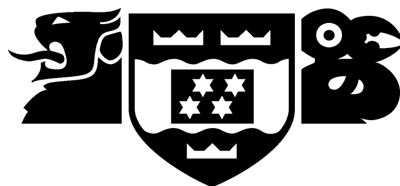
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

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

Genetic Programming for Multiple Class Classification

Yun ZHANG

Supervisor: Dr Mengjie Zhang

May 12, 2005

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

Abstract

Genetic Programming is a relative young machine learning paradigm with many open areas for accepting new ideas. Multiple class classification is a standard though non-trivial real life problem that is good for testing the ability of learning algorithms. In this report, we will describe two developments regarding refining techniques involved in the learning process of genetic programming, along with the estimation of the value of the development by applying the new techniques to multiple-class (object) classification tasks, then comparing the result with conventional GP approaches, in terms of effectiveness and efficiency.

The first development *Modi* regards the invention of a multiple outputs genetic program tree structure. With the use of it on multi-class classification we get the classification accuracy substantially improved. The second development the *Pres* algorithm regards a theoretically linear time program tree simplification algorithm involving using prime numbers. With the use of it we get the training time of GP substantially shortened without affecting the effectiveness of the learning. Ideas behind both developments would be valuable in more general sense, but not only restricted in the area of GP. Reasonable future works are also derived, which are quite worthwhile to go and try.

Comics - Spokesperson

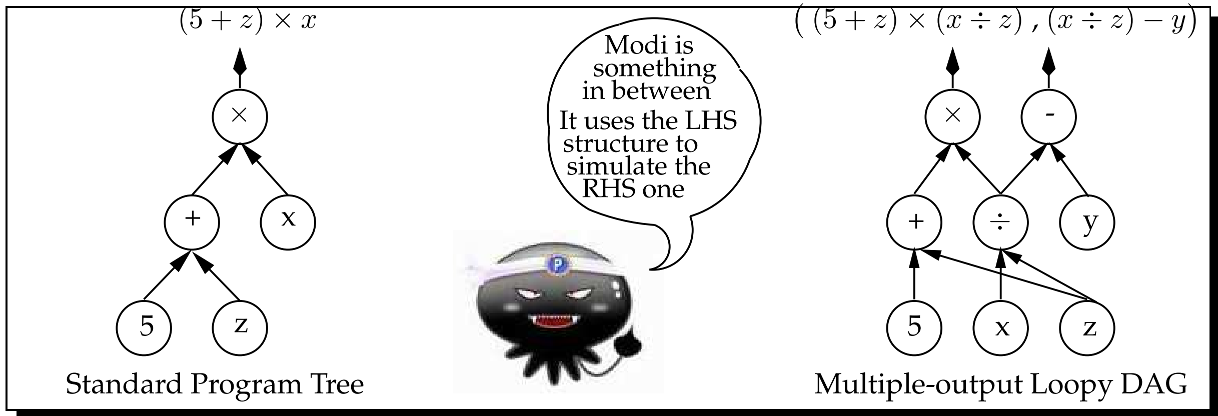


Figure 1: Spokesperson of Modi – **"multiple-output" evil squid**

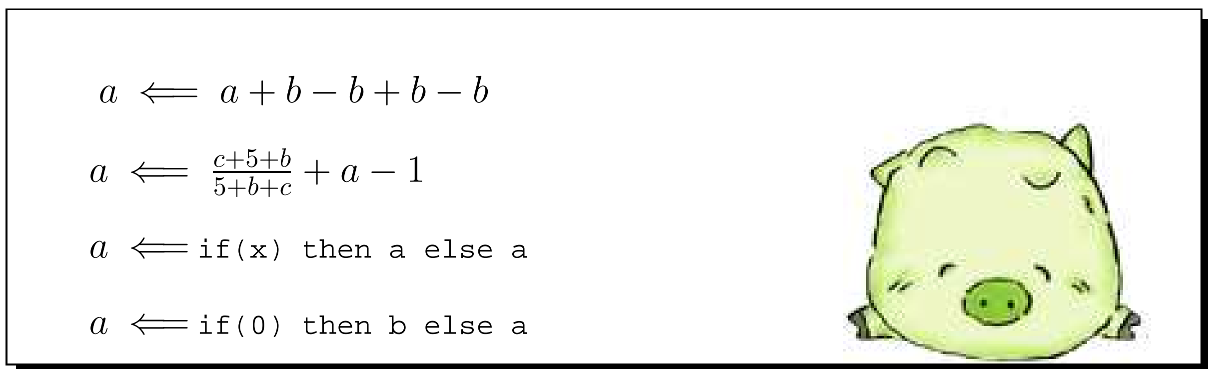


Figure 2: Spokesperson of Pres – **"redundant" poor pig**

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Focus Problems . . . 2
1.3 Goals . . . 3
1.4 Contributions . . . 3
1.5 Structure of Document . . . 4

2 Background Survey **5**
2.1 Genetic Programming . . . 5
2.1.1 The Evolutionary Learning Process . . . 6
2.1.2 Key Concepts . . . 7
2.1.3 Key Features . . . 10
2.2 Classification . . . 11
2.3 Genetic Programming on Multiple-class Classification . . . 12

3 Experimental Design **13**
3.1 Data Sets . . . 13
3.1.1 Shapes . . . 13
3.1.2 Coins . . . 14
3.1.3 Digits . . . 15
3.2 Genetic Programming System Configuration . . . 17

4 Multiple Outputs Program Tree – the *Modi* Structure **18**
4.1 Multiple-outputs Structures in GP . . . 19
4.1.1 Evolutionary Consistency re Datatype . . . 19
4.1.2 Evolutionary Consistency re Program Architecture . . . 19
4.1.3 Multiple-outputs Structures . . . 20
4.2 The *Modi* Program Tree Structure . . . 23
4.2.1 Modi Program Structure . . . 23
4.2.2 Evaluation of the Modi Program . . . 24
4.2.3 The loopy DAG Simulation Effect of Modi . . . 24
4.2.4 Modi Program Generation . . . 25
4.3 Experimental Results and Analysis . . . 28
4.3.1 Overall Classification Performance . . . 28
4.3.2 Effectiveness of the Modi Rate μ . . . 30
4.3.3 Efficiency Analysis . . . 31
4.4 Further Analysis and Discussions . . . 33
4.4.1 Modi Advantages . . . 33
4.4.2 Modi Desires Bigger Tree Depth . . . 33
4.4.3 A Note on the Digit Dataset . . . 34
4.5 Chapter Summary . . . 36

5 Simplification with Prime – the *Pres* Algorithm **37**
5.1 The Pros and Cons of Program Simplification 38
5.2 The *Pres* Algorithm 39
5.2.1 Pres-1: Neat one-level simplification 39
5.2.2 Pres-2: Messy one-level simplification 42
5.2.3 Pres-3: Deeper-level simplification 45
5.2.4 Pres-4: Get back the simplified program 49
5.2.5 Pres-5: Dealing with other operators 49
5.2.6 Summary 50
5.3 Experimental Results 51
5.3.1 The Effectiveness 51
5.3.2 The Efficiency 53
5.4 Further Analysis and Discussion 54
5.4.1 The *restructuring* side effect 54
5.4.2 The Potential of Simplification as a Genetic Operator (Future work) 55
5.4.3 The Practical Achievement of the "Theoretically Linear Time" 56
5.5 Chapter Summary 57

6 Conclusions and Future works **58**
6.1 Modi 58
6.2 Pres 60

A Experimental Result - Tabular **63**

Bibliography **63**

List of Figures

1 Spokesperson of Modi – **“multiple-output” evil squid** i
2 Spokesperson of Pres – **“redundant” poor pig** i

2.1 Flowchart of the Learning Procedure of Genetic Programming 6
2.2 A Toy Genetic Program Tree 7
2.3 Genetic Operators in Action 9
2.4 System Complexity Control — the overfitting problem 11

3.1 Sample Datasets – Shape 14
3.2 Sample Datasets – Coins 15
3.3 Sample Datasets – Digits 16

4.1 Standard Program Tree (SDtree) 19
4.2 Multiple-outputs Structure — Array-Typed Tree 20
4.3 Multiple-outputs Structure — Polytree 21
4.4 Multiple-outputs Structure – Loopy DAG 21
4.5 An example Modi program structure. (a) Modi program tree; (b) Output vector. 23
4.6 Evaluation of the example Modi program structure. 24
4.7 Modi simulated graph classifier. 25
4.8 A toy Modi genetic program when it is being evaluated - full version 27
4.9 Classification Accuracy – Full View 29
4.10 Classification Accuracy – Best Worst View (regarding Modirates) 29
4.11 Modi Rate Analysis – Full View 30
4.12 Modi Rate Analysis – The Improvement 31
4.13 Learning Time 32
4.14 Number of Training Generations 32

5.1 PRES-1, The “prime number” solution for one-level simplification on neat expressions 41
5.2 “Past-free” Bottom-Up Tidy-Up 42
5.3 Bottom-Up Tidy-Up 43
5.4 Double Decked Bus (DDB) 43
5.5 Pres2, “Past-free” Tidy-up Rewriting with DDB 44
5.6 Prime Number Storage Table - The *Primetable* 45
5.7 The Hashtable 47
5.8 Operator Family Rule 48
5.9 Effectiveness in terms of the classification accuracy. 52
5.10 Effectiveness in terms of the terminating generation. 52
5.11 Efficiency in terms of the per-learning CPU time. 53
5.12 A Redundant Program 54
5.13 The *stable* simplification 54
5.14 An *unstable* simplification (PRES) 54

List of Tables

4.1 Results of the new Modi approach over the basic GP. 28

5.1 GP with PRES simplification, on SRS multiclass classification 51

A.1 Result of Basic-SRS 63
A.2 GP on Object Classification: Modi on the Shapes [1/1] 64
A.3 GP on Object Classification: Modi on the Coins [1/1] 65
A.4 GP on Object Classification: Modi on the Digits [1/3] 66
A.5 GP on Object Classification: Modi on the Digits [2/3] 67
A.6 GP on Object Classification: Modi on the Digits [3/3] 68

Chapter 1

Introduction

1.1 Motivation

Genetic Programming (GP) is a relatively young machine learning and searching paradigm pioneered by Nichael Lynn Cramer in 1985 and was firstly explored in depth by John Koza in his 1992 book [?]. The field then grows quickly. So far, GP has been tried on many of AI's problems and has shown to be well qualified for some of them. For example, it has been used to determine classes of image objects for both the object classification and detection, with reasonable success.

Different from traditional learning methods such as *Hill Climbing* [?] and *Neural Networks* [?], which take more advantage of the detail of individual creature hence learn by that individual alters the configuration of itself, the evolutionary learning procedure of GP learns on an entire population of individuals thus produces skillful survival though they do not themselves learn during their individual lifetime. This results in at least two advantages: firstly a bigger chance of escaping from getting stuck around a local optima, secondly multiple solutions in every single learning. However, the soundness, effectiveness, reasonability, and re problem applicability of GP in comparing with traditional methods are still under debate, so attract researches.

Another attribute that makes GP different from the other machine learning algorithm is the way it represents the solution. The searching (solution) space of GP consists of just programs such as Lisp [?], hence connects the field of Machine Learning with Automatic Programming [?]. This makes GP possible to solve problems in a wider range of disciplines, covers AI, mathematics, control engineering, and more.

In the domain of AI especially supervised learning, *classification*, due to its fundamentality in both the theoretical and application domain, is often used for testing the performance of the learner. Classification in an AI sense means a task of Data Mining where a class value is to be sought from details of the given data, which can be a paragraph of plain text, cutouts from digital images, a section of acoustic signal, or whatever whose feature can be extracted and represented in a machine interpretable format.

Multiple-class classification is a subset of classification, in which there are three or more classes of interest. Due to the nature of the technique being used on classification tasks, further difficulty would be added as the number of classes increased. On the other hand, a good solution on multiple-class classification would be more practically valuable, as most of the real-life application problem involves multiple states.

1.2 Focus Problems

As a new discipline, there is still much room for the development of the currently imperfect GP. One typical problem is that in tree-based GP, the evolved program is only able to return a single real number [**?**], as arisen from the restriction of the tree structure of the program. For simple tasks like two class classification, a single real return would be sufficient, as it can just serve as a discriminator between the two classes being classified [**?**]. However, the drawback gets manifested when we are facing more complicated tasks such as multiple-class classification, in which case we do prefer more robust feedbacks from the learnt classifier, rather than just a single real number.

Current researches on tree-GP learning multi-class classifier are mostly about to work out more and more reasonable *classification strategies*, which refer to the algorithm used for converting the single output of the program to a class label. Reasonable success has been achieved in this direction. Typical inventions include *Static Range Selection*, *Dynamic Range Selection*, and *Slotted Dynamic Range Selection*, which will be addressed in more details in chapter two.

We turned around to think about a more natural solution, namely to get out more than one outputs from the program tree. There are studies along this direction also, though not much. A typical development is the *Strongly Typed Genetic Programming* (STGP), yet has not been widely applied because of its burdensome complexity, which makes the customization of the system very complicate.

Therefore to come up with a *flexible* multiple outputs program tree structure is highly desired, as it would greatly relax the constraint on the choice of classification strategy, hence may improve the performance and the applicability of GP on real-life multiple class classification, consequently many higher level intelligent applications.

Another problem that attracts us arises out of the randomnality of the population constructing and learning processes of GP. The point is that in order to pass enough expressiveness to the GP learning system, unless one has deterministic confidential knowledge on the configuration of the task (which rarely happens), we normally do not put detailed control on how exactly would initial programs be generated, nor on (say) which node point in the program tree would we crossover on. These processes are all relatively random. By doing in this way, useless redundancies may be brought in along with the expressiveness we pass to the learning system. Specifically, in the current state of the art of GP, there is no attempt of the learning system to block the appearance of subexpressions like $x + x - x - x$, which can be just reduced to a null.

Leave the program population to carry such *redundancies* along the learning would uselessly enlarge the searching space hence slow down the evolution. Redundancies would also occupy limited space where useful blocks may happen to appear otherwise. With these reasons, it becomes non-trivial to consider trying out to deliberately eliminate redundancies while learning.

If one decide to try the redundancy elimination, there is another issue needed to be consider beforehand. That is the point that separate redundancy elimination by its own would take extra time. Thus we would desire a redundancy elimination algorithm that can do the job as quick as possible. In this way the straightforward text matching approach seemed with a too high time complexity.

1.3 Goals

The overall objective of this project would be to investigate and refine structures and techniques involved in genetic programming, primarily along the way of extending the power of GP on multiple-class classification tasks, in terms of effectiveness, efficiency, and the comprehensibility of program classifiers, comparing with the conventional GP. Specifically, the project is designed to investigate on the two problems raised in the previous section.

- For the first problem arisen, namely the lack of robust feedback from GP learnt multi-class classifier, our research focus will be on to develop a flexible multiple outputs program tree structure.
- For the second problem arisen, namely the carrying of redundancy in program population while learning, our research focus will be on to develop a fast redundancy elimination algorithm, then based on the result to analyse the advantage and disadvantage of doing program simplification while learning.

Without losing of generality, we would like to use *image object classification* for testing our development on the GP system. Basic *Image Processing* techniques would then be involved on doing the feature extraction of images. We would also take several datasets with various properties for the experimenting, in order to gain a more thorough reliable evaluation.

1.4 Contributions

The project has two major contributions. The first one provides a multiple outputs tree structure, which can be used with the Genetic Programming system for a more coherent representation of the individual. With this structure, the classification accuracy of GP learnt classifier is improved for maximumly 11%. Part of this work was submitted to and accepted by the Asia-pacific GP workshop as

Yun Zhang and Mengjie Zhang. A Multiple Outputs GP Tree Structure.

The second one is of a more theoretical aspect. It shows an expression simplification algorithm that is *linear time* with respect to the size of the expression to be simplified. The algorithm can be used to remove redundancies in genetic programs by doing online simplification during the evolutionary learning, also for many other applications.

1.5 Structure of Document

The rest of the report is organized as follows:

Chapter 2 Background Survey, presents the current state-of-the-art in Machine Learning, Genetic Programming, and multiple-class classification. This chapter also serves as a reference of the necessary background knowledge involved in this paper.

Chapter 3 Experimental Design, is a minor section presents the common experimental setting used for estimating both techniques we developed, including a description of the fifteen datasets used, and the basic configuration of the genetic programming system. Specialized setting that differs according to topics would be described in their own section along with the experimental result.

Chapter 4 Modi, is the place for our solution of the first research problem, regarding a flexible *multiple-outputs* program tree structure. A discussion on other possible solutions of the problem were made in the first place to raise the necessity of our development. Experimental results were presented and analyzed after described the core idea.

Chapter 5 Pres, is the place for our solution of the second research problem, regarding a program tree simplification algorithm that can do the simplification in theoretically linear time, by means of prime numbers. The pros and cons of doing program simplification while learning was addressed at the beginning of the chapter, then our solution, afterwards the experimental result and finally the analysis and conclusions.

Chapter 6 Conclusions and Future Works, summarizes the achievement of our works and conclusions derived. Potential problems is also addressed, further works thus derived.

Appendix A presents the full testing results of Modi, though the core part has already been presented in chapter 5.

Chapter 2

Background Survey

2.1 Genetic Programming

Genetic Programming (GP) is an optimization-based machine learning technique whose learning procedure is analogous to biological evolution. Precisely speaking, GP learns by following a *genetic beam search* on a set of candidate solutions known as *genetic programs*. The genetic beam search is in principle an exhaustive search over the solution space, which is all legal compositions over possible functions, constants, and variables, constrained by the prefixed architecture of the genetic program.

The searching process is heuristically guided by the *fitness* of the genetic program, and is performed in a nondeterministic order based on the choice made by *genetic operators*. The search stops by itself if it reaches the goal, namely the case that it has found a genetic program with a perfect fitness. However, in consideration of the tractability of the searching space, the length of the running time is also a criterion to terminate the searching.

There are several families of genetic programming resulting from different representations of the genetic program. *Linear GP* manipulates on solutions in the form of sequence of imperative instructions (C, machine code), whereas *Tree-based GP* manipulates on tree structured programs, say Lisp. It has been shown that the two GPs are equivalent on the experimental performance. In this project, we will only focus on the tree-based one, which will be referred to as GP in the rest of this paper, for convenience.

In the rest of this chapter, we will describe the evolutionary learning process of GP in details, clarify key concepts involved, also to summarize main features that make the evolutionary learning to be different from classical optimization methods, such as neural networks [**?**] and hill climbing [**?**].

2.1.1 The Evolutionary Learning Process

Figure 2.1 shows the flowchart of core procedures involved in the evolutionary learning system of GP. Basically they can be summarized into the following four steps:

1. Generate an initial population of genetic programs.
2. Execute each program in the population and assign it a *fitness value* (page 8).
3. Create a new population of genetic programs by maintaining good solutions through *reproduction* (page 9), creating new solutions through *mutation* (page 8), and combining solutions through *crossover* (page 8).
4. Do step 2 and 3 repeatedly until the best-so-far solution is considered to be acceptable, or run out of the time that the system is allowed to explore on.

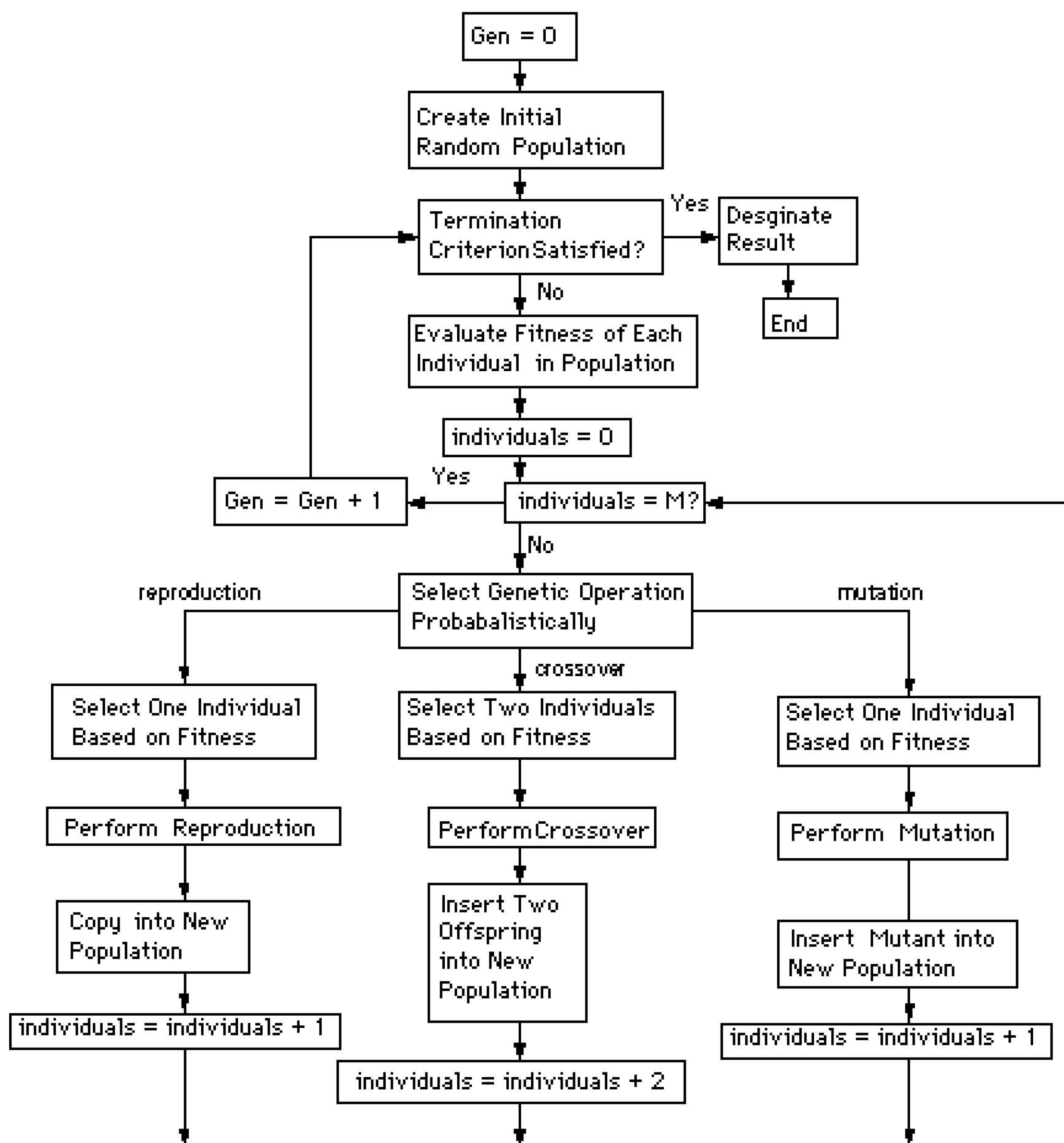


Figure 2.1: Flowchart of the Learning Procedure of Genetic Programming
This figure is extracted from, and owned by, *the GP Tutorial* [?].

2.1.2 Key Concepts

Solution Representation

In evolutionary learning, to somehow code the problem to be solved into the learning system is a primitive concept, also could sometimes be very difficulty. Basically it involves two concepts: the encoding of population (will be addressed immediately), and the choosing of fitness function (will be addressed afterwards).

Encoding of population means to find an appropriate representation of the candidate solution. A good representation should be a one that is powerful enough to represent a good solution, flexible enough so that it could preserve the diversity of the evolution process, and the most important, it has to make easy for the evolutionary learning process (say crossover and mutation) to be carried on.

The representation of solutions can be a very awkward concern in some of the evolutionary learning algorithms, say *genetic algorithm* [**?**], though in GP it is just to define the alphabet of the genetic program, which involves only the customization of two well-defined primitive sets:

Terminal Set: consists of variables and constants of the program. This set is not too strongly task-dependent, as the variables are often whatever can be fed into the program, and the constant are often randomly generated numbers based on some distribution (often a uniform one).

Function Set: consists of the functions of the program. This set is strongly task-dependent and need to be carefully customized for different tasks. A very basic choice of the function set could be $\{+, -, \times, \div, \texttt{iflz}\}$, in which the $\div$ is often defined to be protected (return 0 when divided by 0) for safety; and `iflz` means if less than zero, which takes three arguments, if the first one is less than zero then returns the second one otherwise returns the third one. In the case that we are trying to simulate a `cosec` function, to add `sine` into the function set would be a smart shortcut (cheating) choice. More complex functions or some task dependent functions (say, go north), can be included in a similar way.

The customization on the architecture of the program is sometimes also needed in, say, linear GP. In tree-based GP, programs are just prefix expressions (like Lisp program) that can be represented as a tree, as one shown in figure 2.2.

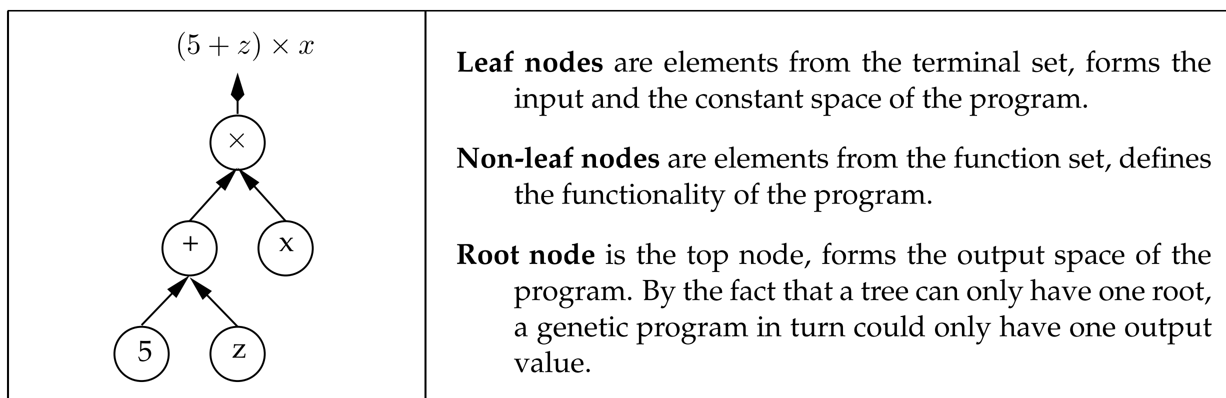


Figure 2.2: A Toy Genetic Program Tree

Fitness Function

The *fitness function* is a mapping from an individual program, to a measurement of how good does the program fit with the learning objective. Fitness function to evolutionary learning is like the heuristic function to heuristic search, thus is considered to be the most important concept of genetic programming.

Fitness function is highly task dependent. For regression problems the *mean squared error* is often used, though for classification problems the *classification accuracy* is the standard choice. Unfortunately, many real-life problems do not have an easy obvious fitness function. Sometimes one may even need to modify the problem a bit in order to find the fitness. For a good fitness function, its computational speed is also important, because the fitness function need to be evaluated on all programs in the population for each evolution, hence is strongly related to the overall learning speed of the system.

Selection

Selection is inspired by the role of natural selection in evolution [**?**]. In short it just means individuals in the population should survive on the fitness. The selection criteria, together with the fitness function, forms the heuristic of the genetic beam search, which guide the evolutionary learning algorithm towards ever-better solutions.

There are many kinds of selection criteria, including *roulette wheel selection* [**?**], *rank selection* [**?**], *tournament selection* [**?**], and many others. Some selection methods, for example the rank selection one, is deterministic. However some of them are not. A typical one of this case is the tournament selection, which uses a model in which individuals compete inside a random subgroup then the fittest one is selected. This criterion is not deterministic, though even carries a certain level of randomness.

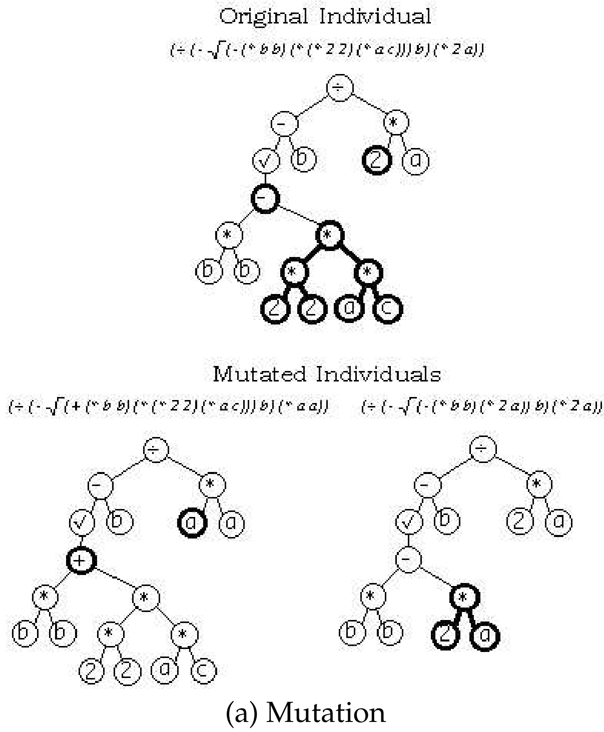
Selection is a basic operation used to decide the partitioning of the population towards different *genetic operators*, namely crossover, mutation, also reproduction in the basic case. The evolutionary learning system often ask for a user defined ordering and ratio over different genetic operators (the ratios for all operators should be added up to one), then choose which of the genetic operation would be performed on which of the individual program based on the selection criterion and the fitness.

Mutation

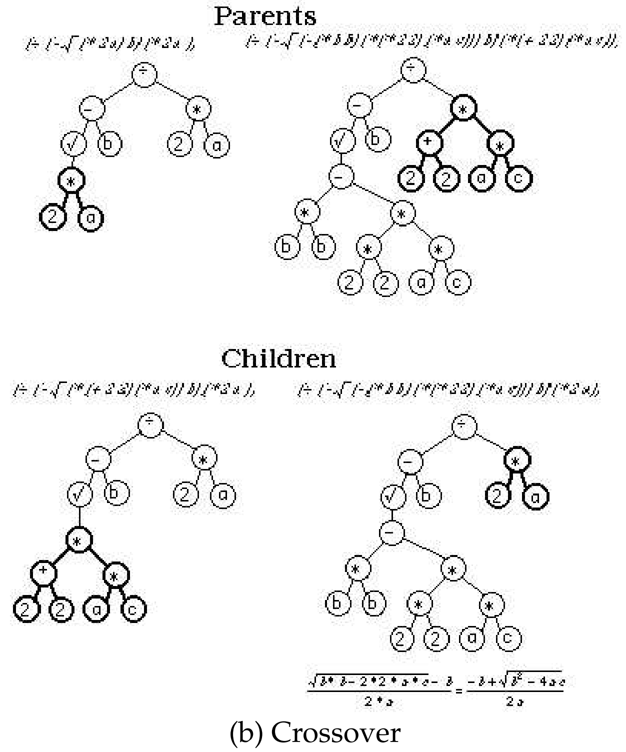
Mutation is inspired by the role of mutation of an organism's DNA in natural evolution. It means an evolutionary algorithm should periodically make random changes (i.e. mutations) in one or more members of the current population, yielding a new candidate solution, which may be better or worse than existing population members. An example that shows how mutation would work to produce new individuals is shown in figure 2.3a. Mutation is currently considered as the most important genetic operator over the others, due to its effect of creating new solutions along with the learning process thence brings diversity into the learning.

Crossover

Crossover is inspired by the role of sexual reproduction in the evolution of living things. It means an evolutionary algorithm should attempt to combine elements of existing solutions in order to create a new solution, with some of the features of each parent. An example that shows how crossover would work to produce new individuals is shown in figure 2.3b. there are many possible ways to perform a crossover operation, referred to as *crossover strategies* which are addressed in details in [**?**].



(a) Mutation



(b) Crossover

Figure 2.3: Genetic Operators in Action
These figures are extracted from, and owned by, *the GP Tutorial* [**?**].

Reproduction

Reproduction is a very simple genetic operator that just copies the selected individuals that is considered to be qualified by the selection criterion into the next generation without any modifications on them. Different from crossover and mutation, the objective of reproduction is not to explore on the searching space, but to preserve the existing achievement on the population fitness, which is something that may be broken by the the other exploring-based operators.

2.1.3 Key Features

Randomness

The evolutionary learning process relies in part of random sampling. This makes it a nondeterministic method, which may yield somewhat different solutions on different runs even if one have not change the learning model nor the starting point. This is different from classical optimization-based learning methods such as neural networks [**?**] and hill climbing [**?**], in which the learning procedure is deterministic under a fixed model and a fixed starting point.

Relative Optimality

In evolutionary learning algorithms, a solution is "better" only in the sense of comparing with the other presently known solutions. Namely, the learning system actually has no concept of an absolutely optimal solution in its learning process. This is a drawback on one hand. However, on the other hand, it makes the evolutionary algorithm to be better employed on problems where it is difficult or impossible to test for optimality.

Population

Where most classical optimization methods concentrate on the refinement of a single solution, evolutionary learning methods maintain a population of candidate solutions. Only one (or a few, with equivalent objectives) of these are considered as the "best", though the other members of the population are still sampled points in other regions of the searching space, where a better solution may later be found. This feature helps the evolutionary algorithm avoid becoming "trapped" at a local optimum [**?**], which is a problem most of classical learning methods are suffering from.

Architecture Based

The evolutionary learning is performed by a heuristic search over all legal *compositions* of primitives. In another word, the learning process is somehow to try through all possible *architectures* over the primitives, under a nondeterministic heuristic. This is different from classical optimization algorithms, which are mostly learning by refining the parameter (say, weights of neural networks) under a prefixed architecture. The feature of architecture-based learning brings in the flexibility on the representation of the solution, along with the ability of learning architectures.

2.2 Classification

Basic Concepts

Classification is the task that takes a feature representation of an object or concept and maps it to a classification label. It arises in a very wide range of applications, such as detecting faces from video images, recognizing digits from the postal codes, and diagnosing tumors in a database of x-ray images. Classification on cut-outs of images has been given its own name, called the *object classification*.

Multiple class classification refers to the classification problem with three or more class of interest, in contrast to the binary classification problems, which has only two classes. Due to the natural of the technique being used on classification tasks, further difficulty will be added as the number of class increases.

Classification in Machine Learning

In a machine learning sense, classification is considered as a supervised learning problem. This means that precise class labels are provided along with details of the pattern to the learning system for it to learn from. On hand (readily-labeled) data examples are often partitioned into two sets, namely the *training set* and the *testing set*. Data examples in the training set are directly used for the system to learn from. Though examples in the testing set are used for measuring the effectiveness of the learning after it terminates. Three-way partitioning of the on hand data with an extra *validation set* is also a common choice in consideration of the system complexity control.

This means to avoid the *overfitting* problem. In machine learning, overfitting means that the learner has exceeded its maximum ability of approaching to the targeted function by using the available training examples, and gets to adjust itself to very specific random features of the training data that have no casual relation to the target function, hence results in that the performance on the training examples still increases while the performance on unseen data becomes worse, as shown in figure 2.4.

Overfitting would come into play if the learning has been performed too long on relatively rare training examples, or that there are unbalanced redundancies in the training dataset. With a validation set, which is used within the learning process, though not directly applied to the training but is purely used to monitor the training fitness on unseen data (i.e. data in the validation set), the overfitting problem can be kind of avoided.

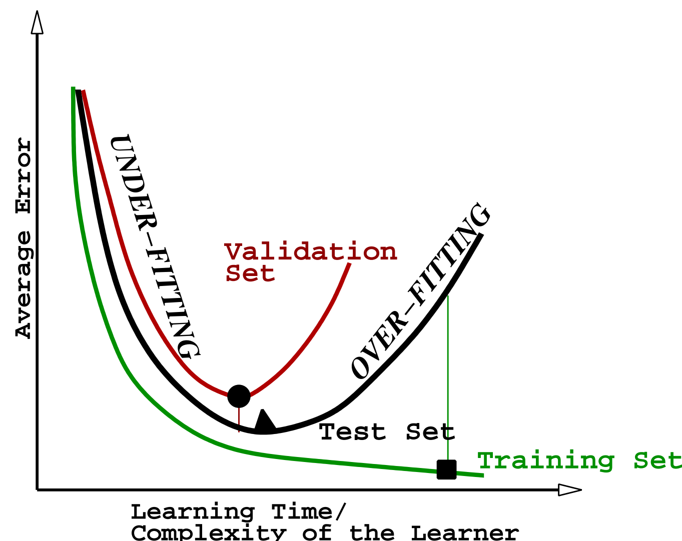


Figure 2.4: System Complexity Control — the overfitting problem

2.3 Genetic Programming on Multiple-class Classification

The research of applying GP on multiple class classification is an active area. Previous works are mainly on how to transform the output from the genetic program classifier, which is often just a single floating-point number, into a class label over multiple alternatives. Algorithms of this kind are called *classification strategy*. Current solutions are mostly to partition the range of the program output into regions, one per class, then consider the range that the output value lies in as the finally classified one. The range partitioning, which needs to decide both region boundaries and the class ordering, can either be manually predefined (hard and expensive), or automatically learnt (time consuming, and often results in unnecessarily complex programs).

The standard classification strategy that converts the output of a genetic program into a class label is the *static range selection* (SRS) . It is very similar with a multiple thresholding approach that defines a fixed ordering on all possible classes with threshold values between each of the pair, so that the output region of the program classifier is fully partitioned over all classes. The fixed ordering is a problem. Consider the classification between `black`, `grey`, and `white` from the average intensity of the pattern. If one carelessly (or more often because of the short of knowledge on the task) defines the class ordering to be, say, `grey`, `black`, and `white`, and put the threshold between the pair to be `1` and `254`, it would be very hard for a classifier to do the right classification.

Other more reasonable though more complicated classification strategies on genetic programs include the *dynamic range selection*, *class enumeration*, and the *evidence accumulation*, as stated in [**?**]. More recent development include *centered dynamic range selection* and the *slotted dynamic range selection*, as stated in [**?**].

Chapter 3

Experimental Design

We use multiple class classification on image objects as our experimental tasks. For a thorough testing, fifteen such datasets are carefully chosen to provide classification problems of varying difficulty. They are: two for *shapes*, four for *coins*, and nine for *digits*. All of them will be described in details in section *sec:alldata*.

Four out of those fifteen datasets are considered as *key datasets*, and will be the concentration when doing detailed dataset-based analysis of the experimental result. They are `squ-4`, `cHard`, `dig15`, and `dig30`, which will all be described in details below.

We do not use early stopping (say validation set techniques) to cope with the overfitting problem. For each of the classification task, our data on hand is partitioned into two equally sized sets for training and testing respectively. We decide to do so because, firstly the overfitting problem does not seem to be a serious issue for our experiments; secondly, it is for the convenience of the experiment.

We use a common GP system setting for all experiments carried. The setting will be described in section 3.2. Noting that to have a common system setting for experiments of all datasets is not a practically reasonable choice. We decide to do so because a common setting is convenient for observing the strength and weakness of the learning methodology, which is our primitive experimental goal.

Each of the experiment is repeated for fifty times with all settings the same except the random seed, which would affect the initial population, as well as the learning procedure wherever the "random" come into play. The averaged result over the fifty runs will then be taken as the final experimental result. The random number generator we use produces pseudo random numbers based on the seed, meaning that all experimental results are repeatable.

3.1 Data Sets

3.1.1 Shapes

Shapes provides two easy object classification problems. Images of this dataset are deliberately generated to give well defined objects against a relatively clean background, as shown in figure 3.1. Pixels of the object are produced using a Gaussian generator with different means (intensity) and variances (fuzziness) for different classes. Objects to be classified are pre extracted, each extracted object is represented by a vector of eight floating-point numbers, referred to as the *feature vector*. Eight values in the feature vector are averaged means and variances over pixel values from four cocentric square windows with different size centralized precisely at each of the object. Specifically, two classification problems of this dataset are:

squ-3: Classification data set is formed by 720 small objects cut out from 24 images similar with one in figure 3.1. It has three classes: dark circles, squares, and light circles.

squ-4: Classification data set is formed by 960 small objects cut out from 24 images similar with one in figure 3.1. It has four classes: dark circles, squares, light circles, and noisy background.

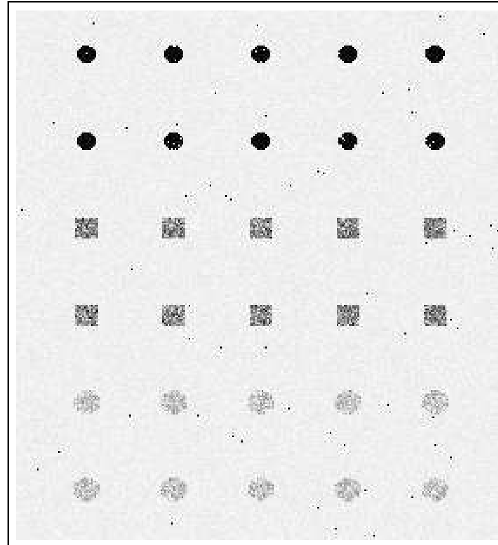


Figure 3.1: Sample Datasets – Shape

3.1.2 Coins

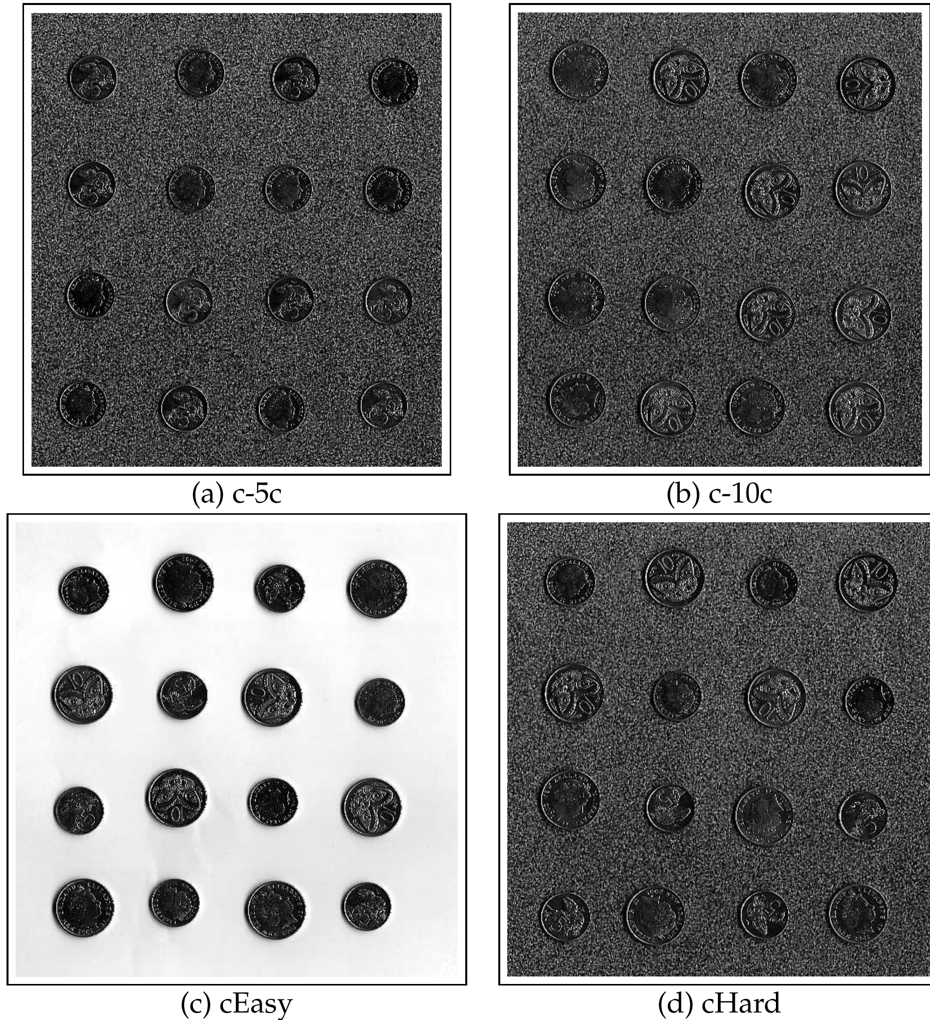
Coins contains images of scanned 5 cents and 10 cents New Zealand coins with or without Gaussian-noisy background. These coins are scanned with 75pt resolution, some of them are even not clear to human eyes, thus is considered as harder classification problems than the shape one. Feature values used to represent the object of this kind are the same as for the shapes, namely, means and variances from four cocentric square windows. Four classification problems of coins with a roughly increasing difficulty are as the following:

c-5c: means coin 5 cents. This classification data set is formed by 384 small objects cut out from 24 images contains 5 cent New Zealand coins with a consistent orientation. The background the objects are placed on is highly noised, but it will not hurt much because for this problem we do not consider the background as a separate class of interest, and the object to be classified has already been (manually) detected and represented as numerical feature vectors ready to be classified. Namely, the problem considers two classes only, which are 5 cent head and 5 cent tail. An example image is shown in figure 3.2a.

c-10c: means coin 10 cents. This classification data set is formed by 576 small objects cut out from 24 images contains 10 cent New Zealand coins with arbitrary orientation. The background the objects are placed on is highly noisy, which makes the classification problems much harder. It has three classes: 10 cent head, 10 cent tail, and noisy background. An example image is shown in figure 3.2b.

cEasy: means easy coins. This classification data set is formed by 480 small objects cut out from 24 images contains 5 cent and 10 cent New Zealand coins with arbitrary orientation, on a clear background. However, the problem is still a hard one because the number of classes is relatively big, namely five. They are 5 cent head, 5 cent tail, 10 cent head, 10 cent tail, and the clear background. An example image is shown in figure 3.2c.

cHard: means hard coins. This classification data set is formed by 480 small objects cut out from 24 images contains 5 cent and 10 cent New Zealand coins with arbitrary orientation, and a highly noisy background. This is considered as the hardest coin problem. It has five classes of interest, which are 5 cent head, 5 cent tail, 10 cent head, 10 cent tail, and the noisy background.



(a) c-5c (b) c-10c

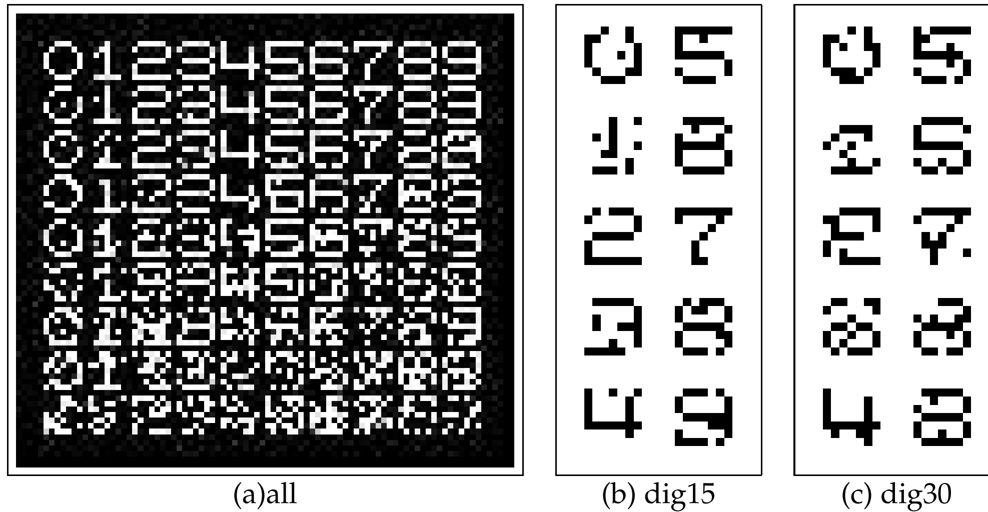
(c) cEasy (d) cHard

Figure 3.2: Sample Datasets – Coins

3.1.3 Digits

Digits contains nine digit recognition tasks, known as `dig00`, `dig05`, `dig10`, `dig15`, `dig20`, `dig30`, `dig40`, `dig50`, `dig60`. Each of those nine tasks involves a collection of binary (i.e. black-white) digit objects, with 100 examples for each of the 10 digits (0,1,2,3,4,5,6,7,8,9), making a total number of 1000 digit examples. Each digit example is an image of 7×7 bitmap, thus the feature vector that represents each of the digit object, as this time we choose to use raw pixel values as feature values, in turn consist of 49 binary numbers.

The nine tasks are chosen to provide hard classification problems of increasing difficulty. In all these problems, the goal is to automatically recognize (correctly classify) which of the ten classes each digit pattern belongs to. Except for the first task which contains clear patterns, all data patterns for the other eight tasks are corrupted by noises. The noise is randomly generated based on the percentage of flipped pixels, and was given by the two numerical suffix of the dataset name, say, `dig10` means



(a)all (b) dig15 (c) dig30

Figure 3.3: Sample Datasets – Digits

the noisy-level is about 10%, so on so forth. It is also the case that the problem gets harder as the level of noise (i.e. flipped-rate) increases.

Sample digit objects for all nine tasks are shown in figure 3.3a ordered up-down by the level of noise, with each line of digits corresponds to a recognition task. It can be seen that, lower lines, which correspond to harder problems, have shown to be difficult for human. Figure 3.3b and 3.3c are the enlarged view of problems of `dig15` and `dig30`. From them we can see that, in the `dig15` problem, although some digits can be clearly recognized by human eyes, such as "0", "2", "5", "7", also possibly "4", it is already not easy to distinguish between "6", "8"and "3", also somehow hard between "1" and "5". The `dig30` one is more difficult. In this problem, human eyes cannot recognize majority of digits, particularly "8", "9" and "3", "5" and "6", and even "1", "2" and "0".

It is obvious that digit problems is hard for human. Though for learnt program classifiers, further difficulty is added by the big number of classes (i.e. 10), as well as the huge feature vector size (i.e. 49). Given these, digits problems are considered to be much harder than shapes and coins, and is supposed to serve as very strict testing problems.

3.2 Genetic Programming System Configuration

Program Population:

- Initial program generation methodology: ramped-half-half.
- Population size: 500.
- Program tree depth limitation: $(max : 7, min : 4)$; Initially $(max : 6, min : 3)$;
- Terminal Node: feature terminals and floating-point random numerical terminals from a uniform distribution between -1 and 1.
- Function Node: uniformally picked from $function\ set = +, -, *, \%, if < 0$
- Program fitness: classification accuracy on the training dataset, namely the number of objects that are correctly classified by the genetic program as a proportion of the total number of objects in the training dataset.

Genetic Operators:

- Elite Reproduction: fixed application ratio = 10%
- Mutation: fixed application ratio = 30%
- Crossover: fixed application ratio = 60%
- Elite-program Selection Criteria: random small scope tournament by using a predefined tournament size, in our case we consistently used 10.

Training Termination Criteria:

- There exists a program in the population with perfect training set fitness of 100%.
- The training has exceeded the pre-defined maximum number of generations, namely 50.
- No early stopping or validation techniques are used.

Chapter 4

A Multiple Outputs Program Tree — the *Modi* Structure

Modi stands for "modify", is a virtual multiple outputs program structure that simulates the effect of loopy acyclic graphs (details later), yet its actual structure is just the standard tree. As its name derives, the idea behind Modi is to take the program as a **Modi***fying* procedure, in which outputs are computed by the program tree *modifies* a predefined sequence of outputs. This makes a contrast with the traditional *outputting* based program tree, which releases its single output from the tree root. The elite of Modi is that, it is structurally equivalent to the standard tree thence freely preserves the consistency of the evolutionary learning process, though it is functionally simulating the loopy acyclic graph, thus is able to reasonably output a sequence of values.

Given that our primitive design goal of the multiple outputs program structures to get more robust feedback from the learnt classifier, the *Modi* structure is tested and evaluated on some typical multiple class classification problems with varying difficulty. A *winner takes all* strategy is used to convert program outputs into class labels, thus get around the problem of having to choose a classification strategy.

The rest of this chapter is organized as follows:

SECTION 4.1 serves as a motivation. In this section we will introduce and evaluate some alternative multiple outputs structures, thus derives the untrivialness of our design of Modi.

SECTION 4.2 will present our multiple outputs program structure – the *Modi* structure.

SECTION 4.3 will show the experimental result of the GP with Modi structured programs on some typical multiple class classification problems with varying difficult, as well as an analysis of the result.

SECTION 4.4 will make further analysis on the experimental result observed, along with some more detailed discussion from the theoretical point of view, thus derives some non-trivial future works.

SECTION 4.5 will be the chapter summary.

4.1 Multiple-outputs Structures in GP

Our primitive design goal is to come up with a program representation that can reasonably output a vector of values, so that the learnt program classifier of such a representation can fit in easier, thus hopefully better, with its task. There are plenty of structures that can do multiple outputs, though hard to be evolved by the evolutionary learning system of GP, also vice versa. In this section, we would like to further clarify how this dilemma comes by going through some of the typical multiple outputs structures, evaluate both their tractability of being embedded into GP, and their suitability of working as a multiple class classifier, thus derives the untrivialness of our design of Modi.

4.1.1 Evolutionary Consistency re Datatype – The Closure Property

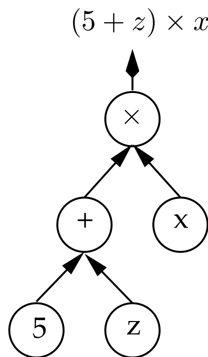


Figure 4.1: Standard Program Tree (SDtree)

In tree-based GP, solutions are in the form of program trees that are similar with (though often much bigger then) the one shown in figure 4.1 on the left. When being evaluated, the program tree takes inputs from leaf nodes, applies functions on them feed-forwardly, finally releases a single output from the tree root. In the standard tree-based GP presented in [**?**], Koza requires that all trees and subtrees have the same datatype to ensure that free crossover and mutation yield valid trees, called the *closure property*. The "typeless" constraint of the closure property does not theoretically reduce the power of the program tree, as many typeless systems are Turing complete. However, many application problems, particularly those involves matrices and lists, are awkward to be represented without types [**?**].

One has developed a technique called *Strongly Typed Genetic Programming* (STGP) [**?**] that allows the use of more than one datatype in genetic programs while still preserves the consistency of the GP system during its evolution. However, as a payment for breaking the closure property, extra constraints on the evolutionary process are needed in place of the closure property to preserve the consistency of the evolution. This extra burden heavily reduces the applicability also even the divergency of the system, thus has not yet been put into widespread use.

4.1.2 Evolutionary Consistency re Program Architecture – The Architecture Evolvability Conditions

In order to preserve the closure property, values being passed through the genetic program must be of an identical datatype. With a standard tree structure, this would result in a "single output constraint" as a tree can have only one root. But if we could make the program *architecture* itself to be of a many-to-many type, the closure property could not hurt us anymore on getting out a list. To cope a new program architecture into the GP system, we still need to be careful not to break the consistency of the evolution. Necessary conditions a program architecture needs to hold in order to preserve the evolutionary consistency are summarized as follows, referred as the *architecture evolvability conditions*.

ARCHITECTURE EVOLVABILITY CONDITIONS

1. The random generation process of an evolvable architecture must be flexible enough so that the diversity of the random initial population can be preserved.

2. Free genetic operations (crossover and mutation) on programs of an evolvable architecture would yield valid programs of the same architecture, thus ensures the evolutionary learning can be carried on under this architecture.

4.1.3 Multiple-outputs Structures

Array-Typed Tree

One way of getting multiple outputs from the standard program tree is to consistently use a fixed-size array as the data type of the entire program, like the one shown on the left side of figure 4.2. Doing in this way, the closure property is preserved because we are still using an identical datatype throughout the entire tree, although it is not a primitive one. This structure also preserves the architecture evolvability conditions, as we have not alter the program architecture at all – it is still a standard tree.

The problem with this approach is on its suitability of serving as a reasonable multiple-class classifier. Firstly, it reduces the flexibility on functions the program could take, as it requires all functions to be defined on vectors, say, dot product instead of simple multiplication. Secondly, program with such a structure would apply exactly a same sequence of functions to each element in its passing-on vector. This means, a program of this kind, has exactly the same functionality as a sequence of basic program trees that are the same for all their function nodes, though only differ on the terminals. These two problems, especially the second one, makes programs of such a structure only be able to output a "trivially-calculated" sequence of values, thus functionally weak to serve as a multiple-class classifier.

Look at the example shown in figure 4.2, the array-typed tree on the left hand side takes vectors and outputs a vector. All of its function nodes thus have to be defined on a vector-basis, this is the first problem. The computational effect of this program is exactly the same as three "only-differ-on-terminals" trees on the right of the figure. Three such programs can not be a reasonable three-class classifier, thus it in turn becomes clear that the array-typed program tree is not a reasonable representation either.

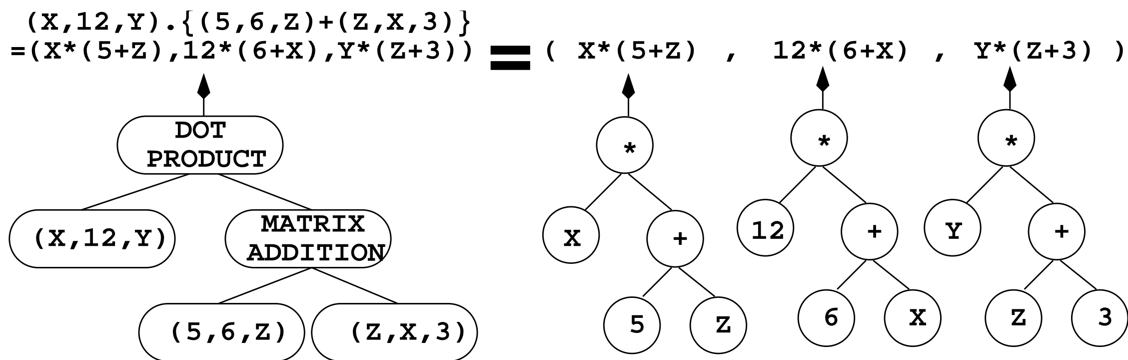


Figure 4.2: Multiple-outputs Structure — Array-Typed Tree

Polytree

A slightly more reasonable multiple-output program structure is the *polytree*, namely a loop-free graph with more than one nodes singled out for special treatment, namely, more than one root nodes, like the one in figure 4.3. Programs of such a structure can do multiple outputs, as to release one output from each of the polytree root would do. It is also easy to see that we could preserve both the closure property and the architecture evolvability condition under this structure. However, the problem is still on its suitability of representing the multiple-class classifier.

A polytree-structured program is actually the same as a sequence of independent program trees, one for each output value. A sequence of independent programs is not sound as a multiple-class classifier, because, according to *Darwinian Co-Evolution Theory* [**?**], relevant spices, like cats and rats, do affect each other on their evolutions. Accordingly, as the multiple outputs we are desiring are also something relevant (otherwise they would belong to independent tasks hence can be solved just separately), it is necessary for different outputting branches to be evolved, also computed, while affecting each other. Given this, the polytree structure should also be ruled out.

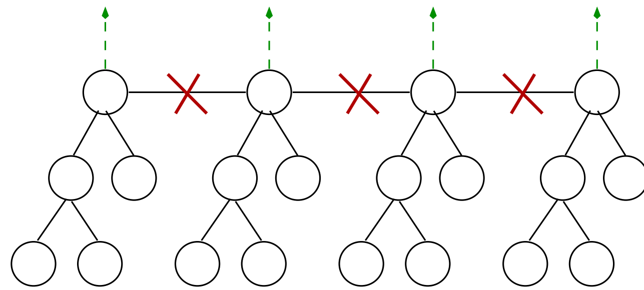


Figure 4.3: Multiple-outputs Structure — Polytree

Loopy DAG

If one could alter the polytree structure, so that the *reusability* between branches under different roots could be brought into the structure, the problem with the polytree would then be solved. This is exactly how the loopy *directed acyclic graph* (DAG) comes. "Directed acyclic graph" means the structure should not contain directed cycle, because it would cause infinite loops when the program runs. "Loopy" means that we do desire undirected cycles, this makes the structure different from polytrees, and is exactly how the reusability can be brought in.

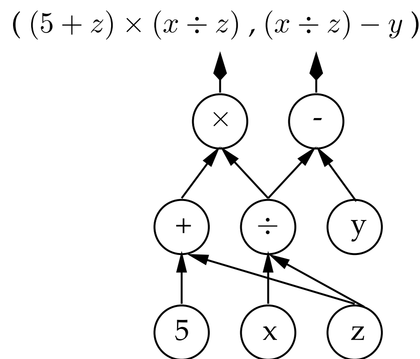


Figure 4.4: Multiple-outputs Structure – Loopy DAG

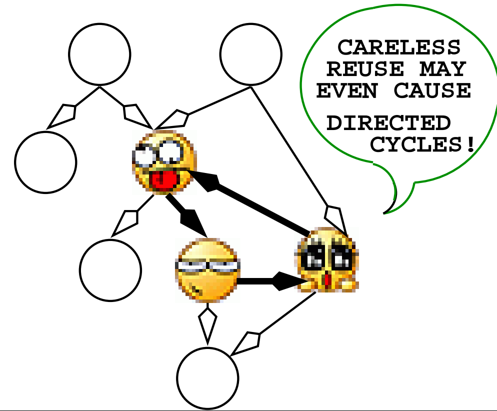
An example program of the loopy DAG structure is shown in figure 4.4. From the example we can see that, with the loopy DAG structure, we are able to get a list of related though differently calculated outputs from the program, thus programs of such a structure is able to reasonably represent multiple-class classifiers. However, when trying to cope this structure into the GP system, problems arise: 4.4.

Firstly, programs of a loopy DAG structure are hard to generate. Because to generate an initial *random* loopy DAG, one need to consider both the reuse of nodes, and the acyclic property. Namely we do want undirected cycles, though definitely not the directed ones. Given these requirements,

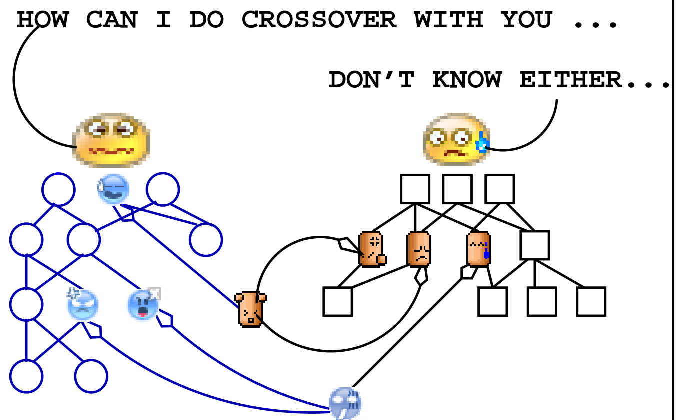
careful considerations and (possibly also) constraints need to be made on the generating process of the random program, thus may affect the flexibility and even the divergency of the learning system. Secondly, although we could preserve the closure property with loopy DAG, tree-based crossover and mutation on programs with such a structure would still yield meaningless descendant, namely it is not an architecture that satisfies the architecture evolvability condition, as shown below:

UNEVOLVABLE ARCHITECTURES

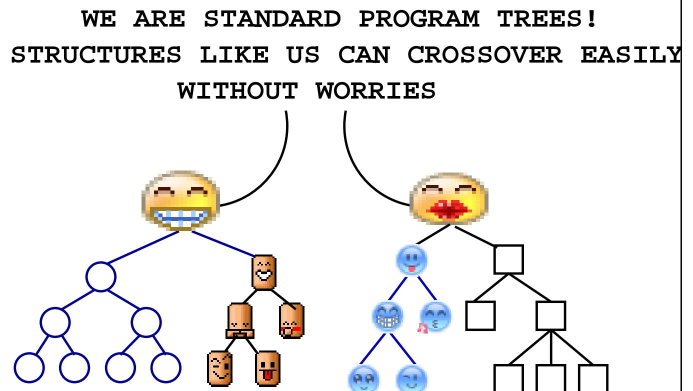
If one use the traditional program tree generation process to generate loopy DAG, directed cycles may appear in the randomly generated program, thus cause infinite loops and corrupt the system, as shown on the right.



If one use the standard crossover operator on two loopy DAGs, it would cause problems, as shown in the figure on the right. The example is an unfortunate case that crossover points are on two nastily-reused nodes, namely, both of the crossover nodes have multiple parents. As the crossover is *decided* between only one pair of their parents, the other parents are feeling unhappy with the dragging-off of their children.



Suffered enough from the payment of not to obey the architecture evolvability constraint? Now let us see a calm positive example. Figure on the right shows that, no problem would happen on the standard program tree structure, which nicely preserves the closure property, as well as the structural tractability constraint.



In the following section, we are going to present the idea of using a virtual structure *Modi* to simulate the loopy DAG, hence achieve an effect of getting multiple related outputs for better serving as a multi-class classifier. As the Modi structure is in truth just the standard tree, the evolutionary inconsistency problem that the real loopy DAG has is freely avoided.

4.2 The *Modi* Program Tree Structure

4.2.1 Modi Program Structure

The Modi program structure has two main parts: (a) a program tree, and (b) an associated *output vector* for holding outputs, as shown in Figure 4.5.

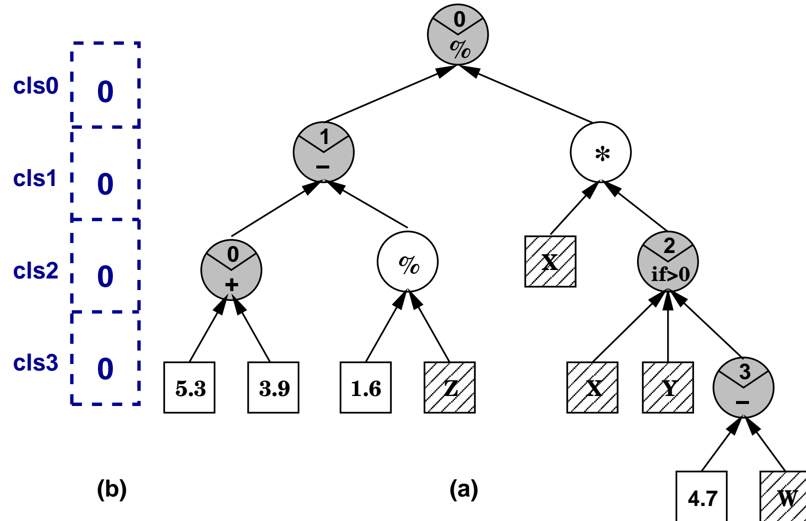


Figure 4.5: An example Modi program structure. (a) Modi program tree; (b) Output vector.

Similarly to the standard program tree structure, the Modi tree also has a root node, internal nodes and leaf nodes. Feature terminals (slashed squares) and random constants (clear squares) form the leaf nodes. Operations in the function set form the root and internal nodes (circles).

Unlike the standard program tree structure, which outputs just one value (often a floating point number) through the root, our Modi program structure takes the *output vector* as the output space, hence produces multiple values, each of which corresponds to a single class in the multi-class classification problem.

As one can see from the figure, two parts of the Modi structure, namely the output vector and the Modi program tree, are not directly *structurally* connected. However, there are value passing between the output vector and some of the tree nodes when the program is evaluated, known as *functional* connections. Functional connections between program tree and the output vector are through some special function nodes called *Modi nodes* (grey circles). Specifically, each Modi node has two roles:

(1) It *updates* an element in the output vector that the node is pre-associated with, by adding its node value to the value of the vector element;

(2) It passes the value of its right child node to its parent, so that the program tree can be continuously preserved.

Note that the output vector is considered *virtual* (that is why it is dashed in the figure), meaning that it does not physically “exist” (i.e.take up memory) other than the moment the program is being evaluated. Which means, during GP’s evolutionary learning , output vectors of all programs “disappear”, only Modi tree parts are active and take part into the learning. Only in the program evaluation time, the output vectis realized and receives updating from the program tree.

4.2.2 Evaluation of the Modi Program

Figure 4.6 shows what happens while the example program is evaluated. Before the evaluation starts, the virtual output vector is realized and initialized with zeros, shown as the dashed vector becomes solid. During the evaluation, each *non*-Modi node passes its value to its parent, exactly the same as in the standard program tree. Modi nodes do differently. Each of the Modi node firstly uses its node value to update the output vector, then passes on the value of its right child to its parent node. The consequence of the program evaluation is that, the output vector gets properly updated based on the input by Modi nodes in the tree. The "size of the output vector" many floating point numbers are then produced, each of which corresponds to a class. Finally, a voting strategy is applied to those outputs. The winner class (the one with the maximum value) is considered to be the class of the input pattern.

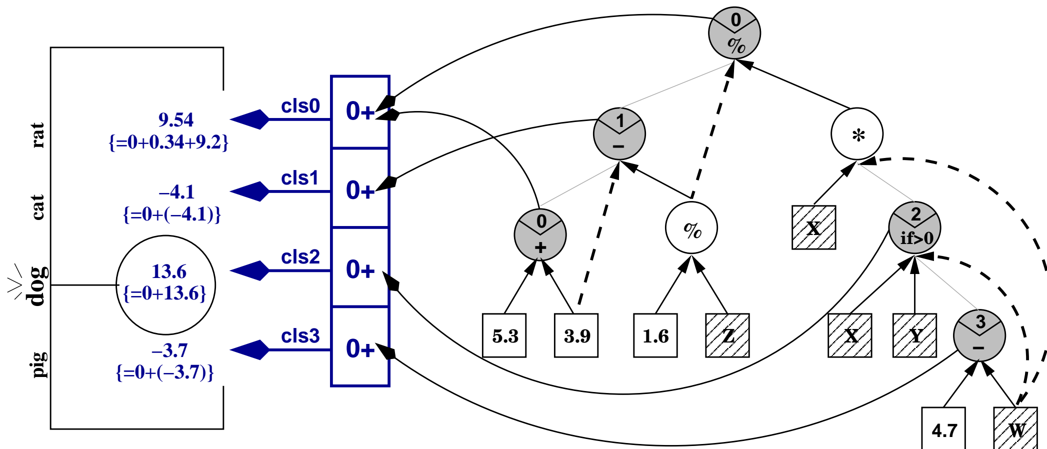


Figure 4.6: Evaluation of the example Modi program structure.

Consider a four-class classification task with possible classes known as {`rat`,`cat`,`dog`,`pig`}. Given an object to be classified, *whatsthat*, which is represented by an input vector contains six extracted feature values `[V,U,W,X,Y,Z]=[0.6,5.7,8.4,2.8,13.6,0.2]`. Taking the learnt genetic program shown in figure 4.5 as the classifier, we feed the input vector *whatsthat* into it, calculating forwardly, and updating the output vector along the way. This would result in the output vector to be finally filled as `[9.54,-4.1,13.6,-3.7]`. Given this result, *whatsthat* should be classified as of the third class `dog`, because the third output `13.6` is the highest *winner*. A more tedious version of figure 4.6, which shows all the detailed value passing of this example, is presented at the end of this section, in figure 4.8 on page 27.

4.2.3 The loopy DAG Simulation Effect of Modi

Figure 4.7 is just a tidy-up redrawing of Figure 4.6 with the structural-used-only fine grey line removed. The figure clearly shows that the dynamic running effect of the Modi program actually simulates a loopy DAG, although the real structure of Modi is just a normal tree plus a vector. Compared with the multi-layer feed forward neural network, which is also a kind of DAG, the Modi simulated DAG also has multi-layers, where leaf nodes forms the input space, internal nodes extract higher level features, and output nodes are associated with class labels. Furthermore, it allows imbalance structure, over-bipartite connections, and non-full connections between neighboring layers, which makes the representation much more flexible.

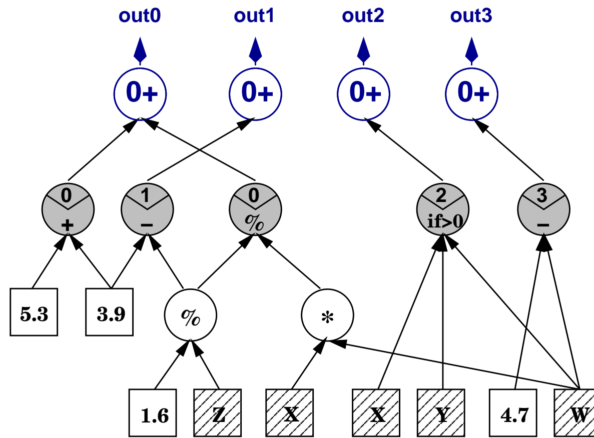


Figure 4.7: Modi simulated graph classifier.

By the effect of *loppy*, the simulated DAG structure also allows the *reuse* of children nodes. Every right-most child of the Modi node is reused by the Modi node itself and the parent of the Modi node, resulting in a two-way reuse. Multi-way reuse is also possible by a sequence of hierarchically connected Modi nodes, as shown at the bottom right conner around the feature terminal node W.

Modi structure gives us only a *proper* subset simulation of the loopy DAG. Which means, Modi can simulate some, but not all program structures the real loopy DAG can represent. This is due to the fact that, in Modi, reuse connections (connections that cause children sharing) are simulated by the way Modi nodes pass values, thus it cannot appears freely in place, but only around *some* (i.e.greater than or equal to one many) Modi nodes, plus at most one non-Modi node. Apart from that, there is also a limitation on where the Modi node can appear in the simulated loopy DAG. In Figure 4.7, it is clear that Modi nodes are always grouped in the layer one level down from the output space due to the fact that only the Modi node is able to (and have to) directly passing value to the output node (i.e. the output vector cell). Therefore, in general, in the Modi simulated loopy DAG, all child sharing can only appears between *some* parent nodes in the second toppest layer plus at most one parent node from a lower layer. This problem is currently considered as a structural shortcoming. Whether or not (or more precisely how much) it would hurt on the effectiveness is still under investigating.

Another minor structural limitation of the current version of Modi comparing with the loopy DAG is that, in Modi simulated loopy DAG, the functionality of the root must be prefixed and consistent through the evolution. In our example we fix them all to *zero plus*. They cannot be as flexible as a tree node, because they are actually output vector cells, which is just storage spaces but not something that can be evolvability randomly assigned and learnt.

4.2.4 Modi Program Generation — Modi Nodes Distribution Law and the Modi Rate μ

Comparing with the standard tree, the randomly generating process of Modi tree has one more step to consider, namely the distribution of Modi nodes. This includes two sub points:

The Whether Problem – How to decide which node should be a Modi. Namely for each of the function node in the program, how could we tell if it should be a Modi node, or just a normal one.

The What Problem – How to assign output vector's cell indices to the Modi nodes we would have chosen by solving the *whether* problem.

The *whether* problem

The solution of the *whether problem* consists of three laws: 1) All leaf nodes are not Modi. They cannot be because to become a Modi node requires the node to have at least one child; 2) The root node is always Modi. It has to be if it is not a leaf, so that we can guarantee that no part of the Modi tree is useless, as we do not make explicit use of the value released from the tree root; 3) For intermediate nodes (i.e. node apart from leaves and the root), the probability of a node to be set to Modi is defined by an offline settable constant μ, the *Modi Rate*. Modi Rate is a manually prefixed setting of the GP evolutionary learning system. It is in the form of a real number $\mu \in [0, 1]$, refers to the probability of an intermediate node to be set as Modi. In another word, μ is the expected percentage of Modi nodes over all intermediate nodes in programs of the *initial* population. Higher the Modi Rate, more function nodes in initial programs would be Modi.

In real experiments, we used an improved Modi rate definition $\mu \in [0, 2]$. For $\mu \leq 1$ we mean the same as above. For $\mu > 1$, we mean that each intermediate node is considered for two independent rounds on if it is a Modi, both with probability $\mu - 1$. This may result in a *Dual Modi node* that upload its value into two (may or may not identical) output cells. For example, $\mu = 1.3$ means two round modi assignment, both with probability $1.3 - 1 = 0.3$. The expected percentage of Modi nodes over all intermediate nodes is then $0.3 + 0.3 = 60\%$, in which $0.3 \times 0.3 = 9\%$ out of those 60% are Dual Modi nodes. Note that Modi rate with $\mu > 1$ is not generally numerically comparable with $\mu \leq 1$ due to the special effect of the *Dual Modi node*, which means that we cannot say, for example, $\mu_1 = 1.3 > \mu_2 = 0.61$, also not the other way round.

The *what* problem

The solution of the *what problem* is just a uniform distribution. Cell indices of the output vector are assigned uniformly across all Modi nodes. As we do not put any other control on the *what* distribution, it is possible for Dual Modi node to upload their value into a same vector cell twice. It is also possible to produce Modi programs that do nothing on some of the output vector element, with bad really luck.

Combine the solution of the *whether* and the *what* problems together we forms the *Modi Node Distribution Law*, as summarized below:

MODI NODE DISTRIBUTION LAW

MNDLaw1 The root node of the program is guaranteed to be a Modi node.

MNDLaw2 Non-root function nodes are considered as Modi Node with probability MR.

MNDLaw3 All terminal nodes, including numerical terminals and feature terminals, are not Modi.

MNDLaw4 Cell indices of the output vector are assigned uniformly across all Modi nodes.

Figure 4.8: A toy Modi genetic program when it is being evaluated - full version

This figure shows what happens when we are running the Modi structured program on a pattern input with size six, the precise value passing and the SBCS part are shown as well:

```
[V, U, W, X, Y, Z] = [0.6, 5.7,8.4, 2.8, 13.6, 0.2]
```

4.3 Experimental Results and Analysis

In this section, we would like to present the experimental result regarding the effectiveness of using Modi structured programs in GP on multiple class classification problems, also to compare the result with the standard GP that uses SRS as the classification strategy. In later discussions, the Modi embedded GP will be referred as *Modi-GP*, the standard one will be refereed as *Basic-GP*, for the convenience of discussion.

As described in chapter three the experimental setting and the dataset, our discussion will be made from two different view, namely based on the observation on the result of all fifteen datasets, and the result of only the four key datasets. The result will mostly be presented in graphical format, with important tabular value underneath. Detailed tabular results can be found in Appendix A.

4.3.1 Overall Classification Performance

Observation on Four Key Datasets

Firstly we would like to compare the Modi-GP and the Basic-GP on four *key* datasets only, with all other common experimental settings the same. Results with the best Modi-rate (details later) are shown in Table 4.1. For the shape data set, both approaches did pretty well as the task is relatively easy. In particular, the Modi approach almost achieved perfect results. For the coin data set, as the task is harder, the Modi approach achieved 93.89% accuracy, 8.67% higher than the basic GP. For two digit data sets, the Modi approach performed much better than the basic GP, with improvements of more than 10%. In particular, for task four, where even human eyes could only recognize a small part of the digit examples, the GP approach with Modi program structure can recognize majority of them, achieved 54.45% accuracy. These results suggest that the new Modi approach can perform better than the basic GP approach for object classification programs, particularly for relatively difficult tasks.

Table 4.1: Results of the new Modi approach over the basic GP.

ClsfAccuracy/ Method	Data Sets			
	Shape	Coin	Digit15	Digit30
Basic-GP (%)	99.40	85.22	56.85	44.09
Modi-GP (%)	99.77	93.89	68.11	54.46
Improvement (%)	0.37	8.67	11.26	10.37

Observation on All Fifteen Datasets

For further testing the behavior of Modi-GP, we experimented over all 15 datasets described in chapter 3. Results are plotted in figure 4.9, shows an overall comparison between Modi-GP with different modi rates and the Basic-GP, in terms of the effectiveness on the classification accuracy. The bottom square labeled curve is for Basic-GP; The top clusters are Modi-GP with different modi rates , as labeled. From those curves we can see that, Modi-GP generally performs better than Basic-GP, presented as the cluster is roughly above the Basic-GP curve. Modi rate does affect the performance as the Modis are clustered but not strictly overlapped. However, the influence is not much comparing with the improvement over the Basic-GP, shown by the big gap between the cluster and the Basic-GP curve.

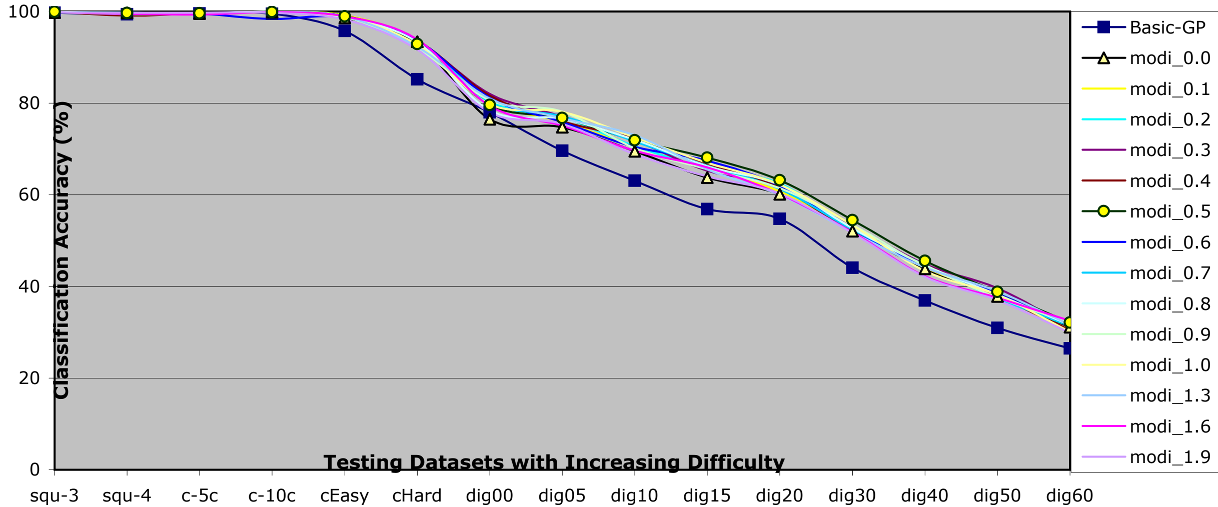
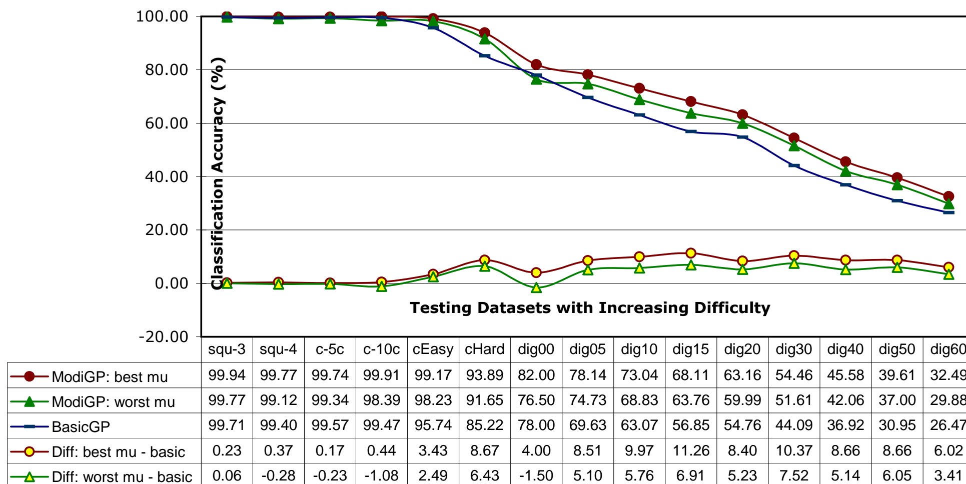


Figure 4.9: Classification Accuracy – Full View

A Best-Worst View

Figure 4.10 shows only the *best* and the *worst* effect that Modi-GP can achieve by presenting the result of the most appropriate and the most inappropriate modi rate for each dataset, namely modi rates that maximize and minimize the classification accuracy of Modi-GP on each of the datasets. The improvement the best and the worst rated curve have over Basic-GP is also shown by the bottom tow curves, called *difference curves*. The precise data value is also presented, shown in the table beneath the graphical region.

From the *best* curve and its corresponding difference curve we can see that, with an appropriate modi rate chosen, Modi-GP is guaranteed to performance better than Basic-GP. The improvement Modi-GP has on the classification accuracy tends to become consistent and maximized on tasks that are hard for Basic-GP. The maximum balancing point is about `10%` improvement on the classification accuracy. From the *worst* curve and its corresponding difference curve we can see that, with a really inappropriate modi rate chosen, Modi-GP still does better on *almost* all object classification tasks. However, for problems that Basic-GP is already able to do pretty well, such as 99% accuracy tasks and the special `dig00` (will be further investigated in the later section), with the worst modi rate of that task, Modi-GP may do slightly worse than Basic-GP, presented as the negative difference between worst Modi and Basic-GP on datasets *squ-4*, *c-5c*, *c-10c*, and *dig00*.



	squ-3	squ-4	c-5c	c-10c	cEasy	cHard	dig00	dig05	dig10	dig15	dig20	dig30	dig40	dig50	dig60
ModiGP: best mu	99.94	99.77	99.74	99.91	99.17	93.89	82.00	78.14	73.04	68.11	63.16	54.46	45.58	39.61	32.49
ModiGP: worst mu	99.77	99.12	99.34	98.39	98.23	91.65	76.50	74.73	68.83	63.76	59.99	51.61	42.06	37.00	29.88
BasicGP	99.71	99.40	99.57	99.47	95.74	85.22	78.00	69.63	63.07	56.85	54.76	44.09	36.92	30.95	26.47
Diff: best mu - basic	0.23	0.37	0.17	0.44	3.43	8.67	4.00	8.51	9.97	11.26	8.40	10.37	8.66	8.66	6.02
Diff: worst mu - basic	0.06	-0.28	-0.23	-1.08	2.49	6.43	-1.50	5.10	5.76	6.91	5.23	7.52	5.14	6.05	3.41

Figure 4.10: Classification Accuracy – Best Worst View (regarding Modirates)

4.3.2 Effectiveness of the Modi Rate μ

Figure 4.11 compares the effect of modi rate on all datasets. Each curve in the figure corresponds to the testing result on a single dataset as labeled on the right, and is plotted against the modi rates in an increasing order. Lower the curve, more difficult the classification dataset. It looks like that all curves are roughly of a *slur liked* shape, except the really funny `dig00` one, which shows up to be affected oppositely as to the others. However, the effect of Modi rate cannot be seen clearly from this plot.

Before continue, we would like to clarify a point on Modi rate = 0.0. Modi rate 0.0 means no function node except the root of whatever randomly generated subtrees (program trees in the initial population or subtrees generated for mutation) are modi. However, evolved programs are able to obtain more Modi nodes through *mutations* and *crossover* in the evolutionary learning process of GP. And this is the reason why the performance of Modi-GP with $\mu = 0.0$ (i.e. curve Modi-00) is still acceptable and even better than the Basic-GP in most of the cases.

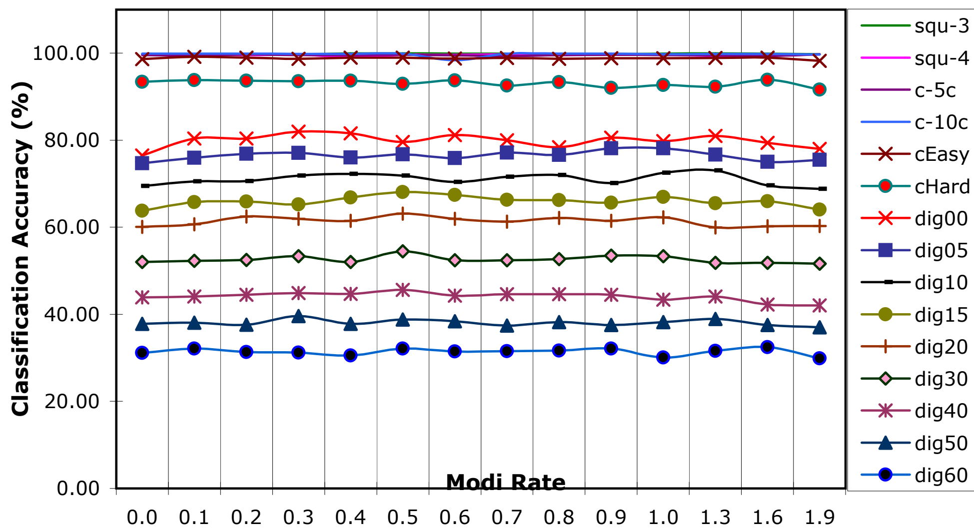


Figure 4.11: Modi Rate Analysis – Full View

To further investigate the effect of Modi rates, we would like to do the plotting in a slightly different way, only focus on the four key datasets, with modi rates ranging from 0.0 to 1.0, as dual Modi has been shown to be bad, namely too big. The plotting is shown in figure 4.12, which shows only the *improvement* of the Modi-GP over the basic-GP. From the figure we can clearly see that, the Modi rate does affect the performance on the classification accuracy. Its influence is not consistently proportional across different tasks, provided by that curves in the figure are not parallel with each other. However, neither too big nor too small Modi rates are the best. No reliable way of choosing a very appropriate Modi rate for a task has been found, except that to do empirical search through experiments. If such a search can improve performance significantly, it is a small price to pay. The experiments suggest that a Modi rate between 0.3–0.6 is a good point to start searching on.

Similar conclusion can be made from Figure 4.9. A good modi rate example is a middle case $\mu = 0.5$, shown as hollow circle labeled curve, which is at the top position of among cluster. A bad modi rate example, a "too small" case with $\mu = 0.0$, shown as hollow triangle labeled curve, with is clearly at the bottom position among the cluster.

The "half modi nodes is good" conclusion refers to the fact that in the initial population, programs with neigher too many modi nodes nor too few modi nodes is convenient for GP to *start evolving on*.

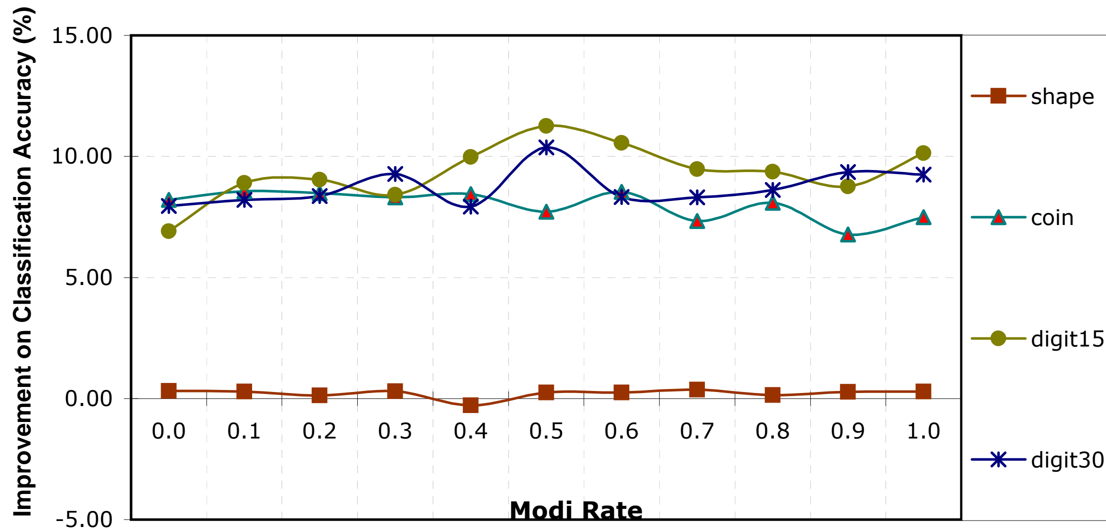


Figure 4.12: Modi Rate Analysis – The Improvement

The result makes sense. Because the interpretation of the modi rate is just the expected percentage of *reuses* and the expected times of output vector updating in the initial programs, too low reuse and output vector updating definitely reduce the power of the program, whereas too high case makes the reusing and updating effect dummy.

4.3.3 Efficiency Analysis

Basically, Modi-GP is more time consuming than Basic-GP in terms of the training time, especially for hard problems, in which double amount of time is used, as shown in Figure 4.13. In principle, there are two possible reason for the slow training time: 1) the running of a single evolution is slower; 2) Modi-GP converges slower on the dataset, namely it needs more evolutions to reach a satisfiable training state (say, perfect classification accuracy). Modi-GP falls into the first case, for the second one, it is actually the other way round.

Modi-GP is relatively slower for single evolution because, the most time consuming part in each evolution is to evaluate the fitness of each of the program in the population. However the program tree evaluating of modi tree is in general slower than the SDtree. Because all nodes need to pass on a value, though modi nodes have an extra task to do, which is to update the output vector.

For the second point, we happily observed that, Modi-GP actually converges much faster then Basic-GP on multi-class classification. This fact is reflected in figure 4.14, in which Modi-GP always guaranteed to take fewer generations then the Basic-GP, or the same if both approaches are not able to fully converge inside the maximum number of generations specified. For simple classification tasks such as the shapes and clear coins, Modi-GP can terminates on just a couple of evolutions. This means that the best program in the randomly generated initial population is already very appropriate in capture the classification task. Therefore, to apply the *No Free Lunch Theorem* [?] inversely, we wold conclude that Modi-GP is more appropriate then Basic-GP on multi-class classification, which is consistent with our experimental result.

Although Modi-GP is able to converge faster to the optimal solution, for hard problems such as fuzzy digits, in which both Modi-GP and Basic-GP are not able to terminate within the fifty generation limits, real life training time of Modi-GP is higher (about double the time) then Basic-GP, purely because its single evolution running time is slower.

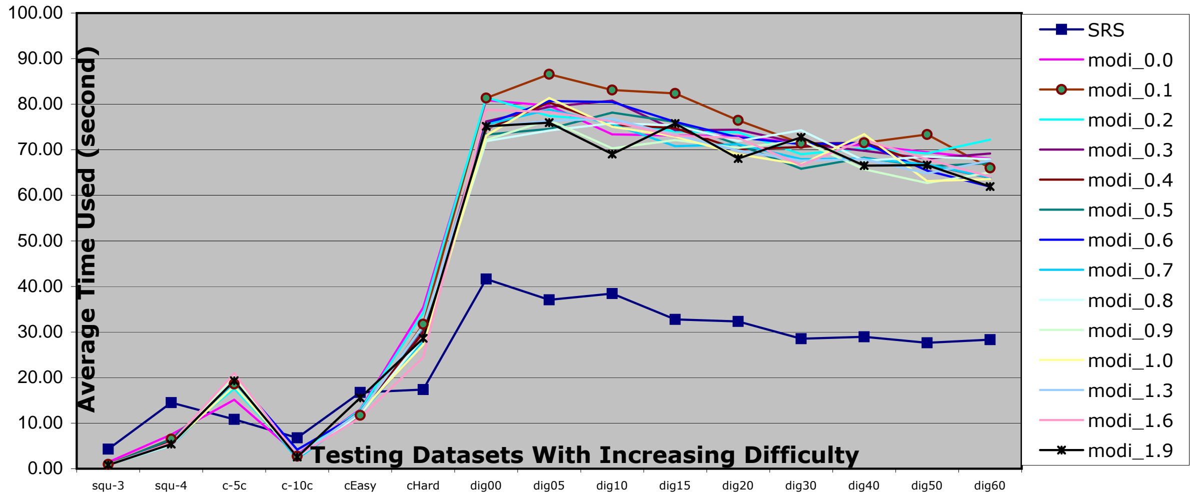


Figure 4.13: Learning Time

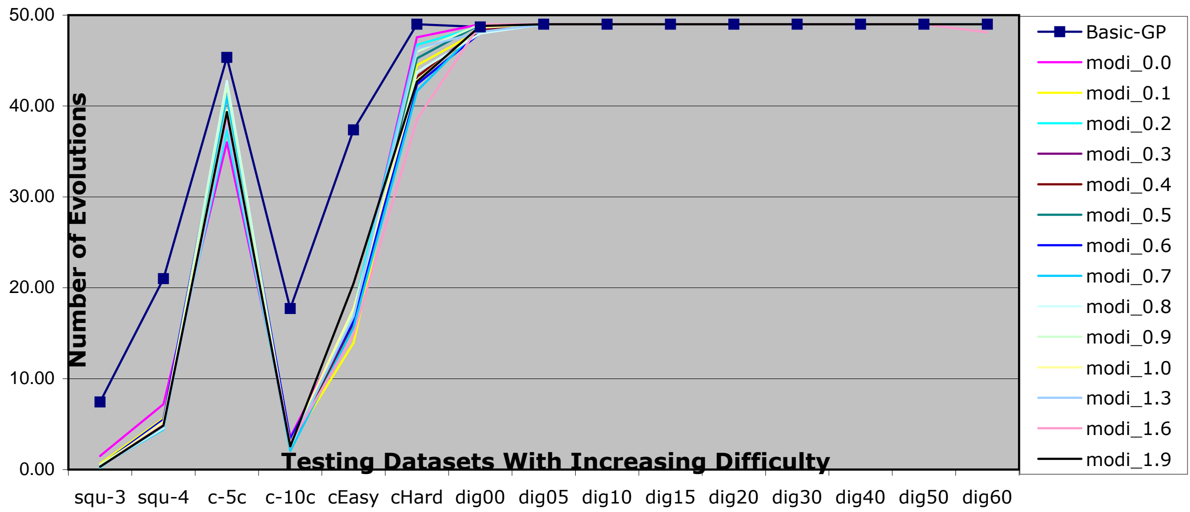


Figure 4.14: Number of Training Generations

4.4 Further Analysis and Discussions

4.4.1 Modi Advantages

Overall, there are three major advantages of the Modi program structure. Firstly, it allows the learnt program to reasonably output a vector of related values, thus gives us a more coherent representation of the multiple-class classifier. Secondly, it allows reusability inside the program tree, thus in somehow compresses the searching space, resulting in a more efficient learning. The third point is that as the Modi structure is structurally equivalent to the standard tree, no extra constraint is needed to embed the structure into the tree-based GP learning system.

Apart from the real-value contribution, the idea behind Modi actually gives us a very flexible family of structures that can all be freely embedded into (hence evolved by) GP. This means, for example, other than simulating the loopy DAG, Modi structure can also simulate polytrees (although not quite practically useful). Modi structure can also simulate the architecture of Artificial Neural Networks, as the architecture of ANN is just a multiple root feed-forward network, therefore, as we have had the multiple root structure (i.e. Modi), we could be able to get a simulated neural network with a bit of concern, say, to move the knowledge stored on the edges of the neural network onto the node, or the other way round, to cope it with the Modi structure.

4.4.2 Modi Desires Bigger Tree Depth

The multiple outputs feature of Modi arises another problem. As there is no free lunch, to get multiple outputs naturally requires bigger-in-size programs. This means the GP learning system has to do the genetic beam search on trees with bigger depth, thus increases the search space and slow down the learning. The "children-reuse" feature of Modi does to some extent help to cope with this problem, though not enough.

To clarify the problem, from the Modi node point of view, in order to "modify" an output vector with bigger size, we would need a program tree to have more tree nodes (as candidate Modi nodes), consequently requires bigger tree depth. If we did not do so, it would result in that some output vector cells never be able to have a chance to be updated by even a single Modi node. An extreme example of this case would be the number of function nodes (candidate Modi nodes) in the program tree is less than the size of the output vector. The nine *digits* datasets we experimented have this problem, as the program tree depth we used is just four to seven ramped-half-half, though the output space is of size ten.

From the loopy DAG simulation point of view, in order to to make a loopy DAG of r "roots" to calculate out the same amount of information as a sequence of binary trees, one would requires the loopy DAG to be in the same depth, though around r times wider than each of the single binary tree in the sequence. Modi is actually a structure that *wraps* the loopy DAG into a standard tree. Therefore in order to achieve an equal computational ability as the loopy DAG that it is simulating (i.e. a much fatter one), the tree structure that really holding the Modi structure should have a much bigger depth. An analogue would be to make a thin man to have the same weight as a fat man we would prefer the thin man to be much taller.

Arithematically, a Modi tree of depth d that is simulating a loopy DAG with r many outputs, would have computational ability around $[\leq \frac{1}{r}]$ times the computational ability of a single output tree with an equal depth d. To the other hand, if we want the Modi tree to have the same computational ability as a standard tree of the same depth for each of its r output, we would better to set it to depth $[\approx r \times d]$. There is something reasonable though we are not doing. We should use a bigger tree depth

for the Modi tree and a smaller one for the standard tree, though in the experiment we set them to be the same.

From the experimental observation, we happily noticed that, for small multi-class classification problems (say four class classification), if we force the Modi program tree depth to be small (say $depth = 7$ instead of $depth = r \times d = 4 \times 5 = 20$, GP can automatically evolve out neat and highly-reused programs that can solve the problem reasonably beautifully. However, for complicate multi-class classification (say ten class), big depth is still necessary. The following is an analysis of the Modi tree depth needed for a r class classification.

Suppose we want to use GP to evolve out a program classifier on r class classification, we would like the evolved program to have r outputs, namely r many functional roots. If we use Modi structured programs, that would mean the size of the *output vector* should be set to r. Without losing of generality, we can also assume our program tree is a full tree with depth d, and all functions are binary, namely, our tree is a binary full tree of depth d. Given these, the number of non-leaf and non-root nodes we can have is $\sum_{i=2}^{d-1} 2^{i-1}$. Suppose the Modi rate is μ, then the expected number of Modi nodes in the tree is $\mathcal{M} = \mu \times \sum_{i=2}^{d-1} 2^{i-1} + 1$ (+1 derived from that the root node has to be a Modi). As we uniformally assigns the class label out, the probability of all output vector cell (r many) will have at least one Modi node in charge of it (hence gets updated based on the input pattern value, thus properly classified) is: $\wp = 1 - \mathcal{P}(\text{at least one of the } r \text{ cell has no Modi node in charge of it}) = 1 - \frac{(r-1)^{\mathcal{M}}}{r^{\mathcal{M}}}$. It is quite a low probability comparing with our intuitively feeling. Lets put some real numbers in: suppose $d = 7$, $\mu = 0.3$, and $r = 10$, which is the real setting of our experiments on the digits dataset, given these and the formulae above, we can easily work out $\mathcal{M} \simeq 19$, and $\wp \simeq 0.865$, not very high. Same setting but the number of classes $r = 4$, then $\wp \simeq 0.996$, much better; Ten class problem, with program depth $d = 10$, then $\wp \simeq 0.999$.

Probabilistically speaking we can never make $\wp = 1$, which means we cannot theoretically avoid the case that some vector cells always be valued to zero regardless to the input pattern the evolved program takes. If we have more than one such "always zero cells" then the SBCS will not able to discriminate between the class those cells are in charge of. However, this is just a theoretical analysis on the initial (randomly generated) population. The problem can normally be greatly improved during the evolutionary learning process of GP.

Theoretically, the learning speed versus learnt output space delimma is a natural problem that all learning algorithm have to suffer from (similiar with Okhams Razor). Therefore the problem is not considered as a *drawback* of the Modi structure. In consideration of the learning speed, in our experiments, we did not try bigger program depth on the digit problems. This will be taken as a primitive future work on Modi.

4.4.3 A Note on the Digit Dataset

As addressed in the previous section, one problem with the *digit* datasets is that the size of its output space (i.e. ten) does not cope with the current setting of Modi tree depth (i.e. four-to-seven ramped-half-half). Another quite similiar problem with them is on its relatively too small input space. As described in chapter three (the datasets), the object feature we used to represent each of the digit is raw pixel values, which means for each of the 7×7 digit object, the input feature vector size is 49. The genetic program tree depth we set is maximum 7 and minimum 4. Therefore even the best evolved program is a full tree of depth 7, there would only be about $2^(7-1) \div 2 = 32$ many input space of the program, which is definitely not enough for handling 49 many features (32 is already the best ideal case, in real experiments the input space of the finally evolved program is much smaller, as the tree depth can rarely be 7-full). For clearer digits sets, in which not all the 49 features are essential to tell

between different digit object, the problem may not be that obvious, however for really fuzzy sets like `dig60`, it would be a big problem. We are currently considering if the unreasonable restriction on the program size could be the reason that Modi cannot make big improvement on `dig60`, it will be future investigated, by means of more experiments.

Another note is on the dataset `digit00`, which contains only clear digits thus has the following two features: 1) the dataset contains only ten distinct examples (as we could have only ten clear digitss), all other data examples are just repetations of them. 2) as a consequence of the first feature, the training dataset and the testing dataset contains exactly the same data examples, thus is totally free from the overfitting problem.

The point with this dataset is, it should be an easy one, though neither Modi nor the standard GP approach is able to perform well on it. It is also the case that the effect of Modi rate shows to be opposite on this dataset comparing with the others. It is quite a superised observation. Currently we could not come out with a explanation that is plausible enough so that everone would be happy with. Will Smart (BSc hons) pointed out that it might because the dataset was too strongly redundant (due to the repetation) on each of the pattern so that the genetic programming system got to put too much concentration on trying to fit those really wee details which could in truth lead the learning nowhere. In another word, the redundancy in the dataset makes the system get lost. This problem is still under investigation, and will be future justified by means of, again, more experiments.

4.5 Chapter Summary

In this chapter, we described a virtual program tree structure that simulates the effect of loopy directed acyclic graphs to make multiple related outputs, yet its actual structure is just the standard tree, thus is naturally evolvable by the tree-based GP. By means of this structure, some problems whose solution is hard to be represented by the traditional typeless program tree, can then be easily embedded into, and learnt by, the tree-based GP.

Multiple class classification is a typical problem that naturally desires a multiple output solution, thus is used to test the effectiveness of the new structure. From experimenting over fifteen datasets with varying difficulty and different features, we observed that the new approach guarantees to outperform the basic approach on all tasks.

Experimental results also show that different Modi rates would lead to different results. Neither too small nor too large Modi rate is good. Although it does not seem to exist a generally Modi rate that is the best for all tasks, rates ranging between 0.3 to 0.6 are good starting points to try on.

For future works, we will further investigate the Modi approach for digit tasks with a larger program size to see if the performance can be improved. We are also considering refining the *modi node distribution law* to gain a fairer assignment of Modi nodes so that it could guarantee to cover the entire output vector. We would also like to investigate ways of extending the idea behind Modi to learn the architecture of more general structures, such as neural networks and belief nets. More future works are listed in the conclusion chapter, i.e. chapter six.

Chapter 5

Program Simplification with Prime — the *Pres* Algorithm

Genetic program trees may pick up redundancies during the evolution, as they are automatically constructed without being told that subtrees like `x-x+x-x` can actually be replaced by a null. This makes us start to consider if it could be nice to eliminate those kinds of redundancies during the evolution, so that the search space of GP can be reduced hence speed up the learning. However, this idea would make sense only if we could do it relatively fast, so that the time saved with reducing the search space will not be over-covered by the time paid for performing the simplification.

In this chapter, we are going to present a redundancy elimination algorithm that can simplify genetic programs in a very efficient way – theoretically linear time with respect to the number of nodes in the program tree to be simplified. The core of the algorithm involves prime numbers, thus is named *Pres*, refers to Prime REfined Simplification.

The rest of this chapter is organized as follows:

SECTION 5.1 will discuss the pros and cons of doing genetic program simplification while learning, by addressing both the problem of carrying redundancies in the evolutionary learning, and the risk may face us if we eliminate redundancies along the way.

SECTION 5.2 will present our linear time redundancy elimination algorithm – the Pres Algorithm. This section is furtherly split into five subsections, describing five aspects of the algorithm.

SECTION 5.3 will present and analyse the experimental result of applying Pres algorithm to simplify programs in the population, periodically during the GP evolution, on some typical multi-class object classification problems.

SECTION 5.4 will make further analysis on the experimental result, along with some more detailed observation on the behavior of the PRES simplification algorithm, thus derives some really interesting future works.

SECTION 5.5 will be the chapter summary.

5.1 The Pros and Cons of Program Simplification

A program, in a GP sense, is often a tree structure describing a complex prefixed expression, which consists of constants, variables, and various operations including (in our case) plus, minus, times, divide, and if. Such expressions will be learnt by the GP system along the way of increasing the fitness of the expression on some tasks. On the other hand, there is no attempt of the learning system to tell that redundancies such as in `x+y-y` is something undesirable.

We were thinking of in some way to tell the system that the redundancy was bad hence made the system be able to eliminate it by itself. However we failed to come out with a teaching approach that was reasonable enough to try. Thus the only way left for eliminating redundancies is to separately deliberately do the program simplification. This solution is less natural and is in doubt if it is nice to do so, as addressed below.

Why do - Redundancy in programs

Randomly generated genetic programs may pick up redundancies, due to the existence of only a restricted number of choices from the variable space, along with the basic nature of arithmetic operations. Types of redundancies in our case, as our program expressions are built on constants, variables, and operations $+$, $-$, $\times$, $\div$, and $if < 0$, are summarized as follows:

- Subexpressions consist of only ground terms (i.e. terms with no variable) can be computed out straightaway. For example:
  $13 + 14 \rightsquigarrow 27$
  $if < 0\ (-1)\ then\ (a)\ else\ (b)\ \rightsquigarrow\ a$
- Negative operations ($-$, $\div$) being applied on different occurrences of an identical expression can be eliminated to unity. For example:
  $a - a = 0$
  $(a + b)\ \div\ (a + b) \rightsquigarrow 1$
  $if(a)\ then\ (b)\ else\ (b) \rightsquigarrow b$
- Some arithmetic rules are able to make the expression much shorter. For example:
  $a \times b - b \times a \rightsquigarrow 0$ commutative rule
  $a \times b + a \times c \rightsquigarrow a \times (b + c)$ distributive rule
  $(a - b)\ \div\ (b - a) \rightsquigarrow -1$

To carry such redundancies in the program uselessly increases the search space, consequently slows down the learning. Spaces taken up by redundant blocks may also hinder the appearance of useful blocks, hence affect the convergency of the learning system. This is why we are considering to simplify the program, as it may be able to speed up the learning, also make more spaces to accept new resources.

Why not to - Break up good blocks

We do have a strong reason of eliminating the redundancy out. However, whether of not it is good to do the elimination *during the learning* is in doubt. The point is that the redundancy elimination process may break down overall redundant though themselves useful sub-blocks.

For instance, `sin(a)-sin(a)` is definitely something redundant that can just be reduced to `0`. However, if we did so, then we would lose the function `sin(a)`, which might by itself a very useful block. Which means, if one of the `sin(a)` can get crossed over later to escape from subtracting towards itself, it may help to increase the fitness of the program quite a lot.

5.2 The *Pres* Algorithm

There are reasonable theoretical arguments both for and against doing simplification while learning. In order to provide further support on the debate, the best way is to try the idea out and analyse on the result. So now in this section we would like to present our simplification algorithm – the *Pres* algorithm first. Results and analysis will come slightly after, in the section following.

Pres stands for *Prime refined simplification*, is an expression simplification algorithm that can eliminate many kinds of redundancy, all together in theoretically linear time with respect to the number of nodes in the program tree to be simplified. As its name derives, the key idea of *Pres* is the use of prime numbers to speed things up. Other hole-fixing ideas of Pres include the *double decker bus* structure, *past-free tidy-up rewriting*, the *operator family law*, and *quartuple hashing*, which will all be addressed in details below.

The entire PRES algorithm is thus separated into five sub-algorithms, referred to as PRES-1 to PRES-5. In the rest of this section, we will firstly look at the simplest *one-level* simplification problem in section 5.2.1 and 5.2.2 regarding PRES-1 and PRES-2, namely the case that all candidates redundancy are primitive terms. We will then go to the *deeper-level* simplification problem in section 5.2.3 regarding PRES-3, in which case the candidate redundancy could be a compound term, say an entire subtree. At the end, we will present how to get the finally simplified program in section 5.2.4 regarding PRES-4, and to show how one could extend the PRES algorithm to handle the simplification on currently undefined operators in section 5.2.5, regarding PRES-5.

5.2.1 Pres-1 — One-level Simplification on Neat Expressions
(Find the Greatest Common Sub-Multiset)

Consider the simplification problem shown in equations 5.1 and 5.2. Because positive operations such as $+$ and $\times$ are commutative, simplification problems as so can thus be considered as eliminating the *Greatest Common Sub-multiset* (GCS) of the multiset[1] representation of the numerator and the denominator (for the $\times\div$ case), as the GCS is exactly the cause of redundancy.

$$\underbrace{\frac{w \times x \times z \times z \times y}{x \times y \times z \times x}}_{redundant\ expression} = \underbrace{\frac{w \times z}{x}}_{simplified} \tag{5.1}$$

$$\underbrace{\big(w + x + z + z + y\big)\ -\ \big(x + y + z + x\big)}_{redundant\ expression} = \underbrace{\big(w + z\big)\ -\ x}_{simplified} \tag{5.2}$$

"Set" means sans ordering. Without ordering the problem fails to preserve the optimal substructure, thus traditional algorithms like Greedy Algorithm [**?**] and Dynamic Programming [**?**] are not applicable in this case. A direct solution of finding the GCS between two multisets would be to go through one of the multiset, for each element in that set we tell if it is a member of the GCS by looking through the opposed multiset for an identical element. But this is polynomial time, $O(m \times n)$ on multisets with length m and n.

A smarter way of finding the GCS is to sort the two multisets first (suppose we do have a comparable key to sort on), then a linear time matching up will give us the GCS of two sorted multisets. Doing in this way, the time complexity is then up to the time of sorting. Divide and Conquer

[1] *Multiset* is a set-like object in which order is ignored, but multiplicity is explicitly significant. — *Mathworld*

sorting algorithms [?] can give us an $O\big(nlog(n)\big)$ sorting, thus the overall GCS finding would be $O\big(nlog(n) + mlog(m) + n + m\big)$. Better, though still not linear.

With counting sort [?] we *may* be able to do even better, with $\Theta\big(max(n,k) + max(m,k) + n + m\big)$ to find the GCS. It could be a nice linear time complexity if k the range of the sorting key is linear, which can be made true in our case. Unfortunately there is another issue that would pull the counting sort approach away from linear on the general simplification problem. Namely the fact that it cannot do deeper level simplification also in linear time. This issue will be addressed more in section 5.2.3 on page 46.

The *Pres* Algorithm does absolutely better. It gives us a consistent linear time solution on the GCS problem with the help of prime numbers: $\Theta\big(min(m,n)\big)$ for GCS finding and $\Theta\big(m + n\big)$ for GCS elimination. Figure 5.1 illustrates how it works. The left hand side is a formal texture description, the right hand side gives an example corresponds to the simplification problem of equation 5.1. The same procedure would apply to equation 5.2, as they are actually reduced into a same GCS elimination problem.

PRES-1: PRIME TO LINEAR TIME GCS

- **Preliminaries:**
  - Pre associate each variable with a unique prime number, so that when one picking up a variable (say) x, a prime number (say) 5 is also picked up, for free.
  - Group variables separated by the negative operator (÷,−) together to form two multisets, whose elements are variables joint by the positive operator (×,+),
  - When constructing a set, calculate the product of the primes along the way.

  Now if we have a set of elements on hand, we would also have a set of prime numbers, and a number that is the multiplication of those primes, i.e. the *prime product* $\mathcal{PP}$.

$$w \rightsquigarrow w_{\cdot 3} \qquad x \rightsquigarrow x_{\cdot 5}$$
$$y \rightsquigarrow y_{\cdot 7} \qquad z \rightsquigarrow z_{\cdot 11}$$

$$\mathbb{A} = \{ w_{\cdot 3},\ x_{\cdot 5},\ z_{\cdot 11},\ z_{\cdot 11},\ y_{\cdot 7} \}$$
$$\mathbb{B} = \{ x_{\cdot 5},\ y_{\cdot 7},\ z_{\cdot 11},\ x_{\cdot 5} \}$$

$$\mathcal{PP}_A = 3 \times 5 \times 11 \times 11 \times 7 \quad = 12705$$
$$\mathcal{PP}_B = 5 \times 7 \times 11 \times 5 \quad = 1925$$

- **GCS Finding:**

  Go through the shorter set $\mathbb{B}$, attempt to divide the prime product of the opposed set $\mathbb{A}$ by each of the prime number in $\mathbb{B}$:
  - If dividable, then consider the divisor to be in the GCS, and continue with the quotient.
  - Otherwise just skip that element.

  In this way we can find the GCS of set $\mathbb{A}$ and $\mathbb{B}$ with size m and n in $\Theta(min(m, n))$ time.

Go through $\mathbb{B}$ *on* $\mathcal{PP}_A$:

$$\rightsquigarrow\ 12705 \div 5 \quad = 2541 \quad GCS = \{x\}$$
$$\rightsquigarrow\ 2541 \div 7 \quad = 363 \quad GCS = \{x, y\}$$
$$\rightsquigarrow\ 363 \div 11 \quad = 33 \quad GCS = \{x, y, z\}$$
$$\rightsquigarrow\ 33 \div 7 \quad = \times \quad GCS = \{x, y, z\}$$

- **GCS Eliminating:**

  If we want not only to *find* the GCS, but also to eliminate all common elements (i.e. do the simplification), we would need to do the GCS Finding on both sets, and get rid of the common element along the way. Thus will take us $\Theta(m + n)$ time.

Go through $\mathbb{A}$ *on* $\mathcal{PP}_B$:

$$\rightsquigarrow\ 1925 \div 3 \quad \times \quad \mathbb{A} = \{w, x, y, z, z\}$$
$$\rightsquigarrow\ 1925 \div 5 \quad = 385 \quad \mathbb{A} = \{w, y, z, z\}$$
$$\rightsquigarrow\ 385 \div 11 \quad = 35 \quad \mathbb{A} = \{w, y, z\}$$
$$\rightsquigarrow\ 35 \div 11 \quad \times \quad \mathbb{A} = \{w, y, z\}$$
$$\rightsquigarrow\ 35 \div 7 \quad = 5 \quad \mathbb{A} = \{w, z\}$$

Figure 5.1: PRES-1, The "prime number" solution for one-level simplification on neat expressions

5.2.2 Pres-2 — One-level Simplification on Messy Expressions (Tidy-Up Rewriting with Double Decker Bus)

In the previous section, we have shown how the Pres algorithm simplifies the neatly grouped-up expressions like equations 5.1 and 5.2 in linear time. However in tree based GP, expressions rarely come out as nice as that, often much messy, like the very left side of equation 5.3 below.

$$\underbrace{\Big(\div \Big(\times \; (\div \; (\div \; w \; x) \; (\div \; y \; x) \,) \; (\div \; (\times \; z \; z) \; (\div \; z \; y) \,) \Big) \; x \Big)}_{a\ simplifiable\ branch} = \underbrace{\frac{w \times x \times z \times z \times y}{x \times y \times z \times x}}_{tidy\ up\ rewriting} = \underbrace{\frac{w \times z}{x}}_{simplified} \tag{5.3}$$

Therefore, in order to bring PRES-1 and the real genetic program together, a *tidy-up rewriting* on the raw tree-parsed expression is necessary. This can be done in linear time, by bottom-up constructing a double layered table that represents (say) the numerator and denominator so far up to each of the node.

The Basic Idea: Bottom-up Tidy-up

An example construction corresponds to equation 5.3 is shown in figure 5.2. For each of the non-leaf node we would have a two-layer storage table associate with it, which tends to represent the entire subtree down up to this node. Each of such table has two layers: a positive layer corresponds to the numerator/dividend (or minuend for subtraction), and a negative layer corresponds to the denominator/divisor (or subtrahend). The bottom-up construction of those per-node tables is as the following:

- If the operator of a node is a positive one like add and times, the positive layer of its table would be the concatenation of positive layers of its children, and its negative layer would be the concatenation of negative layers of its children.
- Otherwise if the operator of the node is a negative one like minus and divide, we concatenate the positive layer of its left child and the negative layer of its right child to form the positive layer of the node, the other way round for the negative layer.

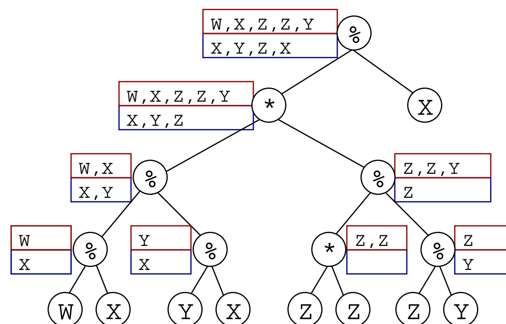


Figure 5.2: "Past-free" Bottom-Up Tidy-Up

By following this procedure from bottom-up, we would get each non-leaf node a double layered table, which could by itself preserve all necessary information of the subtree down from its owning node, though neatly grouped. Thus the two layer table of the root node would form a *Tidy-up Rewriting* of the entire tree, which could be in turn thrown to PRES-1 without worries.

The entire bottom-up construction process has a *past-free* property. Namely the parsing of the double layered table at each of the node depends only on its direct children, not anyone else from the earlier past, like its grandchildren or deeper descendant. With this property, also that we can do the layer (list) concatenation in constant time, the entire Tidy-up Rewriting would be linear with respect to the number of nodes in the tree.

A More Advanced Approach: With Double Decker Bus

Above is the basic idea, though what the PRES Algorithm really does is something slightly smarter: It does the tidy-up rewriting on the prime number associated with each of the variable, rather than the variable itself, as shown in figure 5.3. This enables us to compute the *prime product* along the way to fully satisfy the *preliminaries* of applying PRES-1, as described on top of the figure 5.1.

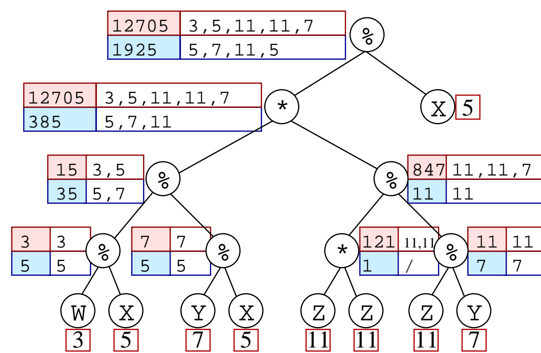


Figure 5.3: Bottom-Up Tidy-Up

Apart from that, the Pres algorithm used a more advanced double layered table with two additional "wheels", like the one shown in figure 5.4. The front wheel stores the *operator family* of this table, namely if it is a table whose positive layer is for plus and negative layer is for minus, or a table whose positive layer is for times and negative layer is for divide, or something else. The back wheel stores the constant calculated along the way, source from numerical terminal nodes, which are not associated with prime numbers as the way we did for feature terminals. This structure looks rather like a double decker bus, so let us call it a DDB, the positive layer the *PosDeck*, and the negative layer the *NegDeck*.

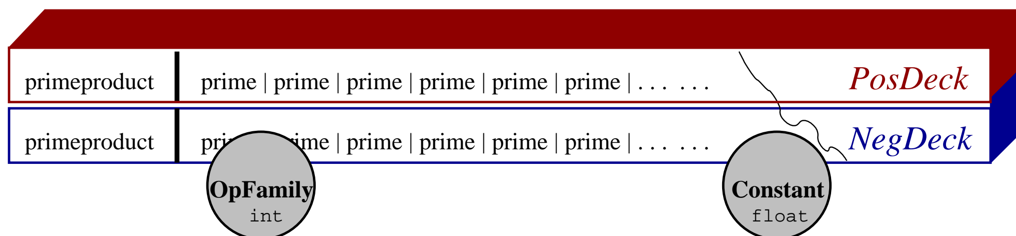


Figure 5.4: Double Decked Bus (DDB)

Figure 5.5 summarizes the structure of DDB (on the left), and the past-free bottom-up algorithm used to construct it (on the right). These ideas together would give us a tidy-up writing along with the control over numerical terminals, thus makes it possible to do one-level linear time simplification on real program tree branches.

PRES-2: TIDY-UP REWRITING

Double Decker Bus (DDB):

DDB is a per-node double layered storage structure looks exactly like in figure 5.4. It has four main components, two decks and two wheels:

- Front wheel:
  `(int) OpFamily:`
  It stores the *operator family* of the DDB, namely if it is a one whose posdeck is for $+$ and negdeck is for $-$, or a one whose posdeck is for $\times$ and negdeck is for $\div$.
- Back wheel:
  `(float) Constant:`
  It stores the numerical constant calculated along the way, source from descendent numerical terminal nodes.
- Top Deck:
  `(Layer) PositiveDeck:`
  - `(list) posprimes:`
    A set of prime numbers, tends to represent all positive descendents in the subtree down up to this node.
  - `(unsigned longlong) primeproduct`
    The product of the prime number set. Its value would uniquely correspond to the `posprimes` set, as that is why we use primes.
- Bottom Deck:
  `(Layer) NegativeDeck:`
  - `(list) negprimes:`
    A set of prime numbers, tends to represent all negative descendents in the subtree down up to this node.
  - `(unsigned longlong) primeproduct`
    The product of the prime number set. Its value would uniquely correspond to the `negprimes` set.

DDB Construction:

Per-node DDBs are built bottom-up, from leaf nodes to the root. Specifically, for each of the non-leaf node, we do the following:

1. Copy the `OpFamily`.
   This step is trivial up to here, as we are currently assuming that the entire tree must be of a same operator family. Namely for example, a mixture of $\times$ and $\div$ is allowed in our toy tree, though not (so far) for a mixture of $\times$ and $+$.
2. Compute the `constant` of the node.
   To do this, simply apply the function of the node to `constants` of its children will do. To guarantee the safety of this operation, we would do the following with leaf children:
   - If it is a numerical leaf then the constant value it should pass on would just be its node value.
   - Else if it is a feature leaf denoting a variable, then the constant value it should pass on would be 1 for operator family of $\times$ and $\div$, and 0 for operator family of $+$ and $-$.
3. Update the `PosDeck` and `NegDeck` by:
   - If the function of a node is a positive one ($+$,$\times$), then concatenates matching decks of its children to form the deck of itself. Multiplying matching primeproducts of its children to form the primeproduct of itself.
   - If the function of a node is a negative one ($-$,$\div$), then concatenates opposite decks of its children to form the deck of itself. Multiplying opposite primeproducts of its children to form the primeproduct of itself.

The above three steps all take constant time, and would be applied to each of the non-leaf node exactly once. Thus the DDB construction of the entire tree would be linear time with respect to the number of nodes in the tree.

Figure 5.5: Pres2, "Past-free" Tidy-up Rewriting with DDB

5.2.3 Pres-3 — Deeper Level Simplification (Quartuple Hashing and the Operator Family Law)

In previous sections, the simplification problem we were looking at was restricted to a single level, namely we assumed that all nodes in the program tree is of an identical operator family, so that all candidates redundancy are primitives. But this is very often not the case. Consider the simplification problem shown below in formula 5.4, one redundancy occurs on the compounded term $(x + y)$, as the numerator has a term $(x + y)$, which can be eliminated with the term $(y + x)$ of the denominator.

$$\begin{aligned} & \underbrace{\Big(\div \Big(\times \; (\times \; (+\, x\, y)\, (-\, z\, 3)\,)\; (\div \; y \; (+\, y\, x)\,)\,\Big) \Big(\times \; (\; \div \; (+\, z\, x)\; w\;)\; y \,\Big)\Big)}_{a\ simplifiable\ branch} \\ = \quad & \underbrace{\frac{w \times (x+y) \times (z-3) \times y}{y \times (z+x) \times (y+x)}}_{tidy\ up\ rewriting} \\ = \quad & \underbrace{\frac{w \times (z-3)}{z+x}}_{simplified} \end{aligned} \tag{5.4}$$

This makes the PRES Algorithm get stuck. Because the the algorithm requires a prime number to be associated with each of the candidate redundancy. This can be done easily for atom nodes by the pre-association, though not for compound terms like the $(x + y)$. This is because compound terms are dynamically formed along the way, thus the *pre* association is not possible at all. Happily the problem is solved with the idea of *dynamic prime number association* along the way of the bottom-up construction of DDB. This involves two supporting ideas:

1. The use of *Quartuple Hashing*, to safely determine *what* prime number to use for a dynamic association along the way, in constant time.
2. The *Operator Family Law*, to tell *when* to do the prime number association.

Quartuple Hashing

Let us look at the problem of *what* first and the problem of *when* shortly after. The *what* problem sounds easy, as to have an ordered list of prime numbers and assign primes out one after another appears to do, i.e. use a *primetable* like a one shown in figure 5.6. But the problem is we need to avoid two different prime numbers to be associated with compound terms like $(x + y)$ and $(y + x)$, or two occurrence of $(x + y)$, as they are actually the same and should be simplified if appear oppositely.

	0 1 2 3 4 ... ... 498 499
prime[] keys	2 3 5 7 11 ... ... 3559 3571
Node*[] vals	Node* Node* ... ... Node* null null ... . null

Figure 5.6: Prime Number Storage Table - The *Primetable*

The way of avoiding this problem is that, if we decide to associate a prime number with a subtree (a compound term), we should firstly check whether or not a same compound term has already been

associated with a prime number, if it is the case then we would associate the new subtree with a same prime number, only if it is not the case would we pick up a new prime number.

This checking procedure would be time consuming if we did it by going through the entire *primetable* for a looking-up every time we wanted a new dynamic prime number association. Happily with the help of hashing on subtrees we can still stay on the constant time, given that the hashing structure and hashing function(s) are reasonably chosen so that the collision is reduced to an acceptable level.

Here is the detail of how we used the hashing: As what we want to do is that given a subtree, deterministically find out the prime number associated with it, if any, in constant time. Thus we should hash from the key of subtrees to their associated prime numbers. This could be constant time only if we can do the key-equality-check in constant time, which is not the case if we use raw subtrees as the key.

The `primeproduct` in the DDB helps on this point. Recall that the DDB of each of the node is able to fully represent the subtree down from it, and the head of a deck (i.e. the primeproduct) is able to fully represent the carriage part (i.e. the prime list), thus the head of the two decks together with the two wheels would be able to fully represent the DDB, thus the entire subtree down from each node. Namely a quartuple: (`PosPrimeproduct, NegPrimeproduct, OpFamily, Constant`) would do. In this way we reduces the linear-sized hash key (raw subtree) into a constant-sized one (with size four), hence ensures the looking-up of the hashing to be constant time.

The hash table that Pres actually uses is a one shown in figure 5.7 on page 47 (color view recommended). It looks awesome though actually no more than a two-level hashing structure both with bucket. The *PosHashtable* hashes on the *PosPrimeproduct* and the *OpFamily*. The *NegHashtable* hashes on the *NegPrimeproduct* and the *Constant*. As the *Constant* key is in the form of floating-point number, user controllable tolerance on the equality of this key is introduced. Values stored in such a hashtable are pointers to the prime-associated subtree roots, which could in turn give us their associated prime numbers. Note that we prefer to use bucket, because we want our hashtable to tell "no there is no such an element in" also in constant time.

Up to this point, the answer of "why the *counting sort* solution we discussed before (in section 5.2.1 on page 40) does not sound on its linearity on deeper level simplification" becomes obvious. As with it we could have no linear-sized key for our hashing, thus have to hash on subtrees which would drive the entire algorithm into polynomial.

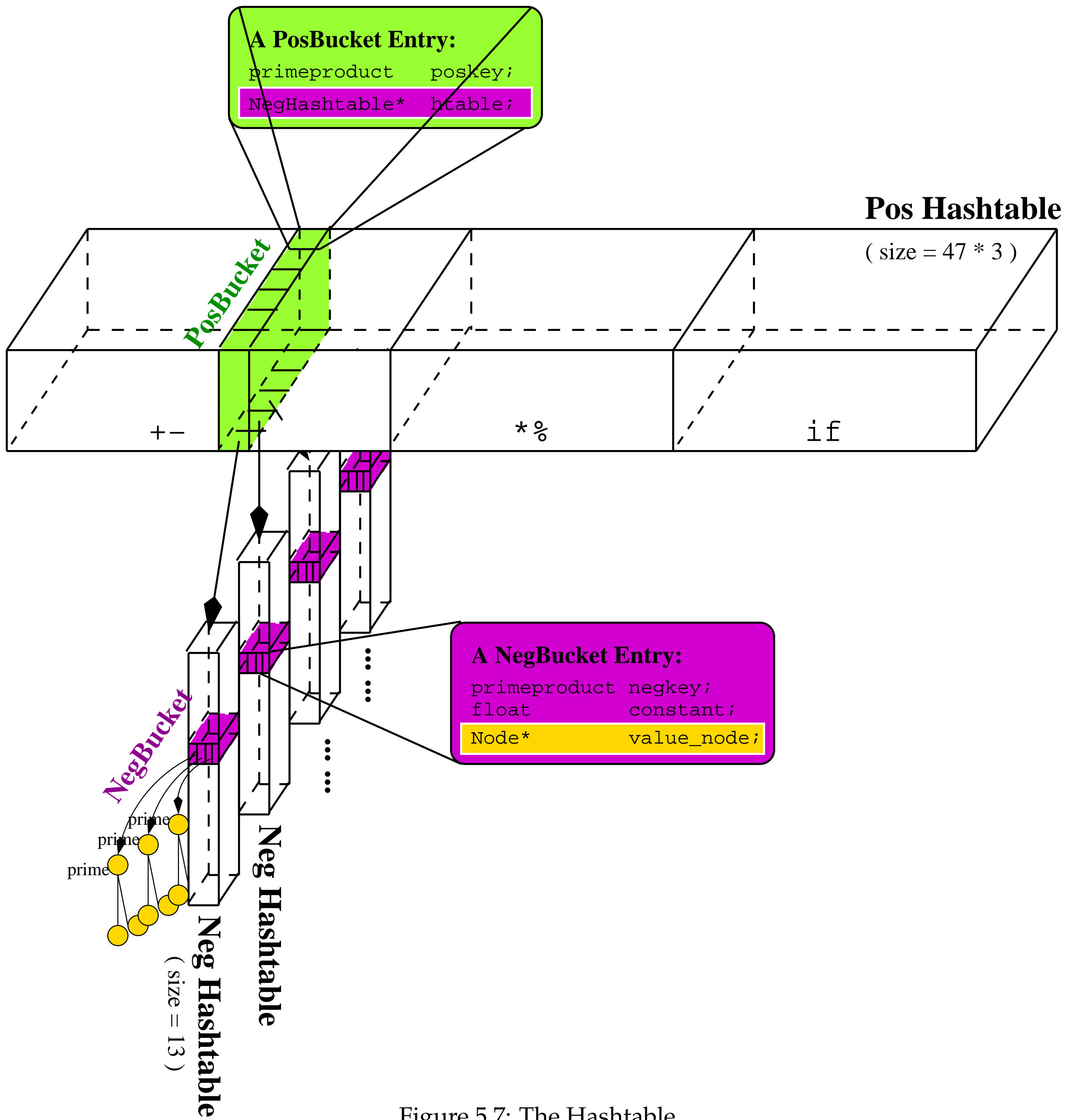


Figure 5.7: The Hashtable

The Operator Family Law

We have solved the *what* problem by using hashing. This means if we want to associate a prime number with a subtree, we can feel free to do so, in constant time. Now it is the time for us to look at the problem of *when* to do the prime number association would be the most reasonable, namely it could maximize the efficiency though without affecting the ability of simplification.

In order to optimize the simplification efficiency, we want to do as few *subtree wrapping* (i.e. dynamic prime number association) as possible. We notice that with operators $+$, $-$, $\times$, and $\div$, to wrap a subtree into a prime number is necessary only if the subtree root and the parent of it are of a different operator family, like the *wrap* example shown in figure 5.8. But in the case that its sibling is eliminatable toward it and its parent is a negative operator, we can just eliminate those subtrees straightaway, like the *arithmetic* example shown in 5.8. This is summarized as the *Operator Family Law*, as shown in figure 5.8.

Note that with PRES-3, we would in some circumstance need to apply the linear time PRES-1 on non-root nodes. This seemed to make the whole process non-linear, though is not in actual fact. The point is, after we applied PRES-1 on the DDB of a node, we would then never need to do PRES-1 again on that subtree, thus overall we would still stay on the linear case. An analogue of this fact could be, instead of summing over ten values at once, we do five summations each of size two, which in general just require a same amount of computation.

PRES-3: OPERATOR FAMILY LAW

This is a plug-in of PRES-2, namely the bottom-up DDB construction for tidy-up rewriting. Together with this PRES-3 the PRES algorithm can then be used for deeper level simplification though still preserves its excellent linear time complexity. Specifically, when doing the bottom-up DDB construction, for each of the non-leaf node we do the following:

- **Merge:**
  For child who has the same operator family as itself, merge the child according to the way described in PRES-2.
- Otherwise we apply PRES-1 on the non-mergable child to simplify the DDB of it, for a preparation of either an *arithmetic*, or a *wrapping*, as described below.
- **Arithmetic:**
  Else if it is a negative operator, and its two children are eliminatable, then eliminate them straightaway. (*eliminatable* would mean identical subtrees, or fit in with various arithmetic rules, like $(a-b)\div(b-a) = 1$, $a\times b\times c - b\times c = a-1$, $(a+b)(a-b) = a^2 - b^2$, and so on, according to the built-in arithmetic rule.
- **Wrap:**
  Otherwise we wrap the subtree into a prime number, update the primetable and the hashtable if necessary, then continually construct the DDB up with this newly associated prime number.

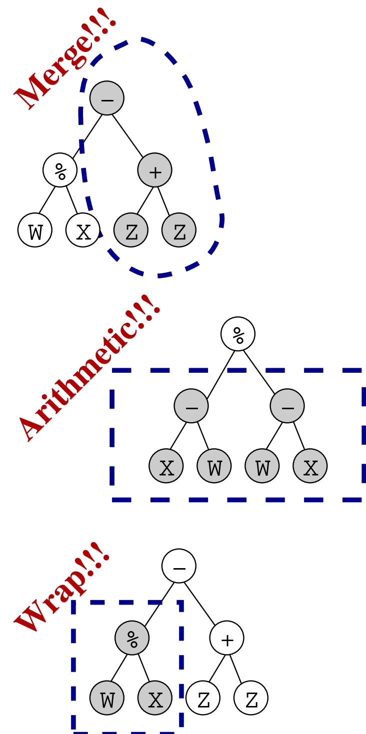


Figure 5.8: Operator Family Rule

5.2.4 Pres-4 — Get Back the Simplified Program (from the Root-DDB)

So far we have shown how to construct the DDB along the way. After we reached the root of the entire tree, we would apply PRES-1 once, for a linear time overall simplification, to gain a simplified DDB. We will then need to reconstruct the simplified program from the simplified DDB.

In PRES the reconstruction is done in a "divide and combine" way. It divides prime numbers in the root DDB into pairs, then from them combining the tree up by making up appropriate function nodes. The reconstruction process may also go down if the prime number itself is a wrapping-up of some subtrees, in which case a retrieval from the hashtable would be performed, in theoretically constant time.

In implementation the reconstruction is done by five functions recursively call each other. The entire procedure goes through each of the reconstructed node at most three times, thus the overall reconstruction process is still linear with respect to the number of nodes in the simplified tree.

5.2.5 Pres-5 — Dealing with Other Operators (`if` etcetera)

Up to here, we only talked about PRES simplification on basic arithmetic operators $+$, $-$, $\times$, and $\div$. Similar procedures can be applied to other simplifiable operators such as the $\{\wedge, \sqrt{}\}$ pair and the `if` function. A good point is that to do this there is no big change needed on the basic structure of the PRES algorithm. Certainly, customization is still necessary for building simplification rules in. The basic procedure needed to extend the PRES algorithm to handle a new operator or a new pair of operators is as follows:

1. Assign a novel operator family to them.
2. Define the positiveness and the negativeness of the newly assigned operator family if necessary (e.g. for operator pairs), or blank the negative part (e.g. for singleton function).
3. Extend the OPERATOR FAMILY LAW by considering the following:
   (a) **Merge:** Define how the DDB merging would go with the new operator, as it might be a bit tricky for (say) the $\{\wedge, \sqrt{}\}$ pair.
   (b) **Arithmetic:** Define in which case the canceling-out between siblings would come into play, also other arithmetic rules if needed.
   (c) **Wrap:** Define the wrapping-up condition, as it may not always be the case that we would wrap up subtrees whenever encountered a different operator family. Sometimes we may want to wrap up things even within a same operator family, like what we are currently doing with the `if` function.

Currently, the only extension we have built into PRES is for the `if` function. It is currently treated as a singleton function, namely all children are stored in the *posdeck* of the DDB, the *negdeck* would always have no passenger, and one would always do the subtree wrapping along the way even within a same operator family. This is a limitation as it treats the following expressions differently although they are actually the same.

$$\texttt{if } (a>0) \texttt{ then } \big(\texttt{if } (b>0) \texttt{ then } c \texttt{ else } d\ \big) \texttt{ else } d$$
$$\texttt{if } (b>0) \texttt{ then } \big(\texttt{if } (a>0) \texttt{ then } c \texttt{ else } d\ \big) \texttt{ else } d$$

This can be easily improved by storing the *condition* part of the `if` function to be in the positive deck, storing the *cases* part to be in the negative deck, then defining the OPERATOR FAMILY LAW carefully on them. However, because of the time issue this idea has not yet been realized and will be addressed more in the future work.

5.2.6 Summary

A combination of PRES-1 in figure 5.1, PRES-2 in figure 5.5, PRES-3 in figure 5.8, PRES-4 and PRES-5 on page 49 forms the entire PRES — a linear time expression simplification algorithm. The core idea of Pres is the use of prime number and the product of prime numbers, which gives us both a linear time GCS solution, and a quartuple representation of the entire program tree.

The advantage of Pres Algorithm over other simplification algorithm is its theoretical linear time complexity, and its quartuple subtree representation, which makes the use of arithmetic rules in simplification becomes possible. PRES will take more advantage if the frequency of the operator family switching along the tree path is low, namely better if we can keep the DDB longer. We also believe that the Pres Algorithm can be applied to more widespread use other than just the genetic program simplification, as it is a very basic level algorithm.

5.3 Experimental Results

We applied the PRES algorithm to simplify all programs in the population, periodically during the evolutionary learning of GP, with simplification frequencies both of every two evolutions and every five evolutions. Tasks we used were fifteen multi-class object classification problems of increasing difficulty, with the Static Range Selection (SRS) as the classification strategy. Basically the experimental setting is the same as what we have described in chapter 3. Detailed results are listed in table 5.1, along with the result of not to do the simplification. All results are averages over fifty runs.

Table 5.1: GP with PRES simplification, on SRS multiclass classification

Dataset		NumCls	AvgNum of Generations			Terminating Time (sec)			Classification Accuracy (%)		
			Simplification Frequency								
			every 2	every 5	never	every 2	every 5	never	every 2	every 5	never
Shape		3	7.60	7.80	7.36	3.57	3.52	4.18	99.78	99.72	99.75
		4	22.56	23.28	20.94	11.10	12.29	14.68	98.95	99.14	99.41
Coin	5c	3	43.92	43.60	45.20	7.27	7.00	10.55	99.61	99.43	99.58
	10c	2	17.32	16.40	16.56	4.93	4.65	6.27	99.57	99.59	99.49
	easy	5	36.12	38.28	37.56	10.61	10.34	17.14	95.74	96.41	95.38
	hard	5	49.00	49.00	49.00	13.66	12.12	16.98	84.57	86.09	85.18
Digit	00	10	48.58	48.84	48.70	32.54	28.72	43.19	78.00	77.00	78.20
	05	10	49.00	49.00	49.00	27.53	25.42	40.43	71.19	70.61	69.61
	10	10	49.00	49.00	49.00	27.63	23.55	35.55	63.93	63.98	63.02
	15	10	49.00	49.00	49.00	24.92	21.92	31.93	57.72	58.64	56.92
	20	10	49.00	49.00	49.00	22.84	21.89	30.41	54.20	54.40	54.18
	30	10	49.00	49.00	49.00	22.21	20.28	28.37	44.51	45.06	43.95
	40	10	49.00	49.00	49.00	20.32	19.44	28.22	36.71	37.75	36.68
	50	10	49.00	49.00	49.00	19.63	18.70	27.23	31.29	32.56	31.20
	60	10	49.00	49.00	49.00	18.80	19.67	27.71	26.74	26.47	26.30

5.3.1 The Effectiveness

A) Effectiveness in terms of the Classification Accuracy

Figure 5.9 compares the effect of doing online PRES simplification with varying frequencies in terms of the classification accuracy. Three curves regarding no simplification and simplification for every two and five evolutions are about to overlap, reflecting the fact that online simplification would not affect the accuracy of the learning much.

Rather than "have not made the thing worth", we are quite supervised that to do simplification has even slightly improved the classification accuracy up to about 2%. It is not clear from the figure, though can be seen by looking at the value presented in table 5.1. Thus we made some detailed observation by tracing on the learning process, and observed that the wee improvement on the classification accuracy is from the fact that, in the later stage of the evolution, when the GP system seems not to be able to make further improvement in several evolutions, doing a simplification, which has a side effect of *restructuring* the entire expression, may result in a further improvement on the best population fitness, although often quite small. Further discussion on this point will be made shortly afterwards, in section 5.4.1.

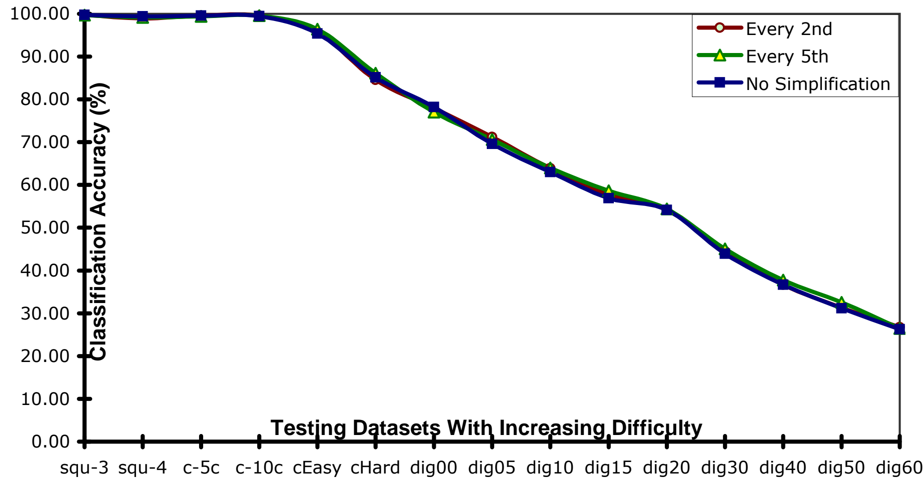


Figure 5.9: Effectiveness in terms of the classification accuracy.

B) Effectiveness in terms of Termination Generations

This is a convergency effectiveness measure about the possibility (ability) for the learning to achieve a certain level of accuracy. It would somehow also take part in the efficiency measure, which will be addressed in the next section.

Figure 5.10 compares the effect in terms of the number of generations used for terminating the learning, in our case it would be either a perfect fitness on the testing dataset, or reached the predefined limit of fifty generations (fourth-nine evolutions). From the figure we can see that the online simplification has little effect on this point either. Thus overall, together with the result on the classification accuracy shown before, the risk we were worrying about for doing online simplification, namely the potential of breaking good building blocks consequently reducing the ability of the learning, have not in any aspect happen to appear, which is quite a happy observation.

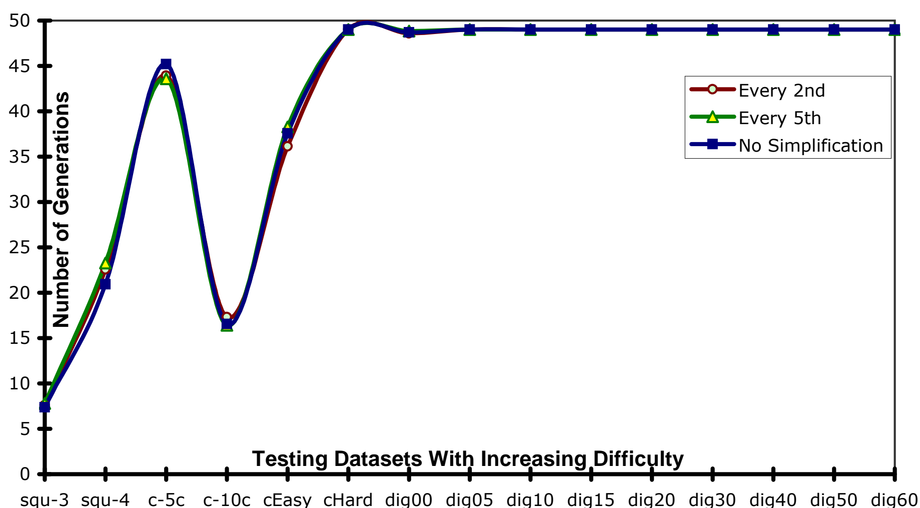


Figure 5.10: Effectiveness in terms of the terminating generation.

5.3.2 The Efficiency

A) Efficiency in terms of Time

Recall that our primitive goal of doing online simplification is to reduce the search space hence speed up the learning. Thus under the freedom from reducing the learning ability, the efficiency in terms of the learning time would be of our primary consideration.

Figure 5.11 compares results in terms of the CPU time (in second) used for the learning. By the fact that the number of generations used for completing the learning are about the same for all three cases, (from the overlapping of curves in figure 5.10), figure 5.11 would also reflect the averaging per-generation search time, although somehow proportionally scaled.

From the figure, it is obvious that the training time gets substantially shortened by doing PRES simplification during the learning. A remarkable point is that, to do simplification for every two evolution makes the entire learning slower than to do the simplification for every five evolution. This reflects the fact that separately deliberately doing online simplification would pay extra time for itself, which points out to us the importance of carefully choosing a reasonable simplification frequency.

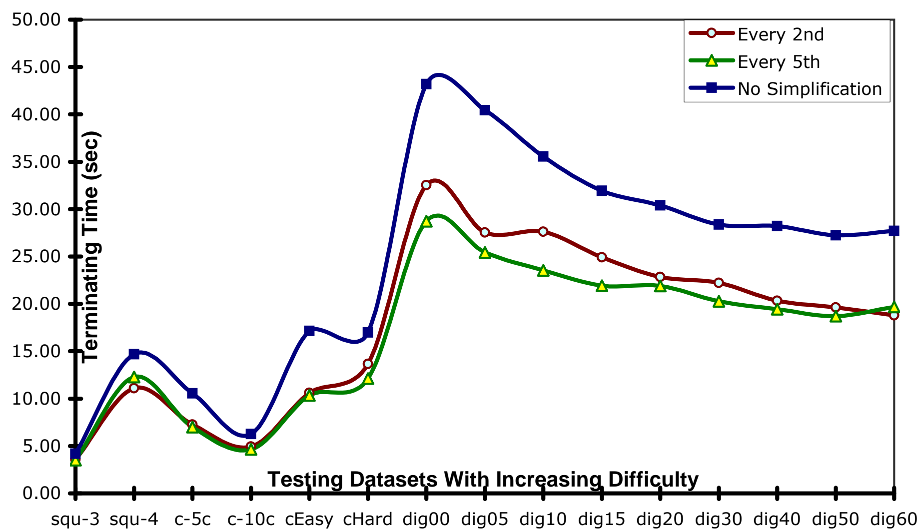


Figure 5.11: Efficiency in terms of the per-learning CPU time.

B) Efficiency in terms of Physical Computational Resources

The PRES algorithm is nice on its linear time complexity. However, it is much more memory consuming than many other polynomial simplification approaches, say, the straightforward string matching. However in these days, as memory is not of a serious concern anymore, the payment of hardware in consideration of saving the time is almost always worthwhile.

5.4 Further Analysis and Discussion

5.4.1 The Side Effect of the *Pres* Simplification — Program Restructuring

Recall that a sorting algorithm is *stable* if it maintains the original order of items with identical keys [**?**]. Such a definition can be extended to simplification algorithms:

> An expression simplification algorithm is **stable** if it never rearranges unsimplifiable subexpressions

The PRES algorithm, also many other expression simplification algorithms such as the basic string matching approach, is doing *unstable* simplification. When applying such an algorithm to a redundant expression, a side effect of *restructuring* will be brought in along with the elimination of the redundancy. For example, consider the redundant expression shown in figure 5.12, a *stable* simplification would result in an expression like in figure 5.13, though what the *unstable* Pres algorithm gives us is as one shown in figure 5.14. They are actually the same in terms of the computational effect, though differ in subtrees.

```
(% (* (if<0 (+ F3 F7) (* (- -0.528897 0.702463) -0.281440)
(if<0 0.548846 F3 F5)) (* (- F6 F3) (if<0 0.447347 0.796984
-0.839771))) (+ (if<0 (- -0.570596 F3) (+ (* -0.666692
0.752424) -0.267440) (+ F3 F4)) (if<0 (- -0.314318 -0.705341)
(- F0 0.951338) (% F0 0.954518))))
```

Figure 5.12: A Redundant Program

```
(% (* (if<0 (+ F3 F7) 0.346554 F5)
(* (- F6 F3) -0.839771)) (+ (if<0
(- -0.570596 F3) -0.769075 (+ F3
F4)) (% F0 0.954518)))
```

Figure 5.13: The *stable* simplification

```
(% (* (* (- F6 F3) -0.839771) (if<0
(+ F7 F3) 0.346554 F5)) (+ (if<0
(- -0.570596 F3) -0.769075 (+ F4
F3)) (* F0 1.047649)))
```

Figure 5.14: An *unstable* simplification (PRES)

Different from sorting algorithms, for simplification algorithms especially in a GP learning sense, we should just prefer an *unstable* solution. Which means, to get a side effect of *expression restructuring* along with the simplification is something we should be happy with. This is in consideration of building-blocks of the genetic beam search. With a stable simplification, we would end up with purely a reduction on the building blocks inside the program population. However, with an unstable simplification, which gives us just the same fitness, we may be able to also get new useful blocks, for free.

For a simple example, restructuring from `(-(+ab)c)` to `(+(-ac)b)` will introduce a somehow new block `(a-c)`, which could by itself be very useful if it gets recombined with, say, `(a+c)`, in the case that we are learning to simulate the polynomial $a^2 - c^2$.

The beauty of unstable simplification thus becomes obvious, as it gives us somehow new building blocks, though totally without taking the risk of reducing the population fitness, which is something we would have to face if we restructure the population by, say, large-scale crossover on an already highly fitted population.

5.4.2 The Potential of Simplification as a Genetic Operator (Future work)

The restructuring side effect of unstable simplification, which could give us new building blocks for free, inspires of the potential for extending the idea of online unstable simplification to a novel genetic operator.

Recall that a genetic operator is a process used in GP (also in GA the genetic algorithm) to maintain the *genetic diversity* [**?**], meaning that there should be many different versions of otherwise similar organisms to ensure the survival of the entire species. Current widely-accepted genetic operators are all biologically inspired, such as crossover and mutation, though the arithmetic-based unstable simplification, as it could somewhat lead to a similar effect, does have the potential of being extended to a genetic operator.

To see how, let us compare the unstable simplification with traditional genetic operators like crossover and mutation to see the gap. The similarity between them is just that they can all achieve a restructuring effect on the population. Differences between them are much richer, from the following aspects:

- The unstable simplification would take the advantage of not in any sense taking the risk of reducing the current fitness, thus it could be freely applied to the *entire* population. However, it has the drawback that its restructuring is only over each single program, thus its maximum restructuring ability is relatively small and restricted. Apart from that, another undesirable property is that the effect of restructuring by unstable simplification is always deterministic, as the instability of the unstable simplification is by itself deterministic, which is a common factor from its arithmetic basis. This is undesirable, because with learning by genetic beam search, we do need a certain level of uncertainty.
- The *real* traditional biologically inspired genetic operators is the other way round. Their restructuring scale is relatively wide, as for crossover that could be around one third of the entire population, up to the crossover ratio preferred. For mutation, the restructuring effect could even go to infinite, in principle. But the point is, with them we would have to take the risk of breaking down the existing achievement on the population fitness, which is something we really do not want to do especially on an already highly-fitted population.

Given these, the advantage of unstable simplification over the traditional genetic operator might be positive in the later stage of the learning. Just imagine, when the current population of programs has been learnt around to its maximum ability, namely when the GP system seems not to be able to make further improvement in a few number of evolutions, however we do believe the result should be able to be improved, it may because the current set of building blocks supported by the population is limited not to be able to represent the better solution, if we ask for new blocks from mutation, then we would take the risk of breaking the existing achievement. In this case if we do the unstable simplification we could just avoid this problem.

Currently, the determinism of the unstable simplification is a fatal bottleneck that blocks the effectiveness of using simplification as a genetic operator. Thus to realize the idea we would at least need to somehow introduce a certain level of uncertainty into the simplification, like some *Monte Carlo*[2] [**?**, **?**] or *Las Vegas* [**?**] way of doing simplification. They can all be interesting future works.

[2] A bonus point with *Monte Carlo Simplification* is that it would have an abbreviation of *MCS* ...

5.4.3 The Practical Achievement of the "Theoretically Linear Time"

The $\Theta(n)$ time Pres Algorithm has not shown a strong advantage on genetic program simplification over some other $O(n^2)$ versions, because Pres would take more advantage only if the "operator family switching frequency" along the tree branch is low, in which case we would end up with a really long *Double Decker Bus* thus could apply the very efficient PRES-1 on a long set.

Otherwise, with a high switching frequency of the operator family, Pres will mainly work on looking up in the hashtable and wrapping subtrees etcetera, which brings a big constant coefficient in front of the linear time complexity. It is also the case that, with a high operator-family switching frequency, the probability of the appearance of the redundancy will be substantially reduced, in which case some of the $O(n^2)$ algorithms will take a big advantage of, even reduce themselves to a $\Omega(n)$ with constant coefficient much smaller than of the PRES algorithm.

Unfortunately, according to our observation, in genetic programs, the operator family actually switches quite frequently along the branch. This makes us start to think of the applicability, or more precisely the *level* of applicability of the PRES algorithm on genetic operator simplification. From our detailed observation we observed the following properties of genetic programs:

- The switching frequency of operator families along the branch is often high.
- The probability of redundancies to appear on a compound term is inversely proportional to the size of the compound term. Namely, the longer the candidate redundancy, the lower the probability that it was a real redundancy.

The first fact heavily reduces the advantage of the PRES algorithm, as it would bring a big constant coefficient before the linear time complexity. However, together with considering the second fact, we start to think if it would be nice to consider only the primitive redundancy and completely not to worry about deeper level redundancies. If we did so, then the switching between operator family will not hurt us, thus the considerably reduced the potential constant coefficient that would go with the linearity.

The *building-block theory* of GP also provides a side-support for that *subtree simplification* (in contrast to the atom simplification) may not be sound, as it states that small-sized subtree (i.e. subtrees of about 5-10 nodes long) are the most possible valuable building blocks that the genetic beam search may prefer. Given this, not to do deeper level simplification (i.e. only simplify on atom nodes), may somehow reduce the probability of breaking good building blocks, which was a worry in the very first place.

To summarize, although the PRES algorithm is theoretically guaranteed to be in a linear time complexity, in real practice we *may* achieve even better results with some polynomial time simplification algorithms (need experimental supports). Another point, due to the fact that there is not many deeper level redundancies, we *may* also be able to get better results by only using PRES-1 and PRES-2 for lower-level simplification, namely, to totally "forget" about the potential existence of deeper-level redundancies (Lets call it *Naive Simplification*). This point, together with the "unstable simplification to genetic operator", are both taken as non-trivial future works.

5.5 Chapter Summary

In this chapter, we presented our linear time genetic program simplification solution – the PRES Algorithm, which consists of five sub-algorithms PRES-1 to PRES-5 regarding an "increasing" on the utility.

The algorithm was experimented on multiple-class classification tasks with increasing difficulty. Generally speaking, the experimental result does consistent with our primitive expectance on the online program simplification. Namely by doing the online simplification with PRES, the searching space of GP is reduced, hence the per-evolution learning time is shortened. The general effectiveness of the GP learning have not been reduced both in terms of the classification accuracy, and in terms of the ability of the convergency.

By the restructuring side effect of the *unstable* PRES algorithm, we were inspired on the potential of a Monte Carlo or Las Vegas liked simplification algorithm as a novel genetic operator. By further analyzing and detailed observation, we started to consider the soundness of doing only the level-one simplification online during the learning process of GP. Both points are richly discussed and be considered as reasonable and interesting future works.

Chapter 6

Conclusions and Future works

This project aims to develop new methods used in the genetic programming learning process, to extend the power of GP for multiple-class (object) classification tasks, for better effectiveness, efficiency, and the comprehensibility of the learnt program classifier, comparing with conventional GP approaches.

Two such methods has been developed and tested. They are the *Modi* program structure, a practical based design that gives substantial improvements on the classification accuracy; and the *Pres* simplification algorithm, which is a design mainly focus on its theoretical achievements. These two methods are summarized in the following two sections.

6.1 Modi

Modi is a structure that uses the standard tree to simulate loopy DAG. There are three major advantages of using the Modi structure as the architecture of genetic programs. Firstly, it allows the learnt program to reasonably output a vector of related values, thus provides us with a more coherent representation of the multiple-class classifier. Secondly, it allows reuse inside the program tree, thus somehow compresses the search space consequently results in a more efficient learning. The third point is that as the Modi structure is structurally equivalent to the standard tree, no extra constraint is needed to embed the new structure into the GP system, thus the applicability of the GP learning system totally has not been affected.

Modi has been tested on fifteen datasets with varying difficulty. The result shows that with Modi structured program, the learning accuracy of the GP system is substantially improved on multiple class classification, especially on relatively difficult tasks which there is still space for improvement.

The idea behind Modi, (viz. to take the program as a modifying-based procedure rather than an outputting procedure), is a very general thought that can be used in many other applications. The work of Modi also to some extent opens a new research direction in the area of genetic programming re multiple class classification, as past concentrations in this area were mostly on the development of new multiple class classification strategies. Modi, as a new program structure, is not in any sense conflicting with those works but can be thought as a side support for them.

Modi is still far from perfect. There is much room for further improvement of the structure. The idea behind Modi also derives a set of nontrivial research topics. Some of them are summarized below:

(1) Full-powered loopy DAG simulation: As pointed out before, Modi structure is only able to give us a proper subset simulation of the loopy DAG, not a full set one. This is due to the fact that, in Modi, reuse connections, namely acyclic loopy connections that cause children sharing, are simulated by the way Modi nodes pass the value, thus have to appear somehow around the Modi node but not free in place. Whether or not this theoretical shortcoming is practically hurt is in doubt. An analogue could be like the tradeoff that the *Naive Bayes* made on the full Bayesian. However, still use this analogue, as many researches have been done on full Bayesian, a full-powered loopy DAG simulation is something non-trivial to be furtherly investigated.

(2) Genetically evolving Neural Networks: Firstly I would like to clarify that, this thought, along with the following one, are purely *thoughts*. They may not be practically valuable for future investigating, but at least the idea is somehow cool and inspirational. The point with this thought is, as Neural Network is also a kind of loopy DAG, if one could learn the architecture of loopy DAG with GP, why couldn't they learn the architecture of Neural Networks, which is something currently has to be prefixed, with the same procedure.

(3) Genetically evolving Belief Nets: Currently the architecture of belief nets would have to be prefixed by human experts. This is okay, however there are still researches on predicating the architecture of belief nets. Therefore the idea behind Modi, namely use GP to learn the architecture of belief nets (as it is still, loopy DAGs in the worst case), is a possible direction.

(4) Vector-based fitness function: This is completely the idea of Malcolm Lett (BSc) for his assignment, though is shamelessly stolen and put in here (with permits) as a future work for Modi. The idea is, with the Modi structure, we are able to get a vector of outputs from the learnt program. However, the fitness decision that the learning system makes for each of the program is still based on a single value, namely the classification accuracy. If one could improve the fitness function (also the training dataset) to consider all output values of the program though separately, the fitness decision making would be more sound, also hopefully could improve the effect of the entire learning.

(5) Fairer modi nodes distribution law It would be great if we could guarantee that all cells in the output vector of the Modi structured program are taken by some Modi nodes. Thus we would like to investigate if it makes sense to add deliberate heuristic controls on the *modi node distribution problem* for fairer assignment. For a naive example, mixture of Gaussian instead of a simple uniform distribution for the *what* subproblem.

(6) More Experiments on the Digits We would like to investigate the digit dataset with a larger program size, to see if the performance can be improved.

(7) Have a classification strategy custom-made for Modi: As Modi can be considered as a side support for the study on classification strategies, whether or not a more plausible classification strategy can furtherly improve the performance of Modi-GP on multiple class classification is something non-trivial to think.

6.2 Pres

Pres is an expression simplification algorithm that can eliminate many kinds of redundancy, all together in theoretically linear time with respect to the number of nodes in the tree-representation of the expression to be simplified. The key idea behind Pres is the use of prime number, which gives us both a linear time Greatest Common Sub-multiset finding, and a quartuple representation of the entire program tree.

The algorithm is experimented on multiple-class classification problems with varying difficulty. Generally speaking, the experimental result does consistent with our primitive expectation on the online program simplification, namely it is able to reduce the searching space of GP hence speed up the learning. However, due the the natural of the types of redundancies in genetic programs, the linear time advantage of the Pres algorithm have not been strongly shown up. This point is seriously considered and extended into future works. Along with the works derived from the idea behind the Pres algorithm, following future works are under consideration:

(1) Monte Carlo Simplification as a Genetic Operator: Unstable simplification causes a program restructuring side effect. In other words, with unstable simplification, we would be able to get new building blocks though totally free from taking the risk of breaking the existing achievement on the fitness. Thus if one could bring some level of uncertainty into the simplification procedure, it would have the potential of becoming a qualified genetic operator.

(2) Build redundancy elimination into the fitness function: This is actually our initial thought, though failed to realize because we cannot figure out an appropriate way to build the redundancy elimination into the fitness function. Our thinking was to define the negative operators to be somehow different from its corresponding positive one, say make $a - b = a - b - 10$. In this way, $a + b - b$ would equal to $a - 10$, which makes the redundant part $b - b$ not as useless as before. This is just the very basic idea and is definitely not applicably by its own, however, to think under this track may be able to come out some valuable thoughts.

(3) Naive Simplification Due to the fact that there is not many deeper level redundancies, plus the theoretical support of the *building-block theorem*, we *may* be able to get better results by only using PRES-1 and PRES-2 for lower-level simplification, namely, to deliberately "forget" about the potential existence of deeper-level redundancies. If this is a sound thought is up to experiments.

(4) Making more use of the quartuple representation: An advantage that the use of prime numbers gives us is the quartuple representation of the entire tree. This makes us start to think about if it could be possible to evolve on the quartuple by using some learning algorithm that is good on learning values, say, genetic algorithms, or neural networks. This is a very raw thought that current sound stupid, but I do believe that we must be able to do something with the quartuple representation.

(5) Better hash function: The current hash function on the quartuple key is just a two-phase `mod` function with two prime numbers `47` and `13` as the base. Future studies on the property of prime numbers to work out a more efficient hash function (and hash structure) would be necessary. Current thoughts of potential hashing functions include the inverse of function $f(n) = n^2 - n + 41$ and $f(n) = n^2 - 79 + 1601$ suggested by Prof. Rob Goldblatt (from a Mathematical sense); and a six-page-long one-to-one (single) formula that maps the first 500 prime numbers into the first 500 natural numbers, provided by my friend QW (from a Physics sense). From a computer science sense I would like to do a bunch of experiments on a set of integer numbers to see which one could be the best to serve as the base of the `mod` function.

(6) Better handling of `if`: As addressed before on page 5.2.5, the current handling on simplifying the *if* function can still be improved. This can be done by storing the *condition* part of the `if` function to be in the positive layer, and store the *cases* part to be in the negative layer, with some links between them. To do this, one may need to slightly alter the structure of the Pres algorithm, or the representation of the function node, though should not be too much work.

(7) Arithmetic Overflows: A shortcoming of using prime numbers and the product of prime numbers that have not been addressed before is that, the product of the prime number could easily get very big and exceed the 8-bytes long long thus cause arithmetic overflows. This can be easily solved by, say, future enlarge the storage space for the prime product when we find it overflows, though it has not yet been coded into the algorithm, as we currently do not have this problem with our small set of prime numbers.

Appendix A

Experimental Result - Tabular

Table A.1: Result of Basic-SRS

Dataset		Number of Classes	Average Number of Evolution	Terminating Time (sec)	TestSet Classification Accuracy (%)
Shape	4squr:se	3	7.44	4.29	99.71
		4	21.00	14.51	99.40
Coin	4squr:5c	2	45.34	10.84	99.57
	4squr:10c	3	17.72	6.78	99.47
	4squr:ce	5	37.38	16.74	95.74
	4squr:ch	5	49.00	17.38	85.22
Digit	dig00	10	48.70	41.62	78.00
	dig05	10	49.00	37.07	69.63
	dig10	10	49.00	38.45	63.07
	dig15	10	49.00	32.78	56.85
	dig20	10	49.00	32.31	54.76
	dig30	10	49.00	28.52	44.09
	dig40	10	49.00	28.95	36.92
	dig50	10	49.00	27.62	30.95
	dig60	10	49.00	28.33	26.47

Table A.2: GP on Object Classification: Modi on the Shapes [1/1]

Dataset	Number of Classes	ModiRate	Average Number of Generations	Terminating Time (s)	TestSet Classification Accuracy (%)
Win :4 square Data:squ BG :clear	3	0.0	1.50	1.42	99.79
		0.1	0.64	0.96	99.85
		0.2	0.20	0.71	99.84
		0.3	0.48	0.89	99.78
		0.4	0.32	0.80	99.94
		0.5	0.44	0.87	99.93
		0.6	0.40	0.85	99.87
		0.7	0.39	0.85	99.88
		0.8	0.36	0.83	99.87
		0.9	0.54	0.93	99.80
		1.0	0.30	0.78	99.89
		1.3	0.28	0.79	99.94
		1.6	0.30	0.82	99.86
		1.9	0.32	0.82	99.77
	4	0.0	7.20	7.50	99.71
		0.1	5.62	6.52	99.69
		0.2	5.46	6.31	99.53
		0.3	5.34	5.93	99.70
		0.4	4.90	5.93	99.12
		0.5	5.48	6.46	99.65
		0.6	5.50	5.59	99.65
		0.7	4.42	5.08	99.77
		0.8	4.54	5.10	99.54
		0.9	5.00	5.45	99.67
		1.0	5.30	5.46	99.69
		1.3	4.72	5.54	99.38
		1.6	4.94	5.69	99.49
		1.9	4.84	5.39	99.70

Table A.3: GP on Object Classification: Modi on the Coins [1/1]

Dataset	Number of Classes	ModiRate	Average Number of Generations	Terminating Time (s)	TestSet Classification Accuracy (%)
Coin 5 cents Win :4 squares Data:5c not-rotated BG :fuzzy	3	0.0	36.00	15.14	99.57
		0.1	38.30	18.59	99.41
		0.2	37.34	17.59	99.67
		0.3	39.30	19.31	99.58
		0.4	38.96	18.97	99.55
		0.5	41.26	18.78	99.61
		0.6	42.12	20.06	99.51
		0.7	41.12	18.72	99.38
		0.8	42.76	19.67	99.56
		0.9	42.12	20.33	99.61
		1.0	39.56	18.18	99.57
		1.3	38.30	19.26	99.50
		1.6	38.88	20.94	99.34
		1.9	39.34	19.31	99.74
Coin 10cents Win :4 square Data:10c rotated BG :fuzzy	3	0.0	3.56	3.14	99.89
		0.1	2.72	2.75	99.88
		0.2	2.80	2.92	99.91
		0.3	2.34	2.32	99.85
		0.4	2.50	2.44	99.82
		0.5	2.50	2.55	99.88
		0.6	3.36	4.19	98.39
		0.7	2.06	2.18	99.89
		0.8	2.48	2.54	99.87
		0.9	2.60	2.61	99.90
		1.0	2.88	2.75	99.75
		1.3	2.64	2.66	99.85
		1.6	2.88	2.80	99.83
		1.9	2.54	2.57	99.72
Coin Easy Win :4 squares Data:5c&10c rotated BG:clear	5	0.0	17.16	13.00	98.61
		0.1	13.94	11.73	99.17
		0.2	17.58	12.89	98.98
		0.3	16.04	12.21	98.72
		0.4	15.76	12.59	98.99
		0.5	16.76	12.57	98.93
		0.6	16.37	12.21	98.78
		0.7	15.66	12.03	98.92
		0.8	17.08	11.94	98.72
		0.9	16.94	12.33	98.83
		1.0	17.70	12.40	98.82
		1.3	16.84	12.45	98.87
		1.6	14.82	11.15	98.93
		1.9	20.48	15.55	98.23
Coin Hard Win :4 squares Data:5c&10c rotated BG:fuzzy	5	0.0	47.56	35.38	93.43
		0.1	44.48	31.72	93.78
		0.2	46.72	34.21	93.70
		0.3	43.22	30.18	93.53
		0.4	43.24	29.85	93.67
		0.5	45.24	31.06	92.93
		0.6	42.38	27.47	93.76
		0.7	41.64	27.97	92.55
		0.8	43.73	28.77	93.31
		0.9	45.98	31.40	92.00
		1.0	42.84	27.28	92.71
		1.3	46.34	31.31	92.29
		1.6	38.54	24.48	93.89
		1.9	42.69	28.61	91.65

Table A.4: GP on Object Classification: Modi on the Digits [1/3]

Dataset	Number of Classes	ModiRate	Average Number of Generations	Terminating Time (s)	TestSet Classification Accuracy (%)
dig00 Fuzziness=00	10	0.0	49.00	80.88	76.50
		0.1	48.32	81.34	80.40
		0.2	48.78	81.49	80.40
		0.3	48.52	76.22	82.00
		0.4	48.26	74.48	81.60
		0.5	49.00	73.33	79.60
		0.6	48.04	75.61	81.20
		0.7	48.70	75.52	80.00
		0.8	47.98	71.88	78.40
		0.9	49.00	72.15	80.60
		1.0	48.62	73.12	79.80
		1.3	48.16	74.98	81.00
		1.6	49.00	78.75	79.40
		1.9	48.80	75.13	78.00
dig05 Fuzziness=05	10	0.0	49.00	79.68	74.73
		0.1	49.00	86.57	75.93
		0.2	49.00	77.42	76.90
		0.3	49.00	79.41	77.11
		0.4	49.00	80.35	76.05
		0.5	49.00	74.56	76.76
		0.6	49.00	80.68	75.87
		0.7	49.00	78.81	77.13
		0.8	49.00	74.23	76.60
		0.9	49.00	76.51	78.14
		1.0	49.00	81.34	78.12
		1.3	49.00	76.36	76.71
		1.6	49.00	78.30	75.06
		1.9	49.00	75.94	75.46
dig10 Fuzziness=10	10	0.0	49.00	73.37	69.47
		0.1	49.00	83.11	70.51
		0.2	49.00	76.40	70.62
		0.3	49.00	80.77	71.87
		0.4	49.00	75.47	72.24
		0.5	49.00	78.16	71.88
		0.6	49.00	80.48	70.43
		0.7	49.00	75.65	71.61
		0.8	49.00	75.84	72.00
		0.9	49.00	70.31	70.14
		1.0	49.00	75.06	72.52
		1.3	49.00	76.80	73.04
		1.6	49.00	76.22	69.61
		1.9	49.00	69.05	68.83

Table A.5: GP on Object Classification: Modi on the Digits [2/3]

Dataset	Number of Classes	ModiRate	Average Number of Generations	Terminating Time (s)	TestSet Classification Accuracy (%)
dig15 Fuzziness=15	10	0.0	49.00	73.12	63.76
		0.1	49.00	82.36	65.76
		0.2	49.00	74.02	65.89
		0.3	49.00	74.30	65.27
		0.4	49.00	74.75	66.83
		0.5	49.00	76.09	68.11
		0.6	49.00	76.13	67.41
		0.7	49.00	70.77	66.32
		0.8	49.00	75.35	66.22
		0.9	49.00	72.13	65.61
		1.0	49.00	72.89	66.98
		1.3	49.00	73.24	65.53
		1.6	49.00	73.08	65.97
		1.9	49.00	75.78	64.04
dig20 Fuzziness=20	10	0.0	49.00	73.02	60.11
		0.1	49.00	76.45	60.70
		0.2	49.00	73.82	62.45
		0.3	49.00	74.41	61.97
		0.4	49.00	70.07	61.50
		0.5	49.00	71.10	63.16
		0.6	49.00	72.72	61.97
		0.7	49.00	71.20	61.28
		0.8	49.00	71.92	62.17
		0.9	49.00	70.40	61.46
		1.0	49.00	69.01	62.26
		1.3	49.00	69.83	59.99
		1.6	49.00	72.61	60.19
		1.9	49.00	68.07	60.28
dig30 Fuzziness=30	10	0.0	49.00	71.35	52.04
		0.1	49.00	71.41	52.29
		0.2	49.00	69.02	52.46
		0.3	49.00	71.34	53.36
		0.4	49.00	70.60	52.02
		0.5	49.00	65.83	54.46
		0.6	49.00	71.23	52.42
		0.7	49.00	67.94	52.40
		0.8	49.00	74.27	52.70
		0.9	49.00	72.02	53.45
		1.0	49.00	66.76	53.34
		1.3	49.00	67.61	51.81
		1.6	49.00	66.65	51.83
		1.9	49.00	72.72	51.61

Table A.6: GP on Object Classification: Modi on the Digits [3/3]

Dataset	Number of Classes	ModiRate	Average Number of Generations	Terminating Time (s)	TestSet Classification Accuracy (%)
dig40 Fuzziness=40	10	0.0	49.00	70.95	43.88
		0.1	49.00	71.53	44.11
		0.2	49.00	70.34	44.46
		0.3	49.00	69.77	44.88
		0.4	49.00	71.58	44.71
		0.5	49.00	68.26	45.58
		0.6	49.00	71.49	44.27
		0.7	49.00	68.06	44.63
		0.8	49.00	67.60	44.64
		0.9	49.00	65.70	44.48
		1.0	49.00	73.40	43.35
		1.3	49.00	67.93	44.12
		1.6	49.00	72.18	42.21
		1.9	49.00	66.53	42.06
dig50 Fuzziness=50	10	0.0	49.00	69.48	37.82
		0.1	49.00	73.35	38.08
		0.2	49.00	69.20	37.62
		0.3	49.00	68.06	39.61
		0.4	49.00	66.73	37.84
		0.5	49.00	66.36	38.82
		0.6	49.00	65.42	38.40
		0.7	49.00	66.78	37.41
		0.8	49.00	68.50	38.19
		0.9	49.00	62.71	37.58
		1.0	49.00	63.12	38.18
		1.3	49.00	64.83	38.92
		1.6	49.00	67.93	37.56
		1.9	49.00	66.65	37.00
dig60 Fuzziness=60	10	0.0	49.00	68.00	31.12
		0.1	49.00	66.02	32.11
		0.2	49.00	72.21	31.36
		0.3	49.00	69.18	31.19
		0.4	49.00	63.84	30.52
		0.5	49.00	67.04	32.13
		0.6	49.00	61.88	31.46
		0.7	49.00	63.85	31.51
		0.8	49.00	67.91	31.64
		0.9	49.00	65.03	32.11
		1.0	49.00	63.56	30.09
		1.3	49.00	67.63	31.61
		1.6	48.12	64.07	32.49
		1.9	49.00	61.93	29.88