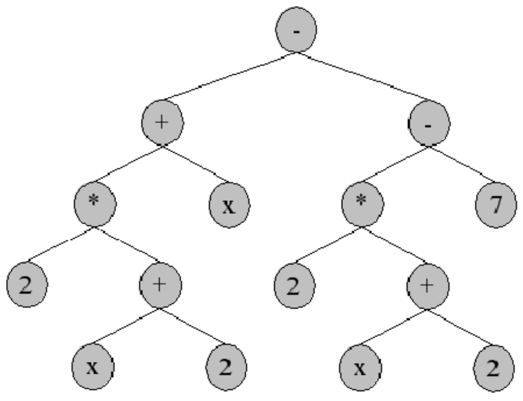

Figure 5.1: Prime Simplification: Original program tree

Also note that the prime product quartuple consists of *(top deck/numerator, bottom deck/denominator, operator family, constant)*. In this report, the prime product quartuple is referred to using only the numerator and denominator, in the form $\frac{p}{q}$.

5.4.1 Encoding

Firstly, the program tree is traversed from the bottom-up (like in the algebraic system), and the *algebraic equivalence hash* calculated. These hash values are used to allocate prime numbers to each of the leaf-nodes. For the example, suppose $Hash(2) = 2$ and $Hash(x) = 6$. It follows that the 2nd prime number is 3 and the 6th prime number is 13.

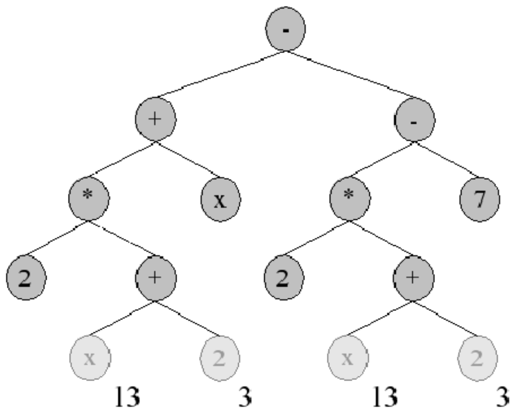

Figure 5.2: Prime Simplification: Encoding leaf-nodes

Handling of the internal nodes is trickier. The prime product quartuple is constructed in such a way that the numerator (*what Zhang describes as the "top deck"*) represents the positive side of an operator family, and the denominator (*"bottom deck"*) represents the negative side (e.g. for addition/subtraction the numerator would hold all things added and the denominator would hold all things subtracted in an expression). "Representing" simply means that the number in the numerator/denominator is a product of all the primes allocated to the nodes being encoded.

Numerically valued nodes are accumulated in the *constant* portion of the prime product quartuple. This effectively combines constants which can be combined together into a single node when the program is reconstructed.

This means that for the expression `(+ x 2)`, the resulting prime product is $\frac{3 \times 13}{1} = \frac{39}{1}$ (x and 2 on the top). While for the expression `(- x 2)`, the resulting prime product is $\frac{13}{3}$ (x on the top, 2 on the bottom). Obviously with this approach, one cannot encode all types of operators into a single prime product (i.e. encoding $\times$ and $+$).

Therefore, during the bottom-up construction, if the current node being looked at is of the same operator "family" (i.e. addition vs. subtraction, multiplication vs. division etc.) then the node is added to the current prime product quartuple. But, if there is a change in operator family then the subtree up to this point is *wrapped*, meaning the subtree is simplified using its prime product quartuple (explained in the next subsection) and then the result is allocated a new prime number to represent it in the simplification system.

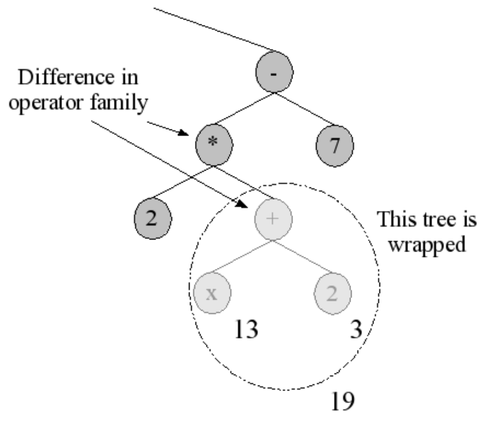

Figure 5.3: Prime Simplification: Encoding internal-nodes

In figure 5.3, the subtree `(+ x 2)` uses the operator family `+/-`, while the parent node uses the operator family `*/%`. This means the subtree is wrapped and allocated the prime `19` (as `Hash((+ x 2)) = Hash(x) + Hash(2) = 2 + 6 = 8`, and the 8^{th} prime is 19). The prime product for the subtree `(* 2 (+ x 2))` is now $\frac{3 \times 19}{1}$ as it uses the newly wrapped subtree.

Each subtree that is wrapped or otherwise encoded as a prime number is stored in a hashtable so that it can be later retrieved for program reconstruction. The prime number is simply modulo hashed to fit it within the tables bounds, and the subtree is stored in that location in the hashtable:

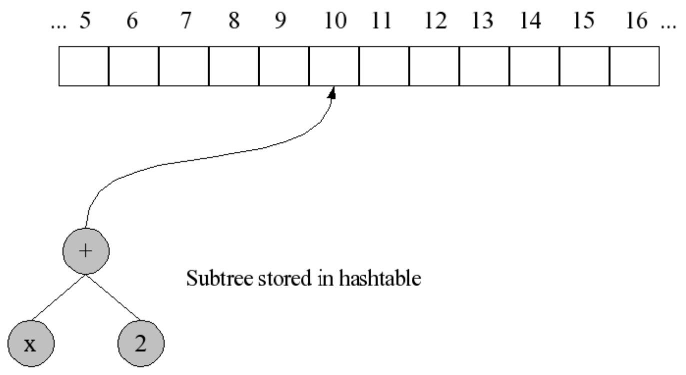

Figure 5.4: Prime Simplification: Storing in hashtable

Figure 5.5 shows the fully encoded program tree. The final prime product constructed for the whole program is: $\frac{11713}{53}$.

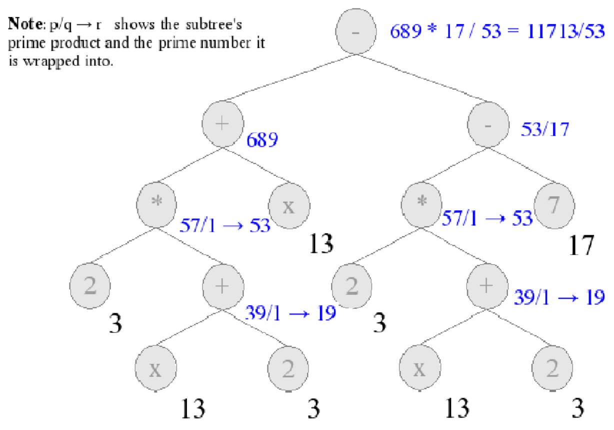

Figure 5.5: Prime Simplification: Fully encoded program tree

5.4.2 Simplification

Once the prime product quartuple has been fully calculated for the whole program tree, it is time to perform the simplification step. This is a *very* simple procedure which consists of factoring out the *Greatest Common Divisor* for the numerator and denominator. Let the prime product be denoted in the form $\frac{p_{old}}{q_{old}}$. Then simplification is:

$$\text{let } d = gcd(p_{old}, q_{old})$$
$$p_{new} = \frac{p_{old}}{d}, q_{new} = \frac{q_{old}}{d}$$

$\frac{p_{new}}{q_{new}}$ form the new prime product. For the example program, the simplification is as follows:

$$\text{let } d = gcd(11713, 53) = 53$$
$$p_{new} = \frac{11713}{53}, q_{new} = \frac{53}{53}$$

Therefore the new prime product quartuple is $\frac{221}{1}$

5.4.3 Reconstruction

Now that the simplification has been performed, the original tree program structure and representation needs to be reconstructed from the new prime product quartuple. Recall that each leaf-node and wrapped subtree that has been associated with a prime number has been stored in a hashtable. In order to reconstruct, one has to obtain a sequence of prime numbers from the prime product. This is done using a trivial prime factoring algorithm ("direct search factorization" or "trial division"):

```
procedure PrimeFactor (Integer q, ListOfFactors f)
  for each prime number p up to the square root of q
    if p divides q then
      add p to f
      q = q / p
```

This algorithm simply tests all prime number up to the square root of the number to be factored. If a prime is a divisor of the number, it is factored out and added to the `ListOfFactors`. This process is repeated until no factors can be found.

This is far from the quickest factorisation algorithm (one would ideally use a quadratic sieve [3, *Boender 1995*] or even a non-deterministic algorithm such as pollard-rho), but in this algorithm the prime factors are usually ″small″ primes (in number theory terms) and so the performance loss is not extremely great.

For the example, the list of factors is:

ListOfFactors = { 13, 17 } for the numerator and
ListOfFactors = { } for the denominator (empty set).

Each prime factor in the outputted list of factors represents a subtree that makes up part of the simplified program. Each subtree is retrieved from the hashtable, and placed in the new program structure, with the subtrees from the numerator forming the left side of the tree and the subtrees from the denominator forming the right side of the tree. These two sides are joined at a new root node, which takes the value of the ″negative″ operator in the operator family (e.g. $-$, /).

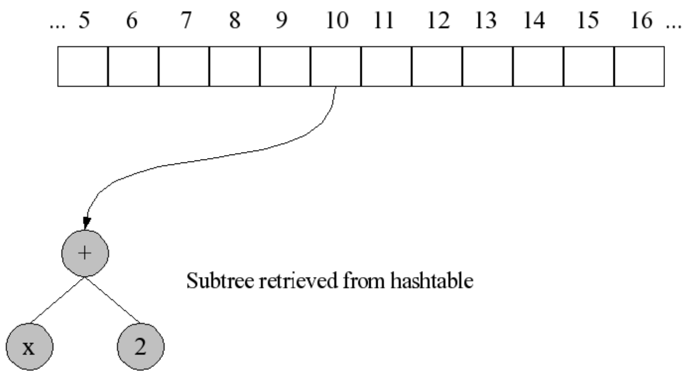

Figure 5.6: Prime Simplification: Retrieving from hashtable

For the example, the reconstructed simplified program simply constitutes of the subtrees stored in `13` and `17`. Also, as the denominator is an empty set, a new root node is unnecessary, and only the left side of the tree is left.

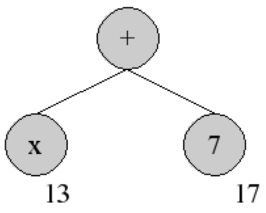

Figure 5.7: Prime Simplification: Reconstructed Simplified Program

After simplification using this algorithm is finished, the program is passed to an algebraic simplification system. This is because, while good at simplifying at multiple-levels, this algorithm is not suited for more specific types of simplification which are not easily represented in the prime product quartuple structure (e.g. combining constants). The two systems complement each others weaknesses, and together form a *Catholic* simplification system. The system comprising of both the prime number simplification algorithm and algebraic simplification algorithm is the system tested in this chapter.

5.5 Testing and Results

The same experiments were executed using the above described algorithm instead of the algebraic simplification system used in the previous chapter. In addition to the GP system parameters, the following additional "simplification parameters" need to be specified:

Parameter	Value
Field Order (for Prime Simp.)	8011
Hashtable Size	20011
Constant Precision	10000000
Proportion	100%
Frequency	Varied: Every [0, 1, 2, 4, 6] Gens.

Table 5.1: Prime Simplification: Simplification Parameters

While *Constant Precision*, *Proportion*, and *Frequency* parameters are identical to those used in chapter 4, the field order had to be drastically reduced. This is because it now has the added responsibility of determining how large the prime factors used in the system are (this needs to be kept reasonably small). As an additional parameter, the *hashtable size* has to be defined. This dictates where subtrees will be stored in the hashtable and how many different slots exist in the table. The value for this is allowed to be considerably higher than the field order, as it is not constrained by the size of the prime factors that the computer system can handle.

The tables of results are as follows (graphs did not provide any additional insight and so have been omitted):

5.5.1 Easy Regression

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	**28.781 ± 13.427**	**0.005 ± 0.013**	**1.221 ± 0.501**	37.611 ± 5.634	421807.87 ± 207935.21
Every 1	29.667 ± 12.853	0.028 ± 0.063	157.694 ± 73.785	**26.749 ± 7.056**	345666.50 ± 148756.53
Every 2	34.500 ± 11.396	0.014 ± 0.088	91.119 ± 34.463	28.521 ± 5.397	**273421.31 ± 67254.80**
Every 4	32.917 ± 12.616	0.091 ± 0.013	49.811 ± 22.965	29.714 ± 7.050	299817.90 ± 94513.05
Every 6	33.333 ± 13.852	0.049 ± 0.021	28.332 ± 15.617	30.458 ± 7.046	311773.40 ± 149803.96

Table 5.2: Easy Regression - PRES Results

In this task, the standard GP system obtained the best results in final generation, final best fitness and execution time. This is markedly different to the results obtained in chapter 4 where simplification was able to produce a system with better fitness. Smaller average program sizes are still obtained.

5.5.2 Hard Regression

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	39.667 ± 9.326	**69.014 ± 72.431**	**5.081 ± 1.126**	102.547 ± 25.519	446443.40 ± 117854.71
Every 1	39.667 ± 9.326	111.035 ± 115.985	403.450 ± 129.810	**69.629 ± 14.209**	435066.31 ± 79681.43
Every 2	**39.250 ± 9.350**	76.010 ± 198.152	227.773 ± 120.131	71.386 ± 47.681	**408031.31 ± 64702.02**
Every 4	39.667 ± 9.326	80.808 ± 44.058	126.664 ± 40.069	76.912 ± 17.285	411292.00 ± 71872.91
Every 6	39.667 ± 9.326	110.356 ± 83.934	106.896 ± 45.003	76.493 ± 25.284	414579.00 ± 73036.77

Table 5.3: Hard Regression - PRES Results

In this task, again the GP system with no simplification attains the best final fitness and execution time. The final generations for each of the systems were identical, with the exception of performing at every 2 generations, which has a minor improvement.

5.5.3 Coins Dataset

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	28.833 ± 13.099	**0.982 ± 0.024**	**5.117 ± 2.152**	43.765 ± 8.157	0.503 ± 0.084
Every 1	33.000 ± 11.083	0.977 ± 0.023	98.041 ± 46.129	36.388 ± 4.915	0.506 ± 0.038
Every 2	32.500 ± 11.264	0.973 ± 0.031	56.190 ± 20.809	**32.837 ± 5.648**	0.487 ± 0.055
Every 4	35.083 ± 11.623	0.958 ± 0.038	32.329 ± 12.341	36.305 ± 5.791	**0.522 ± 0.062**
Every 6	**24.917 ± 11.377**	0.980 ± 0.031	13.928 ± 3.935	35.179 ± 4.518	0.474 ± 0.071

Table 5.4: Coins Dataset - PRES Results

In this task, yet again the GP system with no simplification obtains better final fitness and execution time results. All simplification systems obtain worse final generation results, with the exception of performing at every 6 generations, which had trained slightly faster.

5.5.4 Face Dataset #1

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	39.667 ± 9.326	0.887 ± 0.045	**6.233 ± 1.252**	46.080 ± 8.505	**0.472 ± 0.033**
Every 1	39.667 ± 9.326	0.812 ± 0.127	125.087 ± 25.745	**29.479 ± 2.463**	0.403 ± 0.019
Every 2	39.667 ± 9.326	0.872 ± 0.051	67.629 ± 9.832	31.240 ± 3.317	0.434 ± 0.034
Every 4	39.667 ± 9.326	**0.894 ± 0.078**	38.581 ± 5.671	32.074 ± 2.118	0.440 ± 0.039
Every 6	39.667 ± 9.326	0.891 ± 0.032	25.472 ± 7.563	34.156 ± 5.827	0.447 ± 0.037

Table 5.5: Face Dataset 1 - PRES Results

This task's results are interesting, as unlike the previous three, two of the systems using simplification produce improved final fitness over the standard GP system (Every 4 and Every 6). However, the best execution time is still held by no simplification.

5.5.5 Face Dataset #2

	Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
Without	44.481 ± 5.442	0.897 ± 0.097	**5.612 ± 1.385**	39.566 ± 5.905	**0.430 ± 0.056**
Every 1	44.635 ± 4.835	0.913 ± 0.080	90.600 ± 32.412	**31.066 ± 6.706**	0.417 ± 0.040
Every 2	44.750 ± 5.108	0.924 ± 0.079	51.654 ± 16.930	31.690 ± 6.078	0.423 ± 0.046
Every 4	**43.638 ± 7.194**	**0.925 ± 0.076**	29.314 ± 9.413	32.177 ± 5.963	0.417 ± 0.058
Every 6	44.615 ± 5.486	0.894 ± 0.099	20.182 ± 6.042	32.249 ± 5.682	0.424 ± 0.067

Table 5.6: Face Dataset 2 - PRES Results

This task shows three systems with simplification obtaining improved final fitness results over the standard GP system (Every 1, Every 2 and Every 4). These results, combined with the results for *faces dataset #1*, suggest that relatively difficult tasks benefit from the application of simplification.

As always, the average program size for each simplification system is better that the standard GP system, with simplification at every generation producing the smallest programs.

5.6 Discussion of Results

5.6.1 Effectiveness

In the *Algebraic Simplification Method* in chapter 4, the solutions found by systems using this simplification system are slightly lower in fitness than the standard GP system. Interestingly, the system did not perform as well as the algebraic system in chapter 4, with lower fitnesses that the standard GP occurring more often. The exception to this is in both of the faces datasets, where simplification on the whole does well. This suggests that simplification is well suited for the face dataset tasks and should probably be used for tasks using similar features and of similar difficulty.

5.6.2 Efficiency

This is where the algorithm starts the falter, with very high memory usage and *real/wall-clock* execution times. The high memory usage is caused by a number of things: The necessity of storing subtrees of programs so that they can be reconstructed after simplification, which causes the number of nodes required to be allocated by the GP system to drastically increase. So while the number of nodes being used by programs is being reduced, this is offset by the duplication of nodes when storing them in the hashtable. The use of a arbitrary precision math library also consumes a lot of memory as the prime product quartuple is being constructed and factored. Additionally, the use of a math library instead of using quick hardware supported arithmetic instructions causes heavy slowdown in the construction and especially factoring of the prime product quartuple. This leads to times that are many times that of the standard GP system. As both of the problems can be linked to use of an arbitrary math library which is necessary to solve the problem of "arithmetic overflow", it can be said that the algorithm itself has high hardware requirements. Ideally hardware with large amounts of memory, and high precision math processing (in excess of 64-bits) such as a *grid computing* setup or *supercomputer*. Using a lower frequency of simplification is also a viable solution, as it brings down execution times to only around 3 to 4 times the standard GP execution time.

5.6.3 Comprehensibility

The results gathered in terms of comprehensibility are similar to chapter 4 and so this section is more brief. The combined simplification system performs very well in this area, due to the fact that both *deep level* and *local* simplification is performed. This leads to minimally represented expressions which reduce the amount that a person needs to understand.

For example in the easy regression problem, several solutions from the systems using simplification are very similar to the "ideal"/original equation. This contrasts with a solution from the standard GP system.

```
Every 0
(+ (% (if<0 (% -0.791133 (* 0.184067 -0.508333)) (* (- f0 f0) (* f0 f0)) (+ (+ f0
 f0) f0)) f0) (+ (+ (* f0 f0) (% (- f0 f0) (% f0 f0))) f0)),0.00016

Every 1
(+ 2.582333 (+ (* f0 f0) (+ f0 1.428980))),0.000128

Every 2
(- (+ (* f0 f0) (+ 3.186244 f0)) -0.802933),0.000117

Every 6
  (+ 4.010031 (+ (* f0 f0) f0)),0.0001
```

For the three solutions taken from systems with simplification, it is very clear that these programs closely match the original equation.

5.7 Summary

In this chapter, several improvements were made to the PRES simplification algorithm. The largest of these was the integration of an algebraic equivalence component (using the same hashing method used in chapter 4), which allowed the algorithm to identify and possibly cancel expressions which are functionally equivalent, but appear entirely dissimilar.

Another improvement made was the introduction of *Monte Carlo Reconstruction*, which added indeterminism to the reconstruction process (based on the size of the subtree). This indeterminism allows for new building blocks to be created during the simplification process.

Finally, an arbitrary math library was used for constructing the prime product quartuple. This prevented overflow problems, and allowed application of the algorithm to larger programs.

It was found that when applying this algorithm to GP systems, that while effectiveness was slightly lower on the tasks, superior fitness was obtainable on both face recognition datasets, suggesting that this method is well suited for relatively difficult tasks. Recall that this was also the case for the algebraic simplification system, so the hypothesis can be extended to: *simplification* is well suited for relatively difficult tasks.

This method was able to achieve significantly lower program sizes, which partially addresses the problem of program bloat. However, when the algorithm was implemented, it was found to be resource intensive due to the need to process very large numbers. Applying the algorithm at lower simplification frequencies allows for more reasonable execution times.

The results that have been obtained so far in this report have been gathered using simplification on *all* genetic programs in the population. As varying frequency has had a clear impact on the performance of GP systems, it can be hypothesized that applying simplification to only a *proportion* of programs in a population can be used to further improve performance of a GP system. The next chapter investigates this conjecture.

Chapter 6

Effects of Proportional Simplification

6.1 Proportion

In the previous chapters, the experimentation performed included adjusting the frequency of simplification. This adjustment led to small variations in effectiveness and efficiency, and in some cases an improvement in final fitness results. This chapter looks into the effects of varying the *proportion* of programs that are simplified each generation.

It should be noted that because of the large amount of time it takes to run GP systems with the PRES algorithm, all of the experiments in this section were performed using the simpler *algebraic simplification system* described in chapter 4.

6.2 Using Different Proportion Selection Methods

Varying the proportion introduces yet another question: "Which programs should be selected to be simplified?". In this section, three different program selection schemes are tested to see the effect of changing simplification selection criteria has on the GP system. The usual GP system parameters were used (those outlined in 3.4), but instead of varying frequency, frequency was fixed to 1 (every generation) and instead proportion was varied (in the range of [0%, 20%, 40%, 60%, 80%, 100%]). A proportion of 20% means selecting and simplifying 20% of the program population.

The three selection methods and their results are as follows:

6.2.1 Selection Method 1: Random Selection

This is the most trivial method, where randomly selected programs are selected to fill the simplification proportion. This allows every program to have equal chance in being selected for simplification regardless of their characteristics.

Task		Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
	0%	28.781 ± 13.427	**0.005 ± 0.013**	1.221 ± 0.501	37.611 ± 5.634	421807.87 ± 207935.21
	20%	**28.182 ± 14.246**	0.033 ± 0.096	**1.075 ± 0.494**	30.430 ± 4.057	432062.34 ± 209462.21
Easy	40%	30.818 ± 14.151	0.026 ± 0.095	1.158 ± 0.484	27.994 ± 3.297	402400.09 ± 207574.68
	60%	31.318 ± 12.514	0.005 ± 0.015	1.188 ± 0.463	26.758 ± 3.795	**363001.50 ± 123769.71**
	80%	30.000 ± 12.292	0.013 ± 0.034	1.163 ± 0.421	25.766 ± 2.856	376000.50 ± 127014.46
	100%	32.438 ± 13.119	0.011 ± 0.042	1.232 ± 0.464	**25.606 ± 2.937**	373212.90 ± 147767.37
	0%	44.875 ± 4.756	83.774 ± 75.283	5.141 ± 1.019	104.436 ± 22.171	**414357.68 ± 87175.63**
	20%	44.875 ± 4.756	75.327 ± 77.955	4.710 ± 1.482	87.436 ± 28.735	454719.65 ± 103974.89
Hard	40%	44.875 ± 4.756	131.793 ± 317.490	**4.272 ± 0.763**	**73.824 ± 13.699**	493562.34 ± 124712.84
	60%	44.875 ± 4.756	**65.681 ± 42.750**	4.892 ± 0.977	79.452 ± 16.697	446154.43 ± 102302.37
	80%	44.875 ± 4.756	98.556 ± 138.783	5.049 ± 1.123	77.566 ± 17.797	460109.00 ± 112727.93
	100%	44.875 ± 4.756	92.884 ± 80.624	5.206 ± 0.861	74.362 ± 13.642	465193.18 ± 76491.16
	0%	35.750 ± 11.200	**0.973 ± 0.025**	1.657 ± 0.532	44.476 ± 7.302	0.530 ± 0.066
	20%	37.719 ± 11.791	0.968 ± 0.032	**1.632 ± 0.508**	36.032 ± 5.783	0.531 ± 0.063
Coins	40%	39.406 ± 10.494	0.965 ± 0.026	1.779 ± 0.507	34.479 ± 6.039	**0.541 ± 0.068**
	60%	**34.844 ± 10.796**	0.968 ± 0.044	1.682 ± 0.489	34.666 ± 5.042	0.519 ± 0.055
	80%	36.938 ± 11.204	0.967 ± 0.040	1.787 ± 0.486	33.336 ± 5.530	0.514 ± 0.061
	100%	37.469 ± 10.992	0.964 ± 0.039	1.700 ± 0.452	**32.539 ± 5.622**	0.515 ± 0.063
	0%	46.077 ± 3.578	0.855 ± 0.117	2.639 ± 0.578	37.861 ± 8.755	**0.427 ± 0.052**
	20%	45.981 ± 3.639	0.860 ± 0.091	**2.289 ± 0.356**	29.721 ± 4.579	0.416 ± 0.041
Faces 1	40%	46.077 ± 3.578	0.849 ± 0.112	2.305 ± 0.396	28.413 ± 5.053	0.412 ± 0.056
	60%	46.077 ± 3.578	0.866 ± 0.084	2.465 ± 0.497	29.032 ± 5.993	0.420 ± 0.063
	80%	**45.692 ± 4.497**	0.872 ± 0.102	2.580 ± 0.552	29.334 ± 6.193	0.425 ± 0.054
	100%	46.077 ± 3.578	**0.876 ± 0.095**	2.646 ± 0.488	**28.262 ± 5.305**	0.421 ± 0.050
	0%	44.481 ± 5.442	0.897 ± 0.097	2.699 ± 0.451	39.566 ± 5.905	**0.430 ± 0.056**
	20%	43.962 ± 6.175	0.903 ± 0.089	**2.289 ± 0.389**	31.021 ± 4.729	0.416 ± 0.048
Faces 2	40%	44.096 ± 6.348	0.890 ± 0.096	2.333 ± 0.389	**29.982 ± 3.994**	0.416 ± 0.057
	60%	43.865 ± 7.011	0.904 ± 0.092	2.496 ± 0.473	30.544 ± 4.287	0.429 ± 0.056
	80%	**43.596 ± 6.423**	**0.913 ± 0.093**	2.645 ± 0.458	31.299 ± 4.857	0.423 ± 0.053
	100%	43.673 ± 7.067	0.906 ± 0.096	2.762 ± 0.561	31.060 ± 5.903	0.426 ± 0.057

Table 6.1: Random Selection Method - Results

This table shows that lower run times are achievable for all tasks when using simplification at lower proportions. Almost all of the lowest run times are achieved when using the lowest proportion value (20%).

Lower program sizes were most often achieved when proportion was set to 100% (easy regression, coins dataset and faces #1). While final fitnesses were mostly comparable with the standard GP system, with improved final fitness recorded at using 60% for the hard regression task, 100% for the faces #1 dataset and 80% for the faces #2 dataset.

6.2.2 Selection Method 2: Fitness Based Elitism

This selection scheme is the same as that used in GP *reproduction*. It sorts the programs in terms of fitness (decided by the fitness function), then takes the X fittest programs and sends them to the simplification component to be simplified. The motivation behind this selection scheme is that the fittest programs are more likely to survive and be present in the next generation. Therefore simplification should only be applied to these in order to reduce overhead caused by simplification.

Task		Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
	0%	28.781 ± 13.427	0.005 ± 0.013	1.221 ± 0.501	37.611 ± 5.634	421807.87 ± 207935.21
	20%	**27.364 ± 13.252**	0.003 ± 0.010	**0.935 ± 0.370**	27.320 ± 2.744	425751.34 ± 173370.71
Easy	40%	32.727 ± 13.440	**0.002 ± 0.005**	1.168 ± 0.429	26.935 ± 3.232	359198.31 ± 155816.78
	60%	30.227 ± 15.034	0.007 ± 0.014	1.123 ± 0.508	26.238 ± 2.440	427832.81 ± 249503.70
	80%	34.227 ± 12.126	0.005 ± 0.014	1.447 ± 0.559	27.691 ± 4.082	**331366.68 ± 123966.32**
	100%	32.438 ± 13.119	0.011 ± 0.042	1.232 ± 0.464	**25.606 ± 2.937**	373212.90 ± 147767.37
	0%	44.875 ± 4.756	83.774 ± 75.283	5.141 ± 1.019	104.436 ± 22.171	**414357.68 ± 87175.63**
	20%	44.875 ± 4.756	111.863 ± 117.870	**4.094 ± 0.612**	75.609 ± 12.282	510018.71 ± 130535.07
Hard	40%	44.875 ± 4.756	117.564 ± 198.966	4.170 ± 0.837	**71.814 ± 14.839**	466109.40 ± 96110.77
	60%	44.875 ± 4.756	**77.730 ± 73.513**	4.629 ± 0.942	75.355 ± 15.644	464869.34 ± 101284.79
	80%	44.875 ± 4.756	96.386 ± 73.106	5.296 ± 1.577	81.316 ± 25.186	470749.62 ± 111109.85
	100%	44.875 ± 4.756	92.884 ± 80.624	5.206 ± 0.861	74.362 ± 13.642	465193.18 ± 76491.16
	0%	**35.750 ± 11.200**	**0.973 ± 0.025**	1.657 ± 0.532	44.476 ± 7.302	0.530 ± 0.066
	20%	39.219 ± 11.392	0.953 ± 0.049	**1.584 ± 0.478**	35.398 ± 6.799	**0.536 ± 0.060**
Coins	40%	36.000 ± 11.579	0.970 ± 0.035	1.661 ± 0.588	36.928 ± 7.313	0.532 ± 0.067
	60%	36.500 ± 10.697	0.969 ± 0.040	1.723 ± 0.497	35.492 ± 5.563	0.527 ± 0.060
	80%	35.781 ± 10.260	0.970 ± 0.033	1.736 ± 0.442	34.975 ± 6.472	0.520 ± 0.042
	100%	37.469 ± 10.992	0.964 ± 0.039	1.700 ± 0.452	**32.539 ± 5.622**	0.515 ± 0.063
	0%	46.077 ± 3.578	0.855 ± 0.117	2.646 ± 0.578	37.861 ± 8.755	**0.427 ± 0.052**
	20%	46.077 ± 3.578	0.871 ± 0.091	**2.337 ± 0.488**	30.392 ± 6.589	0.417 ± 0.049
Faces 1	40%	**45.712 ± 4.415**	0.846 ± 0.130	2.366 ± 0.451	29.279 ± 5.496	0.411 ± 0.051
	60%	46.058 ± 3.579	0.872 ± 0.089	2.544 ± 0.583	29.720 ± 7.130	0.418 ± 0.050
	80%	46.077 ± 3.578	0.849 ± 0.090	2.587 ± 0.511	**29.235 ± 5.642**	0.416 ± 0.052
	100%	45.712 ± 4.415	**0.876 ± 0.104**	2.622 ± 0.583	29.798 ± 6.571	0.427 ± 0.057
	0%	44.481 ± 5.442	0.897 ± 0.097	2.681 ± 0.451	39.566 ± 5.905	0.430 ± 0.056
	20%	45.769 ± 3.817	0.890 ± 0.093	**2.376 ± 0.351**	**30.637 ± 4.537**	0.430 ± 0.056
Faces 2	40%	**43.692 ± 5.979**	0.904 ± 0.095	2.460 ± 0.408	31.963 ± 5.575	**0.434 ± 0.049**
	60%	44.115 ± 6.178	0.904 ± 0.092	2.609 ± 0.484	31.903 ± 5.515	0.432 ± 0.055
	80%	44.692 ± 5.450	**0.908 ± 0.093**	2.659 ± 0.458	30.668 ± 5.006	0.428 ± 0.045
	100%	44.365 ± 6.235	0.889 ± 0.093	2.714 ± 0.548	30.653 ± 5.734	0.421 ± 0.053

Table 6.2: Fitness Based Selection Method - Results

When comparing this to the Random Selection Method, the results are very similar. Final fitness for easy regression was better (0.002 vs. 0.005), while final fitnesses for hard regression (77.730 vs. 65.681) and faces #2 (0.908 vs. 0.913) were slightly lower. For the remaining two datasets (coins and faces #1), the best final fitnesses were identical (0.973 and 0.876).

In terms of execution time and average program size, the two sets of results were almost indistinguishable. With very minor differences between the two for each task.

6.2.3 Selection Method 3: Fatness (Size) Based Elitism

This is the most intuitive of the three methods, as one of the problems simplification is targeted at solving is that of *program bloat*. It is sensible to select the largest programs in the population for simplification in order to get *maximum* reduction in program size. This selection method simply sorts the programs in terms of size and takes the X largest programs and sends off to be simplified by the simplification component.

Task		Final Gen	Final Best Fit	Time(s)	Avg. Prog Size	Avg. Prog Fit
	0%	**28.781 ± 13.427**	0.005 ± 0.013	1.221 ± 0.501	37.611 ± 5.634	421807.87 ± 207935.21
	20%	30.727 ± 14.441	0.010 ± 0.029	**1.060 ± 0.409**	26.753 ± 2.990	401906.56 ± 207145.00
Easy	40%	30.000 ± 14.165	0.003 ± 0.009	1.183 ± 0.581	27.581 ± 5.119	404264.93 ± 197351.78
	60%	34.545 ± 12.389	0.003 ± 0.006	1.347 ± 0.437	26.228 ± 3.658	**342317.84 ± 162105.00**
	80%	30.273 ± 14.412	**0.001 ± 0.005**	1.237 ± 0.595	25.982 ± 4.484	408594.25 ± 199233.90
	100%	32.438 ± 13.119	0.011 ± 0.042	1.232 ± 0.464	**25.606 ± 2.937**	373212.90 ± 147767.37
	0%	44.875 ± 4.756	83.774 ± 75.283	5.141 ± 1.019	104.436 ± 22.171	**414357.68 ± 87175.63**
	20%	44.875 ± 4.756	**56.764 ± 45.882**	**4.383 ± 1.534**	77.703 ± 29.461	482652.65 ± 122734.78
Hard	40%	44.875 ± 4.756	85.186 ± 111.938	4.387 ± 0.914	72.026 ± 15.854	471394.15 ± 104739.37
	60%	44.875 ± 4.756	88.535 ± 97.251	4.684 ± 1.382	72.482 ± 22.921	487436.87 ± 113535.41
	80%	44.875 ± 4.756	71.687 ± 82.134	4.865 ± 1.308	**71.575 ± 20.886**	476821.21 ± 91336.75
	100%	44.875 ± 4.756	92.884 ± 80.624	5.206 ± 0.861	74.362 ± 13.642	465193.18 ± 76491.16
	0%	35.750 ± 11.200	**0.973 ± 0.025**	1.657 ± 0.532	44.476 ± 7.302	0.530 ± 0.066
	20%	37.719 ± 11.791	0.968 ± 0.032	**1.633 ± 0.508**	36.032 ± 5.783	0.531 ± 0.063
Coins	40%	39.406 ± 10.494	0.965 ± 0.026	1.779 ± 0.507	34.479 ± 6.039	**0.541 ± 0.068**
	60%	**34.844 ± 10.796**	0.968 ± 0.044	1.681 ± 0.489	34.666 ± 5.042	0.519 ± 0.055
	80%	36.938 ± 11.204	0.967 ± 0.040	1.788 ± 0.487	33.336 ± 5.530	0.514 ± 0.061
	100%	37.469 ± 10.992	0.964 ± 0.039	1.700 ± 0.452	**32.539 ± 5.622**	0.515 ± 0.063
	0%	46.077 ± 3.578	0.855 ± 0.117	2.646 ± 0.578	37.861 ± 8.755	**0.427 ± 0.052**
	20%	46.000 ± 3.616	0.873 ± 0.072	**2.330 ± 0.476**	29.436 ± 5.855	0.420 ± 0.050
Faces 1	40%	46.077 ± 3.578	0.856 ± 0.097	2.394 ± 0.446	**28.443 ± 5.352**	0.412 ± 0.050
	60%	46.077 ± 3.578	0.859 ± 0.092	2.608 ± 0.564	29.786 ± 6.665	0.422 ± 0.050
	80%	46.077 ± 3.578	0.866 ± 0.077	2.696 ± 0.528	29.478 ± 5.646	0.418 ± 0.052
	100%	**45.712 ± 4.415**	**0.876 ± 0.104**	2.622 ± 0.583	29.798 ± 6.571	0.427 ± 0.057
	0%	44.481 ± 5.442	0.897 ± 0.097	2.681 ± 0.451	39.566 ± 5.905	**0.430 ± 0.056**
	20%	**44.115 ± 6.750**	0.899 ± 0.091	**2.357 ± 0.395**	31.023 ± 4.692	0.419 ± 0.058
Faces 2	40%	45.173 ± 4.549	0.870 ± 0.088	2.530 ± 0.368	30.499 ± 4.426	0.414 ± 0.054
	60%	44.692 ± 5.335	0.904 ± 0.092	2.577 ± 0.334	**30.206 ± 4.720**	0.429 ± 0.038
	80%	44.231 ± 6.292	**0.919 ± 0.082**	2.743 ± 0.568	31.147 ± 5.451	0.423 ± 0.052
	100%	44.365 ± 6.235	0.889 ± 0.093	2.714 ± 0.548	30.653 ± 5.734	0.421 ± 0.053

Table 6.3: Fatness Based Selection Method - Results

Compared to the other two methods, again the execution times and average program sizes are minutely different. More interestingly, when using this method, the final fitnesses for easy regression (0.001 vs. 0.002 vs. 0.005), hard regression (56.764 vs. 77.730 vs. 65.681) and faces #2 (0.919 vs. 0.908 vs. 0.913) were all higher. For the remaining two datasets (coins and faces #1), the best final fitnesses remained identical (0.973 and 0.876) for all three methods.

6.2.4 Discussion of the Selection Methods

While all three methods posted similar results, the 'fatness' based selection method did manage to achieve better fitnesses in the easy regression, hard regression and faces #2 tasks then the other two methods. In terms of efficiency (average program size and execution time), all three methods had very similar results (with only minute variations). As fitness was the only real separating factor between the methods, the 'fatness' based selection method shows itself as the best method of the three for selecting programs for simplification.

6.3 Balancing Frequency and Proportion

In essence, variation in *frequency* addresses the question of "how often?..." simplification should be performed in a system run, and variation in *proportion* addresses the question of "...and to which programs?". Experiments where these two parameters are varied may provide guidelines as the what values these parameters should initially be set to when applying simplification to a new GP task.

This section shows the effects of varying these parameters has on the experimentation tasks, while using the *fatness based elitism* selection method described earlier.

Each experimentation subsection contains 3D graphs of *final program size*, *final system fitness*, *execution times* and *number of generations* for each combination of frequency/proportion to provide insight into the effects of frequency and proportion on the efficiency and effectiveness of the GP system.

6.3.1 Results

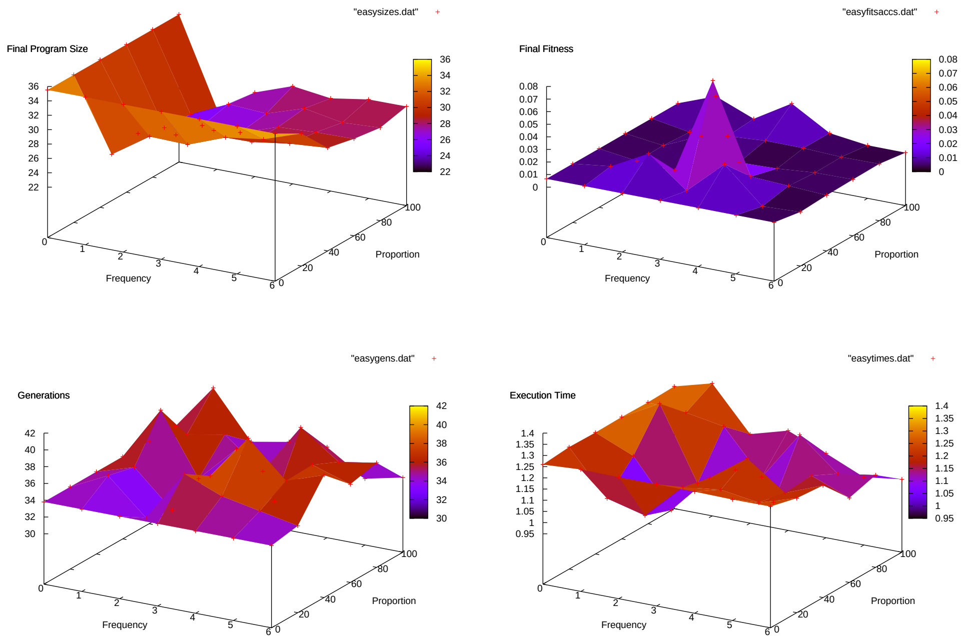

Figure 6.1: Easy Regression - Frequency vs. Proportion

These four graphs show the results for varying combinations of proportion and frequency.

Firstly, the *program size* graph shows clearly that applying simplification every generation to *all* the programs results in the smallest average program size. By lowering either frequency or proportion, the average program size increases, shown by the upward slope of the graph surface.

The *final fitness* graph shows the average final fitness of solutions evolved in the various systems. Several dark regions can be seen interspersed on the surface, and a fitness "spike" is clearly noticeable in the centre of the surface. This suggests that "best" fitness can occur at various combinations of frequency/proportion and that fitness at one combination does not necessarily mean a similar fitness at neighbouring combinations.

The *generations* graph continues to show (as in other results gathered so far) that the number of generations the system requires to evolve a solution unpredictably changes from combination to combination. Although in this case it seems that "at best", simplification systems can generate a solution in the same number of generations as a system without simplification.

The *execution time* graph also fluctuates as frequency or proportion are changed.

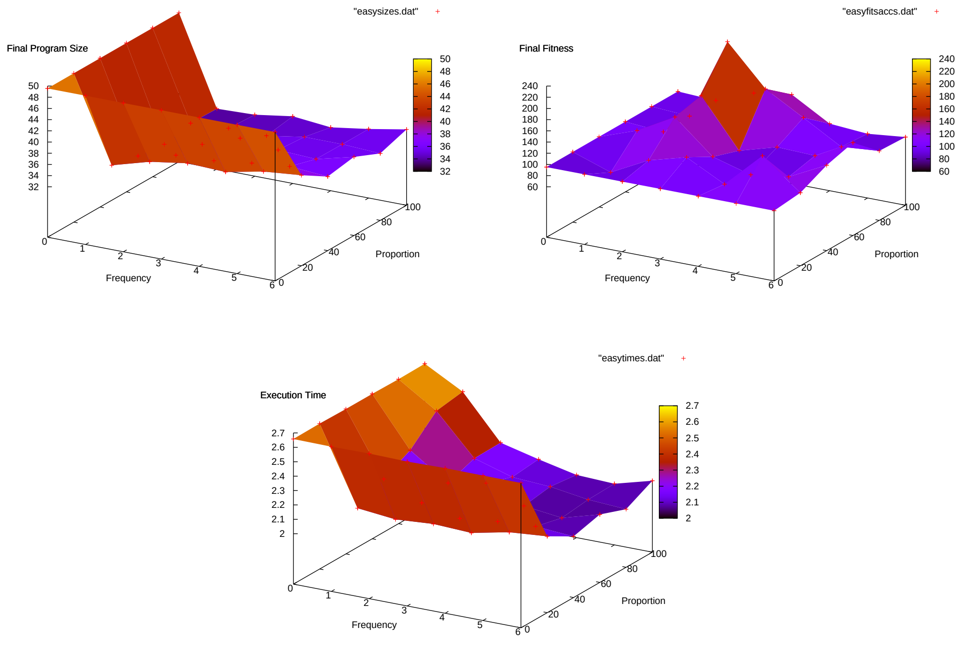

Figure 6.2: Hard Regression - Frequency vs. Proportion

In this task, the *program size* graph exhibits the same trends as the easy regression task and all combinations of frequency/proportion result in lower program size. The *final fitness* graph also shows similar behaviour to its counterpart in the easy regression task. A ”spike” in fitness is again apparent in the middle of the surface.

Unlike the easy regression task, the *execution time* does not fluctuate and shows a clear trend that as frequency and proportion are lowered, the execution time of the system also lowers. This further reinforces the idea that a GP system with lower frequency and proportion still gains benefit from having smaller programs to process, while also benefiting from reduced simplification overhead.

A graph of final generations was not included for this task at they were all identical (at `50`), providing an uninteresting graph.

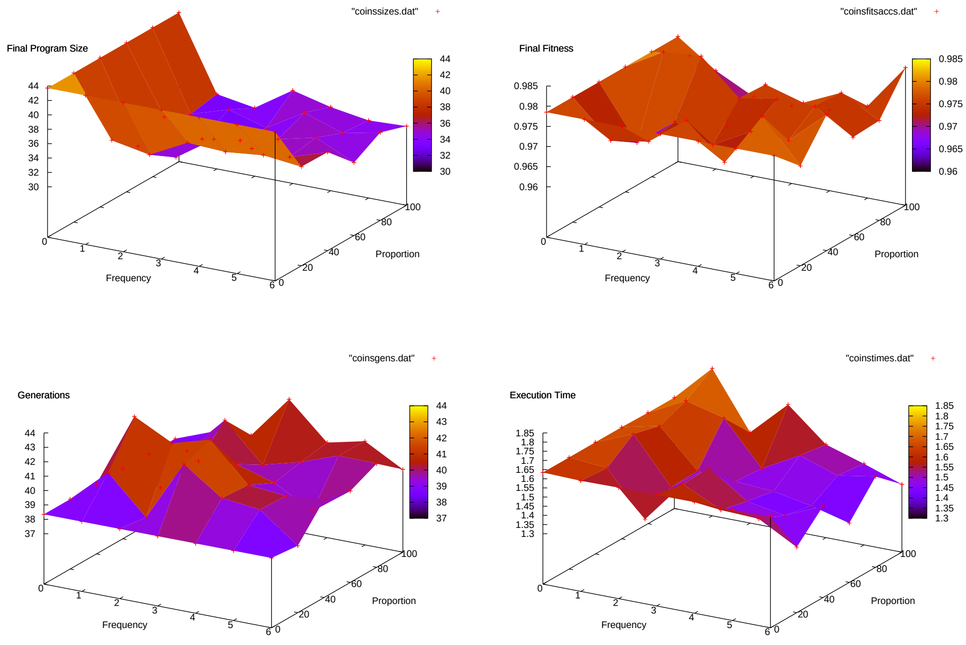

Figure 6.3: Coins Dataset - Frequency vs. Proportion

The *program size* graph exhibits the same trends as the two regression tasks, with higher frequency and proportion providing the smallest programs. The *final fitness* graph is more erratic than the regression tasks, with several "peaks" and "valleys".

Like in the easy regression task, the *execution time* graph fluctuates. This property is only seen in this task and the easy regression tasks (the two "easiest" experimentation tasks). Inspection of the graphs shows that both these tasks have execution times of below 2 seconds. It is theorised that tasks which have evolvable solutions in shorter time have more fluctuation in that time when adjusting frequency and proportion.

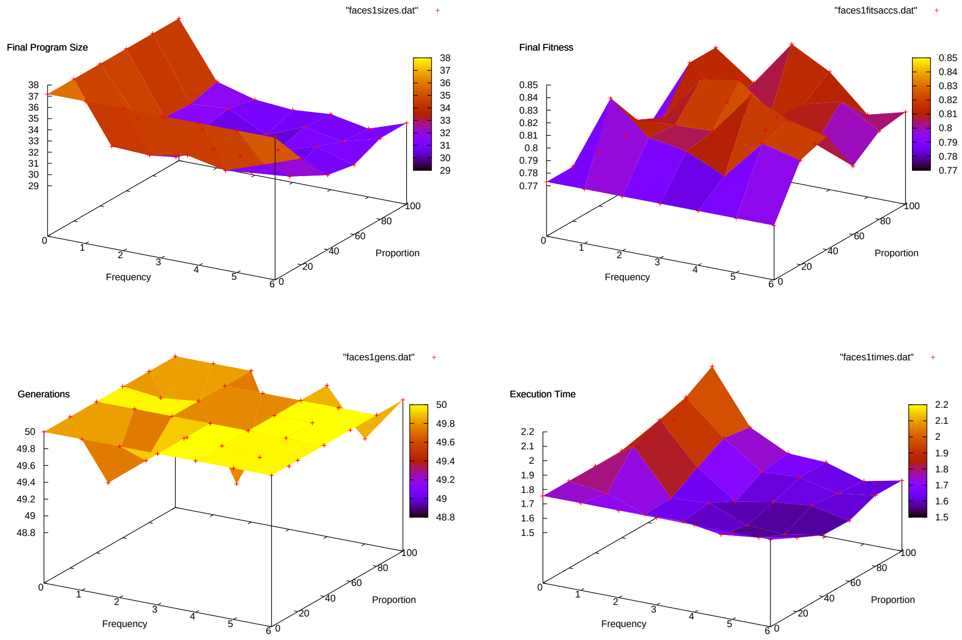

Figure 6.4: Faces Dataset 1 - Frequency vs. Proportion

The *program size* and *execution time* graphs show similar trends to their counterparts in the hard regression task. Whereas the *final fitness* graph is similar to its counterpart in the coins classification task.

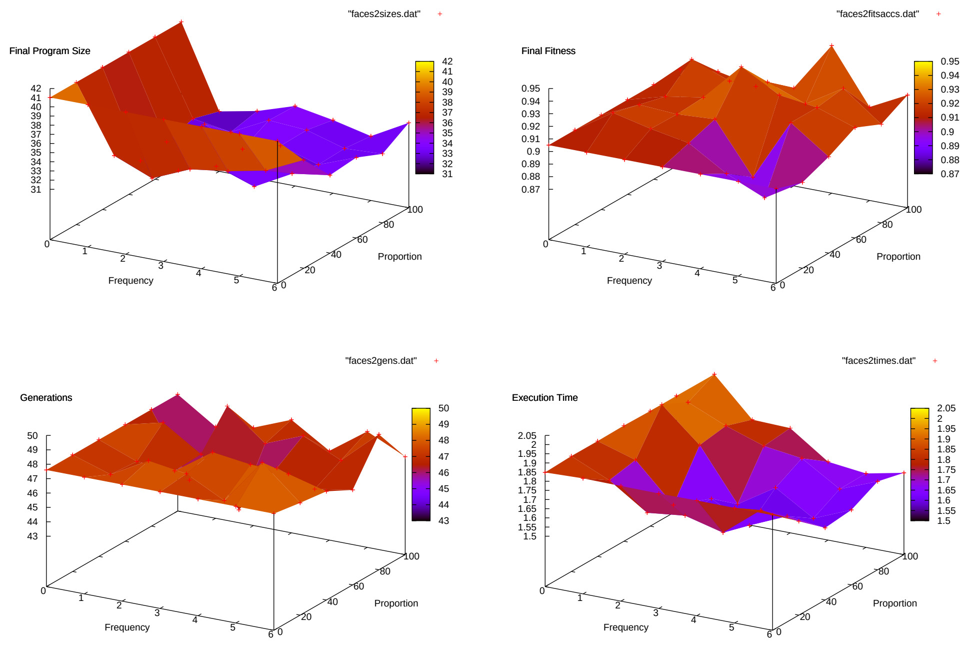

Figure 6.5: Faces Dataset 2 - Frequency vs. Proportion

In this final task, the results are very much like those in face dataset #1. The only difference is that *execution time* has steeper transitions and fluctuates a little more. Inspection of the graph shows that execution time for this task is less that 2.05 seconds, which supports the above theory.

6.3.2 Discussion of Results

Effectiveness

Considering the scale of the axes, the fitnesses of each task do not fluctuate very much at all, but it can be seen that better fitnesses occur in interspersed ”pockets”. This shows that a balance between frequency and proportion is needed to achieve the best system for a task. But the fitness graph surfaces are also unstable and show no clear trend as to what combination of frequency and proportion is best for those tasks. Even though some combinations in the middle of the surface provide better fitness, there are also some combinations that provide noticeably worse fitness (e.g. the fitness spikes in the *easy regression* and *hard regression* tasks). Frequency and proportion appear to be parameters that require a lot of ”tweaking” to get correct.

Efficiency

Unsurprisingly in each task, the smallest average program size occurs when frequency is performed at every generation and on every program. Average program size also tended to increase gradually as either frequency or proportion are lowered, as this allows programs to grow for larger amounts of generations before simplification reduced their size again. The gradient in the graphs for frequency is larger than that of proportion, suggesting that frequency has a larger effect on program size than proportion.

In terms of *generational* time, the results were very erratic, showing no clear trend. This result is similar to the generational time results from earlier chapters and shows that improving a systems efficiency by reducing the number of generations to train it is not an easy task to accomplish.

In terms of *real/wall-clock* time, the behaviour as frequency and proportion changed depended on the experimentation task. For "easier" tasks, in which a solution could be found in small amounts of time, execution time changed erratically as frequency and proportion were changed (though the difference in time is very small). For more difficult tasks, changes in execution time as frequency and proportion were varied were smoother, displaying a trend that the shortest execution times appeared when frequency and proportion were reduced.

This observation shows that if one solely wants the lowest execution times, which is more important in "difficult" tasks, then using simplification with a lower frequency and lower proportion (which combines lessening simplification overhead with the execution time gain from processing smaller programs) is generally the best choice.

Overall, the execution times for systems using simplification at a lower frequency than *every generation* are almost always lower than that of the standard GP system. Therefore, if one wants to achieve *better* execution times with comparable/better effectiveness, then concentrating on optimizing effectiveness alone is generally sufficient to obtain improvements in both.

6.4 Summary

In this chapter, three different methods for selecting programs to be simplified were tested: *Random Selection*, *Fitness Based Elitism* and *Fatness Based Elitism*. The results for all three were all very similar, with *Fatness Based Elitism* having a slight advantage in fitness over the other two methods.

Using this selection method, several combinations of frequency (varied between 0 and 6) and proportion (varied between 0% and 100%, in steps of 20%) were tested for all of the experimentation tasks.

When varying *both* frequency and proportion, it was generally found that simplifying 100% of programs at every generation provided the smallest programs, while simplifying at 0-20% every 6^{th} generation generally provided the fastest execution times for relatively "difficult" programs.

The optimal combination of frequency/proportion to get the best fitness however, was varied for each experimentation task, and was quite erratic (fitnesses for one combination were *not* always similar to fitnesses for neighbouring combinations).

Execution times and program sizes were typically lower for systems using simplification. It was so decided that, when applying simplification to a new GP task, finding a combination of frequency/proportion that results in optimal fitness is all that is required to benefit from both improved efficiency and effectiveness.

Chapter 7

Building Block Analysis

This chapter will use similar methods to those used by Poli and Langdon [26, *Poli, Langdon 1997*] to investigate and discuss the effects (if any) of *simplification* on schema during a GP run. Particularly, whether the simplification process breaks up building blocks in the GP system.

7.1 Poli and Langdon's Schema Definition

The schema definition used by Poli and Langdon follow the GA schema definition more closely than other GP schemas. A schema is defined as a tree, with the function nodes taken from the set $\{F \cup =\}$ and terminal nodes from the set $\{T \cup =\}$. The = node represents an "anything" node, which matches any node in the set of functions/terminals (F/T).

For example, in a system with function set $F = \{+, -\}$ and terminal set $T = \{x, y\}$, the schema `(= x =)` represents the set of programs {`(+ x x)`, `(+ x y)`, `(- x y)`, `(- x x)`}.

The *order* of a schema is defined as the total number of non-= symbols. The *length* is defined as the total number of nodes, and the *defining length* is defined as the number of links in the minimum subtree including all the non-= symbols.

Using the above example (`(= x =)`), the order is 1, the length is 3 and the defining length is 1.

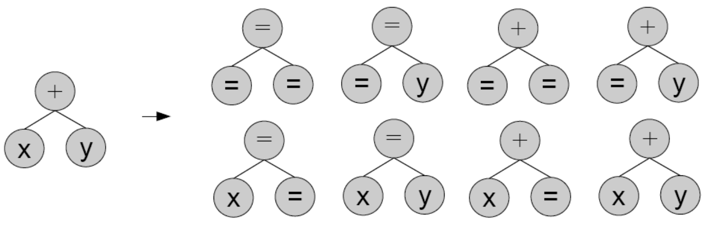

Figure 7.1: List of the 8 schema from a single program

These schema can be viewed as representations of the building blocks in a GP system, and by tracking individual schema and the fitness of schema, the propagation and elimination of building blocks can be investigated.

Matching schema is where the methods used in this project differ from Poli and Langdon's definition and experiments. In their work, a particular schema only matched an exact copy of that schema (e.g. `(= x =)` would only match `(= x =)`). In this project, the *matching scheme* has been relaxed a little to mean that if an existing schema is a *substring* of the other, then the schema match and the appropriate count/fitness tracking variables updated accordingly. This was done as to more closely follow the concept of a 'building block', and it is reasonable to say that if a program yields a schema `(+ y (= x =))`, that this schema also includes $(= x =)$ and so the second schema must also be counted and fitness updated.

By processing and analysing programs from smallest to largest, all schema can be correctly and accurately accounted for.

7.2 Test Results

For this task, it is necessary for the problem used to be a very simple one, with solutions available at very short tree depths. The reason for this is that using the above schema definition, every program has 2^n different schema, where n is the number of nodes in the tree. This means at the usual program depths of 5 or 6 deep, the number of schema runs into the many thousands, making in depth analysis intractable (at a depth of 5, the number of possible schema for a fully grown tree is `9223372036854775808`, generating and analysing each of these is not feasible).

Many tasks can be classed as simple, including very basic regression tasks and the *santa fe ant problem* [18, *Poli and Langdon 1998*]. But as in Poli and Langdons work, the *XOR problem* was finally decided as the experimentation task, as it is fairly simple and required no additional operators or simplification rules to implement.

The XOR problem is a basic machine learning problem wherein the learning system is given the task of creating a solution that can mimic the functionality of the binary XOR logic operation. In other words, for the four possible input patterns, the solution will output the correct value.

Pattern	**Input 0**	**Input 1**	**Output**
1	0	0	0
2	0	1	1
3	1	0	1
4	1	1	0

Table 7.1: The four possible XOR patterns

The function set used was the basic set of arithmetic functions $\{+, -, *, /\}$ and the terminal set was the two inputs required for XOR $\{Input1, Input2\}$, no randomly generated terminals were used. The fitness function was the classification accuracy for the four XOR patterns. Unlike most GP systems, the system was *not* terminated if a solution was found (i.e. a program that handles all four patterns correctly) and terminated only when the 50 generation limit was reached. This was so that schema propagation could be monitored throughout the full 50 generations and not just the few generations it might take to derive a solution.

The results in this chapter are less concerned with the actual performance results of the system, and are more concerned with the inner workings of the system and how schemata are propagating in the system.

7.2.1 Selection Only

The following GP parameters were used for the system:

Crossover and mutation rates were not used in the system (both set to 0%) for this part of the experimentation in order to identify the effects of the simplification method alone and its impact on the schemata in the system.

Instead of strict elitism/reproduction, a *roulette wheel* selection scheme was used. This assigns each program in the population a chance of being promoted to the next generation based in how fit the program is (i.e. fitter programs have a higher chance of being promoted). Programs are then randomly selected until the number of programs to be copied through reproduction is met. Programs are also allowed to be selected more than once, allowing multiple copies of a single program to be present in the next generation.

Generations	50
Population Size	20
Mutation Rate	0%
Reproduction Rate	100%
Crossover Rate	0%
Min. Tree Depth	1
Max. Tree Depth	3

Table 7.2: Genetic Programming System Parameters

The following graph shows the convergence of schema in the XOR problem using selection only with varying frequencies of simplification:

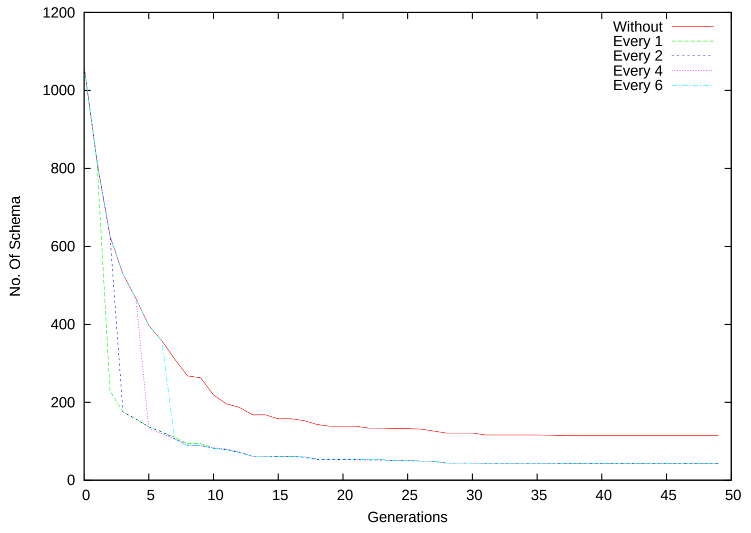

Figure 7.2: Generations vs. No. of Schema - Selection Only

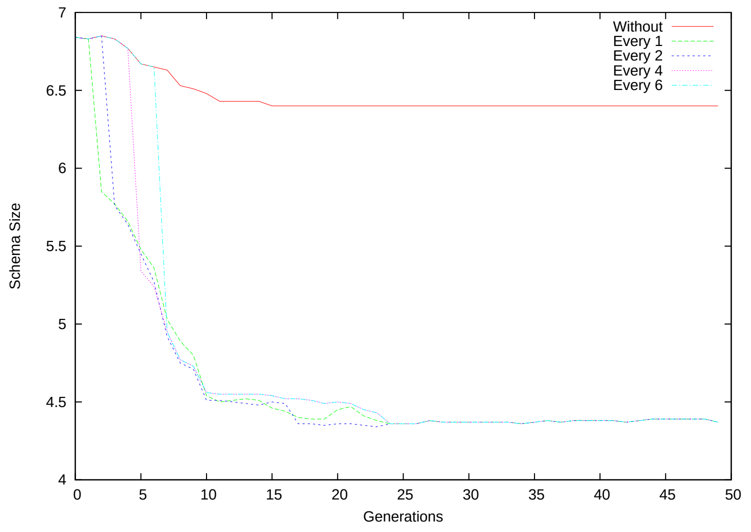

Figure 7.3: Generations vs. Average Schema Size - Selection Only

In the absence of crossover and mutation, the only difference in results between the different frequencies is where the initial drop in number of (distinct) schema and schema size occurs.

The roulette wheel selection genetic operator favours the fittest programs, promoting them to the next generation, sometimes multiple times. This means that the population will eventually converge to a small set of the fittest programs that were originally generated. This is shown by the smooth decline in both number of schema and schema size (smaller programs do not contain as many schema as larger programs).

Without simplification, the number of schema present throughout the GP run is higher than those systems with simplification. This is because the systems with simplification eliminate larger programs from the system and reducing the number of instances that populate larger schema. This either totally eliminates these larger schema or reduces their influence within the population. This reduction in schema size and number of schema is clearly shown by the initial drops in each graph.

However, although these larger programs are no longer in the genetic program population, they have been replaced with smaller programs of equal fitness. These smaller programs populate the smaller schema within the system, causing those schema to become fitter. So although larger, possibly highly fit schema may be removed, they are replaced by shorter schema of relatively equal fitness.

This is further shown by the fact that in all the system runs, the *best* fitness for each generation was the identical. But while simplification itself does not disadvantage the system, the true test of simplification will be its effect on the GP system when combined with crossover.

7.2.2 With Crossover

For these experiments, crossover was applied to 50% of the program population and elitism to the other 50%. Mutation was again left at 0% as its presence would make analysis of schema propagation much harder.

The following GP parameters were used for the system:

Generations	50
Population Size	20
Mutation Rate	0%
Reproduction Rate	100%
Crossover Rate	0%
Min. Tree Depth	1
Max. Tree Depth	3

Table 7.3: Genetic Programming System Parameters

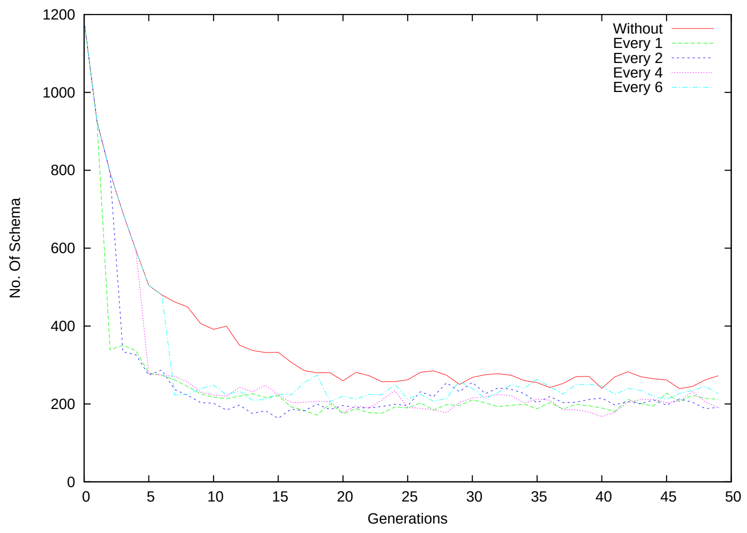

Figure 7.4: Generations vs. No. of Schema - Selection and Crossover

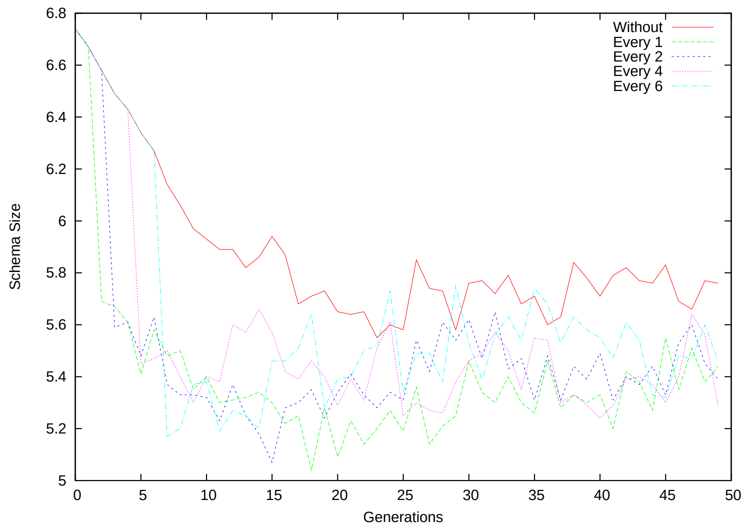

Figure 7.5: Generations vs. Average Schema Size - Selection and Crossover

The inclusion of crossover means that the system never converges on a set of programs (as happens when selection is the only operator). This is shown in the graphs by the fluctuation in both number of schema and schema size. This means that while simplification favours and populates smaller schema by reducing the size of programs, the recombination and creation of larger programs caused by crossover allows larger schema to stay in the system. This also maintains the diversity of building blocks within the system.

7.3 Another Perspective: Tri-Blocks

One of the limitations of using *schema analysis* is that for ″real″ problems, where programs can be around a tree-depth of 7+, the number of schema for a single program is very large. This makes experimental analysis very time consuming, if not intractable.

Therefore, a different way of looking at ″building blocks″ needs to be used. The main reason for the number of schema per program being very large is that it allows schema of the form `(+ (= x y) y)`, that is, it allows unconnected nodes to constitute a building block.

If we disallow these unconnected blocks, then the number of schema from a single program is decreased, and analysis of larger programs can be performed. In this section, *local*, connected groups of 3 nodes were counted and tracked in the GP system, these groups are dubbed *tri-blocks*.

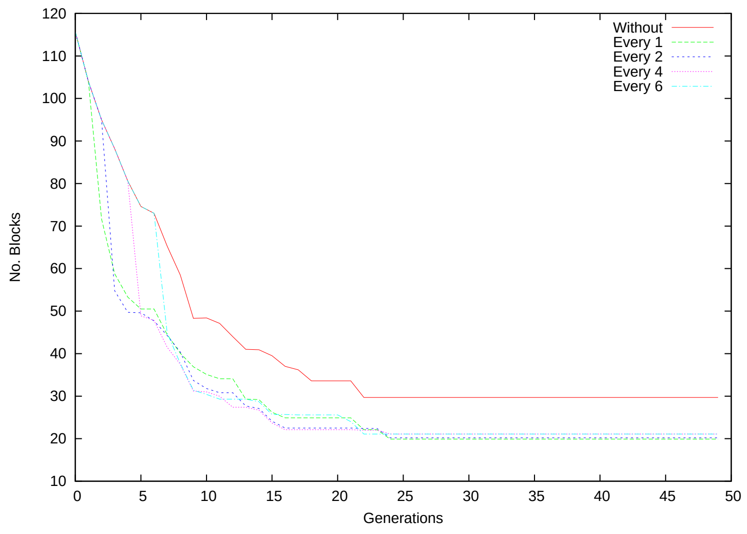

Figure 7.6: Generations vs. Number of Tri-Blocks - Selection Only

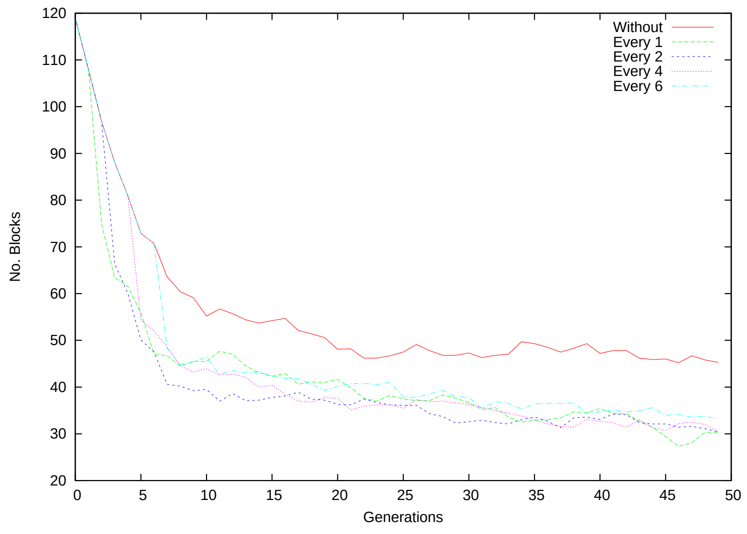

Figure 7.7: Generations vs. Number of Tri-Blocks - Selection and Crossover

These results are similar to those when using Poli and Langdon's schema definition. The difference in these sets of experiments is that all measured building blocks are of the *same size*. So while a reduction in schema in the above experiments could be partially accounted for by the reduction in size of programs, a reduction in (distinct) tri-blocks can only be due to fewer tri-blocks.

Again, when crossover is added, the convergence of tri-blocks is counteracted slightly. Simplification clearly causes a reduction in the number of tri-blocks in the GP system. It can be concluded that simplification causes an overall *reduction* in the number of building blocks within a GP system.

7.4 Summary

In this chapter, using the schema definition put forward by Poli and Langdon's earlier work, schema analysis was performed on a simple GP task (the XOR problem). Results from these analyses suggested that while simplification may remove some larger building blocks, it effective shifts the fit programs from larger schema to shorter schema. Therefore, equally fit, smaller building blocks can effectively replace the larger building blocks. Crossover's recombination effect also rebuilds some of the larger building blocks during the GP run.

A more simple way of defining buildings blocks was then used (tri-blocks) to analyse a larger version of the XOR problem. Tri-blocks are simply locally connected groups of 3 nodes. These tri-blocks were tracked and analysed within GP systems (with and without simplification) running the XOR problem with larger programs (depths of 7 allowed). The results found for this definition strongly suggested that building blocks are removed from the system by the simplification process.

More work needs to be done in this area, most importantly to determine whether *good/fit* building blocks are disrupted by simplification.

Chapter 8

Conclusions

This project aimed to develop simplification techniques and explore the effect that simplification has on a GP system when applied during evolution.

8.1 Using Algebraic Simplification during Evolution

In line with the first goal of this project, chapter 4 developed a simple algebraic approach to the simplification of genetic programs. This system used a set of basic rules along with an algebraic equivalence component to simplify tree-based programs directly without needing to translate it into another format.

The integration of this method into a GP system to simplify all programs was found to slightly improve effectiveness, although achieving this required finding an appropriate value for frequency. Performing simplification every generation led to slightly poorer effectiveness in programs.

The average size of programs and thus the memory usage of the GP system was significantly lowered when simplification was used, regardless of how often the simplification component was invoked (frequency). This reduction in memory and lower amounts of program processing leads in most cases to lower execution times and overall better GP system efficiency.

Comprehensibility is made easier when using simplification as it reduces the amount of information (program) that needs to be interpreted and understood. Other factors can also influence comprehensibility such as what range the features have been normalised to.

Overall, the performance of GP systems was improved with the addition of a simplification component. Programs of comparable or better effectiveness were obtainable for all of the experimentation tasks, regardless of difficulty.

8.2 Using Prime Number Simplification with Algebraic Equivalence during Evolution

Another objective of this project was to show how a numeric hashing method could be used to simplify the genetic programs. Improvements to the PRES algorithm, a numeric hashing method for simplifying genetic programs, were made. These included the integration of an *algebraic equivalence* component, *monte carlo* program reconstruction, and usage of a arbitrary precision math library to prevent overflow problems.

When applying this simplification component to a GP system, effectiveness was usually slightly reduced in most tasks. With the exception of the face datasets. It was thus shown that simplification benefits relatively difficult GP tasks.

Like with the algebraic simplification system, program sizes were significantly smaller than the standard GP system for all tasks.

However, the algorithm uses a very high amount of resources which results in very high execution times. This was caused by the necessity of using an arbitrary precision math library for calculations to prevent arithmetic overflow. Using the algorithm at low frequencies (e.g. every 6) brought the execution times to "acceptable" levels (e.g. `29.332` for easy regression, `13.928` for coins dataset).

Overall, this algorithm is seen as being useful for relatively difficult problems, especially difficult classification problems.

8.3 Effects of Frequency and Proportion

This project also set out to discover how frequency and proportion affect the performance of GP systems. Three different proportion selection scheme were devised (random selection, fitness based selection and fatness (size) based selection). Fatness based selection was found to be marginally better than the other two schemes.

Using this scheme to select program proportions, a balance between frequency and proportion was found to be easily found if efficiency (high speed and low memory usage) is the sole goal, with both execution times and program sizes having "smooth" surfaces that can easily be followed when running multiple tests (or even possibly using hill climbing). However a balance was found to be very difficult to find when attempting to optimize the effectiveness (final fitness of outputted solutions) of the system. The surfaces for all of the experimental tasks were unstable and largely variant, and testing of all possible combinations of frequency/proportion is likely to be necessary to find a good balance.

These findings helped achieve the goal, and provided some guidelines for applying simplification with varied frequency and proportion to new systems.

8.4 Building Blocks

Lastly, this project aimed to determine whether the process of simplification destroys good building blocks in the GP system during evolution. In order to determine this, two different perspectives were used: Poli and Langdon's Schema Definition, and local groups of 3 nodes (Tri-Blocks).

Results using schema analysis showed that simplification reduced the number of building blocks in a GP system, which is a clear disruption of building blocks. Results also showed that crossover counteracts simplification through recombination and creates new building blocks (as has been concluded in other work).

Experiments using selection only showed that while simplification reduced or eliminated larger building blocks, smaller building blocks could be created in the process. Whether the *good* building blocks are disrupted or merely transformed into smaller building blocks remains to be investigated.

Results from tracking tri-blocks also showed that simplification removes building blocks.

Overall, adding simplification to a GP system can significantly reduce evolution time, reduce program size (and thus memory usage), and improve comprehensibility of evolved genetic programs. This can all be achieved without loss of system accuracy/effectiveness, and in some instances can slightly improve the effectiveness of the system, particularly for relatively difficult tasks.

8.5 Future Work

8.5.1 Face Dataset: Removing Lighting Variation

The variation in lighting poses a real problem in evolving a classifier using GP. While using the distance ratios improved the effectiveness of GP, getting these types of features is not a trivial task. It may involve several feature recognisers to be able to centre of the main facial features such as eyes, mouth and nose.

Another approach that is worth looking into is a way to normalise the lighting variation in a preprocessing step, thus removing the variance issue and allowing the GP system to focus on the

$$sin(x) = \frac{e^{ix} - e^{-ix}}{2i}$$

Table 8.1: Exponential form of sine

differences in facial features themselves. This has been looked into by [34, *Wang, Li et al.*] by transforming an image into its *Self-Quotient Image*, which is lighting independent. The following figure shows an example of such an image:

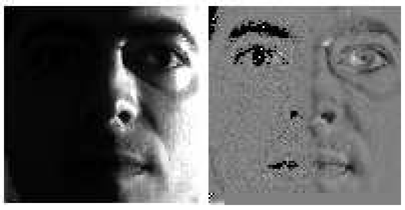

Figure 8.1: Example of an SQI and its deshadow effect

Incorporating this, or a similar method into a GP system will increase the performance that GP has in generating a face recogniser. Something possibly to look into in the future.

8.5.2 Algebraic Simplification: Smarter Rules

In addition to simplification rules that work on *algebraic* rules, additional rules (mostly affecting *conditional operators*) that are task dependent can be added. For example, a program from `Every 0` for the *Coins Dataset* (Algebraic Simplification System) contains the sub-expression `(if<0 f2 (+ 0.069133 0.188200)...)`. If the features in this case were normalised to between 0 and 1 (which is a relatively common practice), we would know that *none* of the features used in the coins dataset are negative. For whenever this sub-expression is evaluated, the first branch will never be executed and so is irrelevant.

Similarly, as another hypothetical example, consider the sub-expression `(if<0 (+ f1 1.0) 0.344 0.577)` (and using the range used in the rest of this project [-1,1]). Since we know `f1` is at the very least `-1`, `(+ f1 1.0)` will never be `<0`, so this expression can be simplified as well.

Adding these *domain/task specific* simplification rules will allow more more compact programs and further redundancy reduction.

8.5.3 Algebraic Equivalence and Hashing: Other Functions

Only the basic arithmetic operators, constant and IF operators/terminals are handled in the current hashing function. Extending this to trigonometric and exponential functions is the next logical step and would require determining an element in $\mathbb{Z}_p$ which can represent *e* and an element that can represent the imaginary element *i*. The properties of these elements must be preserved (e.g. $i^2 + 1 = 0$). Having these two elements would allow not only exponential operators to be hashed, but also trigonometric functions as these can be expressed in terms of *e* and *i*. For example:

8.5.4 Simplification as a search

The algebraic simplification system used in this project is basically a '*greedy* search for the *optimally* simplified expression, as it simply iterates through a list of simplification rules and attempts to apply them sequentially. This will not necessarily lead to an optimally simplified expression, just as a greedy path search will not necessarily lead to an optimal path.

Instead of applying the rules sequentially, one can imagine the system as an *n-dimensional* space, where *n* denotes the number of simplification rules in the system. Simplification can then be treated

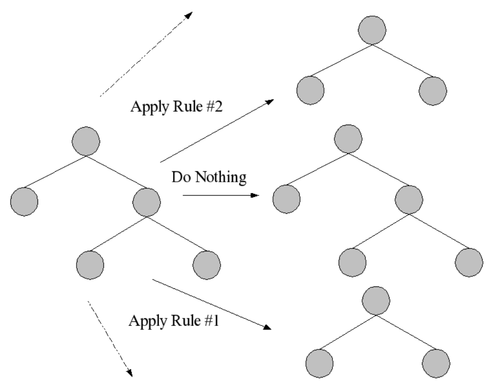

Figure 8.1: Simplification as a search

as a search through this space, where each simplification rule used is a "move" in that direction in the space. Of course, one can also choose to "do nothing" as a "move" as well, which means moving to the parent node without applying any rules to this node. This means more optimal searching methods such as *iterative deepening search* can be used in future designs of the algebraic system.

8.5.5 Prime Number Simplification

Prime number factoring

A problem which is being researched in the number theory community, is that of factoring large numbers into their factors. For this implementation, the trivial method of dividing by primes less than or equal to the square root of the product was used. This is perfectly fine for "small" numbers (i.e. less than 20 digits), but for larger numbers, a faster method should be used.

Several possible methods include the *Pollard-rho* (a non-deterministic algorithm that uses special properties of numbers, and so will not always give a factoring), *Pollard-Strassen* [27] (the fastest known deterministic algorithm) and the *Quadratic Sieve* [3] methods. Using one of these methods will ensure that the prime number hashing method can be used on larger scale problems.

Limitations of Prime Product Quartuple: Using an n-tuple

One of the limitations of the prime product quartuple is the existences of a single positive layer and single negative layer. This essentially only supports operators which act on one axis of "negation" (i.e. + vs. −. `up` vs. `down`).

Consider a simplified *artificial ant* problem similar to the one briefly described in chapter 2. Let this problem contain the function set { `move up`, `move down`, `move left`, `move right` }. Now these are essentially all part of the same operator family, and a sequence `move up`, `move left`, `move left`, `move down` should cancel out the `move up` and `move down` actions. But one cannot encode all four operators in the family into a single prime product, and *wrapping* subtrees will cause the algorithm to fail to "see" that `move up` and `move down` cancel each other out.

Extending the quartuple to an *n-tuple*, which allows multiple negative/positive layer pairs, will allow more than one set of opposing operators to be encoded and canceled correctly. This would allow for easier application of this simplification algorithm into domains that do not use arithmetic or mathematical operators (e.g. robotics, video games).

8.5.6 Dynamic Adjustment of Proportion/Frequency

The results gathered in this project suggest that there is a combination of frequency and proportion that can lead to optimal performance in a GP system. However, like most GP parameters, there may be guidelines to what values these should start at, but overall there is no way to know what combination works best without extensive testing.

One wonders whether it is possible to dynamically adjust the frequency and proportion during the course of evolution to react to the conditions of the GP system (i.e. increasing frequency or proportion if memory/resource usage is becoming too high, or decreasing if best program fitness is stagnating and not becoming fitter for a certain number of generations). This may provide an easier way of applying simplification to a new task, as multiple experiments to discover the optimal combination of frequency/proportion may not be necessary.

Alternatively, a type of *momentum* parameter may be introduced. *Momentum* is a parameter in *Neural Networks* which amplifies the learning rate as the network trains, causing a network to converge on an "answer" more quickly. If simplification is more important in the early stages, it makes sense to perform simplification more often in the early generations and then lower the frequency as time goes on (and vice versa if simplification is more important in the latter stages).

8.5.7 More Building Block Analysis

While initial findings in this project indicate that building blocks are removed from the GP system when simplification is invoked, there are no indications whether *good* building blocks are disrupted. The fact that simplification systemscan improve the effectiveness of tasks suggests that good building blocks must remain in the system, and large building blocks may meerely be tranformed into smaller, equally fit building blocks. More investigation into these aspects, as well as on more difficuly tasks needs to be undertaken.

Bibliography

[1] BANZHAF, W., FRANCONE, F. D., KELLER, R. E., AND NORDIN, P. *Genetic programming: an introduction: on the automatic evolution of computer programs and its applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[2] 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.

[3] BOENDER, H., AND RIELE, H. Factoring integers with large prime variations of the quadratic sieve, 1995.

[4] BUSCH, J., ZIEGLER, J., AUE, C., ROSS, A., SAWITZKI, D., AND BANZHAF, W. Automatic generation of control programs for walking robots using genetic programming. In *EuroGP '02: Proceedings of the 5th European Conference on Genetic Programming* (London, UK, 2002), Springer-Verlag, pp. 258–267.

[5] EKART, A. Shorter fitness preserving genetic programs. In *Artificial Evolution. 4th European Conference, AE'99, Selected Papers* (Dunkerque, France, 3-5 Nov. 2000), C. Fonlupt, J.-K. Hao, E. Lutton, E. Ronald, and M. Schoenauer, Eds., vol. 1829 of *LNCS*, pp. 73–83.

[6] FAYYAD, U. M., PIATETSKY-SHAPIRO, G., SMYTH, P., AND UTHURUSAMY, R., Eds. *Advances in knowledge discovery and data mining*. American Association for Artificial Intelligence, Menlo Park, CA, USA, 1996.

[7] FERNANDEZ, J. J. The genetic programming tutorial, 2003. http://www.geneticprogramming.com/Tutorial/.

[8] FIKES, R., AND NILSSON, N. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence 2* (1971), 189–208.

[9] FORREST, S., AND MITCHELL, M. Relative building-block fitness and the building-block hypothesis. In *Foundations of Genetic Algorithms 2*, L. D. Whitley, Ed. Morgan Kaufmann, San Mateo, CA, 1993, pp. 109–126.

[10] FOUNDATION, F. S. The gnu mp bignum library, 2005. http://www.swox.com/gmp/.

[11] GEORGHIADES, A., BELHUMEUR, P., AND KRIEGMAN, D. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intelligence 23*, 6 (2001), 643–660.

[12] HOLLAND, J. *Adaption in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[13] KEIJZER, M. Efficiently representing populations in genetic programming. 259–278.

[14] KOHAVI, R. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *IJCAI* (1995), pp. 1137–1145.

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

[16] KOZA, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994.

[17] KOZA, J. R., KEANE, M. A., STREETER, M. J., MYDLOWEC, W., YU, J., AND LANZA, G. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[18] LANGDON, W. B., AND POLI, R. Why ants are hard. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (University of Wisconsin, Madison, Wisconsin, USA, 22-25 1998), J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, Eds., Morgan Kaufmann, pp. 193–201.

[19] LECUN, Y., JACKEL, L. D., BOTTOU, L., BRUNOT, A., CORTES, C., DENKER, J. S., DRUCKER, H., GUYON, I., MULLER, U. A., SACKINGER, E., SIMARD, P., AND VAPNIK, V. Comparison of learning algorithms for handwritten digit recognition. In *International Conference on Artificial Neural Networks* (Paris, 1995), F. Fogelman and P. Gallinari, Eds., EC2 & Cie, pp. 53–60.

[20] MARTIN, W. A. Determining the equivalence of algebraic expressions by hash coding. 549–558.

[21] MOSES, J. Algebraic simplification: a guide for the perplexed. *Commun. ACM 14*, 8 (1971), 527–537.

[22] NORDIN, P. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[23] O'REILLY, U.-M. *An analysis of genetic programming*. PhD thesis, 1995. Adviser-Franz Oppacher.

[24] PAULUS, D., AND HORNEGGER, J. *Applied Pattern Recognition (2nd edition)*. Vieweg., 1998.

[25] POLI, R. Genetic programming for image analysis. 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. 363–368.

[26] POLI, R., AND LANGDON, W. B. An experimental analysis of schema creation, propagation and disruption in genetic programming. Technical Report CSRP-97-8, University of Birmingham, School of Computer Science, Feb. 1997. Presented at ICGA-97.

[27] POMERANCE, C. Analysis and comparison of some integer factoring algorithms.

[28] SCHUERMANN, J. *Pattern Classification: A Unified View of Statistical and Neural Approaches*. Wiley&Sons, 1996.

[29] SMART, W. *Genetic Programming for Multi-class Object Classification, BSc (Hons) Research Project*. School of MSCS, Victoria University of Wellington, 2003.

[30] SMART, W., AND ZHANG, M. Genetic programming for image object tracking, n/d.

[31] SOULE, T., AND FOSTER, J. A. Effects of code growth and parsimony pressure on populations in genetic programming. *Evolutionary Computation 6*, 4 (Winter 1998), 293–309.

[32] 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 1996), J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., MIT Press, pp. 215–223.

[33] TELLER, A., AND VELOSO, M. Algorithm evolution for face recognition: What makes a picture difficult. In *International Conference on Evolutionary Computation* (Perth, Australia, 1–3 1995), IEEE Press, pp. 608–613.

[34] WANG, H., LI, S. Z., AND WANG, Y. Face recognition under varying lighting conditions using self quotient image. In *FGR* (2004), pp. 819–824.

[35] 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.

[36] WINKELER, J. F., AND MANJUNATH, B. S. Genetic programming for object detection. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 330–335.

[37] ZHANG, M., AND CIESIELSKI, V. Genetic programming for multiple class object detection. In *12th Australian Joint Conference on Artificial Intelligence* (Sydney, Australia, 6-10 Dec. 1999), N. Foo, Ed., vol. 1747 of *LNAI*, Springer-Verlag, pp. 180–192.

[38] ZHANG, M., CIESIELSKI, V. B., AND ANDREAE, P. A domain-independent window approach to multiclass object detection using genetic programming. *EURASIP Journal on Applied Signal Processing 2003*, 8 (July 2003), 841–859. Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis.

[39] ZHANG, M., AND SMART, W. Multiclass object classification using genetic programming. 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. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, Eds., vol. 3005 of *LNCS*, Springer Verlag, pp. 369–378.

[40] ZHANG, Y. *Genetic Programming for Multiple Class Classification, BSc (Hons) Research Project*. School of MSCS, Victoria University of Wellington, 2004.

Appendix A

Appendix I: Simplification Rules

The following is the list of simplification rules used in the algebraic simplification system. Constants are denoted by lower case letters (a, b, c etc.) and variables by upper case letters (A, B, C etc.). If a numeric value is listed (e.g. $1, 0$), the actual rule matched any expression *equivalent* to that numeric value.

Simplification Rule
$if < 0(a, B, C) \rightarrow B$ if $a < 0, C$ if $a \geq 0$
$if < 0(A, B, B) \rightarrow B$
$a + b \rightarrow c, c = a + b$
$a - b \rightarrow c, c = a - b$
$a \times b \rightarrow c, c = a \times b$
$a \div b \rightarrow c, c = a \div b$
$a + (b + C) \rightarrow c + C, c = a + b$
$a + (b - C) \rightarrow c - C, c = a + b$
$a - (b + C) \rightarrow c + C, c = a - b$
$a - (b - C) \rightarrow c - C, c = a - b$
$a \times (b \times C) \rightarrow c \times C, c = a \times b$
$a \times (b \div C) \rightarrow c \div C, c = a \times b$
$a \div (b \div C) \rightarrow c \times C, c = a \div b$
$a + (B + c) \rightarrow b + B, b = a + c$
$a + (B - c) \rightarrow b + B, b = a - c$
$a - (B + c) \rightarrow b - B, b = a + c$
$a - (B - c) \rightarrow b - B, b = a - c$
$a \times (B \times c) \rightarrow b \times B, b = a \times c$
$a \times (B \div c) \rightarrow b \times B, b = a \div c$
$a \div (B \div c) \rightarrow b \div B, b = a \times c$
$A \div 1 \rightarrow A$
$A \div A \rightarrow 1$
$0 \div A \rightarrow 0$
$0 \times A = A \times 0 \rightarrow 0$
$A \times 1 = 1 \times A \rightarrow A$
$A + 0 = 0 + A \rightarrow A$
$A - 0 \rightarrow A$
$A - A \rightarrow 0$
$A \times \frac{1}{B} = \frac{1}{B} \times A \rightarrow \frac{A}{B}$
$A \times \frac{B}{A} = \frac{B}{A} \times A \rightarrow B$

Table A.1: Simplification system rules

Appendix B

Appendix II: Developed Programs and Utilities

B.1 VGP Analysis Tool

The VGP analysis tool is a command line program developed for this project, designed to take raw generational information from a system run and output several statistics, formatted in *.csv* (comma separated values file), *.dat* (gnuplot graph) or *.tex*(latex table formatted). It was neccesary to construct this tool in order to process and analyse VGP log files in depth, with special attention to schema analyses.

The `vgp` package was modified to produce a log file that contained a raw dump of each generation's set of programs, followed by a summary of the systems final results:

```
Generation: 0
Pop has 500
Prog 0 : (10000000.000000)
(+ (% (* (if<0 0.105133 f0 0.712933) (* f0 f0)) (+ (% f0 -0.977000) (+ -0.292200
 f0))) (- (- f0 -0.829800) (% (+ 0.167600 f0) f0)))
Prog 1 : (10000000.000000)
(* (+ (if<0 -0.244400 (- f0 f0) (% 0.528667 f0)) (+ (* f0 f0) (* -0.947467 f0)))
 (% (+ (* f0 -0.516733) (if<0 f0 f0 0.808400)) (% (+ -0.510733 0.073400)
 -0.571800)))
Prog 2 : (10000000.000000)

...


Prog 499 : (199.916527)
(+ (- (% (- f0 (- (+ (% -0.846667 f0) (if<0 f0 f0 -0.548933)) (* (% 0.238600
 -0.184133) (% -0.379000 f0)))) (* 0.666400 (% f0 (if<0 (+ f0 0.090333) (*
 0.625200 f0) (% -0.591733 0.114533))))) (- (if<0 -0.272533 f0 -0.846667)
 (if<0 (if<0 (- (% f0 -0.455867) (* f0 f0)) (% (- -0.128800 f0) (% 0.600400
 -0.861800)) (+ (if<0 f0 f0 0.983867) (- f0 0.723600))) (% (if<0 (+ f0 0.090333)
 (* 0.625200 f0) (% -0.591733 f0)) (% (if<0 f0 f0 f0) (- f0 f0))) (if<0 (* (*
 0.517733 -0.800733) (if<0 0.932133 f0 -0.408533)) (- (% f0 -0.113733) (% f0 f0))
 (- (+ f0 0.016400) (- 0.481000 f0)))))) (* (if<0 (+ (* (% (% f0 0.240000) (- f0
 -0.906067)) (+ (* f0 -0.902267) (- 0.072067 -0.122200))) (if<0 (if<0 (* -0.576400
 0.967000) (if<0 f0 -0.595533 f0) (- -0.730867 0.823667)) (* (- f0 f0) (* f0 f0))
 (- (if<0 0.228333 f0 f0) (+ -0.611133 -0.443600)))) (* (* (* (+ -0.422200
 -0.843333) (+ -0.529400 0.364200)) (- (if<0 f0 -0.781067 -0.307533) (% f0 f0)))
 (* (if<0 (if<0 -0.655000 0.542133 0.393600) (if<0 f0 f0 f0) (% -0.394933
 -0.511133)) (- (if<0 f0 f0 -0.184133) f0))) (if<0 f0 (* (if<0 (+ f0 f0) (+
 0.248933 f0) (if<0 -0.395333 f0 -0.197267)) (+ (if<0 -0.591733 0.894600 f0)
 (+ -0.409600 f0))) (* (+ (+ 0.090333 -0.713133) (% f0 -0.797467)) (if<0 (if<0
 f0 f0 -0.769867) (% f0 -0.886533) f0)))) (if<0 f0 f0 f0)))

Gens: 50  Final_fitness: 100.506036  Took: 7.49 seconds
Best Program: (+ (- (% (- f0 (- (+ (% -0.846667 0.090333) (if<0 f0 f0 f0))
 (* (% 0.238600 -0.184133) (% -0.379000 f0)))) (* 0.666400 (% f0 (if<0 (+ f0
 0.090333) (* 0.625200 f0) (% -0.591733 f0))))) (- (if<0 -0.272533 f0 -0.846667)
 (if<0 (if<0 (- (% f0 -0.455867) (* f0 f0)) (% (- -0.128800 f0) (% 0.600400
 -0.861800)) (+ (if<0 f0 f0 0.983867) (- f0 f0))) (% (if<0 (+ f0 -0.422067) (*
 0.625200 -0.529667) (% -0.591733 f0)) (% (% -0.131933 -0.681133) -0.443600))
 (if<0 (* (* 0.517733 -0.800733) (if<0 0.932133 f0 -0.408533)) (- (% f0 -0.113733)
 (% f0 f0)) (- (+ f0 0.016400) (- 0.481000 f0)))))) (* (if<0 (+ (* (% (% f0
 0.240000) (- f0 -0.906067)) (+ (* f0 -0.902267) (- 0.072067 0.625200))) (if<0
 (if<0 (* -0.576400 0.967000) (if<0 f0 -0.595533 f0) (- -0.730867 0.823667)) (*
 (- 0.496267 f0) (* f0 f0)) (- (if<0 0.228333 f0 f0) (+ -0.611133 -0.443600))))
 (* (* (* (+ -0.422200 -0.843333) (+ -0.529400 0.364200)) (- (if<0 f0 -0.781067
 -0.307533) (% f0 f0))) (* (if<0 (if<0 -0.655000 f0 0.393600) (if<0 f0 f0 f0)
 (% -0.394933 -0.511133)) (- (if<0 f0 f0 -0.184133) f0))) (if<0 (- (+ f0 f0) (-
 f0 f0)) (* (if<0 (+ f0 f0) (+ 0.248933 f0) (if<0 -0.395333 f0 -0.197267)) (+
 (if<0 f0 0.894600 f0) (+ -0.409600 f0))) (* (+ (+ -0.109733 -0.713133) (% f0
 -0.797467)) (if<0 (if<0 f0 f0 -0.769867) (% f0 -0.886533) f0)))) (if<0 f0 f0
 (% -0.083333 0.019867))))
```

The VGP analysis tool tranforms this log file into a `csv` file of tables for Average Sizes, System Average Size, Best Fitnesses, System Best Fitness, Average Fitness, Execution Times, Best Accuracy, No. of Schema, Schema Fitness, Best Programs in System::

```
Average Size
0,36.876,36.688,36.202,34.202,35.64,36.592,36.022,36.932,36.564,35.472,35.72,
35.682,36.248,35.346,37.93,37.42,36.536,35.592,36.222,38.302,36.572,35.484,
35.928,35.604,38.146,=AVERAGE(B2:Z2)
1,37.544,38.848,37.308,34.512,37.15,38.29,38.198,36.214,36.432,37.228,36.076,
37.634,37.072,36.936,37.814,38.536,38.048,37.232,37.834,40.928,38.1,36.158,
37.818,36.334,41.43,=AVERAGE(B3:Z3)
...

(if<0 (% (if<0 f7 (+ (- f5 f7) f5) (* (+ -0.420600 f3) (+ f3 0.323800)))
 (% 0.289267 0.854200)) (* -0.555667 (- (+ f7 -0.461200) -0.448333)) (*
 (if<0 (+ (% f2 0.323800) 0.738667) (* -0.505200 -0.953800) 0.777000) (if<0
 0.175933 0.175933 (* (if<0 0.643067 f4 0.009667) (+ f2 f1))))),0.96875
(if<0 (% (if<0 f7 (+ (- f5 f7) f5) (* (+ -0.420600 f3) (+ f3 0.323800)))
 0.289267) (* -0.555667 (- (+ f7 -0.461200) -0.448333)) (* (if<0 (if<0
 0.740200 f7 0.009667) (* -0.505200 -0.753867) 0.777000) (if<0 0.175933 (-
 (% f2 f4) (if<0 0.833533 -0.930467 f4)) (* (if<0 0.740200 f4 0.009667) (+
 f5 f1))))),0.96875
```

Finally, the `present`(er) and `gnuplotter` tools use the `csv` files to generate tables and graphs ready for inclusion into a LATEXdocument. Combined with scripts, this streamlines the whole process of results analysis and gathering.

B.2 Present(er)

This utility takes the (usually) 6 `csv` files generation by the analysis tool and puts them into a latex or gnuplot presentable form. It also calculates the *mean* and *standard deviations* for the results tables (the raw `csv` file leaves this task for an application such as *OpenOffice* to calculate).

An example of the output from this utility follows.

```
\begin{table}
\begin{center}
\begin{tabular}{|c|c|c|c|c|c|c|}
\hline
Edit Me & & Final Gen & Final Best Fit & Time(s) & Avg. Prog Size & Avg.
 Prog Fit\\
\hline
\multirow{5}{*}{Edit Me} &
Every 0 & $28.727 \pm 12.685$ & $0.006 \pm 0.015$ & $1.239 \pm 0.458$ &
$37.295 \pm 6.029$ & $391977.188 \pm 164428.188$\\ \hline
& Every 1 & $33.682 \pm 11.552$ & $0.018 \pm 0.050$ & $1.301 \pm 0.420$
 & $24.590 \pm 3.080$ & $335315.906 \pm 98895.508$\\ \hline
& Every 2 & $28.636 \pm 13.096$ & $0.006 \pm 0.013$ & $1.102 \pm 0.490$
 & $28.016 \pm 4.639$ & $401917.156 \pm 158858.312$\\ \hline
& Every 4 & $29.636 \pm 13.488$ & $0.006 \pm 0.018$ & $1.077 \pm 0.411$
 & $29.284 \pm 4.539$ & $388406.812 \pm 151176.547$\\ \hline
& Every 6 & $27.909 \pm 13.249$ & $0.002 \pm 0.005$ & $1.026 \pm 0.415$
 & $30.227 \pm 4.809$ & $409113.812 \pm 155984.328$\\ \hline
\hline
\end{tabular}
\caption{}
\end{center}
\end{table}

\begin{center}
\begin{verbatim}

Every 0
(+ (if<0 (+ (- f0 (+ 0.409400 f0)) -0.553533) 0.762133 (* (+ (% -0.274267
 f0) (+ f0 f0)) (if<0 (- f0 f0) (+ 0.691800 0.587400) (+ f0 -0.794400))))
 (+ (if<0 (* f0 -0.553533) (if<0 (+ 0.533533 0.807933) (- 0.587400
 -0.601733) f0) (- f0 0.056600)) (+ (* f0 f0) (% -0.362467 (if<0 0.845267
 f0 -0.108933))))),0.00013

...

0       10000000.000000 10000000.000000 10000000.000000 10000000.000000
1       18.268148       18.268148       18.268148       18.268148
2       11.468711       11.468711       11.468711       11.468711
3       7.838078        7.840739        7.838078        7.838078
4       5.700742        5.334342        5.754585        5.700742
5       4.249721        4.796186        3.392497        4.249721

...
```

Appendix C

Appendix III: Algebraic Simplification - Best Programs

Easy Regression

```
Every 0
(+ (if<0 (+ (- f0 (+ 0.409400 f0)) -0.553533) 0.762133 (* (+ (% -0.274267 f0)
 (+ f0 f0)) (if<0 (- f0 f0) (+ 0.691800 0.587400) (+ f0 -0.794400)))) (+ (if<0
 (* f0 -0.553533) (if<0 (+ 0.533533 0.807933) (- 0.587400 -0.601733) f0) (- f0
 0.056600)) (+ (* f0 f0) (% -0.362467 (if<0 0.845267 f0 -0.108933))))),0.00013

(+ (if<0 (+ (* f0 (+ f0 0.531267)) (if<0 (+ -0.718200 f0) f0 -0.136667))
 0.676467 (+ -0.315867 (% f0 f0))) (+ f0 (+ (* f0 f0) (% -0.362467 (if<0
 0.845267 f0 -0.108933))))),0.00013

(+ (if<0 (* f0 -0.553533) 0.762133 f0) (+ (if<0 (- f0 (* f0 -1.000000)) (if<0
 f0 0.676467 (* 0.944733 f0)) (- f0 f0)) (+ (* f0 f0) (% -0.362467 -0.109333))))
,0.000124

Every 1
(+ (* f0 f0) -0.017933),0.000107
(+ (+ 3.970196 (* f0 f0)) (- (- (- f0 -0.200133) f0) -3.648647)),0.000112
(+ (+ 3.970196 (* f0 f0)) (- f0 -0.019200)),0.000112

Every 2
(+ (- (+ (+ f0 0.923600) 1.000000) -2.093948) (- (+ f0 (if<0 -0.025733 0.813667
 (% 0.069533 f0))) -1.357800)),0.000102

(if<0 (- (- (if<0 f0 f0 -0.506533) (* f0 (% f0 0.062200))) (% (* (* f0 f0) f0)
 f0)) (+ (- (+ f0 0.558333) -2.093948) (- (* f0 f0) -1.357800)) (if<0 f0 -0.072467
 f0)),0.000102

(if<0 (- (- (if<0 f0 f0 -0.506533) (* (% f0 0.090333) (% f0 0.062200))) 0.371333)
 (+ (- (+ f0 0.558333) -2.093948) (- (* f0 f0) -1.357800)) (if<0 f0 -0.072467 f0))
,0.000102

Every 4
(+ 0.291074 (+ (+ 0.536400 f0) (- (* f0 f0) -3.161922))),0.000112

(+ 0.291074 (+ (+ 0.536400 f0) (- (* f0 f0) -3.161922))),0.000112

(+ 0.291074 (+ (+ 0.536400 f0) (- (* f0 f0) -3.161922))),0.000112

Every 6
(+ (+ (+ (+ f0 (* f0 f0)) (+ (- 0.843333 0.320200) (if<0 -0.108267 -0.684333
 -0.962733))) 1.625600) 2.524733),0.000118

(+ (+ (+ (+ f0 (* f0 f0)) (+ (- 0.843333 0.320200) (if<0 -0.108267 -0.684333
 -0.962733))) 1.625600) 2.524733),0.000118

(+ (+ (+ f0 f0) 0.848333) 3.114133),0.000118
```

Hard Regression

```
Every 0
(+ (if<0 (- (% f0 (* (+ (+ f0 f0) (- -0.914000 f0)) (% (+ f0 -0.247067) (-
 -0.089867 0.724333)))) (* 0.731200 0.065133)) (+ 0.609333 (if<0 (- (% (-
 f0 f0) -0.459533) (* 0.731200 0.065133)) (+ 0.609333 f0) (* (if<0 f0 f0
 0.579600) (* f0 f0)))) (* (if<0 f0 f0 0.579600) (* f0 f0))) (- (% (- (*
 (+ (% 0.972867 f0) f0) (% f0 f0)) (if<0 (* (- -0.224933 f0) 0.015933)
 0.568800 (- (- -0.367467 f0) f0))) (if<0 0.579600 -0.960733 -0.710800))
 (if<0 (* 0.617267 f0) (+ f0 f0) (% (% (- (% 0.060533 0.206467) (% 0.692067
 f0)) (if<0 (if<0 f0 f0 f0) (+ -0.282267 -0.397067) (* -0.073400 0.758067)))
 f0)))),4.36864

(+ (if<0 (- (% f0 -0.459533) (* 0.731200 0.065133)) (+ 0.609333 (if<0 (- (%
 (- f0 f0) -0.459533) (* 0.731200 0.065133)) (+ 0.609333 f0) (* (if<0 f0 f0
 0.579600) (* f0 f0)))) (* (if<0 f0 f0 0.579600) (* f0 f0))) (- (% (- (* (+
 (% 0.972867 f0) f0) (% f0 f0)) (if<0 (* (- -0.224933 f0) 0.015933) 0.568800
 (- (- -0.367467 f0) f0))) (if<0 0.579600 -0.960733 -0.710800)) (if<0 (*
 0.617267 f0) (+ f0 f0) (% (% (- (% 0.060533 0.206467) (% 0.692067 f0))
 (if<0 (if<0 f0 f0 f0) (+ -0.282267 -0.397067) (* -0.073400 0.758067)))
 f0)))),6.27138

(if<0 (* 0.617267 f0) (+ f0 f0) (% (% (- (- f0 f0) (% 0.692067 f0)) (if<0
 (if<0 f0 f0 f0) (+ -0.826333 -0.956667) (* -0.073400 0.758067))) f0)),4.36864

Every 1
(% (- -0.724333 (if<0 (* (- (+ f0 0.617400) (* f0 f0)) (+ f0 -0.894467)) (%
 (if<0 f0 f0 (% -0.637800 (* f0 f0))) -0.139747) (- (- (% f0 0.281800) f0)
 (* (* (* f0 f0) 0.427333) (- (- f0 0.334600) -0.624733))))) (if<0 (* (% f0
 0.034600) (* (* (* (* f0 f0) 0.427333) (- f0 -0.624733)) -0.188467)) (* (%
 0.393267 f0) (+ f0 (* (+ (% 0.202867 f0) f0) (* (% -0.006533 f0) (* f0 f0))
))) (+ f0 -0.894467))),9.33643

(% (- (+ f0 (- (+ (* (% -0.843267 f0) (- 0.757933 f0)) (% (+ 0.153867 f0) (-
 f0 -0.250333))) f0)) (if<0 (* 0.617400 (+ f0 -0.894467)) (% (if<0 f0 f0 (%
 -0.637800 (* f0 f0))) -0.139747) (- (- f0 -0.894467) (* (* (* f0 f0) 0.427333)
 (- f0 -0.624733))))) (if<0 (* (- f0 -0.624733) (* f0 -0.188467)) (* (%
 0.393267 f0) (+ f0 (* f0 (* (% -0.006533 f0) (* f0 f0))))) (+ f0 (* (%
 -0.006533 f0) (* f0 f0))))),9.33643

(% (- (+ f0 (- 0.872479 f0)) 0.691733) (if<0 (* (- f0 -0.624733) (* f0
 -0.188467)) (* (% 0.393267 f0) (+ f0 (* (+ (% 0.202867 f0) f0) (* (% -0.006533
 f0) (* f0 f0))))) (+ f0 -0.894467))),9.33643

Every 2
(+ (if<0 (- (% f0 -0.459533) 0.047625) (- (- (* -11.862152 (% (if<0 f0
 -0.405200 1.000000) (* f0 f0))) (- (% f0 -0.459533) 0.047625)) (+ -0.395400
 f0)) (* (if<0 f0 f0 (+ f0 -3.618123)) (* f0 f0))) (- (+ (- (if<0 (+ (*
 -0.663800 f0) 0.579600) 0.015980 (- (- 0.685933 f0) f0)) f0) 1.000000) (if<0
 (* f0 -0.663800) (if<0 (+ f0 f0) (+ f0 f0) 0.759467) (+ -0.684043 (% (+ (*
 0.648667 f0) f0) (if<0 f0 0.579600 f0)))))),0.561069
```

```
(+ (if<0 (- (% f0 -0.459533) 0.047625) (- (- (* -11.862152 (% (if<0 f0
 -0.405200 1.000000) (* f0 f0))) (- (% f0 -0.459533) 0.047625)) (+ -0.395400
 f0)) (* (if<0 f0 f0 (+ f0 -3.618123)) (* f0 f0))) (- (+ (- (if<0 (+ (*
 -0.663800 f0) 0.579600) 0.015980 (- (- 0.685933 f0) f0)) f0) 1.000000) (if<0
 (* f0 -0.663800) (if<0 (+ f0 f0) (+ f0 f0) 0.759467) (+ -0.684043 (% (+ (*
 0.648667 f0) f0) (if<0 f0 0.579600 f0)))))),0.561069

(+ (if<0 (- (% f0 -0.459533) 0.047625) (- (- (* -11.862152 (% (if<0 f0
 -0.405200 1.000000) (* f0 f0))) (- (% f0 -0.459533) 0.047625)) (+ -0.395400
 f0)) (* (if<0 f0 f0 (+ f0 -3.618123)) (* f0 f0))) (- (+ (- (if<0 (+ (*
 -0.663800 f0) 0.579600) 0.015980 (- (- 0.685933 f0) f0)) f0) 1.000000) (if<0
 (* f0 -0.663800) (if<0 (+ f0 f0) (+ f0 f0) 0.759467) (+ -0.684043 (% (+ (*
 0.648667 f0) f0) (if<0 f0 0.579600 f0)))))),0.561069

Every 4
(+ (if<0 (- (% f0 -0.459533) f0) (+ 0.609333 (+ (+ (+ (% 0.728200 f0) f0)
 0.007049) (% -4.955772 (* f0 (* 0.412867 f0))))) (* (if<0 f0 f0 0.579600) (*
 f0 (% (* -0.486267 (- f0 f0)) f0)))) (- (+ (- (if<0 (+ (% 0.728200 f0) f0)
 0.015980 -0.133067) f0) 0.809267) (if<0 (% (if<0 (+ -0.328800 f0) (* (% f0
 0.922800) (- 0.000533 f0)) 0.513733) (* (- f0 (if<0 f0 0.527200 f0)) (- (+
 -0.865200 f0) -0.498600))) (+ (+ (if<0 (- f0 0.875067) (- f0 0.776733)
 -0.783133) (+ (+ f0 f0) f0)) f0) (% 0.275200 (- f0 -0.989800))))),0.281126

(+ (if<0 (- (% f0 -0.459533) 0.047625) (+ f0 (+ (% (+ 0.609333 f0) f0) (%
 (if<0 (% 0.275200 f0) 1.000000 -4.955772) (* f0 (* 0.412867 f0))))) (* (if<0
 f0 f0 0.579600) (* (if<0 (+ -0.653933 f0) f0 -1.548328) f0))) (- (+ (- (if<0
 (+ f0 f0) (% (+ f0 f0) f0) -0.591867) f0) (+ (if<0 (* (* f0 0.932200) f0) (+
 f0 f0) f0) (- 0.382733 f0))) (if<0 (* 0.617267 f0) (+ 0.617267 f0) (%
 0.275200 (- -0.724383 f0))))),0.260796

(+ (if<0 (- (% f0 -0.459533) 0.047625) (+ f0 (+ (% (+ 0.609333 f0) f0) (%
 (if<0 (% 0.275200 f0) 1.000000 -4.955772) (* f0 (* 0.412867 f0))))) (* (+
 f0 f0) (* (if<0 (+ -0.653933 f0) f0 -1.548328) f0))) (- (+ (- (if<0 (+ f0
 f0) (% (+ f0 f0) f0) -0.591867) f0) (+ (if<0 (* (* f0 0.932200) f0) (+ f0
 f0) f0) (- 0.382733 f0))) (if<0 (* 0.617267 f0) (+ (+ (if<0 (- f0 -0.989800)
 (- f0 -0.132933) -1.200943) (+ (+ f0 f0) f0)) f0) (% 0.275200 (- -0.724383
 f0))))),0.328898

Every 6
(+ (if<0 (- (% f0 -0.459533) 0.047625) (+ 1.300351 (+ (- (% (- -0.412667 f0)
 (% f0 f0)) (+ (% f0 f0) (% f0 f0))) (% 0.808200 (* (* 0.085000 f0) (* f0
 -0.797333))))) (* (if<0 f0 f0 0.579600) (* f0 f0))) (- (+ (- (if<0 (%
 -0.554400 f0) 0.015980 (+ (% f0 -0.166333) 1.000000)) f0) (- (if<0 (*
 -0.524969 f0) 0.003275 (if<0 (if<0 f0 f0 -0.289800) (+ 0.310000 f0) 1.000000))
 (% f0 (* 3.323473 f0)))) (% (% (if<0 (% -0.814267 f0) 0.134219 -0.661267)
 (- (% (if<0 f0 0.328467 -0.936333) -0.225834) 1.128670)) (if<0 (* (if<0 (%
 -0.814267 f0) 0.134219 -0.661267) (* 3.323473 f0)) (% -0.745733 f0) -0.700533)
))),0.476262

(+ (if<0 (- (% f0 -0.459533) 0.047625) (+ 1.300351 (+ f0 (% 0.808200 (* (*
 0.085000 f0) (* f0 -0.797333))))) -0.797333) (- (+ (- (if<0 (* -0.348819 f0)
 0.015980 (+ (% f0 -0.166333) 1.000000)) f0) (- (if<0 (* -0.524969 f0) 0.003275
 (- (+ f0 f0) f0)) (% f0 (* f0 f0)))) (if<0 (+ (if<0 (- -0.148933 f0) (if<0 f0
 -0.463733 -0.785733) (% f0 0.218733)) (- (% f0 0.205067) (* f0 -0.630467))) (%
 (if<0 (% (* f0 0.914133) 0.015980) 0.215467 (+ (* f0 f0) (+ f0 f0))) (if<0 (*
 0.047163 f0) (+ (* f0 f0) (if<0 f0 -0.477200 -0.364067)) (+ f0 (% -0.745733 f0
)))) (% (+ -0.455933 (* -0.067203 f0)) f0)))),0.420479

(+ (if<0 (- (% f0 -0.459533) 0.047625) (+ 1.300351 (+ f0 (% 0.808200 (* (*
 0.085000 f0) (* f0 -0.797333))))) (* (if<0 f0 f0 0.579600) (* f0 f0))) (- (+
 (- (if<0 (% -0.554400 f0) 0.015980 (+ f0 1.000000)) f0) (- (if<0 (* -0.524969
 f0) 0.003275 (- (+ f0 f0) f0)) (% (* 0.085000 f0) (* f0 f0)))) (% (% (% (%
 -0.047467 f0) (% 0.554533 (+ f0 f0))) (- (% (if<0 f0 0.328467 -0.936333)
 -0.225834) 1.128670)) (* (+ (% 0.296867 f0) (+ f0 (* -0.182667 f0))) 0.127133)
))),0.330665
```

Coins Dataset

```
Every 0
(- (% (if<0 (if<0 f2 (+ 0.069133 0.188200) (if<0 f5 0.261400 0.316067))
 0.755667 (* -0.180333 (- f1 0.261400))) 0.904267) (if<0 (* (- (* f6
 0.224133) (if<0 -0.356267 0.076467 -0.383400)) 0.224133) -0.180333
 0.418067)),0.992188

(- (% (if<0 (if<0 f2 (+ 0.069133 0.188200) (if<0 f5 0.261400 0.316067))
 0.755667 (* -0.180333 (- f1 0.261400))) 0.904267) (if<0 (* (- (* f6
 0.224133) (if<0 -0.356267 0.076467 -0.383400)) 0.224133) -0.180333
 0.418067)),0.992188

(- (% (if<0 (if<0 f2 (+ 0.069133 0.188200) (if<0 f5 0.261400 0.316067))
 0.755667 (* -0.180333 (- f1 0.261400))) 0.904267) (if<0 (* (- (* f6
 0.224133) (if<0 -0.356267 0.076467 -0.383400)) 0.224133) -0.180333
 0.418067)),0.992188


Every 1
(* (% (- -0.775733 (% f4 -0.462933)) (if<0 (+ f3 (% f6 f2)) (- (+ f7 f7)
 (% f3 f6)) (+ (if<0 f0 -0.846400 0.859467) (- f2 -0.113733)))) (if<0
 (+ (+ (- f2 -0.113733) 1.000000) (% f1 0.169933)) 0.169933 (- (-
 0.369267 (* f5 0.254600)) (* (* -0.720733 f3) 0.292600)))),0.992188

(* (% (- -0.775733 (% f4 -0.462933)) (if<0 (+ f3 (% f6 f2)) (- (+ f7 f7)
 (% f3 f6)) (+ (if<0 f0 -0.846400 0.859467) (- f2 -0.113733)))) (if<0 (+
 (+ (- f2 -0.113733) 1.000000) (% f1 0.169933)) 0.169933 (- (- 0.369267
 (* f5 0.254600)) (* (* -0.720733 f3) 0.292600)))),0.992188

(if<0 (* 0.763400 f6) (if<0 (- (% f3 -0.720733) (if<0 (- f5 0.825467)
 0.373511 (+ f7 f7))) -0.462933 0.412600) (+ f3 (* -0.855267 f3)))
,0.992188

Every 2
(if<0 (% (- f6 (if<0 f3 (- 0.431600 f1) -0.648067)) (- (- f6 (* f6
 0.474733)) (- 0.003160 f6))) (* 0.003160 f5) (if<0 (% 0.289333 (if<0
 (- f7 0.516733) -0.205333 -0.473000)) (* (* (- f6 f2) f7) -0.941965)
 (- (% (+ f5 0.103667) f5) (if<0 f1 f0 0.681067)))),0.992188

(if<0 (% (- f6 (if<0 f3 (- 0.431600 f1) -0.648067)) (- (- f6 f2) (-
 0.431600 f6))) (* 0.003160 f5) (if<0 (% 0.289333 (if<0 (- f7 0.516733)
 -0.205333 (* 0.516733 f7))) (* (* 0.516733 f3) -0.941965) (* (- f7
 0.274078) (+ -0.789873 f5)))),0.992188

(if<0 (% (- f6 (if<0 f3 0.289333 -0.648067)) (- (- f6 f2) (- 0.431600
 f6))) (* 0.003160 f5) (if<0 (% 0.289333 (if<0 (- f7 0.516733) -0.310867
 (- f3 0.994600))) (* (* 0.516733 f7) -0.941965) (- (% 0.535267 f5)
 (if<0 f1 f0 0.681067)))),0.992188

Every 4
(if<0 (- (* (% f6 f1) -0.587423) -0.295655) (* (% (- (- -0.203667 f4)
 0.927267) -1.055533) (* -0.158200 (- (- f6 f5) (* f3 -0.740267)))) (*
 -0.587423 f7)),0.992188

(if<0 (if<0 (* 0.887867 (* (+ f3 -0.387667) (+ 0.424333 f3))) (+ (% (%
 f0 f2) (+ f3 -0.495000)) (% (+ -0.922000 f4) 0.365133)) (+ f4 f6)) (*
 -0.352110 (if<0 f7 -0.740267 f6)) (* f3 0.052200)),0.992188

(if<0 (- (* (% -0.584733 f1) (if<0 (+ f4 -0.221533) (- -0.648067 f7) f0))
 -0.295655) (* (% f7 -1.055533) (* -0.158200 (- (- f6 f5) (* f3
 -0.740267)))) (% -0.584733 f1)),0.992188

Every 6
(- (% (* -0.100933 (- f1 0.575133)) 0.571400) (if<0 (* (if<0 (if<0 f7
 f0 0.803933) -0.654600 (% f5 -0.698933)) 0.224133) -0.180333 0.418067))
,0.992188

(- (% (* -0.100933 (- f1 0.575133)) 0.571400) (if<0 (* (if<0 (if<0 f7 f0
 0.803933) -0.654600 (% f5 -0.698933)) 0.224133) -0.180333 0.418067))
,0.992188

(- (% (if<0 (if<0 3.837545 0.257333 0.316067) f1 (* -0.100933 (- f1
 0.575133))) 0.571400) (if<0 (* f6 3.837545) -0.180333 (if<0 (* f6
 0.224133) (- -1.250067 (if<0 f6 -0.650667 0.211200)) 0.418067)))
,0.992188
```

Faces Dataset #1

```
Every 0
(% (if<0 (- (% -0.428067 f4) (* 0.801200 -0.971133)) (if<0 (+ (+ -0.344333
 f5) (if<0 0.921400 0.429133 -0.414133)) (if<0 f0 (- 0.890733 0.627133)
 -0.736267) (+ (+ -0.344333 f13) 0.097600)) (- (- 0.401867 0.712067) (*
 0.351400 f0))) (+ -0.736267 (% 0.440067 (- (if<0 0.689200 0.440067
 -0.347800) 0.965800)))),0.996528

(% (if<0 (- (% -0.428067 f4) (* 0.801200 -0.971133)) (if<0 (+ (+ -0.344333
 f5) (if<0 0.921400 0.429133 -0.414133)) (if<0 f0 (- 0.890733 0.627133)
 -0.736267) (+ (+ -0.344333 f13) 0.097600)) (- (- 0.401867 0.712067) (*
 0.351400 f0))) (+ -0.736267 (% 0.440067 (- (if<0 0.689200 0.440067
 -0.347800) 0.965800)))),0.996528

(% (if<0 (- (% -0.428067 f4) (* 0.801200 -0.971133)) (if<0 (+ (+ -0.344333
 f5) (if<0 0.921400 0.351400 -0.414133)) (if<0 f1 (- 0.890733 0.627133)
 -0.736267) f10) (- (- (* 0.801200 -0.971133) 0.712067) (* 0.351400 f0)))
 (+ -0.736267 (% 0.440067 (- (if<0 0.689200 f0 -0.347800) 0.965800))))
,0.996528

Every 1
(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 f0 0.351400 -0.832400) (-
 -0.310200 (* 0.351400 f0))) -1.185417),0.996528

(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 f0 0.351400 -0.832400) (-
 -0.310200 (* 0.351400 f0))) -1.185417),0.996528

(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 f0 0.351400 -0.614400) (-
 -0.310200 (* 0.351400 f0))) (+ (- (+ 0.050090 f14) (* f14 f1)) (+
 0.050090 f14))),0.996528

Every 2
(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 (* 0.351400 f0) 0.253133
 (- f4 1.000000)) (- -0.310200 (* 0.351400 f0))) -1.288493),0.996528

(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 (* 0.351400 f0) 0.253133
 (- f4 1.000000)) (- -0.310200 (* 0.351400 f0))) -1.288493),0.996528

(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 (* 0.351400 f0) 0.253133
 (- f4 1.000000)) (- -0.310200 (* 0.351400 f0))) -1.288493),0.996528

Every 4
(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 f0 0.298267 -0.778667) (-
 -0.310200 (* 0.351400 f0))) -1.149821),0.996528

(% (if<0 (- (% -0.428067 f4) -0.778072) (if<0 f0 0.298267 -0.778667) (-
 -0.310200 (* 0.351400 f0))) -1.149821),0.996528

(% (if<0 (- (% -0.428067 (% -0.428067 f4)) -0.778072) (* (if<0 f14
 -0.487133 -0.536592) (if<0 f4 (* -0.104800 f4) 0.298267)) (- -0.310200
 (* 0.351400 f0))) -1.300260),0.996528

Every 6
(* (if<0 f4 (* (* (* 0.373733 f14) 0.202133) (+ (* f11 0.097200) (-
 -0.362400 f2))) 0.298267) (if<0 (+ (% 0.129335 f4) -0.564533) (if<0
 (% -0.360733 (* -0.696200 f6)) (if<0 (+ 0.989133 f4) (* f6 -0.519133)
 (- -0.307200 f4)) (if<0 (+ f0 -0.747400) 0.684606 1.871153)) (+ (- f0
 -0.681400) (+ (* f0 -0.519133) (- f0 -0.681400))))),0.947917

(* (if<0 f4 (* (* (* 0.373733 f14) 0.202133) (+ (* f11 f7) (- 0.974533
 f14))) 0.298267) (if<0 (+ (% -0.008933 f4) -0.564533) (if<0 (% -0.360733
 (* -0.696200 f6)) (- -0.307200 f4) (if<0 (+ f4 -0.747400) 0.684606
 1.871153)) (+ (% (* f9 f17) (- -0.008933 0.038600)) (+ (- f6 -0.374067)
 (- f0 f4))))),0.947917

(* (if<0 f4 (* (* (* 0.373733 f14) 0.202133) (+ (* f11 1.871153)
 0.989539)) 0.298267) (if<0 (+ (% 0.129335 f4) -0.564533) (if<0 (- (-
 f17 0.913867) (- -0.162600 f13)) (if<0 (+ 0.989133 f4) -0.274200 (-
 -0.307200 f4)) (if<0 (+ f0 -0.747400) 0.684606 1.871153)) (+ (% (% f3
 0.311200) f6) (+ (* -0.307200 0.298267) (- f0 -0.681400))))),0.947917
```

Faces Dataset #2

```
Every 0
(- (* (* -0.056600 f19) (- (+ (+ f0 0.756200) (+ f4 -0.670867)) (if<0
 0.151467 (% -0.713333 f13) -0.056600))) (if<0 (if<0 (* -0.056600 (-
 -0.056600 f0)) f18 -0.713333) (if<0 f18 -0.157200 -0.713333) 0.151467))
,0.996528

(- (* (* -0.056600 f19) (- (+ (+ f0 0.756200) (+ f4 -0.670867)) (if<0
 0.151467 (% -0.713333 f13) -0.056600))) (if<0 (if<0 (* -0.056600 (-
 -0.056600 f0)) f18 -0.713333) (if<0 f18 -0.157200 -0.713333) 0.151467))
,0.996528

(* (- (+ (% f2 (if<0 -0.538600 0.604733 f6)) (% (% 0.606667 0.261867) (*
 f6 0.841267))) f4) (if<0 (- (+ -0.953600 -0.039000) (- -0.200533 f19))
 (if<0 f18 0.689200 (if<0 -0.039000 -0.567667 (if<0 -0.200533 -0.285067
 0.298267))) (+ (+ 0.987333 f0) f9))),0.996528

Every 1
(if<0 (* 0.725933 (+ (+ (* f17 0.351667) -0.108933) (* (% f19 0.652400)
 -1.423800))) (* (- 0.918267 (* 0.107333 f2)) (if<0 f0 -0.139467
 0.657067)) (if<0 (% (% f19 0.104200) (+ f0 -0.577600)) f16 (- (- f19
 -0.139467) -0.849133)))),0.996528

(if<0 (* (% f19 0.652400) -1.423800) (* (- 0.918267 (* 0.107333 f2))
 (if<0 (if<0 f11 f0 (+ f0 0.725933)) f0 0.657067)) (if<0 (% (% f19
 0.104200) (+ f0 -0.577600)) -0.110333 (- (if<0 -0.664733 (- f6 f19)
 (+ f18 0.524467)) -0.766667)))),0.996528

(if<0 (* 0.351667 (+ (+ (* f8 0.351667) (* f16 f7)) (* (% f1 0.313133)
 -1.423800))) (* (- 0.918267 (* 0.107333 f2)) (if<0 (+ f0 -0.577600)
 -0.139467 0.657067)) (if<0 (- (- f2 -0.748733) (if<0 f18 f2 -0.190850))
 f16 (- (- f18 -0.139467) -0.849133)))),0.996528

Every 2
(if<0 (+ (+ -0.064070 f18) 0.215200) (+ f19 0.961933) (* 0.554667 (if<0
 (- f2 (* f18 -0.582000)) (+ (+ f0 f19) -0.443771) 0.853800))),0.996528

(if<0 (+ (+ (+ -0.064070 f18) f4) (+ (+ f0 (+ f9 0.554667)) f18)) (+ f19
 0.961933) (* 0.554667 (if<0 f6 -0.328674 0.961933))),0.996528

(if<0 (+ (+ -0.064070 f18) 0.215200) (+ f19 0.961933) (* 0.554667 (if<0
 (- f2 (* f18 -0.582000)) (+ (+ f0 f19) -0.443771) 0.853800))),0.996528

Every 4
(if<0 (* (* 0.184400 f19) -0.782867) (if<0 (% (- (if<0 f2 -0.923800
 0.533800) -0.164600) (- (* -0.905267 f19) (+ f5 0.962267))) 0.538533
 (if<0 (- (* -0.444933 f18) (* f19 f7)) (* (if<0 f19 f19 0.622467)
 0.981867) (if<0 (if<0 f3 f0 0.882867) -0.213733 0.763200))) (+ 0.934867
 f18)),0.996528

(if<0 (* (* 0.184400 f19) -0.782867) (if<0 (% (- (if<0 f2 -0.923800
 0.533800) -0.164600) (- (* -0.905267 f19) (+ f5 0.962267))) 0.538533
 (if<0 (- (* -0.444933 f18) (* f19 f7)) (* (if<0 f19 f19 0.622467)
 0.981867) (if<0 (if<0 f3 f0 0.882867) -0.213733 0.763200))) (+ 0.934867
 f18)),0.996528

(if<0 (* (* 0.184400 f19) -0.782867) (if<0 (% (- (if<0 f2 -0.923800
 0.533800) -0.164600) (- (* -0.905267 f19) (+ f5 0.962267))) 0.538533
 (if<0 (- (* -0.444933 f18) (* f19 f7)) (* (if<0 f19 f19 0.622467)
 0.981867) (if<0 (if<0 f3 f0 0.882867) -0.213733 0.763200))) (+ 0.934867
 f18)),0.996528

Every 6
(if<0 (* (if<0 (+ 0.024733 (% 0.603867 f18)) (+ (* f13 f13) f18)
 0.603867) 0.460733) (+ 0.934867 f18) (if<0 (- f0 0.603867) -0.262272
 (* 0.274400 (+ 0.931467 (- f6 -0.798267))))),0.996528

(if<0 (* (if<0 (+ 0.024733 (% 0.603867 f18)) (+ (* f13 -0.185600) f18)
 0.603867) 0.460733) (+ 0.934867 f18) (if<0 (- f0 0.603867) -0.262272 (*
 0.274400 (+ 0.931467 (% 0.947933 f18))))),0.996528

(if<0 (* 0.274400 (+ (if<0 (% f5 f7) f6 (if<0 f6 f6 -0.202467)) (% (-
 0.324267 -0.673533) f18))) (+ 0.934867 f18) (if<0 (- f0 0.603867)
 -0.262272 (* 0.274400 (+ 0.934867 (% 0.947933 f18))))),0.996528
```

Appendix D

Appendix IV: Prime Number Simplification - Best Programs

Easy Regression

```
Every 0
(+ (% (if<0 (% -0.791133 (* 0.184067 -0.508333)) (* (- f0 f0) (* f0 f0)) (+ (+ f0
 f0) f0)) f0) (+ (+ (* f0 f0) (% (- f0 f0) (% f0 f0))) f0)),0.00016

(+ (+ (if<0 (* 0.164000 -0.312133) (* 0.164000 -0.312133) (- -0.750333 (- f0
 0.463600))) (- (- (if<0 f0 0.469067 f0) (if<0 f0 f0 f0)) (if<0 (* f0 f0) (if<0 f0
 0.420067 f0) (* f0 f0)))) (- (* f0 f0) (+ (% 0.697667 -0.476933) (- -0.969200 f0))))
,0.000205

(+ (if<0 (if<0 (% (+ f0 f0) -0.850533) (% f0 f0) (if<0 0.434333 0.465467 0.784667))
 (if<0 (if<0 (if<0 -0.450867 -0.925467 f0) (if<0 f0 0.397600 f0) (% -0.111133 f0))
 (% f0 (+ 0.261067 0.133667)) (+ f0 f0)) (% -0.482533 -0.253600)) (+ 0.600733 (+ (-
 (* f0 f0) (if<0 -0.627533 -0.443200 -0.854333)) (+ 0.933733 f0)))),0.000138

Every 1
(+ 2.582333 (+ (* f0 f0) (+ f0 1.428980))),0.000128

(+ 2.582333 (+ (* f0 f0) (+ f0 1.428980))),0.000128

(+ 2.582333 (+ (* f0 f0) (+ f0 1.428980))),0.000128

Every 2
(- (+ (* f0 f0) (+ 3.186244 f0)) -0.802933),0.000117

(if<0 (- (% 0.190781 f0) 0.429000) (- (+ (* f0 f0) (+ 3.186244 f0)) -0.802800) (%
 (* -0.865067 (+ -0.388867 f0)) (- (% f0 0.958667) (% -0.432333 f0)))),0.000117

(- (+ (* f0 f0) (+ 3.186244 f0)) -0.802933),0.000117

Every 4
(+ (if<0 (% (+ 0.045467 f0) f0) (+ 2.896934 f0) (% (+ 0.604400 f0) f0)) (+ (* f0
 f0) (+ 0.160154 f0))),0.000137

(+ (* f0 f0) (+ (+ 0.999572 f0) (if<0 (% (+ f0 f0) f0) (+ 2.230156 (if<0 f0
 0.604400 f0)) (% (+ (+ f0 f0) f0) f0)))),0.000137

(+ (if<0 (% (+ 0.045467 f0) f0) 2.845800 (% (+ (+ f0 f0) f0) f0)) (+ (+ 1.011821
 f0) (* f0 f0))),0.000137
```

```
Every 6
(+ 4.010031 (+ (* f0 f0) f0)),0.0001

(+ (+ f0 3.676363) (+ (* f0 f0) f0)),0.0001

(+ 4.010031 (+ (* f0 f0) f0)),0.000101
```

Hard Regression

```
Every 0
(+ (+ -0.874800 (* (+ -0.613600 (* (if<0 (* -0.291667 f0) (% 0.876800 f0) (if<0
 f0 f0 f0)) (if<0 (% f0 f0) f0 (- f0 -0.505267)))) f0)) (if<0 (- (% 0.718000 f0)
 0.055800) (% (- (- (if<0 (+ f0 0.009267) (% 0.892267 0.539533) (* f0 -0.383400))
 -0.291667) (if<0 (* (* 0.334467 f0) f0) (+ (* f0 f0) 0.754533) (* (% 0.896800
 0.121133) (+ f0 f0)))) (- 0.433600 f0)) (* (% (* (if<0 (* -0.900467 f0) (%
 -0.774400 f0) (if<0 f0 f0 f0)) (- (+ 0.114667 0.936000) (- f0 f0))) f0) (- (*
 (* (% f0 -0.616533) (- -0.752467 0.038067)) (% (% 0.718000 0.084067) (* f0
 0.777600))) (* (- 0.260867 0.718000) (% (if<0 -0.564467 0.190400 -0.480067) 0.261733)))))

(+ (+ (- (if<0 (+ (% (* -0.717400 -0.588533) (- f0 f0)) (+ (if<0 f0 f0 0.864467)
 (- 0.883933 f0))) (+ (+ (% f0 f0) (* f0 -0.375933)) (% (% 0.358267 f0) (- f0
 f0))) (- (+ (if<0 0.716133 f0 -0.816467) (if<0 f0 f0 0.736533)) (if<0 (+ f0 f0)
 (+ f0 f0) (* 0.205000 f0)))) (% (if<0 (if<0 (* f0 f0) (% f0 f0) (* 0.541600
 0.062067)) (if<0 (if<0 f0 -0.140733 -0.543733) (if<0 0.144400 f0 0.245400)
 (if<0 f0 -0.147200 f0)) (if<0 (* -0.019133 -0.046333) (% 0.206267 f0) (* f0
 -0.007000))) (% (+ (- f0 0.326733) (* f0 f0)) (+ (- 0.465067 0.569800) (*
 0.747400 f0))))) (* (+ -0.613600 (* (if<0 (* -0.291667 f0) (% 0.718000 f0)
 (if<0 f0 f0 -0.505267)) (if<0 (% f0 f0) (+ 0.084067 f0) (- f0 -0.505267))))
 f0)) (if<0 (- f0 (% f0 -0.506667)) (% (if<0 (* (- 0.260867 (if<0 f0 f0
 -0.505267)) (% 0.919333 -0.781133)) 0.870667 f0) (- 0.433600 0.231000))
 (* (% (% f0 f0) f0) (- (- (% (* f0 -0.777133) (* f0 f0)) (% (if<0 0.038067
 -0.992533 0.986467) (* f0 0.089467))) -0.291667)))),6.99765

(+ (+ -0.874800 (* (+ -0.616533 (* (if<0 (* -0.291667 f0) (% -0.122600 f0)
 (if<0 f0 f0 f0)) (if<0 (% -0.506667 f0) f0 (- f0 -0.505267)))) f0)) (if<0
 (- (% 0.718000 f0) f0) (% (- (- (if<0 (+ f0 0.009267) (% 0.892267 0.539533)
 (* f0 -0.383400)) (% (- 0.153867 0.740200) (+ f0 f0))) (if<0 (* (* 0.334467
 f0) f0) (+ 0.718000 0.754533) (* (% 0.896800 0.121133) (+ f0 f0)))) (-
 0.433600 f0)) (* (% (* (if<0 (* -0.900467 f0) (% -0.774400 f0) (if<0 f0
 f0 f0)) (- (+ 0.114667 0.936000) (- f0 f0))) f0) (- (* (* (% f0 -0.616533)
 (- -0.752467 0.038067)) (% (% 0.718000 0.084067) (* f0 0.777600))) (* (-
 0.260867 (% f0 -0.506667)) f0))))),7.34199

Every 1
(+ -0.407600 (+ (* -0.977533 (if<0 (if<0 (* (- f0 -0.505267) (- f0
 0.513067)) (% f0 -0.314733) (if<0 (* f0 0.890000) 0.036303 -0.713253)) (-
 (* 1.766469 (- (% 0.561733 f0) f0)) (% (% 0.899867 f0) (* f0 -0.076067)))
 (+ (- (+ f0 f0) (if<0 f0 f0 0.509133)) (* 0.503667 f0)))) (* f0 (+ -0.820000
 (* (if<0 (* -0.291667 f0) (% -0.774400 f0) f0) (- f0 -0.505267)))))),29.8924

(+ -0.407600 (+ (* -0.977533 (if<0 (if<0 (* (- f0 -0.505267) (- f0 0.513067))
 (% f0 -0.314733) (if<0 (* f0 0.890000) 0.036303 -0.713253)) (- (* 1.766469
 (- (% 0.561733 f0) f0)) (% (% 0.899867 f0) (* f0 -0.076067))) (+ (- (+ f0
 f0) (if<0 f0 f0 0.509133)) (* 0.503667 f0)))) (* f0 (+ -0.820000 (* (if<0
 (* -0.291667 f0) (% -0.774400 f0) f0) (- f0 -0.505267)))))),29.8924

(+ (- f0 -0.505267) (+ (* -0.977533 (if<0 (if<0 (* -1.004890 (- f0 0.513067))
 (* f0 -0.311667) 0.890000) (- (* 1.766469 (- -0.505267 f0)) (% (% 0.561733
 f0) (* f0 -0.076067))) f0)) (* -0.977533 (if<0 (if<0 (* (- f0 -0.505267)
 (- f0 0.513067)) (% f0 -0.314733) (if<0 (* f0 0.890000) 0.036303 -0.713253))
 (- (* 1.766469 (% (% f0 0.687800) (* f0 f0))) (% (% 0.899867 f0) (* f0
 -0.076067))) (+ (- f0 (if<0 f0 f0 0.509133)) (* 0.503667 f0))))))
,30.6921

Every 2
(if<0 (if<0 f0 0.448498 (+ -0.685800 (+ (* -1.657868 (+ f0 f0)) 0.855352)))
 (% (+ (if<0 f0 (if<0 f0 0.242067 (if<0 (+ f0 -0.657000) 1.000000 0.542171))
 f0) (* 15.486235 (- 0.141733 (* 2.229490 f0)))) (* f0 f0)) (* (+ (- (-
 0.685800 (* (% 0.164067 f0) (% f0 0.045400))) (- (- (* -0.132867 f0) f0)
 -0.691000)) (* f0 f0)) (- -0.996045 f0))),1.33094

(if<0 (if<0 f0 0.448498 (+ -0.685800 (+ (* -1.657868 (+ f0 f0)) 0.855352)))
 (% (+ (if<0 f0 (if<0 f0 0.242067 (if<0 (+ f0 -0.657000) 1.000000 0.542171))
 f0) (* 15.486235 (- 0.141733 (* 2.229490 f0)))) (* f0 f0)) (* (+ (- (-
 0.685800 (* (% 0.164067 f0) (% f0 0.045400))) (- (- (* -0.132867 f0) f0)
 -0.691000)) (* f0 f0)) (- -0.996045 f0))),1.33094

(if<0 (if<0 f0 0.448498 (+ -0.685800 (+ (* -1.657868 (+ f0 f0)) 0.855352)))
 (% (+ (if<0 f0 (if<0 f0 0.242067 (if<0 (+ f0 -0.657000) 1.000000 0.542171))
 f0) (* 15.486235 (- 0.141733 (* 2.229490 f0)))) (* f0 f0)) (* (+ (- (-
 0.685800 (* (% 0.164067 f0) (% f0 0.045400))) (- (- (* -0.132867 f0) f0)
 -0.691000)) (* f0 f0)) (- -0.996045 f0))),1.33094

Every 4
(if<0 (* (+ -0.871333 f0) (* (+ 1.000000 f0) (+ f0 f0))) (* (- (* 0.424733
 (% (if<0 (% f0 -0.527533) -1.115000 f0) (* (* f0 f0) -0.023067))) (* f0
 f0)) (if<0 (% -0.191400 (* (* f0 f0) -0.768267)) (* f0 -0.889867) (% f0
 (% f0 (- -0.521867 (if<0 f0 f0 0.025867)))))) (- (% (if<0 (if<0 (* f0 (-
 -0.989067 f0)) f0 (- 0.593334 f0)) 1.000000 (if<0 (+ (- -0.088933 f0) (+
 f0 -0.135333)) 3.243011 (- (+ f0 f0) (if<0 f0 -0.248067 -0.016867)))) (-
 0.696267 f0)) (- (- f0 0.322423) f0))),26.2377

(if<0 (* (+ -0.871333 f0) (* (+ 1.000000 f0) (+ f0 f0))) (* (- (* 0.424733
 (% (if<0 (% f0 -0.527533) -1.115000 f0) (* (* f0 f0) -0.023067)))
 (* f0 f0)) (if<0 (% -0.191400 (* (* f0 f0) -0.768267)) (* f0 -0.889867)
 (% f0 (% f0 (- -0.521867 (if<0 f0 f0 0.025867)))))) (- (% (if<0 (if<0 (*
 f0 (- -0.989067 f0)) f0 (- 0.593334 f0)) 1.000000 (if<0 (+ (- -0.088933
 f0) (+ f0 -0.135333)) 3.243011 (- (+ f0 f0) (if<0 f0 -0.248067
 -0.016867)))) (- 0.696267 f0)) (- (- f0 0.322423) f0))),26.2377

(if<0 (* (+ -0.871333 f0) (* (+ 1.000000 f0) (+ f0 f0))) (* (- (* 0.424733
 (% (if<0 (% f0 -0.527533) -1.115000 f0) (* (* f0 f0) -0.023067))) (* f0
 f0)) (if<0 (% -0.191400 (* (* f0 f0) -0.768267)) (* f0 -0.889867) (% f0
 (% f0 (- -0.521867 (if<0 f0 f0 0.025867)))))) (- (% (if<0 (if<0 (* f0 (-
 -0.989067 f0)) f0 (- 0.593334 f0)) 1.000000 (if<0 (+ (- -0.088933 f0) (+
 f0 -0.135333)) 3.243011 (- (+ f0 f0) (if<0 f0 -0.248067 -0.016867)))) (-
 0.696267 f0)) (- (- f0 0.155437) f0))),25.6071

Every 6
(if<0 (+ -0.140534 (if<0 f0 f0 (- f0 -0.990267))) (* (- (* f0 f0) (+
 3.003267 f0)) (% f0 0.920467)) (- (+ (- f0 (+ (- (+ -0.871200 f0)
 -0.923467) f0)) (+ -2.723933 (if<0 f0 -0.196667 (- f0 -0.196667)))) (+
 -3.265909 (% (% 1.291734 (if<0 f0 -0.512933 0.108733)) (if<0 (% f0
 -0.624333) (* f0 f0) (if<0 f0 f0 0.218867)))))),4.83683

(if<0 (+ -0.140534 (if<0 f0 f0 (- f0 -0.990267))) (* (- (* f0 f0) (+
 3.003267 f0)) (% f0 0.920467)) (- (+ (- f0 (+ (- (+ -0.871200 f0)
 -0.923467) f0)) (+ -2.723933 (if<0 f0 -0.196667 (- f0 -0.196667)))) (+
 -3.265909 (% (% 1.291734 (if<0 f0 -0.512933 0.108733)) (if<0 (% f0
 -0.624333) (* f0 f0) (if<0 f0 f0 0.218867)))))),4.83683

(if<0 (+ -0.140534 (if<0 f0 f0 (- f0 -0.990267))) (* (- (* f0 f0) (+
 2.721467 f0)) (% f0 0.920467)) (- (+ (- f0 (+ (- (* f0 0.949333)
 -0.923467) f0)) (+ (- -1.446133 (* 0.234133 f0)) (+ (if<0 f0 (-
 -0.245800 f0) (- f0 -0.196667)) -0.739267))) (+ (+ 1.109734 (* (% f0
 0.171467) (% -0.739067 f0))) (% (% 1.291734 (if<0 f0 -0.512933
 0.108733)) (if<0 (% f0 -0.624333) (* f0 f0) (if<0 f0 f0 0.218867)))
))),4.79333
```

Coins Dataset

```
Every 0
(+ (- (if<0 (* f7 0.461333) (* -0.248000 -0.442467) (* -0.828800 0.461333)) (*
 (- f5 0.522600) (if<0 -0.746733 0.128733 f5))) (if<0 (if<0 (% (% 0.215000 f7)
 -0.011000) (if<0 0.117933 (+ f6 -0.634333) f5) -0.676333) (* (% 0.171867
 -0.634333) (* f0 0.195800)) (% (% -0.373133 (if<0 0.226533 f4 0.637067)) (% (*
 f0 f2) -0.676333)))),0.953125

(+ (- (if<0 (* f7 0.461333) (* -0.248000 -0.442467) (* -0.828800 0.461333)) (*
 (- f5 0.522600) (if<0 -0.746733 0.128733 f5))) (if<0 (if<0 (% (% 0.215000 f7)
 -0.011000) (if<0 0.117933 (+ f6 -0.634333) f5) -0.676333) (* (% 0.171867
 -0.634333) (* f0 0.195800)) (% (% -0.373133 (if<0 0.226533 f4 0.637067)) (% (*
 f0 f2) -0.676333)))),0.953125

(+ (- (if<0 (* f7 0.461333) (* -0.248000 -0.442467) (* -0.828800 0.461333)) (*
 (- f5 0.522600) (if<0 -0.746733 0.128733 f5))) (if<0 (if<0 (% (% 0.215000 f7)
 -0.011000) (if<0 0.117933 (+ f6 -0.634333) f5) -0.676333) (* (% 0.171867
 -0.634333) (* f0 0.195800)) (% (% -0.373133 (if<0 0.226533 f4 0.637067)) (%
 (* f0 f2) -0.676333)))),0.953125

Every 1
(if<0 (* (- (* f3 f6) f6) (% f3 -0.806867)) (% 0.112933 (- (if<0 (% -1.078400
 f3) (% -0.466467 f7) -0.250600) (% f3 -0.806867))) (- (if<0 (% -0.218067 f3)
 (* f1 -0.237467) 0.584577) (if<0 (- (+ f7 -0.242600) (* 0.244133 f7)) (* f1
 -0.242600) (* 0.244133 f7)))),0.960938

(if<0 (* (- (* f3 f6) f6) (+ (* f3 (+ f2 f5)) 0.977134)) (% 0.112933 (- (if<0
 (% f4 f3) (+ f1 -0.794267) -0.250600) (% f3 -0.806867))) (- (if<0 (% -0.218067
 (if<0 f4 f0 f3)) (* f5 0.244133) 0.584577) (if<0 (- (+ f6 f0) (% f7 -0.109133))
 (* f5 -0.242600) (if<0 (- 0.610667 f0) (- f6 f3) (- f1 0.939067))))),0.960938

(if<0 (* (- (* f3 f6) f6) (+ f7 0.977134)) (% 0.112933 (- -0.305867 (% f3
 -0.806867))) (- (if<0 (% -0.218067 f3) (* f5 -0.237467) 0.584577) (if<0 (-
 -0.473600 (% f7 -0.109133)) (* f5 -0.242600) (* 0.244133 f7)))),0.960938

Every 2
(* (- -1.212734 (* f3 f7)) (* (if<0 (if<0 (- f5 f7) f5 (* f7 f0)) 0.364600 (*
 (* 0.932733 f7) (- f7 f1))) f7)),0.929688

(* (if<0 (if<0 (- -1.212734 f7) f5 (- f5 0.364600)) 0.364600 (* (* f3 f7)
 -1.212734)) (* (- -1.212734 (* f5 (* (- f7 f1) f7))) f7)),0.929688

(* (- -1.212734 (* (if<0 (% f3 f3) (- -0.466467 -0.002000) (- f4 -0.347133))
 f7)) (* (if<0 (if<0 (- 0.364600 f7) -0.777667 (+ f3 f6)) 0.364600 (* f7 (* f3
 f7))) f7)),0.929688

Every 4
(- (* (- -1.212734 (* f5 f7)) (* (- -1.530467 f2) (* f7 -0.289333))) (* 0.044259
 (if<0 (if<0 f6 (- f7 f2) f5) (* f7 f1) (- -2.428534 (* f7 f2))))),0.945312

(- (* (- -1.212734 (* f5 f7)) (* (- -1.530467 f2) (* f7 -0.289333))) (* 0.044259
 (if<0 (if<0 f6 (- f7 f2) f5) (* f7 f1) (- -2.428534 (* f7 f2))))),0.945312

(- (* (- -1.212734 (* f5 0.044259)) (* (- -1.530467 f2) (* f7 -0.289333))) (*
 (if<0 (if<0 f6 (- f7 f2) f5) (* f7 f1) (- (+ -0.898067 f6) (* f7 f7)))
 0.044259)),0.945312

Every 6
(* (if<0 (- (+ (% f6 f2) f1) (* 0.182400 f2)) 0.170200 (% -0.058067 (- 0.115666
 f4))) (if<0 (- (+ f0 f1) (+ (if<0 f3 f1 f0) (% -0.160067 -0.972133))) (* (- f3
 f1) -0.461375) (* 0.933733 (- 0.993400 f5)))),0.929688

(- (% (if<0 (if<0 (- f0 f1) 0.257333 (if<0 f5 0.261400 0.316067)) (% (- -0.616800
 f0) (+ 0.101800 f6)) (* -0.100933 (- f1 0.575133))) 0.575133) (if<0 (* f6
 0.224133) -0.180333 0.418067)),0.929688

(- (% (if<0 (if<0 (- f0 f1) 0.257333 (if<0 f5 0.261400 0.316067)) (% (-
 -0.202600 f0) (+ 0.101800 f6)) (* -0.100933 (- f1 0.575133))) 0.575133) (if<0
 (* f6 0.224133) -0.180333 0.418067)),0.929688
```

Face Dataset #1

```
Every 0
(% (if<0 0.571600 (+ (+ (- f5 f4) (+ 0.569667 -0.775867)) (+ (if<0 -0.808133
 -0.832267 f1) (% 0.930133 f1))) (+ (% (* f4 f0) 0.845067) (- 0.223200 0.744800)))
 (if<0 (+ f0 -0.405867) (% f0 -0.064200) (if<0 (% -0.064200 (* f7 0.729600)) f4
 f2))),0.78125

(% (if<0 0.571600 (+ (* (% -0.379933 f9) (if<0 0.223200 f3 f13)) -0.862000) (+
 (% (* f4 f0) 0.845067) (- 0.223200 0.744800))) (if<0 (+ f0 -0.405867) (% f0
 -0.064200) (if<0 (% 0.744800 f3) f4 f2))),0.78125

(% (if<0 0.571600 (+ (+ (- f5 f4) (+ 0.569667 -0.775867)) (+ (if<0 -0.808133
 -0.832267 f1) (% 0.930133 f1))) (+ (% (* f4 f0) 0.845067) (- 0.223200 0.744800)))
 (if<0 (+ f0 -0.405867) (% f0 -0.064200) (if<0 (% -0.064200 (* f7 0.729600))
 f4 f2))),0.78125

Every 1
(if<0 (+ f4 f4) (* (% (if<0 f9 0.690733 f2) (- 0.488267 f2)) (* (* (* f8
 0.171467) (+ 0.037533 f7)) (if<0 (- -0.550333 f12) (+ f13 f11) (* 0.803533
 f16)))) f0),0.579861

(if<0 (+ f4 f4) (* (% 0.690733 (- 0.488267 f2)) (* (* (* f8 0.171467) (+
 0.190867 f7)) -0.359133)) f0),0.579861

(if<0 (* -1.019733 (- (% 0.775200 f7) (% 0.207800 f2))) (* (% (if<0 f9 0.690733
 f13) (- 0.488267 f2)) (* (- (* f5 -0.164933) 0.514600) (- 0.121934 f3))) f0)
,0.579861

Every 2
(if<0 (* 0.274400 (+ (+ 0.274400 f4) f2)) (* -1.019733 (if<0 (- f0 (- f7 f2))
 -0.104600 -0.757134)) (if<0 f0 -0.262272 0.603867)),0.770833

(if<0 (* 0.274400 (+ (+ 0.274400 f4) f2)) (* -1.019733 (if<0 (- f0 (- f7 f2))
 -0.104600 -0.757134)) (if<0 f0 -0.262272 0.603867)),0.770833
```

```
(if<0 (* 0.274400 (+ (+ 0.274400 f4) f2)) (* -1.019733 (if<0 (- f0 (- f7 f2))
 -0.104600 -0.757134)) (if<0 f0 -0.262272 0.603867)),0.770833

Every 4
(if<0 (* f0 (- (if<0 (* -0.577133 -0.347933) 0.050867 (+ 0.521200 f13)) (%
 0.591933 f2))) (if<0 (* f0 0.147267) (% 0.044310 f0) (if<0 f0 (* f15 0.105667)
 0.591933)) (* f0 0.591933)),0.809028

(if<0 (* f0 (- f6 (* 13.358903 f0))) (if<0 (* f0 0.147267) (% 0.044310 f0)
 (if<0 (* (- 0.950333 f2) -0.297933) 0.770386 0.591933)) (if<0 f9 (* (+ (*
 f16 0.046400) (* 0.124933 f9)) 0.107600) 0.645933)),0.770833

(if<0 (* f0 (- -0.377867 (% 0.591933 f2))) (if<0 (* f0 (- (* 0.055267 0.289600)
 (if<0 f4 f4 f1))) (% 0.044310 0.770386) (if<0 f0 0.770386 0.591933)) 0.770386)
,0.770833

Every 6
(+ (% (if<0 (* f1 0.422133) -0.075376 0.109800) 0.684933) (if<0 (- 0.058207 (+
 f4 f0)) (* 0.721333 f0) (* 0.076267 (if<0 (% -0.950133 f14) -0.616467 (* 0.109800
 f13))))),0.78125

(+ (if<0 (- 0.058207 (+ -0.322733 f0)) (* 0.721333 f0) (* 0.076267 (if<0 (*
 -0.616467 f4) -0.235000 0.721333))) (% (if<0 (* f1 0.422133) -0.075376
 0.109800) 0.684933)),0.78125

(+ (% (if<0 (* f1 0.422133) -0.075376 0.109800) 0.684933) (if<0 (- 0.058207
(* 0.721333 f0)) (* 0.721333 f0) (* 0.076267 (if<0 (* -0.616467 f4) -0.235000
 (* 0.109800 f13))))),0.78125
```

Face Dataset #2

```
Every 0
(* (if<0 (if<0 (+ (+ -0.007400 0.333867) 0.855533) -0.316867 (- -0.131733 f0))
 f4 -0.316867) (if<0 (+ f0 0.385533) -0.131733 (+ 0.248600 (+ (+ -0.905600
 0.855533) 0.333867)))),0.770833

(* (if<0 (if<0 0.333867 -0.854467 (- -0.131733 f0)) f4 (% (+ -0.965267
 0.266267) f0)) (if<0 (+ f0 f4) -0.131733 (+ 0.248600 (+ -0.007400 (+
 -0.007400 0.333867))))),0.770833

(* (if<0 (if<0 (+ (+ -0.007400 0.333867) 0.855533) -0.316867 (- -0.131733
 f0)) f4 -0.316867) (if<0 (+ f0 0.385533) -0.131733 (+ 0.248600 (+ (+
 -0.905600 0.855533) 0.333867)))),0.770833

Every 1
(if<0 (+ f0 (* (if<0 f18 0.451133 (+ -0.258600 f11)) (* 0.441449 f4))) (* (-
 (* (* 0.441449 f4) f3) 0.283933) (% (+ f0 0.631046) 0.920467)) (* 0.441449
 f4)),0.777778

(if<0 (+ f0 (* (if<0 f18 0.451133 (+ -0.258600 f11)) (* 0.441449 f4))) (* (-
 (* (* 0.441449 f4) f3) 0.283933) (% (+ f0 0.631046) 0.920467)) (* 0.441449
 f4)),0.777778

(if<0 (+ f0 (* (if<0 f18 0.451133 (+ -0.258600 f11)) (* 0.441449 f4))) (* (-
 (* (* 0.441449 f4) f3) 0.283933) (% (+ f0 0.631046) 0.920467)) (* 0.441449
 f4)),0.777778

Every 2
(if<0 (- (if<0 f6 -0.141760 0.014614) (% -0.704733 (+ f18 (+ f18 -0.383667))))
 (* (+ (- (% f18 -0.513800) 1.000000) (- (* 0.655467 f4) (* f0 -0.840000)))
 0.141563) 0.666067),0.760417

(if<0 (- (if<0 f6 0.074786 0.141563) (% -0.704733 (+ f18 (+ f18 -0.383667))))
 (* (+ (- (% f18 -0.513800) 1.000000) (- (* 0.655467 f4) (* f0 -0.840000)))
 0.141563) 0.666067),0.760417

(if<0 (- (if<0 f6 -0.141343 0.014614) (% -0.704733 (+ f18 (+ f18 -0.383667))))
 (* (+ (- -0.195552 1.000000) (- (* 0.445000 f4) (* f6 -0.840000))) 0.141563) 0.666067),0.

Every 4
(* (- (if<0 (- f4 f19) -1.559267 -0.060236) f4) (* -0.602800 (if<0 f4 (*
 -0.085120 f4) (% (* (if<0 f10 0.749600 f7) 0.951933) 0.857067)))),0.746528

(* (- (if<0 (- f4 f19) f13 -0.060236) f4) (* -0.602800 (if<0 (- f4 f10)
 -0.068374 (% (* (if<0 f10 0.749600 f7) f19) 0.857067)))),0.746528

(if<0 (+ (if<0 f0 f4 0.603467) 0.483764) (* (if<0 (if<0 f9 f1 0.900200)
 -0.689267 f5) 0.483764) (* (if<0 (if<0 (% f6 -0.500867) (+ f4 f0) f1) (if<0
 f2 -0.154658 (+ f16 f16)) 0.219733) (+ (if<0 f19 0.483764 1.528936)
 0.483764))),0.756944

Every 6
(if<0 (* f4 0.772333) (* f4 0.534600) (* (* f10 (% (* f4 0.202333) (% f19
 0.549400))) (* (* f0 1.117133) 0.951933))),0.625

(% (if<0 (if<0 f0 0.284954 (if<0 f5 f16 f9)) 0.514400 (* (- 0.165600 f1)
 0.074649)) (% (if<0 (- 1.117133 (* f16 0.637333)) f5 f0) 0.952734)),0.625

(% (if<0 (if<0 f0 0.284954 (if<0 f5 f16 f9)) 0.514400 (* (- 0.165600 f1)
 0.074649)) (% (if<0 (- 1.117133 (* f16 0.637333)) f5 f0) 0.952734)),0.625
```