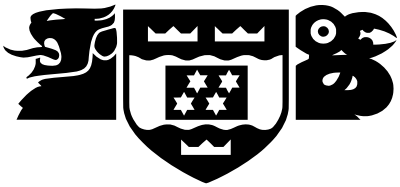
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



School of Mathematics, Statistics and Computer Science
Te Kura Tatau

PO Box 600
Wellington
New Zealand

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

New Operators in Linear Genetic Programming for Multiclass Classification

Carlton Downey

Supervisor: Mengjie Zhang

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

Abstract

Genetic programming (GP) has had some limited success with solving multi-class classification problems, however the performance of GP classifiers still lags behind that of alternative techniques. This project investigates an alternative form of GP, Linear GP (LGP), which demonstrates great promise as a classifier. By utilizing the structure of LGP programs, we develop several new LGP operators which provide a significant performance improvement. These include a new mutation operator which identifies bad instrutions for mutation, and a new crossover operator which mimics biological crossover between alleles.

Acknowledgments

First and foremost I would like to thank my supervisor, Mengjie Zhang, for the time and effort he has invested in teaching me how to perform research. I would also like to thank my parents for their continued love and support though the highs and lows of Honours, as well as their help in last minute proof reading. My comrades in Memphis must be mentioned for their many helpful suggestions, particularly their knowledge on the many peculiarities of LaTeX, and their suggestions on appropriate Honours music. Finally, I would like to thank my girlfriend Xiaowen for her love, support, and understanding when time and again I was forced to plead work.

Contents

Chapter 1

Introduction

Identifying and classifying objects is a common and important task which humans perform daily. Knowing the difference between the cereal and the rat poison, identifying which person is your mother and which is your father, these and many similar classification tasks are all extremely important. These problems are so trivial for humans that we do not even realize we are performing them all the time. However, there are other classification tasks which only some human experts with specialized knowledge can perform, and where even experts often make mistakes. An example of such a problem is determining if a medical test result is positive for cancer. For many years people have been designing algorithms and machines to assist in classifying data where domain experts are rare and expensive [13]. This approch is also applied to data which is too complex, too abstract, or too copious for human understanding.

This project focuses on solving classification problems of a particularly challenging type: multiclass classification problems. These are classification problems where the number of possible classes is large, such as digit recognition or face recognition. It is relatively easy to solve classification problems where we need only decide between two alternatives. However when the number of alternatives is large the problem difficulty typically increases as well [6].

The problems we are most interested in training machines to solve are those which are sufficiently complex that we cannot design machines to precisely solve them. We have to *train* some machine on existing data, much as a human learns how to read and write. Improving the performance of an existing machine by training it on data is the goal entire goal of machine learning. Two things are important when training machines: how quickly we can train them and how well they perform after training. Clearly a machine which has perfect performance after after many centuries of training is of little use to us, and similarly a machine which can be trained in a matter of seconds but has terrible performance is also of no use.

GP is a method for *evolving* computer programs as potential solutions, in order to solve a user defined problem [15]. GP is inspired by biological evolution, where life forms adapt to optimize their performance in their environment. Similarly we wish to evolve computer programs to give the best possible performance on the problems we want to solve [7, 10].

Genetic Programming (GP) can be used to find a good solution to many problems quickly and automatically. GP has several advantages over alternative methods. In particular, it is a more flexible technique and allows for creation of potential solutions not envisioned by the programmer [15]. Therefore the ability to use GP methods to solve multiclass classification problems is a highly desirable outcome.

Conventional GP methods using Tree based GP (TGP) are effective at solving many binary classification tasks, but often perform poorly on multiclass classification problems [33].

An alternative form of GP, Linear GP (LGP) has been shown to have superior performance to TGP on many classification problems [8] while retaining the good features which make conventional GP attractive as a search technique. However despite its promise, according to our investigations it appears that relatively little research has been done to improve the performance of LGP. Hence we believe that LGP is a rich ground for research into improving the performance of GP on multiclass classification problems.

1.1 Goals

We aim to investigate LGP and compare it with conventional TGP. We then aim to improve the LGP learning algorithm by developing several new operators for LGP and determining their effectiveness.

Specifically, this report aims to achieve the following research goals:

- To compare the performance of conventional TGP techniques to conventional LGP and establish a set of baseline results.
- To devise a model of instruction correctness and use it to develop a new mutation operator which preferentially mutates poor instructions.
- To determine an abstract structure for LGP programs, and use this structure to develop a new crossover operator which alleviates the problem of building block disruption.
- To improve our new crossover operator by devising a heuristic which predicts which parts of the parent programs should be exchanged in order to maximize the probability of high fitness offspring.
- To further improve the new crossover operator by developing elitism and selection operators which complement previously developed techniques.

In order to investigate these goals, three multiclass classification problems of increasing difficulty will be used to empirically compare performance in a series of expierments.

1.2 Contributions

This project makes the following contributions:

- This project shows that basic LGP performs comparably to a state-of-the-art TGP classifier. To do this we describes how to use linear genetic programming (LGP) to solve multiclass object classification problems and develop a package in java for performing LGP. This and an existing TGP package are used to empirically compare the performance of several TGP classification strategies to LGP. Statistical significance testing is used to show that LGP can significantly outperform a simple TGP approach.
- This project develops a new mutation operator based on the principle of preferentially mutating the badly performing parts of programs. As far as we have been able to ascertain this new operator is significantly different to all existing techniques. This new operator significantly outperforms the existing mutation operator on all problems. To develop this operator we devise a model of instruction correctness for LGP programs. A paper on this method was submitted to the IVCNZ '09 conference for review and publication and we are currently awaiting acceptance.

- This project develops a new form of crossover based on the LGP program structure devised during development of the selective mutation operator. As far as we have been able to ascertain this new operator is significantly different to all existing techniques. It is shown that the this new crossover operator outperforms the conventional crossover operator on all problems. A combinatorial argument is used to show that this new form of crossover restricts the number of possible exchanges and is less disruptive to building blocks than conventional crossover.
- This project develops a new crossover operator which increases the likelihood of high fitness offspring without significantly increasing the computational cost. We develop a heuristic which can be used to predict which program code should be exchanged to give the best offspring. It is shown that the this new crossover operator outperforms the conventional crossover operator and the previously developed crossover operator on all problems.
- This project develops two new operators which focus on increasing the likelihood that the two programs chosen as parents for crossover have diverse strengths. It is shown that LGP with these new operators and our new crossover operator has significantly superior performance to LGP with only the new crossover operator. We show how we can increase the diversity of the population through a new elitism operator. We also show how we can select the second parent to complement the first parent through a new selection operator.

1.3 Organization

The remainder of this paper is organized as follows:

- Chapter 2 covers a survey of the relevant literature in this area.
- Chapter 3 describes the data sets and parameter configurations used for empirical testing during the course of this project.
- Chapter 4 describes the first of the contributions, a comparison of TGP and LGP as methods for solving multiclass object classification problems.
- Chapter 5 devises an abstract model for LGP programs and uses this to develop a selective mutation operator which focuses on mutating poor instructions.
- Chapter 6 further develops the abstract structure of LGP programs and uses this structure to develop a new crossover operator which alleviates the problem of building block disruption.
- Chapter 7 devises a heuristic which can be used to predict the benefits of exchanging two class trees. This heuristic is used to improve the crossover operator developed in chapter 6.
- Chapter 8 develops a new elitism operator and a new selection operator both of which act to increase the likelihood that crossover occurs between parents with diverse strengths.
- Chapter 9 presents the project conclusions as well as possible future work.
- An outline of the new program package JVUWLGP is described in Appendix A.

Chapter 2

Literature Survey

This chapter reviews the current knowledge and previous research upon which this project builds. Firstly, we give an overview of *Machine Learning*, *Evolutionary Computation*, *Genetic Programming* (GP) and Linear Genetic Programming (LGP). This is followed by an overview of classification problems and GP work related to solving such problems. We conclude with a brief section on current limitations.

2.1 Overview of Machine Learning

Machine Learning is a subfield of Artificial Intelligence (AI) which deals with automating the development of some part of a system which performs some task [4, 8]. The part of the system being learnt can be the algorithm itself, the parameters to the algorithm, or the learning process. Any automated process which learns from data is performing machine learning, be it teaching a robot to walk or training a computer to recognize cancerous tissue. This data can be anything from a large database of customer data to real time video feeds, anything which exhibits patterns which can be learnt. There are three major types of Machine Learning: *Supervised Learning*, *Unsupervised Learning* and *Reinforcement Learning*[27]. This project focuses on Supervised Learning, but we will briefly describe and contrast all three methods.

2.1.1 Main Types of Machine Learning

Supervised Learning/Classification

In supervised learning [25], elements of the input data set are labeled with their desired output. The machine reads in the inputs and computes an output, which it can then compare to the desired output. If the generated output differs from the desired output, the machine then adjusts its internal structure to correct the generated output. This can be visualized as someone supervising a machine while it tries to learn, and telling it when it makes a mistake. A typical example of supervised learning might be to identify cancerous tissue in medical testing. The machine would be presented with a series of test results which have already been labelled by a medical expert.

Classification is an area of Supervised Learning. A classification problem is one where we want to determine the class or type of an object given some data about the object. Typical examples include determining the gender of a person based on their taste in clothing, or recognizing handwritten digits from their individual pixels. When attempting to solve a classification problem we train a machine to correctly out the class of various objects given

that data about them is provided as input. Classification problems come in two types: binary classification problems and multiclass classification problems.

- Classification into $n = 2$ classes is called binary classification.
- Classification into $n > 2$ classes is called multiclass classification.

Many machine learning techniques, particularly genetic programming, excel at binary classification tasks but struggle with multiclass classification[8].

The aim of classification problems is to maximize the number of test set instances correctly classified, as this is a measure of the classifiers performance. Hence the fitness of a classifier is proportional to the number of incorrectly classified training set instances.

We can measure the performance of a learned classifier using a variety of techniques. The most elementary approach is the classification accuracy. This is the number of objects which are correctly classified over the total number of objects. A perfect classifier has 100% classification accuracy, i.e. it classifies every object correctly. The worst possible classifier has 0% classification accuracy, i.e. it classifies every object incorrectly.

Classification is difficult because two members of the same class can vary greatly, or two members of different classes can look very similar [6]. For instance if you look at any two peoples handwriting, you will immediately see differences. It is hard for machines to learn to ignore these differences and focus on the particular features which are what really characterize the digit.

Unsupervised Learning

In unsupervised learning, elements of the data set are unlabeled. This means that unlike supervised learning, there is no expected output which the machine can use to evaluate its current performance. Instead, in unsupervised learning we use some distance metric to measure the distance between data points. We can then look for patterns in the data based on the distances between data points. Unsupervised learning is typically interested in finding correlations in the data set, for instance determining the relationship between age and income level [6].

Reinforcement Learning

In reinforcement learning, the machine is rewarded every time it performs a useful action [25]. In this way we guide the machines' learning without expecting specific behavior. The machine will start off performing random actions, and over time will use the rewards associated with these actions to shape its future behavior. These rewards may be either positive or negative, for instance a positive reward may occur when a robot soccer player scores a goal, and a negative reward when the other team scores.

2.1.2 Paradigms of Machine Learning

There are four main paradigms in Machine Learning:

- Case-based Learning Paradigm: In Case-based learning we directly compare the test instance to the training data. An example of case-based learning is the Nearest Neighbor Algorithm, where we classify a new object by finding the closest existing object with a known label.

- **Inductive Learning Paradigm:** In Inductive Learning we derive a rule from the training data based on generalization, and apply this rule to instances of the test set. An example of Inductive Learning is the Decision Tree classifier [22].

- **Connectionist Learning Paradigm:** In Connectionist Learning we learn by making and changing connections between nodes. An example of Connectionist Learning is Neural Networks (NNs). NNs are based on mathematical models of groups of neurons. Typically we decide on a network structure before training, and learning is purely based on changing the values, or *weights*, on the connections between nodes [24].

- **Evolutionary Learning Paradigm:** In Evolutionary Learning we learn by improving individuals in a population over many generations, where individuals are candidate solutions to a problem [14, 5, 9]. This results in a number of individuals who are well suited to solving our problem. This can be easily visualized in terms of biological evolution, where the species better suited to the environment survive, and the less well suited ones die off. This type of learning is discussed in more detail in section 2.2.

2.1.3 Data Sets

Data sets are required in Machine Learning to train the machine and test its effectiveness once trained. Typically a data set is divided up into three parts; a *training set*, a *validation set*, and a *test set*. The training set is used to train the machine by comparing outputs and adjusting internal structure. The test set is used to evaluate the performance of the machine on unseen data. The validation set is used to prevent over-fitting during training. Over fitting occurs when the machine stops generalizing and starts learning specific details about the elements in the training set. When over-fitting occurs the performance on the training set increases, but the performance on the test set decreases.

2.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are a Machine Learning paradigm inspired by biological evolution devised in the 1950's. EAs mimic Darwinian natural selection for the purpose of optimizing a solution to a predefined problem [7, 10]. General EA learning procedes as follows:

- Initialize a population of random *individuals* (an individual is a candidate solution). Examples of individuals include computer programs and the parameters for algorithms.

- Select individuals from the population to reproduce, favoring individuals with better *fitness* (the fitness of an individual is proportional to how well they perform on the training data). Basically, better programs should reproduce more often.

- Generate a new population of individuals by applying *genetic operators* to the selected individuals. Genetic operators change the "DNA" or code of our programs in a similar way to what occurs in biological reproduction. Standard genetic operators are:

  - Reproduction: Copy an individual without change into the new population.
  - Recombination: Randomly exchange substructures between individuals and place both into the new population.
  - Mutation: Randomly replace a substructure in an individual and place it into the new population.

- Iterate steps 2-3 until either a perfect solution is found, or the iteration limit is reached. Each iteration is referred to as a *generation* due to the obvious biological parallels.

All EAs follow this general model, and the borders between different types of EA's have blurred over the years, however several distinct approaches still exist. These approaches include Genetic Algorithms (GAs), Evolutionary Strategies (ES), Genetic Programming (GP), particle swarm optimization, and ant colony optimization. In this project we review only the most common ones such as GA, ES and GP. GP will be discussed in the next section in detail as it is the focus of this project.

2.2.1 Genetic Algorithms

GAs, first popularized in 1975 by J. Holland [11], are another approach to Evolutionary Algorithms. In GA the individuals are fixed length bit strings known as *chromosomes* and the primary genetic operator is crossover. The emphasis placed on recombination is the key distinguishing feature of Genetic Algorithms. In crossover two *parent* individuals exchange parts of their chromosomes to create two new individuals.

2.2.2 Evolutionary Strategies and Evolutionary Programming

ES and EP are two very similar EA approaches. Both use vectors of real numbers to represent individuals. Both use mutation as the primary genetic operator. The key difference is that in ES the vectors of real numbers themselves are the solutions, while in EP the vectors are translated into finite state machines.

2.3 Genetic Programming

GP is the technique of evolving computer programs to solve problems. GP is the focus of this research project and will be covered in much more depth then the previous three EA approaches. GP as a technique was first mentioned by Stephen F. Smith in 1980 and Forsyth in 1981 [9], but the first instance of GP in its current form was given by Cramer in 1985 [5]. The technique was expanded on and popularized by Koza in 1989 [16]. GP is a relatively new approach to Machine Learning. The distinguishing features of GP are its representation and its operators; the individuals in GP are stand-alone programs and the operators are modified accordingly. There are several different representations in GP which have been explored to varying degrees, the three major ones are detailed below.

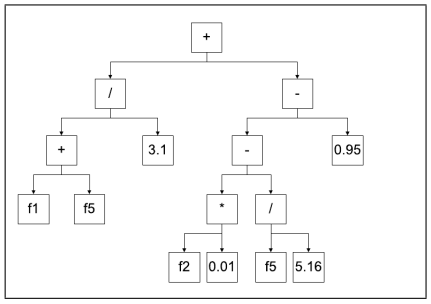
2.3.1 Tree-Based GP

The original representation for GP programs/individuals, and still the most common, is *Tree-Based GP (TGP)* [16]. In TGP programs are either trees or Lisp S-expressions, with these two representations being interchangeable. An example TGP program is shown in Figure 2.1.

A tree is made up of functions, constants and feature values. The leaf nodes of the tree are constants and feature values and the non-leaf nodes are functions. Constants return a constant value set when the individual is initialized, feature values return the value associated with the training example being evaluated, and functions return the result of the function applied to its children. The functions are drawn from a pool determined by the programmer and depend heavily on the program domain. Typical examples include addition, subtraction, multiplication, protected division and simple conditionals such as if statements. The

```
(+ (/ (+ f1 f5)
     3.1)
  (- (- (* f2 0.01)
        (/ f5 5.16)
     )
     0.95
  )
)
```

(a) Program code



(b) Functional tree

Figure 2.1: An example TGP Program

arguments include image features, constant real numbers (constants), and other arithmetical expressions.

The tree has a single root node, so the output of a program for a given training example will be a single value. In basic Tree-Based GP all nodes return the same type and all functional nodes take the same type as arguments. This type is often floating point numbers. There is also a variant of GP called Strongly Typed Genetic Programming (STGP) which allows nodes to input and output different values [12]. This has the advantage of allowing for more powerful functions, at the cost of increased complexity and decreased flexibility of genetic operators.

2.3.2 Linear Graph GP

Linear GP (LGP) [2, 3] is another widely used cousin of Tree-Based GP. In LGP, programs are finite sequences of instructions from an imperative programming language or machine language, quite different to the functional trees which characterize TGP. Typical languages which are often used for LGP are such as Java, C++, or register machines, in this project we use the language of register machines.

- Register machines consist of an array of values, known as the *registers*, and a sequence of instructions which operate on these registers.
- Register Machine instructions consist of one destination register, one operator, and two arguments. The arguments can be constants, image features, or other registers.
- An LGP program is executed by initializing all registers to the value 0, then executing the instructions in order. The output consists of the final register values: the numbers in the registers after all instructions have been executed.
- The number of registers is determined by the user based on the problem.

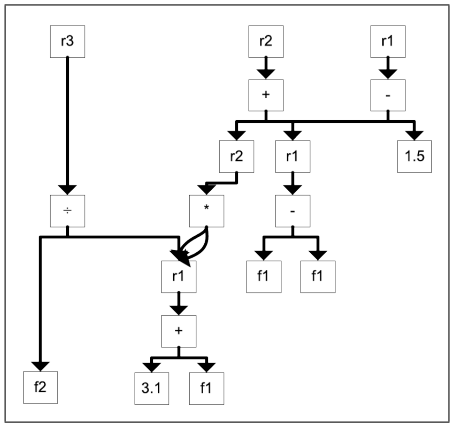
The program is a finite sequence of instructions, and the output is read from the appropriate registers when the program terminates. An example LGP program is shown in Figure 2.2.

Graph GP

Linear GP can also be seen as Graph GP, because a LGP program can be translated into a multi rooted directed graph. This gives LGP a distinct advantage in flexibility compared to Tree-Based GP, where all programs must conform to the tree model with a strict hierarchy and a single output.

```
r[1] = 3.1 + f1;
r[3] = f2 / r[1];
r[2] = r[1] * r[1];
r[1] = f1 - f1;
r[1] = r[1] - 1.5;
r[2] = r[2] + r[1];
```

(a) Program code



(b) Directed Acyclic Graph

Figure 2.2: An example LGP program

Code Introns

Linear GP has the distinctive feature of code *introns*, regions of non-effective code resulting from the *structure* of the program [2]. Intron instructions have no effect on the program output simply by virtue of the program structure. For instance, if we assign a value to a register, any previous value held in that register is deleted. If the value was not previously used then the instructions which calculated it are redundant and hence introns. Introns are so named because they are similar to *biological introns*, redundant areas of genetic material appearing seemingly at random within an organism's DNA. Conflicting views are held over code introns, with some experts holding that they are vital to code evolution, and others holding they serve no useful purpose and simply slow down execution. Those who believe introns aid program evolution do so for two reasons: introns may reduce the effects of variation on effective code, and introns allow for code variation which does not change code fitness.

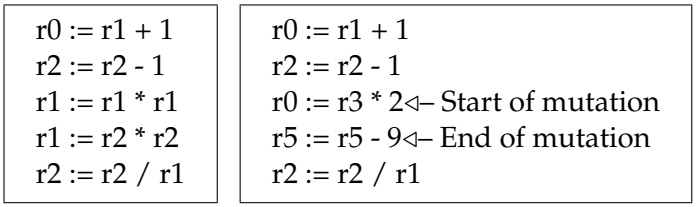
2.4 Terminal Set and Functional Set

Two important aspects of GP are the terminal and functional sets. The functional set is the set of all possible functions which can appear in evolved programs. The terminal set is the set of possible arguments which can appear in evolved programs.

The terminal set consists of constants as well as feature values. Constants are randomly generated from a predefined range, either at initialization or when a mutation occurs. Feature values are variables in the program which take their value from the object being processed.

Functionals operate on inputs to produce outputs. Limiting the functionals to ones which input and output the same types increases interoperability but decreases expressiveness. This is known as closure: A set of functionals and terminals has closure if every function output and every terminal can be used as an input to any other function. A functional or terminal set without closure may result in programs which are invalid.

The set of possible functionals and terminals determines the expressiveness of the programs in the population. A more diverse variety of complex functionals and a larger set of potential terminals increases the expressiveness of the program, however it also increases the complexity of the program. We must include a *sufficient* set of functions and terminals, one out of which it is possible to construct as solution, but we also want the minimal suffi-



(a) Parent (b) Child

Figure 2.3: The LGP mutation operator

cient set, as this will have minimal complexity. Thus when performing GP the aim is always to include the minimum terminal/functional set which is sufficient to express the optimal solution.

2.4.1 GP Program Generation

GP relies on having an initial population of individuals on which to operate. This initial population is usually randomly generated [16], with the method of generation of an individual within the population differing based on which GP approach is being used.

In Tree-Based GP there are three standard approaches: *Grow*, *Full*, and *Ramped*. Grow builds a tree from the root down by randomly selecting terminals or non-terminals as children for nodes. Full ensures that each program tree branch has the same depth by selecting only non-terminals until a predefined depth is reached. Ramped involves performing Full starting with an initial depth and increasing this depth periodically. Ramped results in equal numbers of programs at varying depths. A common extension to Ramped is *Ramped Half-and-Half*: A compromise between Grow and Full, in which half the individuals in the population are generated using Full, and half are generated using Grow.

In LGP there is a predefined program size range (i.e. a max allowed program size and a minimum allowed program size). Random programs are a sequence of instructions where the number of instructions is a random number selected to lie within a predefined range.

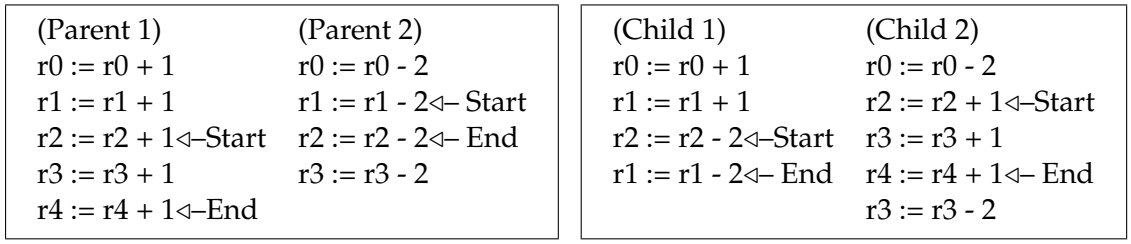
2.5 GP operators

The genetic operators reproduction, mutation and crossover described in section 2.2 take on new meaning when applied to the GP method.

- Reproduction: In Reproduction, some number of the best individuals in the current generation are copied whole to the next generation. Reproduction ensures the fitness of the population does not decrease.

- Mutation: In Mutation, an individual is copied to the mating pool, following which a random part of the individual's code is replaced with randomly generated code. In Tree-Based GP this involves replacing a subtree with a new randomly generated one. In LGP a subsequence of instructions is replaced with a randomly generated one. Constraints are placed on both the size of the subcomponent being replaced and the size of the component which is replacing it. Mutation ensures the population always contains a variety of individuals, this is known as maintaining genetic diversity.

- Crossover: In Crossover, two individuals are copied to the mating pool. Following this a subcomponent from each of them is selected and the two subcomponents are switched. In Tree-Based GP this means exchanging two subtrees, in LGP this means exchanging



(a) Parents (b) Children

Figure 2.4: The LGP crossover operator

two sequences of instructions. Crossover allows mixing of genetic material. An example of crossover in LGP is shown in Figure 2.4 and the code to be exchanged is delimited.

2.6 Selection in GP

An important facet of GP is the method used to select the individuals which form the basis of the next generation. Before we can perform crossover or mutation to produce the next generation of individuals, we must first select a number of individuals from the current generation on which to perform these operations. Fitter individuals should contribute more than poorer individuals to later generations, hence fitter individuals should have a higher likelihood of being selected.

There are many existing selection methods, here we describe three of the most common: Proportional, Rank, and Tournament Selection [18].

- Proportional Selection: In proportional selection a given individual in the population has a likelihood of being selected directly proportional to its fitness. This can be visualized as spinning a roulette wheel containing a segment for each individual. The size of an individual's segment on the wheel is directly proportional to its fitness.

- Rank Selection: In rank selection the rank of each individual within the population is calculated by ordering the individuals according to fitness. The likelihood of a given individual being selected is based on a function of the rank of that individual.

- Tournament Selection: In tournament selection we choose a set number of individuals from the population and hold a tournament between them. The individual with the highest fitness wins the tournament automatically and is selected.

2.7 GP for Classification and Related Work

GP has been particularly successful in the problem domain of classification, regularly outperforming other forms of machine learning [17, 23, 28, 1]. This is particularly true of Tree-Based GP in binary classification problems. TGP programs typically output a single floating point number, and this has a natural class interpretation; if we choose a *class boundary* such as 0, then any number smaller then the class boundary is of class 1, and any number larger than the class boundary is of class 2. This *classification strategy* breaks down with Multiclass classification problems and other solutions are required [26]. Three techniques of varying effectiveness [27] are described below.

- **Program Classification Map (PCM)** is the natural extension of the above approach. If there are K classes then the natural number line is divided up into K regions, each of which corresponds to a class. PCM is easy to implement but often has poor performance.

- **Slotted Dynamic Range Selection** is similar to PCM, but with a much larger number of regions. Thus each class has many slots assigned to it. This technique has been shown to be more effective then PCM on many problem types.

- **Probabilistic Multiclass (PM)** [32] is a classification strategy where the outputs of a program are used to form K normal distributions where K is the number of classes. Test instances are classified to the highest probability density function at the value of the classifier's output. This technique has been shown to be among the most effective classification strategies for solving multiclass classification problems.

LGP performs well on binary classification tasks [3]. LGP is also a natural fit for Multiclass Classification; LGP programs can be seen as multi-rooted directed graphs, where each root can be seen as an output. We can therefore construct programs which have K outputs, where K is the number of object classes present. The class predicted will be the one corresponding to the highest output. Research in this area has already provided promising results [3, 8]. However all that said, the performance of LGP is still sub-optimal on many problems.

2.8 Limitations

GP has proved to be both effective and flexible for solving various tasks, particularly many classification tasks. Despite this, there are still several limitations in GP which could potentiall benefit from further research. Tree-Based GP works well for many problem types, and has been researched extensively, however it often performs poorly on some problem types such as multiclass classification problems. Research has shown that other GP representations such as LGP have superior performance on many of the problems Tree-Based GP struggles with [3, 8], however the research on these representations has been limited thus far.

Little to no development has been performed on the mutation operator. The mutation operator is still treated as a black box, used solely for the purposes of maintaining diversity in the population. Because mutation occurs at random, mutation will often result in a decrease in the performance of the mutated individual. Because the number of good programs is much lower then the number of poor programs, as program fitness increases the likelihood of a poor mutation occurring increases. This is an undesirable result, and it would be better if the likelihood of a favorable mutation occurring could be increased.

Crossover in GP has been shown to be disruptive to fit programs, particularly as the number of generations becomes large. This means that crossover is unlikely to combine existing good sections of program code into new fit programs, but instead is likely to break up these good code sections. This is highly likely to have a negative impact on both the training time and the final classification accuracy of GP, and hence on the performance of GP as a whole.

Work on the crossover operator has focused on new, intelligent forms of crossover which improve the performance of GP by increasing the likelihood that good offspring will result from the crossover process. However these new "intelligent" crossover operators often increase the computation cost of crossover by a large amount. This is particularly troubling as GP is already a computationally costly technique.

When two individuals are selected from the population for crossover, they are selected based on their merits as individual programs. The issue is that two individuals performing well individually does not mean that performing crossover on them will result in good offspring. However at present we have no way of determining whether or not two individuals will produce good offspring.

2.9 Summary

This chapter presents a survey of current work in the field of Machine Learning, focusing specifically on Genetic Programming. GP has emerged as an effective and flexible technique for a variety of tasks, in particular for classification problems. However GP still has several limitations; these include difficulty coping with multiclass classification problems and and tendency towards code bloat. Both of these areas are the subject of current research.

One approach to GP which has shown promise for Multiclass Classification is LGP, a variant of GP where programs are represented as sequences of instructions from an imperative programming language. However LGP in general and specifically LGP for multiclass classification are poorly explored areas of research.

This research project will investigate Linear GP for the purposes of multiclass classification. In particular it will focus on developing new operators for LGP which improve the classification accuracy of LGP on multiclass classification problems.

Chapter 3

Data Sets and Experimental Setup

In order to empirically compare the effectiveness of the GP methods described in this report as techniques for performing multiclass classification, we conducted a series of experiments. Here we describe the data sets and parameters used during the course of these experiments.

3.1 Data Sets

In the experiments, three image datasets containing classification problems of increasing difficulty were used. These obtained these data sets from the UCI machine learning repository [29]. All of the datasets consist of multiclass classification problems from the computer vision problem domain. Object detection, feature extraction, and class labeling have all performed by other parties prior to this project.

Multiclass classification problem difficulty is typically determined by three key factors: the number of classes, the variation in features, and feature overlap[13]. Problems with a larger number of classes are typically more difficult to solve, particularly for GP classifiers. Problems with more feature variation within the instances of a single class are also typically more difficult to solve. Finally, problems where two instances from different classes have similar features are also typically difficult to solve.

The problems used in the experiments are all intentionally difficult to solve, in order to make performance comparison between various methods easier.

3.1.1 Artificial Characters

This data set consists of 5000 artificially generated characters from the English alphabet. The data set was generated using a first order theory which describes the structure of 10 capital letters from the English alphabet and a random choice theorem prover which accounts for variation in the instances. This is a class balanced problem, so each class has an equal number of instances in the data set. Some examples are shown in Fig. 3.1.

Figure 3.1: Artificial Characters Instance Examples

The instances in this data set consist of vectors each with two coordinates and two higher level features where the high level features are:

1. The length the vector.
2. The length of the diagonal of the smallest rectangle which includes the picture of the character. This will be the same for every vector component of a given character.

Each object instances consists of 8 vectors each of 6 floats. The first 4 floats are the start and end points of the vector, the third float is the vector length, and the final float is the size of the character. This gives us a grand total of 48 features for each object instance.

This problem has both a large number of features (48) and a large number of classes (10), however the feature spaces for different classes are relatively well separated. Hence we expect this problem to be the easiest to solve of all the problems used.

3.1.2 Image Segmentation

This data set consists of 2100 3x3 pixel regions extracted at random from seven large images of outdoor areas. These images were hand segmented to create a classification for every pixel. Example "Segmentation" images are shown in figure 3.2 An instance is labeled with the type of segment its 3x3 region is located. The 7 possible classes are: brickface, sky, foliage, cement, window, path, or grass. This is a balanced class problem, so each class has an equal number of instances in the data set. From each instance 19 features have been extracted to form the feature vector used in classification. These features are a mixture of low level pixel features such the position of the 3x3 within the image, and high level features such as the number of edges in a 3x3 segment.

Figure 3.2: Image Segmentation Instance Examples

This problem has fewer classes (7) and fewer features (19) then the other two problems, but has a larger amount of variation in its features. Hence we expect this problem to be of medium difficulty to solve.

3.1.3 Handwritten Digits

This data set consists of 3750 examples of handwritten digits. NIST preprocessing programs were used to extract normalized bitmaps from the handwriting of 43 people on preprinted forms. Each character instance consists of an 8x8 bitmap which is a grey scale quantized image of the original handwritten digit. This is a balanced class problem so there are an equal number of instances of each class in the data set. Some examples are shown in Fig. 3.3.

Figure 3.3: Handwritten Digit Instance Examples

The feature vectors for instances in this data set consists solely of the bitmap itself. In other words the only features available for use in classifying the object are the pixels themselves. This means that this problem has a large number of features (64), a large number of

classes (10) and the features are likely to vary widely and have a great deal of overlap with those of other classes. Hence we expect this problem to the be the hardest of all the problems used.

3.2 Experimental Setup

In this report we perform a series of experiments using the data sets described above. These experiments aim to empirically confirm or disprove the hypotheses discussed throughout this report. The setup and parameters used in these experiments are described in this section.

3.2.1 TGP/LGP Parameters

We attempted to keep the settings as similar as possible between methods for comparison purposes. Parameters common to all methods are identical, and parameters of the same kind are equivalent. The parameter values used in the experiments are shown in table 3.1. These values were determined based on common settings and empirical search via initial experiments.

Table 3.1: parameter configurations

Parameter	LGP	TGP
Population	500	500
Max Gens	200	200
Mutation	60%	60%
Elitism	10%	10%
Crossover	30%	30%
Max Size	32	8
Tournament Size	4	4
Runs	30	30

LGP and TGP have very different program representations, and hence very different measures of maximum program "size". TGP measures maximum program size as the maximum permissible tree depth. LGP on the other hand measures maximum program size as the maximum permissible number of instructions. We need to choose a maximum size for each type of GP so that the average size of individuals in the population will be the same for both techniques. Note that in both cases the average size can be approximated as the maximum size as individuals tend towards the maximum permissible size as evolution proceeds.

In TGP programs are represented as trees, so there is no direct reuse of code. If two parts of the operator tree have identical structure that code must be repeated twice. On the other hand though, all code in a TGP program is structurally effective, insofar as it contributes to the final result. So in TGP if the maximum tree depth is n, then a tree of maximum depth has up to $2^n - 1$ nodes. Up to $2^{n-1} - 1$ of these nodes will be operator nodes, the remaining 2^{n-1} nodes will be terminal nodes, either features or constants.

On the other hand in LGP programs, code may be reused. Once values have been calculated and stored in registers, many future instructions may use the stored value to calculate future results. However in LGP programs it is also the case that some code may not contribute to the final result, i.e. there exist structural introns in the program code. The combination of these factors make it very difficult to precisely calculate the effective number

of nodes in a LGP program, so we are forced to make some assumptions. Assume $1/4$ of the instructions in an LGP program are introns, and that the average number of nodes used to calculate the instruction held in a register is $n/4$, where n is the number of instructions. Finally, assume that half of all instruction arguments are registers. Note that these values were determined using based on initial testing. Then the size of an LGP program with n instructions can be calculated as follows:

- Avg. number of register arguments per instruction = 1.
- Avg. number of nodes per instruction = $n/8 + 1 + 1$. I.e. one register, one operator, and one terminal.
- Avg. number of effective instructions = $3n/4$.
- Therefore avg. number of nodes = $(3n/4) * (n/4 + 1 + 1) = 3n^2/16 + 6n/4 = (3n^2 + 24n)/16$.

So a TGP program of depth 8 has $2^8 - 1 = 255$ nodes, and a LGP of size 32 has $(3 * 32^2 + 24 * 32)/16 = 240$ nodes. Hence a TGP program of depth 8 is approximately equivalent to a LGP program of length 32. Note that these are also the largest program sizes we could realistically use, as anything larger would take too long to run on the accessible machines to be plausible for a project of this duration.

3.2.2 Experimental Configurations

Each data set was divided into a training set, a validation set, and a test set, with all sets being of equal size. After the maximum number of generations was reached, the test accuracy of the program which performed best on the validation set was recorded. This process was repeated for at least 150 runs, each time with a random seed. The final statistics for each run were recorded, then averaged over all runs to provide the final statistics displayed.

3.3 Significance Testing

In order to show that our results are meaningful, we are required to demonstrate their statistical significance. In other words if we develop a new operator which demonstrates improved performance on our test sets, we need to show that this performance improvement is unlikely to have occurred by chance. To this end we use tests of statistical significance to determine whether or not our results constitute a significant improvement. The test used in this project is a 2-tailed students t-test with a significance criterion of 0.95. The significance criterion is an arbitrary measure of how different our results must be before we consider them significant. A significance criterion of 0.95 means we consider our results significant if the probability that they occurred by chance is less then 0.05 or 5%. This is the conventionally used significance criterion, and hence the one used in this project.

Chapter 4

Basic LGP Approach

4.1 Introduction

In this project we seek to improve on the basic LGP approach by developing new LGP operators that improve the performance of LGP on multiclass object classification problems. Once we have developed these operators we will need to confirm that in actual fact they do constitute an improvement over the basic LGP approach. In order to do this we need to determine the performance of conventional LGP on the problems in our data sets. Once we have these results we can use them as a baseline for comparison with later results.

In addition we wish to show that seeking to improve the conventional form of LGP is a worthwhile endeavor. When given two alternative but similar techniques which could potentially be used to solve a problem, we would clearly choose the better one. So in order to justify improving the LGP technique in terms of its performance multiclass object classification problems, we need to demonstrate it is indeed the GP technique of choice for solving such problems. Specifically we need to show that even without any improvements, the conventional form of LGP outperforms TGP as a method for solving multiclass object classification problems. Since LGP and TGP have comparable computation time [8] we use classification accuracy to measure performance.

4.1.1 Chapter Goals

Hence in this chapter we aim to demonstrate the superiority of LGP over TGP as methods for solving multiclass object classification problems. In the process we will establish a set of results which can be used as a baseline for empirically testing future developments. Specifically this chapter has the following research goals:

- To demonstrate the superiority of LGP over TGP as methods for solving multiclass object classification problems.
- To establish a set of baseline results for the performance of LGP on the multiclass object classification problems given in chapter 3.

4.2 TGP for object classification

TGP programs by definition have a single output, typically a single floating point number. In order to use TGP for classification we are required to use a function which maps the program output value to a number of possible classes.

If we are performing binary classification, a natural mapping function exists. We can simply choose a boundary value and classify objects based on which side of boundary value their outputs lie on. However when performing classification with multiple n classes, we must choose at least $n-1$ boundary values so as to have n regions to map to the classes.

The issue arises because there is no way of knowing if the output values for one class will lie close to those of another class, and hence whether their class regions on the real number line should lie close together. If two classes have similar features, their output values are likely to be similar for any program. If their output values are similar but their class regions are widely spaced, any evolved program will misclassify many instances of these two classes. Hence a poor mapping function will result in programs with poor classification accuracy, regardless of other factors. Investigating new mapping functions is an area of active research [30], and there have been several recent improvements, however the results still leave much to be desired.

The upshot of this is that it would be preferable to use a GP method which does not rely on a complicated mapping function, and instead has a natural interpretation of program outputs as classes.

4.3 Linear GP for Classification

In LGP a program has multiple outputs: the set of final register values. When using LGP for classification we associate each class with a distinct register. If we have n classes, a program classifies an instance as the class associated with the register with the largest final value. Figure 4.1 shows an example of this kind. Clearly to use LGP for classification we must have at least as many available registers as there are classes.

LGP is proposed as a natural alternative to TGP for multiclass classification problems because LGP allows arbitrarily many outputs. This allows us to sidestep the mapping problem encountered in TGP entirely by removing the dependence on a mapping function. Moreover, LGP has the power of many tree based expressions executed in parallel, and with common code reused to form a DAG. There are additional benefits to the LGP output structure: we can have a probabilistic classifier, a reject option, or a more focused fitness function, however these are not the focus of this paper.

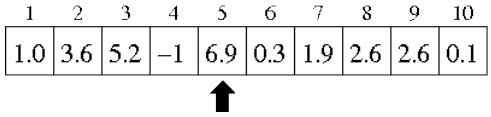


Figure 4.1: *Final register values:* Object would be classified as Class 5

4.4 TGP Classification Strategies

One of our goals for this section is to compare the performance of LGP to that of TGP. However there are many forms of TGP insofar as there are many different classification strategies we could use. Since we do not have time to compare to all of them, we have chosen to compare LGP to two TGP strategies, one at either end of the performance spectrum.

The first classification strategy we will use is TGP with the most naive of classification strategies, program classification map (PCM). Under a PCM classification strategy the real number line is divided up into as many sections as their are classes, and each class is associated with a unique section. This is the simplest of mapping functions to implement

and one of the most commonly used, as it works well with binary classification problems. However it has been previously demonstrated to perform poorly on multiclass classification problems, particularly those with a large number of classes [8]. Hence in order for LGP to be considered an effective method for solving multiclass object classification problems it is clearly required to outperform basic TGP.

The second comparison we will perform is between the performance of LGP and the performance of TGP with an advanced classification strategy. If the performance of basic LGP is comparable to the performance of TGP with an advanced classification strategy, then we can justifiably claim that LGP is the superior method for multiclass object classification. The advanced classification strategy that we have chosen is probabilistic multiclass (PM). This method has been shown to have significantly improved performance over that of PCM on multiclass object classification problems [27]. Hence if we can outperform PM using LGP, we can claim to have achieved state of the art performance.

4.5 Results and Analysis

In this section we report on the results of a series of experiments designed to compare the performance of TGP and LGP as methods for performing multiclass object classification. The first set of experiments compare LGP to TGP with PCM, while the second set of experiments compare LGP to TGP with PM.

4.5.1 Program Classification Map TGP vs. Basic LGP

The results in table 4.1 were obtained by running TGP using PCM, and LGP on the three problems. The first line shows that for the artificial characters problem, TGP with PCM achieved an average accuracy of 55.91% with a s.d. of 8.79% on the *test set* over 150 runs. It also shows that the LGP approach achieved an average accuracy of 82.02% with a s.d. of 5.72%.

Table 4.1: *Classification Accuracy:* Tree-based GP with Program Classification Map vs. Linear GP

Data Set	TGP with PCM		LGP	
	Mean	S.D.	Mean	S.D.
Artificial Characters	55.91%	8.79%	82.02%	5.72%
Segmentation	68.69%	7.51%	75.46%	2.81%
Digit Recognition	45.20%	8.34%	65.46%	3.64%

The results in table 4.2 were obtained by performing a two tailed students t-test based on the results summarized in table 4.1. The first line describes the results of the t-test on the artificial characters problem. It shows that the improvement in classification accuracy that we have obtained on this problem has only a 0.001 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95.

These results show that LGP outperforms TGP using PCM by a significant amount on all data sets. In fact, LGP outperforms TGP by a significant and *very large* amount. This clearly demonstrates the superiority of LGP over TGP as methods for solving multiclass object classification problems.

Table 4.2: *Significance of Results:* TGP with PCM vs. LGP

Data Set	2-Tailed P value	Significant
Artificial Characters	0.0001	Yes
Segmentation	0.0001	Yes
Digit Recognition	0.0001	Yes

4.5.2 Probabilistic Multiclass TGP vs. Basic LGP

The results in table 4.3 were obtained by running TGP using PM, and LGP on the three problems. The first line shows that for the artificial characters problem, TGP with PCM achieved an average accuracy of 81.83% with a s.d. of 5.19% on the *test set* over 150 runs. It also shows that the LGP approach achieved an average accuracy of 82.02% with a s.d. of 5.72%.

Table 4.3: *Classification Accuracy:* Tree-based GP with Probabilistic Multiclass vs. Linear GP

Data Set	TGP with PM		LGP	
	Mean	S.D.	Mean	S.D.
Artificial Characters	81.83%	5.19%	82.02%	5.72%
Segmentation	85.99%	8.23%	75.46%	2.81%
Digit Recognition	50.66%	8.38%	65.46%	3.64%

The results in table 4.4 were obtained by performing a two tailed students t-test based on the results summarized in table 4.3. The first line describes the results of the t-test on the artificial characters problem and it also shows that this does not constitute a significant improvement according to our significance criterion of 0.95.

Table 4.4: *Significance of Results:* class graph crossover vs. selective crossover

Data Set	2-Tailed P value	Significant
Artificial Characters	0.7634	No
Segmentation	0.0001	Yes
Digit Recognition	0.0001	Yes

These results show that LGP outperforms TGP with PM on one problem, TGP with PM outperforms LGP on one problem, and the two methods have similar performance for the third problem. So these two methods overall achieve similar results on our problems, and hence this result suggests that these two methods have similar performance.

4.6 Chapter Summary and Future Work

In this chapter we have demonstrated that LGP has significantly superior performance to TGP using a simple classification strategy, and similar performance to TGP using a state-of-the-art classification strategy. This tells us that the basic LGP approach is at least as good as one of the best TGP approaches we currently know. Hence LGP is the superior method for solving multiclass object classification problems. In addition we have established a set baseline results for the purposes of comparison with later work. We can use this baseline to empirically determine whether or not the techniques developed in later chapters significantly improve program performance.

Chapter 5

Selective Mutation

5.1 Introduction and Motivation

Conventionally, mutation has been seen as the less important operator in GP. It is often treated as simply a tool for maintaining diversity in the population, and preventing a homologous population. It is usually expected that the major improvements in program fitness will come from the crossover operator. However, it has been shown that crossover is often a destructive force, and that for many problems, mutation outperforms crossover as a method for improving program fitness [19].

As far as we have been able to determine, much work has been done on new crossover operators but relatively little on new mutation operators. Many new crossover operators attack the long held belief that crossover is a black box. For instance, intelligent crossover attempts to keep building blocks intact. However this kind of intelligent approach has yet to be applied to the mutation operator. Diversity requires randomness, so it is still widely held that mutation should be a completely random process.

We believe this assumption is flawed, and that it is possible to direct the mutation process in order to improve the performance of the GP algorithm while still maintaining sufficient population diversity.

5.1.1 Background and Motivation

In any given program in the population some instructions will be good, and others will be bad. To be precise, instructions will vary in correctness, ranging from perfect code, though to totally useless.

The conventional mutation operator does not take instruction correctness into account, and chooses an instruction to mutate at random. This means that sometimes instructions which are currently vital to the success of a program are selected for mutation. This sort of mutation is unlikely to result in an improved individual, hence unlikely to directly benefit the population as a whole except for maintaining diversity.

The reason that selecting an instruction to mutate at random is a poor approach stems from two facts.

Firstly, at the instruction level, as instruction correctness increases mutation becomes an increasingly disruptive force. The number of possible correct instructions is very small compared to the number of possible instructions, and as instruction correctness increases, the number of superior instructions becomes smaller still. Hence the mutation of a very correct instruction is overwhelmingly likely to result in a decrease in instruction correctness. Conversely, mutating an instruction with poor correctness is much more likely to result in a highly correct instruction and hence a higher program fitness. Hence mutating instructions

with poor correctness generally produces better results then mutating instructions with high correctness.

Secondly, at the program level, as program fitness improves, mutation becomes an increasingly destructive force. At the start of evolution, the number of highly correct instructions in a given program will be minimal at best, so most mutations will occur in instructions which have low correctness. However, fitter programs are almost certain to consist primarily of instructions with high correctness. This means the likelihood of a disruptive mutation taking place becomes very large. So as program fitness increases, the likelihood that mutation will take place in a highly correct instruction also increases. As we just discussed mutation of a highly correct instruction is usually a disruptive force. Hence as program fitness improves, the disruptive influence of mutation increases.

Mutation of instructions which have high correctness is a disruptive force, so mutating randomly becomes increasingly disruptive as program fitness increases. Hence selecting which instruction to mutate based on instruction correctness could result in improved performance for the GP algorithm. By focusing on mutating those instructions which are causing errors, and leaving intact those instructions which are performing correctly, fit programs are more likely to be evolved.

5.1.2 Chapter Goals

This chapter aims to investigate the hypothesis: A new GP mutation operator based on the principle of selecting instructions to mutate based on their correctness outperforms the conventional mutation operator on a sequence of multiclass classification problems.

Specifically, it aims to answer the following research questions:

- How can a model of correctness for the instructions of a program be developed?
- How can we use this model to develop a selective mutation operator?
- Does this selective mutation operator outperform the conventional mutation operator on a sequence of multiclass classification problems?

5.2 A Model for Instruction Correctness

We desire some model of the correctness of the instructions in a program. For example, given an arbitrary program and an arbitrary instruction in the program, we need a method which lets us determine the degree of positive influence that instruction has on the program output. It is impractical to attempt to find a precise measure of instruction correctness, however it is possible to find an effective approximation.

The only information we have to work with is the program's performance on the training data set. Specifically, for any given program, we can determine how many training instances of each class were misclassified by the program.

Clearly if a program misclassifies a large number of training instances from class i then one possibility is that the instructions which are responsible for the class i output are flawed. We know that an LGP program which is a potential solution to an n-class classification problem can be represented as n overlapping operator graphs, each with a single root. Define the *nth class graph* of a program to be the operator graph whose root is the nth register. An example of an LGP program with the class graphs highlighted can be found in figure 5.1. Thus we blame all of the instruction in the i'th class graph equally if a program performs poorly on class i training instances. For simplicities sake we blame them all equally. So each

instruction in the i'th class graph can be associated with the number of incorrectly classified training instances of class i.

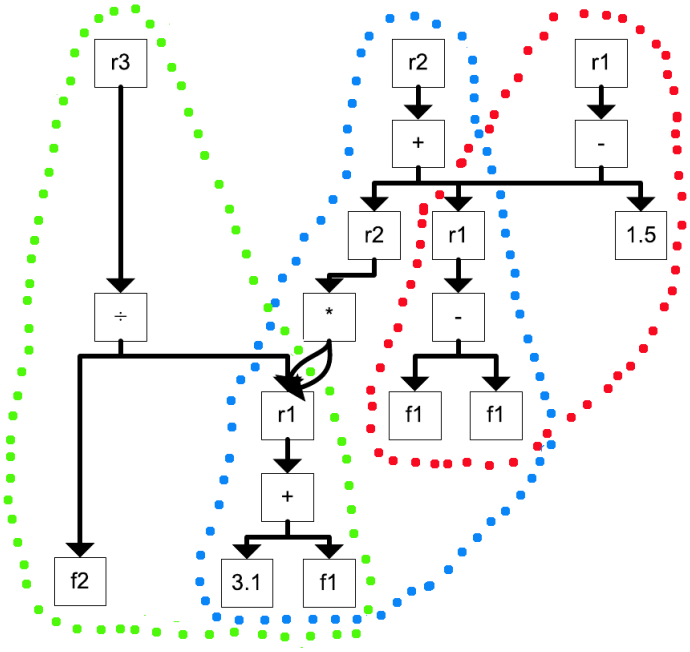


Figure 5.1: The 3 overlapping class graphs of a LGP program with 3 output registers, r1, r2, and r3

This all works well for class graphs which are distinct, i.e. have no shared program code. However, LGP programs by definition are comprised of n overlapping class graphs, so we need to deal with the case where an instruction is part of two or more class graphs. We refer to the method used to deal with overlapping class graphs as the *propagation strategy*. Several propagation strategies are open to us, including taking the minimum, average, or maximum value of all class graphs sharing the instruction. The option we choose is to take the minimum both for theoretical reasons and because it has been shown to work well in practice. If we modify an instruction, then every class graph which uses the instruction will be affected. Hence by taking the minimum instructions will only have high mutation probability if they do not affect good outputs.

So to summarize, we count the number of training instances for each class which are incorrectly classified and call this number the class value. We then assign to all instructions in each class graph the corresponding class count. If an instruction is part of two or more class graphs, we assign that instruction the minimum class count of all the trees it is part of. This gives us a model of instruction correctness for a given LGP program.

5.3 A New Selective Mutation Operator

Once we can generate a model of instruction correctness for any LGP program, a selective mutation operator is readily implemented. Instructions in a LGP program are of two possible types, they are either part of one or more class graphs, or they are not, in which case they are introns. The latter follows because instructions which are not part of any class graph have no effect on the program output, and so are introns by definition. By generating a

model of instruction correctness for this program we generate a correctness value for each non-intron instruction. The intron code has no effect on program output, so we have no way of determining its correctness. Therefore we assign to all intron instructions the average instruction correctness. This has both a theoretical basis and is the result of intial testing. We want good instructions to have low mutation likelihood and bad instructions to have high mutation likelihood. The intron instructions are neither good or bad, but are instead neutral. Mutating them may result in a good instruction or a bad instruction, and we certainly want them to have some probability of being mutated, therefore assigning an average correctness value makes sense.

So now every instruction in the program has a number assigned, indicating its correctness. By normalizing these correctness values we arrive at a probability distribution over all program instructions! Sampling from this distribution results in instructions being selected with probability directly proportional to the number of misclassifications the instruction is responsible for. *Under this selective mutation scheme, the likelihood of an instruction being selected for mutation is inversely proportional to its positive influence on the program output*.

5.3.1 Algorithm

The selective mutation operator involves 3 key steps in addition to the steps present in conventional mutation:

- Determine the number of misclassified training examples for each class, known as the class values.
- Determine the class graphs and assign correctness values.
- Normalize this distribution.

Once these steps have been completed, to perform selective mutation we need only sample from the normalized distribution over instructions, and perform a normal mutation on the selected instruction.

Determining class values

This is done easily and efficiently when program fitness is calculated. Let p be a program, to calculate the class values we use the following algorithm:

```
array classValues;

for each training set instance t
  if t is misclassified by p
    add 1 to classvalues[correctClass]
```

Assign correctness Values

Because we store programs as a linearly ordered sequence of instructions, we do not have direct access to the class graphs. According to the theory we established earlier, in order to assign responsibilities we need to calculate the class graph for each class. We know only that the root node for the class graph must be the last instruction in the program which assigns a value to the i'th register. Therefore by iterating through the program instructions in reverse order we can find the "first" instruction easily. If this instruction takes any registers as arguments, then the "next" instruction (under reverse order) which assigns any such register a value is also part of the class graph. Doing this repeatedly will result in all of the

instructions which are part of the class graph being identified. Hence the class graph for an instruction i can be determined in a single backwards pass through the instructions. The important thing to note is that if instruction i has destination register d, and instruction $i + n$ takes register d as an argument, then i influences the execution of $i + n$. From an alternative perspective, i is a child of $i + n$ in some class graph.

Let p be a program, let c be a class label.

```
List registers;
List classTree;

for each instruction ins in reverse order
  if the destination register is in registers
    add ins to the class tree;
    remove ins from registers;
    if argument 1 is a register
      add argument 1 to registers;
    if argument 2 is a register
      add argument 2 to registers;
```

However we wish to do this for n many class graphs and we want to assign each instruction a responsibility at the same time. Rather then being required to iterate backwards through the instructions n times, we can find all n class graphs simultaneously in a single backwards pass through the instructions. Because we are following a minimum propagation strategy and assigning each instruction the minimum responsibility possible, the process is also somewhat simplified:

Let n be the number of classes, let r be the number of registers, let x be the number of instructions in the program and let avg be the average class value.

```
 array registers;
for each class c
  registers[c] = the class value of class c;
for each register r not associated with a class
  registers[r] = -1;
for each instruction ins in reverse order
  if ins is an intron
    give ins the average class value;
  else
    let d = the destination register of ins;
    let v = the value held in registers[d];
    give ins the value v;
    set registers[d] = a big number;
    for each argument which is a register
      let i be the register number;
      if registers[i] is empty
        set registers[i] = v;
      else if v < registers[i]
        set registers[i] = v;
```

5.3.2 Further Discussion

In the resulting distribution, the probability of an instruction being selected for mutation is directly proportional to how beneficial the instruction is to the program. The basic theory is that selective mutation builds on the existing program by improving those parts that are performing worst. This can be thought of as a form of hill climbing, although it should be noted that the process is stochastic: The worst instruction will not always be selected for

mutation, it is simply *more likely* to be selected. An example is shown in figure 5.2. Part (a) displays the number of misclassifications for each of the 3 classes in the problem, part (b) displays the program after each instruction has been assigned a responsibility, and part (c) displays the mutation probabilities after normalization.

Class	1	2	3
Counts	1	10	4

(a) Misclassification counts

Destination		Value	Responsibility
r[1]	=	3.1 + f1;	4
r[3]	=	f2 / r[1];	4
r[2]	=	r[1] * r[1];	10
r[1]	=	f1 - f1;	1
r[1]	=	r[1] - 1.5;	1
r[2]	=	r[2] + r[1];	10

(b) Responsibilities

Destination		Value	Probability
r[1]	=	3.1 + f1;	13.3%
r[3]	=	f2 / r[1];	13.3%
r[2]	=	r[1] * r[1];	33.3%
r[1]	=	f1 - f1;	3.3%
r[1]	=	r[1] - 1.5;	3.3%
r[2]	=	r[2] + r[1];	33.3%

(c) Probabilities

Figure 5.2: Mutation probabilities after normalization

5.4 Results and Analysis

We attempted to solve the three problems described in chapter 3 first using GP with normal mutation, then secondly with GP using 50% selective mutation and 10% normal mutation. Initial results indicated these values to give better results then 60% selective mutation, we believe this is because the 10% normal mutation helps maintain diversity. All other parameters remained constant between the two experiments.

Table 5.1: Parameter Configurations

Parameter	LGP	LGP + Selective Mutation
Normal Mutation	60%	10%
Selective Mutation	0%	50%

The results in table 5.2 were obtained by running LGP using conventional mutation and LGP using selective mutation on the three problems described in chapter 3. The first line shows that for the artificial characters problem, LGP using conventional mutation achieved an average accuracy of 82.02% on the *test set* over 150 runs with a standard deviation (s.d.) of 5.72%. It also shows that the LGP with selective mutation approach achieved an average accuracy of 85.86% with a s.d. of 4.92%.

The results in table 5.3 were obtained by performing a 2 tailed students t-test based on the results summarized in table 5.2. The first line describes the results of the t-test on the artificial characters problem, It also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 5.2: Classification Accuracy

Data Set	Conventional Mutation		Selective Mutation	
	Mean	S.D.	Mean	S.D.
Artificial Characters	82.02%	5.72%	85.86%	4.92%
Segmentation	75.46%	2.81%	77.91%	4.19%
Digit Recognition	65.46%	3.64%	67.32%	4.29%

Table 5.3: Significance of Results using students t-test

Data Set	2-Tailed P value	Is Result Significant
Artificial Characters	0.0001	Yes
Segmentation	0.0001	Yes
Digit Recognition	0.0001	Yes

The improvement gained by using the selective mutation technique is cumulative with the improvement already achieved by normal LGP over TGP. Hence this confirms our hypothesis that selective mutation as described above is a more effective technique for solving multiclass classification problems then normal mutation.

5.5 Chapter Summary and Future Work

We conclude that selective mutation is an effective technique for improving LGP performance on multiclass classification problems. It is a simple, yet effective technique that can be easily implemented, and results in a negligible increase in the computational complexity of mutation. While many people have investigated and developed new crossover operators, mutation has been left largely untouched. Here we have demonstrated that it is also possible to improve the performance of GP through modifications to the mutation operator. This is a particularly important development because we expect that we can combine the selective mutation operator with an improved crossover operator to increase performance above what could be obtained by using either by itself. We also hope that this will inspire others to shift the focus from improving only the crossover operator, to improving both the crossover and mutation operators.

Two outstanding questions remain unanswered about the selective mutation technique:

- For classification problems, does the arity of the problem (number of classes) affect the performance of the selective mutation operator.
- Is the selective mutation operator limited to the classification problem domain, or is it a generally applicable technique for improving LGP performance on arbitrary problem types.

We are particularly interested in the answer to the second question. The selective mutation operator has at this time only been tested on multiclass classification problems. However the theory behind the technique is applicable to GP in general, and as such selective mutation may potentially improve LGP as a technique for solving arbitrary problem types. This would render selective mutation an important and fundamental technique in LGP. Work subsequent to this project would likely focus on answering the second of these questions.

Chapter 6

Class Graph Crossover

6.1 Introduction

This chapter has two primary motivations.

On the one hand, we are motivated by a desire to address the problem of building block disruption in LGP. Arguments about the existence of building blocks in GP and the questionable performance of the crossover operator have raged back and forwards for many years in academic circles. Building blocks are small, fit pieces of program code, which implicitly may be combined to create larger, fit programs. Building block theory states that the GP process results in building blocks being highly likely to appear in programs in the population. I.e. good bits of code will appear more often then bad bits of code. If building blocks do exist in the population, then crossover allows these smaller pieces of program code to be combined into the larger fit programs. This potential for building block recombination is one of the primary reasons GP is lauded as a search technique by supporters. No other search technique has a similar ability.

While it is believed that GP does in fact obey the building block hypothesis, it is not clear that crossover has a positive effect on building blocks. We desire crossover to keep existing building blocks intact, and combine them into new, fit programs. It is entirely possible, however, that crossover has a purely destructive influence. In other words crossover is breaking up existing building blocks and hence retarding the formation of fit programs. Hence we desire to remove or at least reduce the number of building blocks disrupted during crossover.

On the other hand we wish to develop a crossover operator which operates at the class graph level instead of the instruction level. This goal is motivated by a desire to take the concepts developed in chapter 5 and apply them to the process of crossover in order to develop a selective crossover operator. These techniques deal with LGP programs as sets of class graphs instead of sequences of instructions. Therefore without a crossover operator which deals with class graphs instead of instructions, the work in chapter 5 cannot be successfully applied. Hence the work in this chapter is also a first step towards a selective crossover operator.

6.1.1 Chapter Goals

In this section we discuss the failings of conventional crossover insofar as it disrupts building blocks and focus on developing a new crossover operator which uses the class graph program structure to alleviate this problem. Specifically, this chapter aims to answer the following research questions:

- How do we decompose a LGP program into abstract components which contain entire building blocks?
- How do we use this decomposition to develop a structured crossover operator for LGP?
- Does this new crossover operator alleviate the problem of building block disruption?
- Does this crossover operator outperform the conventional crossover operator on a sequence of multiclass classification problems?
- Is this work compatible with the selective mutation operator developed in chapter 5.

6.2 Background and Rationale

A crossover is considered destructive if one or more building blocks are disrupted by the exchange of genetic material. A building block is disrupted if part of the building block is selected in the code to be exchanged, and part is not. Hence in order for crossover to be non disruptive, when choosing the code to be exchanged, building blocks must be selected in entirety or not at all. Clearly using the conventional crossover operator may result in disruptive crossovers occurring. The conventional crossover operator selects the instruction(s) to exchange at random. This means that whether or not an entire building block is selected for exchange is entirely random. If x instructions are selected at random for crossover out of a program with n instructions, then the probability that a building block of size b is disrupted can be calculated.

A building block is not disrupted if either it is entirely exchanged or none of it is exchanged.

- Number of possible code segments which could be exchanged = C_x^n.
- Number of possible exchanges containing entire building block = C_x^{n-b}. (clearly if $b > n$ there is no way the entire building block can be exchanged).
- Number of possible exchanges disjoint from building block = C_{x-b}^n.
- Therefore prob of no disruption = $(C_x^{n-b} + C_{x-b}^n)/C_x^n$.
- and prob of disruption = $1 - (C_x^{n-b} + C_{x-b}^n)/C_x^n$.

So clearly as the numerator $(C_x^{n-b} + C_{x-b}^n)$ increases, the probability of disruption decreases. Assuming $x \leq n/2$, the top term increases as b (the size of the building block) decreases. I.e. as building block size increases, the likelihood the building block is disrupted by crossover also increases. Certainly, once building blocks exceed a very minimal size, the likelihood they will be disrupted by crossover becomes overwhelming. This is an undesirable situation, as this effectively limits the size of building blocks in the population. Successfully solving a problem usually requires us to build larger building blocks from small ones, but this process will clearly be retarded by conventional crossover. Clearly we desire a crossover operator which has a less disruptive influence on larger building blocks.

There have been many new crossover operators developed over the years, each with its own unique form and benefits. Many of these crossover operators aim to improve GP performance by decreasing the destructive influence of crossover. While some success has been achieved in this area, many of these techniques require orders of magnitude more computation time then the original crossover operator. One of the strengths of the conventional

crossover operator is that once program fitness has been calculated the computational cost of crossover is linear in program size. However newer, alternative forms of crossover such as headless chicken crossover or brood crossover require calculating fitness values for multiple offspring during the crossover process [31]. This means each crossover operation takes significantly longer, a major disadvantage in a GP operator (this will be shown later). What is really required is an improved crossover operator which preserves building blocks but only requires a single fitness evaluation.

6.3 Class Graph Crossover

6.3.1 Biological Crossover

In biology, crossover is limited to exchange of similar DNA. Each physical feature has many different DNA sequences, called alleles, which code for variations of that feature. When crossover occurs, the alleles of one parent which code for a certain physical characteristic are exchanged with the alleles of another parent which code for the same features. In other words biological crossover always exchange DNA which codes for different variations of the same features. E.g. DNA for blue and brown eye color might be exchanged. If we consider the DNA building blocks as sequences within a single allele, then we can clearly see that biological crossover never destroys building blocks. Either the entire building block is swapped or it is left in the original DNA sequence.

In future, we will refer to two sequences of code, DNA or program, as *position equivalent* if they code for different versions of the same feature, i.e. they are alleles for the same feature.

6.3.2 Class Graphs

Biological crossover is in stark contrast to conventional GP crossover, where the program code to be exchanged is chosen at random. By ignoring the abstract structure of the program, we are making no distinction for building block boundaries. This means it is highly likely that we will be breaking up building blocks left, right, and center. Therefore conventional crossover is likely to be a destructive influence. If we take our inspiration from biological crossover, it makes sense to define some sort of abstract structure within our programs, and to respect this structure when choosing code to exchange.

Fortunately, in LGP for multiclass classification there is a natural structure already imposed on the program. We already discussed how a n-class classification problem can be viewed as a DAG composed of n overlapping graphs. Recall that the *nth class graph* in any program is the graph whose root is the nth register. In other words the graph associated with the class n. Therefore sequences of instructions are position equivalent if and only if they are class graphs for the same feature. So we aim to develop a new crossover operator based on the class graphs, and we call it Class Graph Crossover (CGC).

6.3.3 Algorithm

In CGC we use program class graphs as the basis for an exchange of program code. Under a CGC scheme, only positional equivilent code is exchanged. In other words, given any two programs, we select one or more classes, and exchange the corresponding class graphs in the two programs. An example can be found in Figure 6.1. When performing class graph crossover we could produce two children, but choose to only produce one. Basically we build on class graph crossover to create selective crossover, and selective crossover requires

we only produce one child. The reasons for this are explained in chapter 7 when we discuss one child crossover.

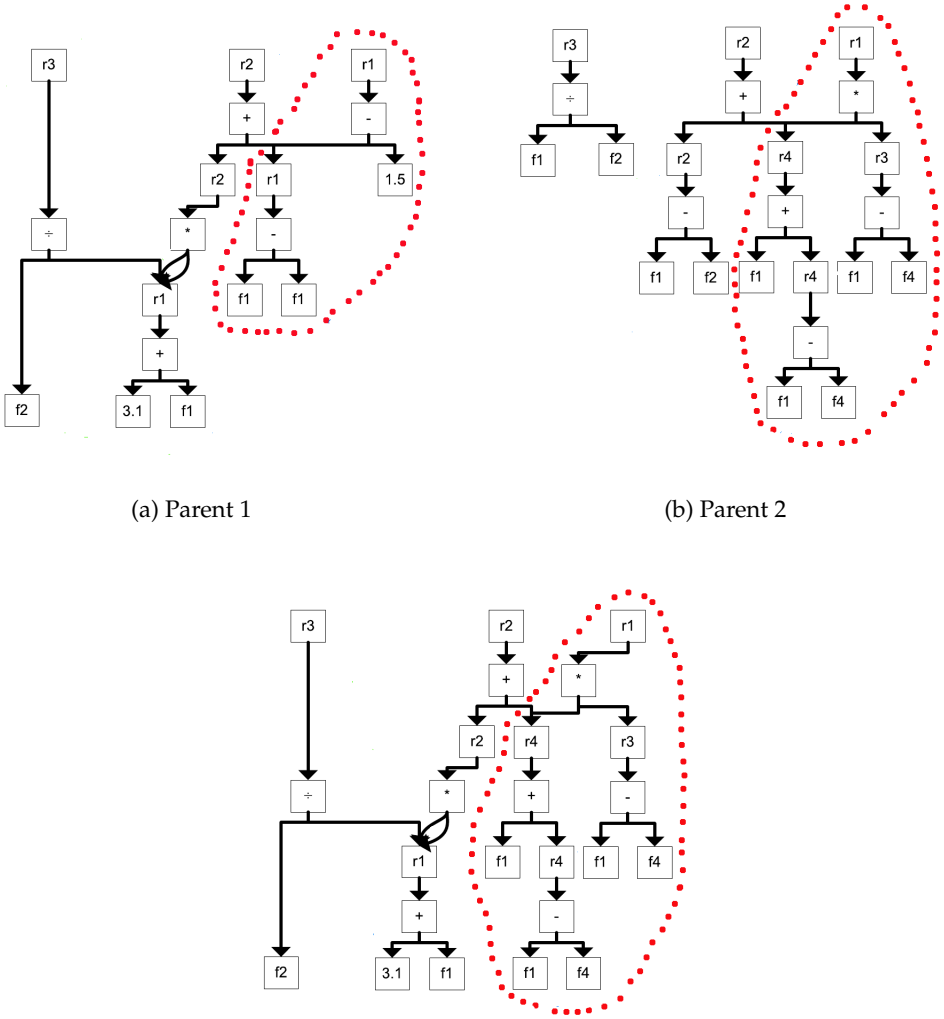
Class Graph Determination

In order to perform crossover we must determine which instructions we should exchange. In CGC this means determining which instructions belong to the class graphs we are exchanging. We can find all of these instructions in a single backwards pass through the program by building up each class graph from its root node.

```
List instructions; //those instructions in any class tree we want
Set registers;

add all the indexes for the classes we want class trees for to registers;

for each instruction ins in reverse order
  if ins is not a intron
    flag = false;
    if ins is a conditional
      if instructions contains the next instruction
        add ins to instructions;
        flag = true;
    else
      let d = the destination register;
      if registers contains d
        add ins to instructions;
        remove d from registers;
        flag = true;
    if flag
      for argument which is a register
        let r be the register;
        if registers does not contain r
          add r to registers;
```

Class Graph Crossover

```
let first, second be two programs;
let classes be the set of classes we are exchanging alleles from;

for each class c
  add c to classes with probability 0.5;

let ins1 be the instructions to exchange from program 1;
let ins2 be the instructions to exchange from program 2;

let n = min of |ins1|, |ins2|;

for 1<=i<=n
  replace instruction ins1[i] with instruction ins2[i] in program 1;

if |ins1| > |ins2|
  for n<=i<=|ins1|
    remove instruction ins1[i] from program 1;
else if |ins2| > |ins1|
  for n<=i<=|ins2|
    add instruction ins2[i] to the end of program 1;
```



(a) Parent 1

(b) Parent 2

(c) Offspring

Figure 6.1: Example of Class Graph Crossover where the r1 class graph is exchanged

6.4 Results and Analysis

We attempted to solve the three problems described in chapter 3 first using GP with conventional crossover, then secondly with GP using class graph crossover. All other parameters remained constant between the two experiments. The parameter configurations can be found in table 6.1. Note that we are required to use a larger number of runs in this set of experiments in order to demonstrate the significance of our results.

Table 6.1: Parameter Configurations

Parameter	LGP	LGP + Class Graph Crossover
Conventional Crossover	30%	0%
Class Graph Crossover	0%	30%

The results in table 6.2 were obtained by running LGP and LGP + CGC on the three problems. The first line shows that for the artificial characters problem, LGP with conventional crossover achieved an average accuracy of 82.02% with a s.d. of 5.72% on the *test set* over 150 runs. It also shows that the LGP with CGC approach achieved an average accuracy of 84.61% with a s.d. of 5.75%.

Table 6.2: Classification Accuracy

Data Set	Normal Crossover		Class Tree Crossover	
	Mean	S.D.	Mean	S.D.
Artificial Characters	82.02%	5.72%	84.61%	5.73%
Segmentation	75.46%	2.81%	76.30%	3.90%
Digit Recognition	65.46%	3.64%	68.04%	3.86%

The results in table 6.3 were obtained by performing a 2 tailed students t-test based on the results summarized in table 6.2. The first line describes the results of the t-test on the artificial characters problem. It shows that the improvement in classification accuracy that we have obtained on this problem has only a 0.0001 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 6.3: Significance of Results

Data Set	2-Tailed P value	Is Result Significant
Artificial Characters	0.0001	Yes
Segmentation	0.0331	Yes
Digit Recognition	0.0001	Yes

These results show that GP using CGC demonstrates improved classification accuracy over GP using conventional crossover on all test data sets, and the improvement is statistically significant in all three problems. This clearly indicates that GP using CGC is superior to GP using conventional crossover for solving multiclass classification problems.

So in Summary, we have structured and restricted the crossover process with a negligible increase in complexity, and the resulting operator shows significant improvement over the original operator. So despite restricting crossover, performance has actually increased, which indicates that crossover restricted to the alleles defined in this chapter is superior to the conventional unrestricted crossover operator. We believe that this indicates that the restriction on code exchange is being compensated for by a decrease in building block disruption.

6.5 Further Discussion

6.5.1 Building Blocks

In CGC, we are swapping the class graphs for some random number of classes. If a building block is present in the program, it must affect the final value of a class register. This means it must be entirely contained within the class graph of that class. So if that class graph is exchanged, the entire building block is exchanged as well. Since we are swapping entire class graphs, this means that we are exchanging nothing but entire building blocks. This is a distinct improvement on conventional crossover where partial building blocks may be exchanged

While CGC ensures that only entire building blocks are exchanged, we have said nothing of those building blocks not exchanged. Because LGP programs consist of overlapping class graphs, it is possible that part of a building block may belong to more then one class graph. This means building blocks not exchanged may be disrupted by CGC. However CGC replaces every class graph with a position equivalent class graph, and position equivalent class graphs code for the same class. Two code sequences which aim to perform the same function are statistically more likely to be similar than two random instruction sequences. This means that position equivalent class graphs are more likely to have similar code then two random sequences of instructions. It is possible that disrupted building blocks may be repaired by the code which replaces the amputated part of the building block. Because position equivalent class graphs are more likely to have similar code, the likelihood of repair is higher under CGC. So even though building blocks which are not exchanged may be disrupted by CGC, they are more likely to be repaired by the substituted code.

Hence it is likely that CGC is superior to conventional mutation as a crossover operator in terms of the number of building blocks it disrupts.

6.5.2 Crossover Complexity

The difference between CGC and conventional crossover becomes most apparent if we consider the number of possible crossovers which can occur under each technique.

Under a conventional crossover scheme, instructions are exchanged at random. Assume for simplicities sake that the number of instructions exchanged is the same for both parents, i.e. n instructions from program 1 are exchanged with n instructions from program 2. Then if we exchange x instructions from programs of size n, then:

- Number of possible sets of instruction from a single program = C_x^n.
- Number of possible crossovers = $C_x^n \times C_x^n$.

Under a CGC scheme, we exchange only class graphs during crossover. Hence if we exchange x class graphs for a problem with c classes:

- Number of possible crossovers = C_x^c.

Notice how the number of possible crossovers depends only on the number of classes and the number of classes to be exchanged, i.e. it is independent of the program size!

To give some idea of the difference this makes, we calculate an example based on the very modest maximum program size used during these experiments.

- Max program size = 32, number of classes = 10.
- Let crossover be 50% of program size, so 16 instructions or 5 class graphs.
- Conventional Crossover = $C_{16}^{32} = 6.01 \times 10^8$
- Class Graph Crossover = $C_5^{10} = 2.52 \times 10^2$

So we have reduced the number of possible crossovers by *6 orders of magnitude*, even for this very modest program size. This gives some idea of how severely the CGC operator restricts code exchange compared to conventional crossover. Crossover is all about searching for the code to exchange which results in the best children, with the conventional crossover operator doing this by blind trial and error. Hence the problem complexity is proportional to the number of possible children, and hence the number of possible crossovers. So if we can decrease problem complexity by 6 orders of magnitude and still achieve similar results we have made major progress in improving the crossover operator.

6.6 Chapter Summary and Future Work

In this section we have introduced a new form of crossover, class graph crossover. This form of crossover limits the sets of valid instructions which can be swapped by introducing a knowledge of abstract program structure into the crossover procedure. To be precise we only exchange position equivalent class graphs during crossover. By following this procedure and restricting code exchange, we believe that the disruptive effects of crossover are minimized. We have shown that this new form of crossover significantly improves the performance of LGP on multiclass classification tasks.

So in short we have severely restricted which sections of code can be exchanged, yet the average program performance has improved! We attribute this to the postulated decrease in building block disruption under a CGC scheme. If we have significantly reduced building block disruption, this represents an important milestone. Future work would focus on building block analysis and determining if CGC really does decrease building block disruption, likely through empirical means.

In addition to decreasing building block disruption and improving program performance, the introduction of *abstract structure* to the crossover operator, at negligible complexity cost, is an important developmental step in and of its own right. This structure can form the basis of further improvement to the crossover operator. Specifically the CGC operator is used to develop a selective crossover operator in chapter 7.

Chapter 7

Selective Crossover

7.1 Introduction

Historically, the conventional crossover operator was introduced as a black box, a stochastic process which could result in solutions which may or may not be superior to the original ones [15]. More recently developed crossover operators focus on improving GP performance by increasing the likelihood that crossover will result in children superior to the parents [31]. These are directed crossover operators: the process of generating children is no longer entirely random, instead there is some sort choice or direction involved. Such methods include brood recombination crossover, where many children are created, and only the best selected, and headless chicken crossover, where the children replace the parents only if they outperform them. While these types of methods have been shown to give improved performance on many problem types [31], they suffer from several known problems.

Firstly, many of these methods such as the headless chicken crossover method use a hill climbing approach to improve classification accuracy. Hill climbing methods attempt to exploit existing potential solutions to find better solutions. When using a hill climbing search algorithm we only ever update our potential solutions if they are better then the previous one. In the context of GP, this means that crossover will only replace the parents with the children if the children outperform the parents.

Hill climbing techniques give excellent results on many problems, particularly those where there is only one hill to climb, because hill climbing in such a context will always result in an optimal solution [20]. However, a large number of interesting search problems have many hills, some peaking lower than others. This means hill climbing techniques can get stuck at a low peak, known as a local optima, and thus be unable to find a good solution. To avoid this problem, known as the local optima problem, a search technique must be able to search in all directions, not just uphill. This is can be achieved by use of randomness, with solutions being selected stochastically instead of deterministically. Existing GP operators which exhibit hill climbing behavior rarely take this into account and hence often fail on problems with many local minima [20].

Secondly, these methods typically come with a large increase in computational overhead. Hence although they improve GP performance in terms of increased classification accuracy after a given number of generations, such a comparison is not a fair one. This comparison does not take into account the difference in the number of function evaluations performed by the two methods.

In GP, the primary computational cost is fitness evaluation. Evaluation of a population of n individuals with x training instances requires each individual to be evaluated against each instance, total cost nx. Under many of these schemes such as brood recombination crossover, we are required to evaluate the fitness of many potential children in order to

select the actual children. This means many more fitness evaluations, and thus a higher computational cost. If we have a 50% crossover rate, and each crossover operator requires evaluation of 4 children, then the new computational cost will be $nx + 0.5 * n * 4 * x = nx + 2nx = 3nx$. Hence we have effectively tripled the computational cost of GP. So in effect, performing GP with brood crossover for x generations is equivalent to performing GP with normal crossover for $3x$ generations. When GP is run for a larger number of generations, the classification accuracy typically increases, so the performance of normal GP will almost certainly improve from x generations to $3x$ generations. Hence comparing performance based solely on classification accuracy after a given number of generations is a flawed test.

Fair tests are likely to reveal that such computationally intensive crossover operators don't give classification accuracy improvements on the scale originally thought. Clearly what is required to give a real performance improvement is a crossover operator which lets us select for superior children without the necessity of evaluating the fitness of additional programs.

In short, a good crossover operator has two key properties. Firstly, it exploits good solutions by attempting to find better ones nearby by using some form of hill climbing, but does so stochastically to avoid getting stuck at local optima. Secondly, it has a low computational cost, which means it cannot require performing any extra fitness evaluations. We believe that a crossover operator with both of these properties should outperform conventional crossover in a fair test.

7.1.1 Chapter Goals

In this chapter we aim to develop such an operator by building on the Class Graph Crossover operator developed in chapter 6. Specifically, this chapter aims to answer the following research questions:

- How can we create a crossover operator which exploits good existing solutions whilst avoiding the local optima problem?
- How can we create a crossover operator which has increased likelihood of good offspring with at most a negligible increase in computational cost?
- Does this selective crossover operator outperform the conventional crossover operator on a sequence of multiclass classification problems?
- Does this selective crossover operator outperform the class graph crossover operator on a sequence of multiclass classification problems?
- Does the combination of selective crossover and selective mutation outperform either operator alone on a sequence of multiclass classification problems?

7.2 Selective Crossover

As it turns out, we can develop a crossover operator that exploits existing solutions and has low computational cost by modifying the Class Graph Crossover (CGC) operator developed in chapter 6. However before we discuss this we need to cover an important point.

7.2.1 One Child Crossover

Normal crossover takes two parents and produces two children by an exchange of code, however this has a major issue if we are trying to optimize the offspring produced. In order

to optimize one child, we will have to give all of the good code to that child. This means the second offspring will receive the remaining, rejected code. Hence the more we manage to optimize one offspring, the worse the second one will be likely to get. So then let's consider the case where we have two parents, but produce only a single child, by substituting code from the second program into the first one. In this way we avoid the problem of almost always producing a poor child.

7.2.2 Directed Class Graph Crossover

In CGC, crossover is constrained to the exchange of position equivalent class graphs. If there are n classes, each program has n class graphs. We are choosing x of these to exchange, so there are ${}^nC_x$ possible combinations of instructions we could potentially exchange. Of the ${}^nC_x$ possible children which would result from these combinations, some will result in good children, and some will result in poor children. What we really want is to exchange the subset of class graphs which results in the children with the best fitness. To solve this problem precisely we would be required to try every possible subset of class graphs, and choose the subset which results in the best children. While this is a significantly lower number of possibilities than conventional crossover, this method requires exponentially many fitness evaluations, so is clearly undesirable. An alternative, used in brood recombination crossover, is to try some small number of children and select the best from this sample. Brood crossover is a computationally feasible compromise of this method where we use some small sample of subsets instead of all possible subsets, however we have already rejected brood crossover because of its computational overhead.

7.2.3 Main Idea of Selective Crossover: Introducing Heuristics into CGC

A superior technique, and the one developed in this chapter, is to use a heuristic to *predict* which subsets will result in good children without having to actually generate and evaluate those children! A more accurate heuristic can replace the costly process of generating and testing children and hence solve our computational cost problems. All we need to do, then, is to find a heuristic which allows us to cheaply predict the best subset of class graphs to exchange.

The heuristic we have developed is based closely on the work in chapter 5 on selective mutation. By definition, each class graph has a class, and hence each class graph is responsible for a certain number of misclassified training instances (see chapter 5 for details). Good class graphs are responsible for a low number of misclassifications, bad class graphs are responsible for a large number of misclassifications. A good program has good fitness, and hence a small number of misclassifications. So all of the class graphs in a good program are good class graphs, i.e. each class graph is responsible for only a small number of misclassifications. So in other words we are aiming for a program which consists entirely of good class graphs. Replacing a poor class graph with a good class graph should improve program fitness. Therefore a crossover can be considered good if it replaces one or more class graphs with better class graphs, and bad if it replaces one or more good class graphs with a poorer ones.

Because the correctness of each class graph is represented by a single integer, the difference in correctness between class graphs can be calculated by simple subtraction. If we then order the class graphs in descending order according to this difference, it is clear that taking the first x class graphs as our subset should give the optimum children. In other words, we find the class graphs in the second program which are better then the position equivalent class graphs in the current program, and exchange these class graphs preferentially. So sup-

pose parent one is poor at classifying certain class instances, whereas parent two is good at classifying these same class instances. Then the selective crossover operator would be expected to replace the poor code in program one with the good code in program two.

7.2.4 Discussion of Stochasticity

It should be noted that this is not strictly hill climbing because the process is stochastic insofar as it will sometimes result in children which are worse then the parents. There are three primary sources of stochasticity in selective crossover.

Firstly, just because a class graph gives good performance in program two, there is no guarantee that same class graph will perform well in program one. In other words, our prediction is just that, a prediction, and as such may be wrong. In fact sometimes it may be the case that exchanging the class graphs *predicted* to be worst could even turn out to give the best possible child! However the theoretical basis of selective crossover lies in the fact that a class graph which performs well in one program is likely to perform well in another program. Certainly such a class graph should be more likely to perform well than a randomly selected class graph would be. What this means is that while SC should often be producing high fitness children, this will not always be the case. Hence this is a semi-stochastic process and not true hill climbing.

Secondly, we are choosing the number of class graphs we are exchanging at random. A true hill climber would only exchange those class graphs which lead to an increase in the fitness of the child. So if there are n class graphs in the second program which outperform the structure equivalent class graphs in the first program, a true hill climber would exchange only these n graphs. In contrast, the selective crossover operator will exchange a random number of class graphs. This means it may make poor exchanges, or not make exchanges which our heuristic predicts would have been beneficial. Some work was done on a deterministic selective crossover operator which made exactly those exchanges our heuristic predicted would be beneficial. However preliminary results indicated this overly constrained the permissible code exchanges, and hence gave poor results.

Finally, we are selecting class graphs to exchange semi-stochastically. To be precise the likelihood that we select any two position equivalent class graphs is directly proportional to the expected improvement to be gained from such an exchange. The expected improvement from exchanging any two position equivalent class graphs is simply the difference in class values as discussed above. Therefore the likelihood that we select any two position equivalent class graphs is directly proportional to the difference in their class values. Note that if the class graph in the original program has a higher class value then the class graph in the second program, then the difference will be negative. In this case we simply make the difference one, as do we if the class values are equal. This gives such class graphs a minimal likelihood to be selected for exchange, but does not preclude the possibility entirely. In other words we will occasionally end up exchanging good class graphs for poor class graphs.

7.2.5 Algorithm

We find then that we have arrived at our algorithm. By finding the difference in class values for each pair of position equivalent class graphs, and ordering the difference, we can predict the optimum set of class graphs to exchange, achieving our first goal. In addition, we can do all this for negligible cost, so we have also achieved our second goal. The details are as follows.

Determine Class graphs

The algorithm for determining class graphs is described in detail in chapter 6 on CGC. To recap: it consists of building all the class graphs simultaneously from the root up by iterating backwards through the program. Clearly the same algorithm is applicable to selective crossover since we are building on the CGC operator.

Determine Class Values

The algorithm for determining class values is described in detail in chapter 5 on Selective Mutation. To recap: it consists of counting the number of training instances of each class which are misclassified. Clearly we can use this same procedure to assign a class value to each class graph.

Performing Crossover

Once we have decomposed each program into a set of class graphs with assigned class values, the actual crossover process is relatively straight forward:

```
let first, second be two programs
let fClass, sClass be their respective arrays of class values

array diff = fClass - sClass;
let list1, list2 be two lists of instructions;

normalize diff to generate a prob distribution over the class numbers;
sample from this ditribution to get the required number of class numbers;
for each class number cn in result
  add to list1 the instructions in the program 1 class tree with index cn;
  add to list2 the instructions in the program 2 class tree with index cn;
exchange the instructions in list1 and list2;
```

7.3 Complexity Analysis

In order to demonstrate that our tests are fair tests, we need to show that using the selective crossover operator results in a negligible increase in computational cost. In GP the primary computational cost is in fitness evaluation. If we have n training instances, and programs have length l, then evaluating the fitness of a single program takes $n * l$ steps. Meanwhile performing conventional crossover on a program involves exchanging $x < l$ instructions, so takes x steps. Typically values for these parameters would be 1,000 training examples, with programs of at most length 100. So fitness evaluations takes $1000 * 100 = 100{,}000$ steps, whereas crossover takes at most 100 steps. We can clearly see that conventional crossover has negligible computational cost compared to the cost of fitness evaluation.

Selective crossover clearly has a higher computational cost than conventional crossover, as we are required to do the extra work of calculating class values for the class graphs in each parent. It remains to show that even though the computational cost is increased, this cost is still negligible compared to the computational cost of fitness evaluation. Calculating the class values for each class graph can be done during fitness evaluation. Instead of keeping track of simply the number of misclassifications, we can instead keep track of the number of misclassifications for each class, for no extra cost. The only cost will be a small increase in memory usage since we have to store a larger number of fitness values for each class. Determining which class graphs to exchange involves comparing the class values of position

equivalent class graphs. If c is the number of possible classes, then this comparison will take c steps, note that $c < l$. Finally, we must determine which instructions belong to the class graphs we are exchanging. This process requires a single backwards pass through each program, so computational cost will be linear in program length. In other words calculating which instructions to swap will take l steps. So the cost of selective crossover will be the cost of conventional crossover + the sum of the new costs described, i.e. at most $l + l + l = 3l$. So on the same example data the cost of selective crossover will be 300 steps, which is still negligible compared to the 100,000 steps required for fitness evaluation.

Hence we have shown that selective crossover results in a negligible increase in computational cost, and hence comparing selective crossover directly to CGC or conventional crossover constitutes a fair test.

7.4 Results and Analysis

We attempted to solve the three problems described in chapter 3 firstly using GP with conventional crossover, then secondly with GP using class graph crossover, and finally with GP using selective crossover. All other parameters remained constant between the experiments. The parameter configurations can be found in table 7.1. Note that we are required to use a larger number of runs in this set of experiments in order to demonstrate the significance of our results.

Table 7.1: Parameter Configurations

Parameter	LGP	LGP + Class Graph Crossover	Selective Crossover
Population	500	500	500
Max Gens	200	200	200
Normal Mutation	60%	60%	60%
Elitism	10%	10%	10%
Conventional Crossover	30%	0%	0%
Class Graph Crossover	0%	30%	0%
Selective Crossover	0%	0%	30%
Max Size	32	32	32
Tournament Size	4	4	4
Runs	150	150	150

7.4.1 Selective Crossover vs. Normal Crossover

The results in table 7.2 were obtained by running LGP and LGP with selective crossover on the three problems. The first line shows that for the artificial characters problem, LGP with conventional crossover achieved an average accuracy of 82.02% with a s.d. of 5.72% on the *test set* over 150 runs. It also shows that the LGP with selective crossover approach achieved an average accuracy of 86.65% with a s.d. of 4.74%.

The results in table 7.3 were obtained by performing a 2 tailed students t-test based on the results summarized in table 7.2. The first line describes the results of the t-test on the artificial characters problem. It shows that the improvement in classification accuracy that we have obtained on this problem has only a 0.0001 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 7.2: *Classification Accuracy:* conventional crossover vs. selective crossover

Data Set	Class Graph Crossover		Selective Crossover	
	Mean	S.D.	Mean	S.D.
Artificial Characters	82.02%	5.72%	86.65%	4.74%
Segmentation	75.46%	2.81%	77.47%	3.91%
Digit Recognition	65.46%	3.64%	69.16%	3.21%

Table 7.3: *Significance of results:* conventional crossover vs. selective crossover

Data Set	2-Tailed P value	Significant
Artificial Characters	0.0001	Yes
Segmentation	0.0001	Yes
Digit Recognition	0.0001	Yes

First we compare the performance of LGP using the selective crossover operator to the performance of the LGP using the conventional crossover operator. The results, summarized in table 7.2 show that LGP using selective crossover outperforms LGP using conventional crossover on all data sets. In addition, a simple 2 class students t-test of statistical significance with a 95% confidence threshold demonstrates that this improvement in classification accuracy is a significant improvement.

While this is a good result, we developed the selective crossover operator by extending the class graph crossover operator, which already outperforms the conventional crossover operator by a significant amount. Hence all we have really demonstrated is that our modifications have not adversely affected the performance of our crossover operator. To really show that LGP using selective crossover is a worthwhile technique, we need to demonstrate its superiority to LGP using class graph crossover. In other words we need to show that our modifications have improved the performance by a *significant amount*. This leads us to our second set of experiments.

7.4.2 Class Graph Crossover vs. Selective Crossover

The results in table 7.4 were obtained by running LGP with CGC and LGP with selective crossover on the three problems. The first line shows that for the artificial characters problem, LGP with CGC achieved an average accuracy of 84.61% with a s.d. of 5.73% on the *test set* over 150 runs. It also shows that the LGP with selective crossover approach achieved an average accuracy of 86.65% with a s.d. of 4.74%.

Table 7.4: *Classification Accuracy:* class graph crossover vs. selective crossover

Data Set	CGC		Selective Crossover	
	Mean	S.D.	Mean	S.D.
Artificial Characters	84.61%	5.73%	86.65%	4.74%
Segmentation	76.30%	3.90%	77.47%	3.91%
Digit Recognition	68.04%	3.86%	69.16%	3.21%

The results in table 7.5 were obtained by performing a 2 tailed students t-test based on the results summarized in table 7.4. The first line describes the results of the t-test on the artificial characters problem. It shows that the improvement in classification accuracy that we have obtained on this problem has only a 0.0009 probability of having occurred by

chance. It also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 7.5: *Significance of Results:* class graph crossover vs. selective crossover

Data Set	2-Tailed P value	Significant
Artificial Characters	0.0009	Yes
Segmentation	0.0099	Yes
Digit Recognition	0.0067	Yes

In our second set of experiments, we compare the performance of the selective crossover operator to the performance of the class graph crossover operator. The results, summarized in table 7.4 show that LGP using selective crossover outperforms LGP using conventional crossover on all data sets. In addition, a simple 2 class students t-test of statistical significance with a 95% confidence threshold demonstrates that this improvement in classification accuracy is a significant improvement on all problems.

These results clearly demonstrate the superiority of LGP using selective crossover over LGP using class graph crossover and conventional crossover, as a method for performing multiclass classification. In addition, this is a fair test, because our modifications to the CGC operator result in only a negligible increase in computation complexity. So LGP using selective crossover results in a *genuine and significant improvement* in classification accuracy over LGP using class graph crossover.

7.4.3 Combining Selective Crossover with Selective Mutation

In our final set of experiments we compare the performance of LGP using both selective crossover and selective mutation to the performance of LGP with either selective crossover or selective mutation but not both.

The results in table 7.6 were obtained by running LGP with both selective crossover and selective mutation and against LGP with just one of the two new operators on the three problems. The first line shows that for the artificial characters problem, LGP with selective mutation achieved an average accuracy of 85.86% with a s.d. of 4.92% on the *test set* over 150 runs. It also shows that the LGP with selective crossover approach achieved an average accuracy of 86.65% with a s.d. of 4.74%. Finally it shows that LGP with both new operators achieved an average accuracy of 87.69% with a s.d. of 4.10%.

Table 7.6: *Classification Accuracy:* class graph crossover vs. selective crossover

Data Set	Selective Mutation		Selective Crossover		Both	
	Mean	S.D.	Mean	S.D.	Mean	S.D.
Artificial Characters	85.86%	4.92%	86.65%	4.74%	87.69%	4.10%
Segmentation	77.91%	4.19%	77.47%	3.91%	78.45%	4.00%
Digit Recognition	67.32%	4.29%	69.16%	3.21%	70.61%	3.59%

The results in table 7.7 were obtained by performing a 2 tailed students t-test based on the results summarized in table 7.6. The first line describes the results of the t-test on the artificial characters problem. Firstly it describes the significance of the performance improvement obtained by changing from using only the selective mutation operator, to using both new operators. It shows that the improvement in classification accuracy which occurs has only a 0.0005 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95. In addition it shows that

the improvement Secondly it describes the significance of the performance improvement obtained by changing from using only the selective crossover operator, to using both new operators. It shows that the improvement in classification accuracy which occurs has only a 0.0430 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95

Table 7.7: *Significance of Results:* class graph crossover vs. selective crossover

Data Set	Selective Mutation		Selective Crossover	
	2-Tailed P value	Significant	2-Tailed P value	Significant
Artificial Characters	0.0005	Yes	0.0430	Yes
Segmentation	0.0478	Yes	0.0327	Yes
Digit Recognition	0.001	Yes	0.0003	Yes

We can see that LGP using LGP using both selective mutation and selective crossover outperforms LGP using only one of the two methods by a significant amount on all problems. These results clearly demonstrate the superiority of LGP using both new operators over LGP using only one of the two new operators, as a method for performing multiclass classification.

7.5 Chapter Summary and Future Work

In this chapter we have developed a new crossover operator that has positively answered all of the research questions asked at the start of this chapter. By using a heuristic to predict which class graphs should be exchanged in order to maximize offspring fitness, we can direct the crossover process without requiring any extra fitness evaluations. This selective crossover operator has a computational complexity negligible higher than that of class graph crossover, yet achieves significantly improved classification accuracy. Hence it genuinely outperforms both the conventional crossover operator, and the CGC operator from which it was developed.

The use of a heuristic to predict the optimal crossover points is a major development, but would not be possible without the work done in chapter 6. This work is specific to the domain of object classification problems, and cannot be directly applied to other problem domains such as regression. This means that to apply the technique of heuristically directed crossover to a given LGP problem type, a model must first be developed for the programs evolved. This model must decompose programs into distinct components which can somehow be given an estimate for how much they contribute to program correctness. Once a model has been developed which satisfies this property, developing a heuristically driven crossover operator is a simple task. Future work will focus on developing such models for problems not in the object classification problem domain.

While selective crossover demonstrates significant improvement on all data sets, we believe that the technique can be improved further. At the heart of selective crossover is the heuristic used to predict the optimal offspring. More precise predictions are likely to result in a selective crossover operator that has higher classification accuracy. Hence if we can improve the precision of our heuristic, we should be able to further improve the performance of the selective crossover operator. That said, we need to increase prediction precision without introducing a significant increase in the computational cost, or any improvements we achieve will be negated. Future work will focus on developing a better estimate of which components should be exchanged to produce highly fit offspring.

Chapter 8

Additional Extensions to LGP

8.1 Introduction

Improvements to the GP learning algorithm are often developed in a modular fashion. By this we mean that research into GP will often focus on improving one feature of GP in isolation from all others. This can be seen from the large number of new crossover operators which have been developed in isolation such as brood crossover [31], headless chicken crossover [21]. We believe that it is possible to improve the performance of one area of GP by a complementary modification to another area. Specifically, we believe that it is possible to improve the performance of the class graph crossover operator, through modifications to other components of the LGP algorithm.

In class graph crossover, some number of position equivalent class graphs are exchanged between two programs. A good program is composed entirely of good class graphs, hence a successful crossover is one that results in an offspring consisting entirely of good class graphs. In order for a good class graph to occur in an offspring, it must first appear in at least one of the parent programs. Hence if n is the class number, and both parents have a bad nth class graph, then any offspring will be almost certain to also have a bad nth class graph. So what we really want is for both parents to complement each other. In other words, if parent one has a weak nth class graph, then parent two should have a good nth class graph, and vice versa. Hence two mediocre parents which are strong in different areas are more desirable than two parents which are stronger individually, but weak in the same areas.

So if we can encourage the mating of programs which have strengths in different areas, we should be able to improve the performance of both class graph crossover and selective crossover. It should have a particularly pronounced effect on selective crossover, because selective crossover heuristically selects the stronger parent allele to form the offspring. Hence selective crossover will exploit the strengths of both parents to form an offspring which contains these strengths.

There are at least two ways we can improve the likelihood that the programs chosen to mate complement each others strengths. The first option is to increase the diversity of the population. In other words we encourage the emergence of programs with diverse strengths to appear in the population. The second option is to select complementary programs to be parents. In other words once we have selected the first parent, we select the second parent based on how well it complements the first one. Both of these approaches require us to make changes to the LGP learning algorithm which are outside the scope of the crossover operator, however both approaches indirectly influence the crossover operator.

8.1.1 Chapter Goals

In this chapter we aim to develop two new modifications to the LGP learning algorithm which complement the class graph crossover and selective crossover operators developed in chapters 6 and 7. Specifically this chapter aims to answer the following research questions:

- How can we modify the LGP learning algorithm to encourage individuals with diverse strengths to occur in the population?
- How can we modify the LGP learning algorithm to increase the likelihood that complementary programs are selected as parents during crossover?
- Does LGP with selective crossover and either or both of the above modifications significantly outperform LGP with only selective crossover on a sequence of multiclass classification problems?
- Does LGP with selective crossover, selective mutation and either or both of the above modifications significantly outperform LGP with selective crossover and selective mutation on a sequence of multiclass classification problems?

8.2 Diversive Elitism

Our first question asks whether and how we can diversify our population to preferentially contain individuals with strengths in different area. What we really care about is preserving the programs with the best performance so far in each area. Hence we are really dealing with a modification to the elitism operator. The existing elitism operator copies the best x individuals directly from the current generation to the next generation. Hence conventional elitism ensures that the fitness of the next generation is at least as good as the fitness of the current generation.

We propose a new form of elitism that selects individuals based on the performance of their class graphs, instead of based on program performance as a whole. To be precise, for each class we would copy to the next generation the x/n (where n is the number of classes) programs with the best performance on training instances of that class. This means that we encourage the emergence of programs which are good at classifying each of the n classes of instances. Hence we ensure that for every class i there are programs present in the population with good *i'th* class graphs which can be potentially selected as parents. We refer to this new form of elitism as Diversive Elitism (DE), and postulate that it should outperform conventional elitism on a sequence of multiclass classification problems.

8.2.1 Algorithm

```
let n be the number of classes
let x be the number of programs generated by elitism

for each class 1<=i<=n
  order the programs by the performance of the $i$'th class tree;
  select the top x/n individuals and place them in the next gen;
```

8.3 Class Graph Selection

Our second questions asks whether and how we can increase the likelihood that the programs chosen to be parents will be complementary to each other. If we assume the first

parent is chosen at random, then what we are really asking is can we increase the average amount by which the second parent complements the first parent.

Assume we are using tournament selection. In conventional tournament selection, the program with the best fitness wins the tournament and is selected for reproduction. But is this really the best strategy? What we are really trying to do is to select the parents which will give us the best offspring. We already discussed how two very fit programs with the same weaknesses typically give worse results then two weaker programs that specialize in different areas. Hence selecting parents based solely on fitness is clearly not always a good strategy. What we really need is a heuristic that given any two parents, can be used to *predict* the fitness of offspring.

As it turns out, we have already developed such a heuristic in chapter 7! In chapter 7, we calculate the difference in class tree values and use it as a heuristic to predict which class trees to exchange. By summing the possible improvement for each class tree, we arrive at a prediction of how much improvement is possible for the entire program. So we can then predict the possible improvement to be gained from mating with each individual in the tournament, and select the one with the greatest possible predicted improvement. Mating with this individual is more likely to produce high fitness offspring than mating with any other individual in the tournament pool. We call this new selection operator Class Graph Selection (CGS).

8.3.1 Algorithm

```
select parent one using normal tournament selection;
select n random individuals from the population;
for each individual
  determine the predicted improvement from mating with that individual;
select the individual with highest predicted improvement as parent 2;
```

8.4 Results and Analysis

Because these techniques are designed to complement selective crossover, applying them to crossover which does not use selective crossover does not make sense. Hence we only apply them to LGP using selective crossover. In addition both techniques complement each other, being two facets of the same idea. Hence both techniques would always be applied together so testing them individually is pointless. Therefore our primary set of experiments consist of comparing LGP with selective crossover to LGP with selective crossover and both the diversive elitism and class graph selection methods. We also rerun the same tests, but with LGP using all of the above and additionally selective mutation. This final test represents the culmination of all of the work in this project, insofar as it uses every new operator we have developed, and hence we should expect it to achieve the best results of all experiments so far.

8.4.1 LGP with SC, DE and CGS vs LGP with SC

The results in table 8.1 were obtained by running TGP with selective crossover, diversive elitisim and class graph selection, and LGP with selective crossover on the three problems. The first line shows that for the artificial characters problem, TGP with selective crossover, diversive elitism and class graph selection achieved an average accuracy of 86.65% with a s.d. of 4.74% on the *test set* over 150 runs. It also shows that the LGP with selective crossover approach achieved an average accuracy of 88.72% with a s.d. of 5.19%.

Table 8.1: *Classification Accuracy:* LGP with SC, DE and CGS vs. LGP with SC

Data Set	LGP with SC		LGP with SC, DE and CGS	
	Mean	S.D.	Mean	S.D.
Artificial Characters	86.65%	4.74%	88.72%	5.19%
Segmentation	77.47%	3.91%	78.50%	4.85%
Digit Recognition	69.16%	3.21%	69.99%	3.13%

The results in table 8.2 were obtained by performing a 2 tailed students t-test based on the results summarized in table 8.1. The first line describes the results of the t-test on the artificial characters problem. It shows that the improvement in classification accuracy that we have obtained on this problem has only a 0.0009 probability of having occurred by chance. It also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 8.2: *Significance of Results:* LGP with SC, DE and CGS vs. LGP with SC

Data Set	2-Tailed P value	Significant
Artificial Characters	0.0009	Yes
Segmentation	0.0099	Yes
Digit Recognition	0.0067	Yes

So LGP with SC, DE and CGS outperforms LGP with SC by a significant amount on all test sets. This indicates that DE and CGS are superior techniques for solving multiclass classification problems.

8.4.2 LGP with all techniques vs LGP with SC and SM

The results in table 8.1 were obtained by running GP with selective crossover, selective mutation, diversive elitisim and class graph selection, and LGP with selective crossover and selective mutation on the three problems. The first line shows that for the artificial characters problem, TGP with selective crossover, selective mutation, diversive elitism and class graph selection achieved an average accuracy of 86.65% with a s.d. of 4.74% on the *test set* over 150 runs. It also shows that the LGP with selective crossover and selective mutation approach achieved an average accuracy of 88.72% with a s.d. of 5.19%.

Table 8.3: *Classification Accuracy:* LGP with Selective Crossover and Selective Mutation vs. LGP with Selective Crossover, Selective Mutation, Diversive Elitism and Class Graph Selection

Data Set	LGP with SC and SM		LGP with SC, SM, DE and CGS	
	Mean	S.D.	Mean	S.D.
Artificial Characters	87.69%	4.10%	89.86%	4.49%
Segmentation	78.45%	4.00%	79.03%	3.93%
Digit Recognition	70.61%	3.59%	71.46%	3.85%

The results in table 8.4 were obtained by performing a 2 tailed students t-test based on the results summarized in table 8.3. The first line describes the results of the t-test on the artificial characters prob, and it also shows that this is a significant improvement according to our significance criterion of 0.95.

Table 8.4: *Significance of Results:* LGP with Selective Crossover and Selective Mutation vs. LGP with Selective Crossover, Selective Mutation, Diversive Elitism and Class Graph Selection

Data Set	2-Tailed P value	Significant
Artificial Characters	0.0001	Yes
Segmentation	0.2062	No
Digit Recognition	0.0489	Yes

So LGP with all techniques outperforms LGP with SM and SC by a significant amount on two problems, and a non-significant amount on oneproblem. This is a strong indication that using DE and CGS is a superior techniques for solving multiclass classification problems, although less strong than if the performance improvement on all three problems had been significant.

8.5 Chapter Summary and Future Work

In this chapter we have developed two new operators which encourage the crossover of individuals with diverse strengths, with the aim of complementing the selective crossover operator and producing fitter children. The first of these operators is a new elitism operator called diversive elitism, which selects the fittest individuals in each area, instead of just the fittest individuals overall. The second of these operators is a new selection operator called class graph selection, which selects two programs which best complement each other for crossover, instead of the two fittest children. Diversive Elitism and Class Graph Selection together clearly improve the performance of LGP on a series of multiclass classification problems. Hence this indicates that LGP with these two techniques is a superior method for performing multiclass classification.

Future work would investigate each of these techniques in more detail and on and individual basis. While it is likely that primarily these two new operators would be applied together, it would be nice to know whether using just one new operator also results in a significant performance improvement. In addition, we would like to determine the optimum settings for these the parameters for both of these methods. In other words, do changes such as increasing the elitism rate, or increasing the tournament size improve the performance of DE, or CGS respectively.

Chapter 9

Conclusions and Future Work

In this chapter the research, results, and conclusions of chapters 4-8 are summarized, and a number of overall conclusions are presented. In addition, the future work which is revealed and motivated by the work in this project is discussed.

9.1 Conclusions

Multiclass classification problems occur naturally in many computer vision applications tasks and potential good solutions are of great importance. Currently GP methods are not favored for solving multiclass classification problems due in large part to the program structure of the conventional TGP method. Experiments with an alternative form of GP, LGP, have led us to explore a better program structure and learning algorithm for these types of problems.

The overall goal of this project was to investigate a new approach in LGP for multiclass classification problems. This goal was successfully achieved by developing five new genetic operators and corresponding learning algorithms. These methods were examined and ompared with the canonical algorithms. These methods were examined and compared to the canonical LGP and TGP on three multicass (object) classification problems of increasing difficuly. THe empirical results and theretical analyses show that these methods significantly outperformed the standard LGP and TGP on these problems.

In the rest of the section, we describe the major conclusions derived from the project.

9.1.1 Tree based GP vs. Linear GP

Theory shows LGP is naturally suited to multiclass classification problems because it allows for arbitrarily many outputs and our results back this theory up. Basic LGP outperformed TGP using a naive mapping function all of our problem sets, moreover basic LGP had comparable performance to TGP with a state-of-the-art mapping function. In other words the basic LGP approach which has not been specialized for solving multiclass classification problems has comparable performance to a heavily researched TGP approach which has been optimized for solving such problems. So if the basic LGP technique performs so well, it should be possible to achieve excellent results by optimizing LGP. This motivates our later research into improving the basic LGP method, and we also hope it motivates other researchers to investigate the area.

9.1.2 Selective Mutation

The conventional mutation operator mutates program instructions at random. This is problematic because conventional mutation of fit programs is very likely to cause good instructions to be changed into bad instructions. We have developed a new mutation operator which focuses on mutating the parts of an LGP which are bad, and keeping intact those parts which are good. Our results show that LGP using this new selective mutation operator has significantly superior performance to LGP using conventional mutation on all the problems investigated in this project.

9.1.3 Class Graph Crossover

The conventional crossover operator exchanges program instructions at random between two LGP programs. This is problematic because exchanging instructions at random is likely to break up good sections of code known as building blocks. We have developed a new form of crossover, called Class Graph Crossover (CGC), which we believe alleviates this problem. CGC is inspired by biological crossover, where code exchange only occurs between two code sequences which determine the same feature. In other words both code sequences must determine eye color, or both code sequences must determine gender etc. In the case of LGP programs, both code sequences must determine the same register output. Our results show that LGP with this new crossover operator has significantly superior performance to conventional LGP on all problems.

9.1.4 Selective Crossover

Current advances in crossover operators focus on improving GP performance by increasing the likelihood that the offspring produced by crossover have high fitness. The issue with these methods is that they dramatically increase the computational cost of the crossover operator because they calculate the fitness of many offspring. We have developed a new crossover operator, called selective crossover, which addresses this problem by using a computationally cheap heuristic to predict offspring fitness. This means we no longer have to evaluate offspring, instead we simply use our heuristic to predict how to get better offspring. Our results show that selective crossover outperforms both conventional crossover and class graph crossover by a significant amount on all problems.

9.1.5 Diversive Elitism and Class Graph Selection

Classification accuracy is not good grounds for selecting programs for crossover. Just because two programs each perform well individually does not mean they are well suited to produce good offspring. Crossover, and particularly selective crossover, works best if the two parents have strengths in different areas. We have developed a new elitism operator, and a new selection operator, which both focus on increasing the likelihood that the parents selected for crossover have contrasting strengths. The former acts by increasing the diversity of the population, and the latter acts by selecting the second parent to contrast the first. Our results show that together these two new operators improve the performance of LGP with selective crossover by a significant amount on all problems. Hence we conclude that increasing the likelihood that the parents selected for crossover have contrasting strengths increases LGP performance.

9.2 Future Work

In this section we discuss the future work which this project has motivated.

9.2.1 Selective Mutation

The concept behind selective mutation is simple in its elegance: if we can rate instruction correctness, we can choose which instructions to mutation. Hence while currently this technique can only be applied to classification problems, we believe that it could be adapted and used to improve the performance of LGP on many different problem domains. Future work could focus on developing a model of instruction correctness for LGP programs in other problem domains and using this to expand the the number of problem domains selective mutation can be applied to.

9.2.2 Class Graph Crossover

CGC massively reduces the number of possible code exchanges which could take place, while significantly improving performance. We hypothesize that this is because CGC increases the likelihood of fit offspring by reducing the likelihood of building blocks being disrupted. Future work would focus on confirming this hypothesis, preferably using a combination of statistical proofs and empirical tests.

9.2.3 Selective Crossover

The performance of selective crossover is directly related to the effectiveness of the heuristic used. We hypothesize that further improvement could be gained by designing new and better heuristics that allow us to predict with more accuracy which crossovers would result in the best offspring. Future research would focus on designing, implementing and testing such heuristics. Additionally the concept of using heuristics to predict which instructions should be exchanged during crossover is not limited to multiclass classification problems. Future work could also focus on developing a selective crossover operator for use on other problem domains.

9.2.4 Diversive Elitism and Class Graph Selection

The We are not sure which parameter values result in optimal results for either of these operators. It could be the case the changing the diversive elitism rate, or changing the number of programs considered when using the class graph selection operator could improve performance. Future work could focus on determining the optimal parameter settings.

9.3 Final Remarks

In this report we have developed a number of new operators, all with the goal of improving the performance of LGP on multiclass classification problems. All five operators have demonstrated significantly improved performance on several multiclass classification problems. We believe that this demonstrates the potential in LGP for improvement and we hope it motivates future researchers to further develop the LGP technique, both for multiclass classification problems and problems in other domains.

Bibliography

[1] AGNELLI, D., BOLLINI, A., AND LOMBARDI, L. Image classification: an evolutionary approach. *PRL 23*, 1-3 (January 2002), 303–309.

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

[3] BRAMEIER, M., AND BANZHAF, W. *Linear Genetic Programming*. No. XVI in Genetic and Evolutionary Computation. Springer, 2007.

[4] CARBONELL, J. G., MICHALSKI, R. S., AND MITCHELL, T. M. An overview of machine learning. In *Machine Learning: An Artificial Intelligence Approach*, R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, Eds. Springer, Berlin, Heidelberg, 1984, pp. 3–23.

[5] CRAMER, N. L. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the 1st International Conference on Genetic Algorithms* (Hillsdale, NJ, USA, 1985), L. Erlbaum Associates Inc., pp. 183–187.

[6] DENNET, D. C. Real patterns, 1991.

[7] FOGEL, D. B. *Evolutionary Computation: The Fossil Record*. Wiley-IEEE Press, 1998.

[8] FOGELBERG, C., AND ZHANG, M. Linear genetic programming for multi-class object classification. In *AI 2005: Advances in Artificial Intelligence: Proceedings of the 18th Australian Joint Conference on Artificial Intelligence, Lecture Notes in Computer Science, Vol. 3809.* (Sydney, Australia, December 2005), S. Zhang and R. Jarvis, Eds., vol. 3809 of *LNAI*, Springer Verlag, pp. 369–379.

[9] FORSYTH, R. BEAGLE A Darwinian approach to pattern recognition. *Kybernetes 10* (1981), 159–166.

[10] FRIEDBERG, R. M. A learning machine: Part i. *IBM Journal of Research and Development 2, 1* (1958), 2–13.

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

[12] J., M. D. Strongly typed genetic programming, technical report 7866. Tech. rep., Bolt Beranek and Newman, Inc, 1993.

[13] KOTSIANTIS, S. B. Supervised machine learning: A review of classification techniques. *Informatica 31* (2007), 249–268.

[14] KOZA, J. Genetic programming: A paradigm for genetically breeding populations of computer programs to solve problems, 1990.

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

[16] KOZA, J. R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection (Complex Adaptive Systems)*. The MIT Press, December 1992.

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

[18] LU, H., AND YEN, G. Dynamic population size in multiobjective evolutionary algorithms. *CEC '02: Proceedings of the 2002 Congress on Evolutionary Computation, 2002 2* (2002), 1648–1653.

[19] NORDIN, P., FRANCONE, F., AND BANZHAF, W. Explicitly defined introns and destructive crossover in genetic programming. *Advances in genetic programming: volume 2* (1996), 111–134.

[20] O'REILLY, U.-M., AND OPPACHER, F. A comparative analysis of GP. In *Advances in Genetic Programming 2*, P. J. Angeline and K. E. Kinnear, Jr., Eds. MIT Press, Cambridge, MA, USA, 1996, ch. 2, pp. 23–44.

[21] POLI, R., AND MCPHEE, N. F. Exact gp schema theory for headless chicken crossover and subtree mutation. In *In Proceedings of the 2001 Congress on Evolutionary Computation CEC 2001, Seoul, Korea* (2000).

[22] QUINLAN, J. R. *C4.5: Programs for Machine Learning (Morgan Kaufmann Series in Machine Learning)*, 1 ed. Morgan Kaufmann, January 1993.

[23] ROSS, B. J., GUALTIERI, A. G., FUETEN, F., AND BUDKEWITSCH, P. Hyperspectral image analysis using genetic programming. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference* (San Francisco, CA, USA, 2002), Morgan Kaufmann Publishers Inc., pp. 1196–1203.

[24] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. Learning internal representations by error propagation. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition 1* (1986), 318–362.

[25] RUSSELL, S. J., AND NORVIG, P. *Artificial intelligence : a modern approach*, 2nd ed. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, N.J. ; [Great Britain], 2003. GBA3-9270 Stuart J. Russell and Peter Norvig ; contributing writers, John F. Canny ... [et al.]. Previous ed.: 1995. Includes bibliographical references and index.

[26] SMART, W., AND ZHANG, M. Using genetic programming for multiclass classification by simultaneously solving component binary classification problems.

[27] SMART, W. R. Genetic programming for multiclass object classification : a thesis submitted to the victoria university of wellington in fulfilment of the requirements for the degree of master of science in computer science /. Master's thesis, School of Mathematics and Computer Science, 2005.

[28] SONG, A., AND CIESIELSKI, V. Texture segmentation by genetic programming. *Evol. Comput. 16*, 4 (2008), 461–481.

[29] http://archive.ics.uci.edu/ml/datasets.html, October 2009.

[30] ZHANG, M., AND CIESIELSKI, V. Genetic programming for multiple class object detection. In *AI '99: Proceedings of the 12th Australian Joint Conference on Artificial Intelligence* (London, UK, 1999), Springer-Verlag, pp. 180–192.

[31] ZHANG, M., GAO, X., AND CAO, M. D. Experiments on brood size in GP with brood recombination crossover for object recognition. Tech. Rep. CS-TR-06-6, Computer Science, Victoria University of Wellington, New Zealand, 2006.

[32] ZHANG, M., AND SMART, W. Using gaussian distribution to construct fitness functions in genetic programming for multiclass object classification. *Pattern Recogn. Lett. 27*, 11 (2006), 1266–1274.

[33] ZHANG, M., AND SMART, W. D. Multiclass object classification using genetic programming. In *EvoWorkshops* (2004), G. R. Raidl, S. Cagnoni, J. Branke, D. Corne, R. Drechsler, Y. Jin, C. G. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, Eds., vol. 3005 of *Lecture Notes in Computer Science*, Springer, pp. 369–378.

Appendix A

JVUWLGP

The research described in this report was carried out using a package called JVUWLGP developed by Carlton Downey in 2009. This package is based heavily on the VUWLGP package developed by Christo Fogelberg in 2005. The architecture and usage of JVUWLGP is described in this appendix.

A.1 Introduction

VUWLGP is a LGP package written in ANSI STL C++. *Java VUWLGP (JVUWLGP)* is an LGP package developed in Java based off the architecture of VUWLGP. JVUWLGP is an extendable library for performing linear genetic programming of all types including regression, classification, multiclass classification etc. I have also written an extension of this library for multiclass classification using LGP.

A.1.1 Motivation

The VUWLGP package was written in an archaic version of C++ which resulted in hundreds of errors when it was compiled with a more recent compiler. I could potentially have modified the package to run with a modified compiler, however the nature of these errors and the complex way in which the package was written made writing my own implementation based off the package architecture in a more user friendly language a better option. By writing my own implementation in Java I have decreased program complexity, improved readability, improved error checking ability and made the task of extending the package easier. In addition it allowed me to gain an integral understanding of how the package works, which will be of enormous value when the time comes to extend the package. The package was previous written in C++ because it was historically much faster then Java. However with modern advances in Java compilers, I believe that there will not be a significant runtime discrepancy between a Java and a C++ implementation. With this disadvantage removed, a Java package is superior because it is easier to read, debug and extend.

A.1.2 Major changes from vuwlgp

In writing Jvuwlgp I have stuck as closely as possible to the original architecture of the vuwlgp package. However in many cases the difference between the two languages (C++ and Java) made it necessary to write parts of the program in a different way. The changes listed below were largely motivated by three factors:

In Java we no longer have to deal with memory management because of the Java garbage collector. This means a great deal of code which dealt with explicit memory management was not included in the Java translation because it became redundant.

There exists implicit code in C++ classes which must be translated into explicit code in Java because in Java, the implicit code does not exist.

There are several language features present in C++ which we do not have access to in Java. These include pointers, structs ands unions, function pointers, and multiple inheritance. Also some language features in C++ such as templates allow more freedom then their equivalent features such as generics in Java. In order to translate code written using any of these features into Java, several workarounds were required. In particular, I have made extensive use of the factory pattern.

A.1.3 Class Diagram

The following is a class diagram of the JVUWLGP package. Readers will note that most of the classes are abstract and must be extended by an implementation of the library.

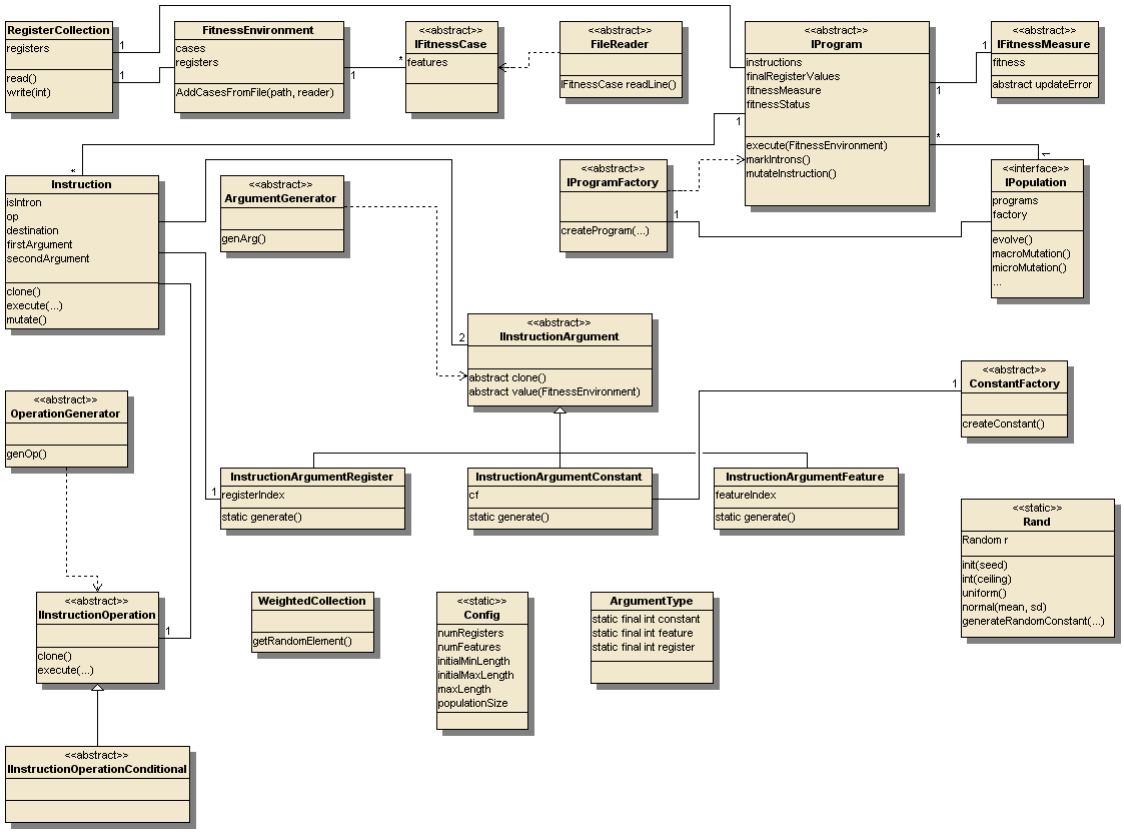


Figure A.1: Class Diagram for the JVUWLGP library

A.1.4 Description of the classes in Jvuwlgp

The following is a brief discription of each of the main classes in JVUWLGP and their purpose. This is aimed to help programmers wishing to use or extend the JVUWLGP library.

ArgumentGenerator: A factory class which exists solely to create instances of IInstructionArgument. Each time IInstructionArgument is subclassed, ArgumentGenerator

will also be subclassed.

ArgumentType: An Enumerated Type, but expressed using integers. Different integers represent different argument types.

Config: Holds the various user defined settings required for linear genetic programming. These include the population size, the maximum program length etc.

ConstantFactory: A factory class which exists solely to create instances of InstructionArgumentConstant.

FileReader: Deals with reading data in from a file and performing preprocessing on this input, before using it to create FitnessCase instances.

FitnessEnvironment: Stores the FitnessCase instances used to evaluate the fitness of an IProgram instance.

FitnessCase: A single labeled data instance which can be used to partially evaluate the fitness of an IProgram. Contains a list of features to be used in the evaluation.

IFtinessMeasure: A measure of a program?s fitness at some point in time. Instances of this class are used to cache the fitness of IProgram instances. This cacheing is done to improve library performance.

IInstructionArgument: An argument for an instruction. Arguments provide values to be used in the execution of an Instruction. Examples of arguments are registers, constants and features.

IInstructionOperation: An operation to be performed on Arguments to produce a value. Examples of operations include +, -, /, *, lt.

Instruction: An instruction in a program. LGP programs are comprised of a list of instructions to be executed in order. An instruction consists of a destination register, an operation to perform, and 2 arguments to perform the operation on. The result of performing the operation on the arguments is written to the destination register upon execution of the program.

IPopulation: This class and IProgram are the core of the JVUWLGP library. Instances of this class represent a population of individual programs. Contains a list of the programs in a population and methods for operating on the population of individuals as a whole. The primary loop for iterating through the generations is contained within this class in the evolve() method.

IProgram: Instances of this class represent individual programs in the population. Contains a list of the instructions to be executed in order, as well as methods which operate on individual programs. Execution of the instructions results in a change to the values held in the program registers. These values are the output of the program.

IProgramFactory: A factory class which exists solely to create instances of a IProgram. Users of this library must extend both the IProgram class, and the IProgramFactory class.

OperationGenerator: A factory class which exists solely to create instances of IInstructionOperation.

Rand: A utility class which contains methods for producing the sorts of random numbers required for this program.

RegisterCollection: Contains a list of registers and methods for operating on this list, such as adding registers, writing to registers, etc.

WeightedCollection: A utility class. Is very similar to a set but each element is given a weight when it is added to the collection. These weights allow random selection of elements from the collection proportional to the weight. Used for randomly selecting programs for reproduction according to fitness.