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

Computer Science

PO Box 600
Wellington
New Zealand

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

Linear Genetic Programming for Multi-class Classification Problems

Christopher Fogelberg

Supervisor: Dr. Mengjie Zhang

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

Abstract

Multi-class image classification is an important research topic within computer science and artificial intelligence. A wide range of images of different types need to be classified, and the techniques developed for multi-class image classification can often be applied to other forms of multi-class classification. Genetic programming has had some success with these problems in the past, however multi-class image classification is still a difficult task and the resulting classifiers are often very hard for humans to understand. This project develops a methodology using linear genetic programming for multi-class image classification. A new fitness function is developed to improve this methodology and the standard tree-based genetic programming methodology. Two heuristics are found to guide initial decisions on a linear genetic programming configuration and to aid in comparing tree-based genetic programming and linear genetic programming configurations. The resulting methodology is compared to the standard genetic programming approach to multi-class image classification. The methodology developed outperforms the standard genetic programming methodology on all six of the varying tasks and does so significantly on five of them. A hill climbing algorithm is also developed and this algorithm augments the powerful evolutionary beam search linear genetic programming uses.

Acknowledgments

As befits a report of this length, there are a number of individuals whom I'd like to thank. I'll list only some of them here. In no particular order:

- Dr. Mengjie Zhang, for always so patiently restraining my urges to solve all the worlds problems in this one report and for spending the time to teach me how to research.
- Graeme Fogelberg, Robert Owen Fillbridge Haylock, Joan Anne Jennifer Hare and Mary Adelstein — the four parents I have been blessed with. What little wisdom I have is due only to them.
- Emily May Rainsford. Despite my repeated use of the excuse of yet more work she never once raised her voice, not even at the very end.
- Hayden, Darren, Phillip, Ash and Chris — for noticing too many errors to count and providing a plethora of helpful comments and feedback on all areas of this report. It would be a lot worse without them.

Contents

Figures

List of Tables

Chapter 1

Introduction

Classification tasks (problems) are tasks which involve classifying some example as one of a number of classes. Each example is typically represented by a set of features (a *feature vector*). Finding an algorithm which can map a feature vector to a class requires the use of machine learning and search techniques. Multi-class image classification is a kind of classification task which involves processing an image and then assigning this image to one of the more-than-two possible classes.

The ability to perform such classification tasks programmatically is becoming increasingly important as greater and greater quantities of information is digitised and requires analysis. Human analysis of this information is often impractical: the information may be too voluminous or expert analysis may be overly expensive, and the classes may be too similar or the images too noisy for a human to perform well on the task. Some examples of multi-class classification tasks are:

- Optical character recognition of typed or handwritten digits and letters.
- Face classification (by personal identity or by some other characteristic, such as ethnicity).
- Analysis and classification of experimental data, such as the spectral patterns of astronomical phenomena.
- Object recognition, such as types of military aeroplane and vehicle.
- Quality evaluation of agricultural products, such as fruit: ″ripe“, ″too ripe“, ″unripe“.

Genetic programming (GP) [23] is a relatively recent artificial intelligence technique which has been used for a range of tasks, including classification problems. Binary classification tasks in which there are only two classes and some multi-class classification tasks have been well solved. For example, the detection and classification of the heads and tails of New Zealand 5 and 10 cent coins can be done with approximately 80-95% accuracy [47]. Similarly, recent research [49] has achieved very good results on a 5 class problem which is not noisy. However as the number of classes or the noise level is increased results are degraded. For problems in which the objects which need to be classified are noisy or very similar to members of another class the problem is particularly severe. These kinds of problems will be considered in this research.

1.1 Issues and Motivation

Many of the most interesting and important types of object images which need to be classified are noisy or very difficult. These include face recognition and the classification of noisy images of complex objects.

Genetic Programming (GP) [23, 1] is a promising approach for building reliable classification programs quickly and automatically, given only a set of examples on which a program can be evaluated. GP uses ideas analogous to biological evolution to search the space of possible programs to evolve a good program for a particular task. A strength of this approach is that evolved programs can be much more robust [23] and flexible than the highly constrained, parameterised models used in other techniques such as neural networks and support vector machines. GP has been applied to a range of image classification tasks [16, 11, 19, 39] with some success.

There are at least two limitations in the currently-used GP *program structures* and *fitness functions* that prevent GP from finding acceptable programs in a reasonable time.

The programs that GP evolves are typically tree-like structures [24], which map a vector of input values to a single real-valued output [50, 29, 52, 56]. For classification tasks, this output must be mapped into a set of class labels through the use of an *output interpretation algorithm*. For binary classification problems, there is a natural mapping of negative values to one class and positive values to the other class. For multi-class classification problems, finding the appropriate boundaries on the number line to separate the classes is very difficult. Several new translations have recently been developed in the interpretation of the single output value of the tree-based GP [29, 57, 58], with differing strengths in addressing different types of problem. While these translations have achieved better classification performance the evolved programs are hard to interpret, particularly for more difficult problems or problems with a large number of classes.

In solving classification problems, GP typically uses the classification accuracy, error rate or a similar measure as the fitness function [52, 57, 58]. The fitness function should approximate the true fitness of an individual program as accurately as possible. Given that the training set size is often highly limited, such a function frequently fails to accurately approximate a program's classification of the true feature space.

In addition, it is often very difficult to understand why a genetic program works (or does not) and the role played by various features and terminals is unclear. Likewise, despite the strength of the the evolutionary beam search as a searching algorithm, hardware constraints means it cannot guarantee to find the local maximum. The solution will generally be near some optimum but the stochastic search process means that it will not be guaranteed to be *at* the optimum. This means that a solution may frequently be improvable by some small margin.

To summarise, the problems faced by existing GP approaches to multi-class classification problems are as follows:

- The resulting programs often classify poorly and inconsistently from run-to-run.
- Existing traditional fitness functions often perform poorly on some types of image data set, leading to poorer results.
- It is difficult to understand how these programs work and which features are important to the classification of each class.
- Due to hardware constraints the evolutionary process can not guarantee local optimisation — any specific configuration for a problem can only ensure that it at least operates in a *ratchet* fashion [53] through the use of *elitism*.

1.2 Research Goals

To address the problems outlined in section 1.1 this research will investigate the use of linear genetic programming [1] for multi-class classification. This will achieve the following goals:

- Develop and analyse a linear genetic programming methodology for multi-class classification problems (chapter 4 and 8, respectively).
- Evaluate a new fitness function to address the *hurdle problem* (chapter 5).
- Ascertain the relationship between program length, accuracy and problem difficulty and the value of extra registers beyond those interpreted as part of the output (chapter 6)
- Create and apply an explicitly hill climbing algorithm to the methodology for multi-class classification developed in chapter 4 and improved in chapter 5. Evaluate this algorithm (chapter 7).

1.3 Contributions

This project has made the following major contributions:

1. This project shows how to use linear genetic programming (LGP) to construct genetic programs for multi-class classification problems. A new methodology for addressing multi-class classification tasks using LGP is introduced. The performance of this methodology is compared to the standard GP methodology using a heuristic developed which aids in creating comparable TGP and LGP configurations. The new methodology performs better on all data sets evaluated when compared to a standard GP methodology.

2. This project investigates the aspects of a classification task which make it more or less difficult. A new fitness function (the *decay curve*) has been developed to more accurately estimate the proportion of the feature space which is correctly classified. This fitness function minimises the impact of one of the aspects which can make a classification task difficult and significantly improves results.

3. This project shows how to determine the optimal maximum program length and number of registers in an LGP configuration, based on the number of classes in the classification task. The value of extra registers and the relationship between the length of an LGP program and the number of classes in a multi-class classification task has been investigated.

4. This project shows how to apply hill climbing techniques to improve performance in LGP systems. A hill climbing algorithm which fine tunes an LGP program has been developed. The value of this algorithm has been evaluated and the results show the expected improvements.

5. This research has been carried out using an LGP package which needed to be developed. This package is called VUWLGP and was written using ANSI Standard C++. It has been flexibly developed and can be used for non-classification tasks as well as for classification tasks.

6. A paper titled "Linear Genetic Programming for Multi-class Object Classification" has been accepted for presentation as a full paper at the 18th Australian Joint Conference on Artificial Intelligence and will be published in the *Lecture Notes in Artificial Intelligence*. A copy of this paper is provided in appendix B.

1.4 Structure

This chapter has briefly introduced the concepts of a classification task and genetic programming and the problems faced by existing GP approaches to classification tasks. The following chapters describe the research carried out to address these limitations.

Chapter 2 summarises existing and related research. Section 2.1 provides a more in-depth and general overview of search algorithms and multi-class classification. Types of multi-class classification are discussed in section 2.2. Genetic programming is introduced more fully in section 2.3. Existing methods of performance evaluation are discussed in section 2.4.

In chapter 3 the tasks used in this research are presented. The features extracted from each image are described and some general characteristics of the experimentation which are true of all experiments for any task are presented. The data sets for some tasks are sourced from the Yale Face Database B [14] and this database is described here.

The results of the experiments which meet the goals and answer the research questions outlined in section 1.2 are presented and discussed in chapters 4–8. Each of these chapters is readable largely independently of the others, provided the reader is somewhat familiar with genetic programming, evolutionary computation and multi-class classification. However readers are advised to scan chapter 4 first to orient their reading.

In chapter 9 the results presented and discussed in chapters 4–8 will be summarised and synthesised into a number of overall conclusions and recommendations.

Chapter 2

Background

This chapter introduces some of the basic concepts behind machine learning, focusing on techniques which can be applied to multi-class classification problems. Multi-class classification problems are then discussed. Following that a brief literature review of genetic programming and performance evaluation is presented. Note that details pertinent to only one of the research questions being addressed are covered in the chapter which presents the results of the experimentation for that question.

2.1 Machine Learning

2.1.1 What is Machine Learning and Why is it Necessary?

Machine learning is the process of automating the development of some part of a system which performs some task. The algorithm, parameters to an algorithm or process can be learnt adaptively over a period of time [46]. There are two types of situation where machine learning is necessary.

The first kind of situation is one in which a human cannot easily, quickly or reliably create a system (including the algorithm and its parameters) which performs some task. Most multi-class classification tasks, described in section 2.2 fall into this category. Note that there is a significant difference between a human's inability to do some task (trained experts can perform most tasks) and a human's inability to *create an algorithm* which can do the same. Such systems are needed because, while experts are normally able to perform these tasks, the cost and number of experts may be insufficient compared to the need for their ability.

In the second situation a human is perfectly able to create a system which performs rapidly, reliably and quickly on some task. However, because this algorithm does not change, if the situation changes it will start to generate inaccurate or bad results. These kinds of "Red Queen"-type [42] situations are very common, especially if humans are involved in some way in the process the system is being applied to. Detecting credit card fraud, for example, is the detection of a "moving target", as fraudsters learn new techniques and adapt old ones to circumvent changed protective measures.

2.1.2 Performing Machine Learning

The overall structure of a machine learning approach to a problem involves three steps [46]:

1. The generation of some representation of a solution to the problem.
2. The evaluation of the generated solution.

3. If the evaluated solution is not good enough, the solution is iterated (i.e. almost always improved) and the machine learning process goes to step (2).

I will now discuss each of these three steps in more detail.

2.1.3 Types of Generation and Representation

Given the selection of some representation of a solution to the problem, the initial generation is usually random but constrained by some parameters. For example, in a neural network the structure is fixed and the weight associated with each link is generated according to some algorithm which ensures that the initially generated solution will almost certainly not be the same from time-to-time. However, there are a wide variety of possible representations, including feed-forward neural networks, genetic algorithms, support vector machines, simulated annealing, decision trees, Bayesian networks and genetic programming. Excluding genetic programming I will now discuss each of these representations briefly. Genetic programming is discussed in more depth in section 2.3.

Feed-forward Neural Networks

Feed-forward neural networks [45] are based on a model of the biological brain. Inputs are passed to some input neurons. These input neurons generate some output which is passed along links (which will scale the output) to another layer of neurons. This process of applying some function to a neuron's input and then passing the value of this function on to the next layer, moderated by some weight, continues until the output layer is reached. The results of applying the processing function to the output layer's input is considered the output of the neural network. This output then needs to be interpreted. In multi-class classification there is usually one output neuron for each class and a winner-takes-all algorithm is used, with the class represented by the output neuron with the largest value being interpreted as the output of the neural network.

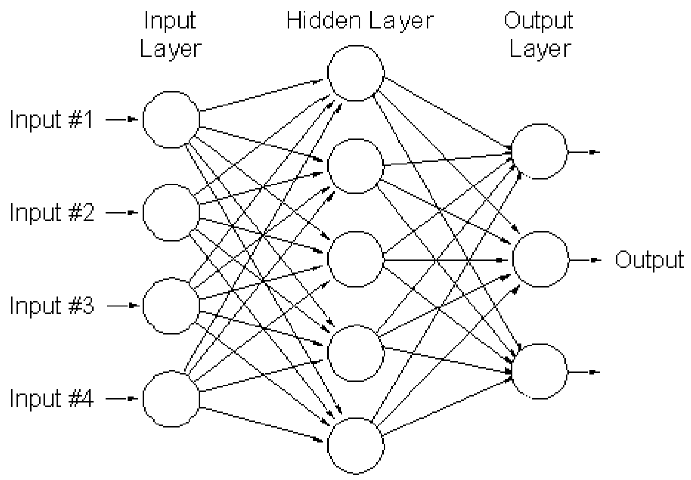


Figure 2.1: A feed-forward neural network [43].

Genetic Algorithms

Genetic algorithms (GA) [15] require a solution to be encoded into a fixed-length string of alleles (typically base-2 bits). A decoding algorithm, d is created by the programmer, and d can be applied to any genotype of the appropriate length and format to create the solution that genotype specifies. Creating d and the representational schema is complex and can make working with genetic algorithms difficult. This genotype-phenotype translation is unnecessary in genetic programming. For multi-class classification tasks, most GA ″phenotypes“ are systems which output one value [25, 28, 38] and the output interpretation algorithms discussed in the context of GP in section 2.3.6 can be used.

Support Vector Machines

Support vector machines (SVM) [8] can generate either binary classifications or regress a function from some set of training data. In the context of binary classifications, an SVM is used to find a hyperplane in the dimensional space which the features sketch out (the *feature space*, discussed in sections 5.2 and 5.3.1). This hyperplane separates the two classes into two regions so that the nearest example of each class is as far from the hyperplane as possible. Applying binary-class classification systems to multi-class classification problems is difficult and can only be done by training multiple binary-class classifiers for one multi-class classification problem.

Simulated Annealing

Simulated annealing [30] is a generalised Monte Carlo method. Annealing is a process applied to metals and involves heating them to a very high temperature and then slowly cooling them so that the system remains in thermodynamic equilibrium. A system which is in thermodynamic equilibrium is in the same state, i.e. at the same temperature, in all parts. Cooling the metal sufficiently slowly ensures that the metal attains and retains the most ordered (and therefore strongest) state possible. Computationally, this is modelled by generating an initial solution randomly. Random changes are then made to this solution and automatically accepted if they improve the solution or accepted with some probability p if they worsen it. As the ″temperature“ is decreased p is decreased as well.

Decision Trees and Bayesian Networks

Decision trees and Bayesian networks [46] work with discretely valued inputs. Continuous inputs (height, temperature, size, brightness, etc.) need to be made discrete prior to them being usable in a decision tree or Bayesian network. As such decision trees and Bayesian networks are ideally suited to some kinds of multi-class classification task, but it is very difficult to use them for tasks involving image processing and classification.

A decision tree is evaluated by beginning at the root [46]. Each internal node of the tree has some condition and a node has one child per possible answer to this question. Depending on the answers to these questions the tree is traversed until a leaf is reached. The leaves of the tree are either discrete answers (such as a class) or a probability distribution across two or more possible answers.

A Bayesian network is a directed acyclic graph [46]. Each node in this graph represents some variable which can have a finite number of discrete values. The causal relations between nodes are shown by arcs. Given some evidence regarding the state of some nodes and through the use of Bayes Law and algorithms which allow evidence to be propagated the updated probabilities of other nodes in the Bayesian network can be determined. When using a Bayesian network for multi-class classification one node will represent all of the possible classes and other nodes will represent features whose values can be known and intermediate levels of causal linking.

2.1.4 Types of Iteration

There are, broadly, three ways in which the iteration from one solution to the next can be performed. This iteration is how the search for a good solution is carried out. Each of the three types of iteration will be summarised briefly and one way of carrying out two of these types will then be discussed in more detail.

unsupervised Unsupervised learning is normally used to locate patterns in the input data. No information is given to the system which finds the patterns as to the correctness or incorrectness of the patterns. The patterns it finds may therefore be arbitrary or they may actually be representative of some real underlying process which caused them to appear. See [9] for more detail.

reinforcement In terms of the quantity of information given to the systen regarding the correctness of its output, reinforcement learning [46] is intermediary between supervised and unsupervised learning. When reinforcement learning is used some information is given to the system at some time regarding the correctness of a prediction it made. This information ranges in precision from a categorisation of a response as ″right“ or ″wrong“ to a precise amount of error, expressed numerically. At the latter degree of precision it differs from supervised learning only in the way the information is presented.

supervised When supervised learning is used the precise, correct output which should have been given for any particular training input is known to the system and used by the system to adjust the answer it will give to other training examples [46].

Hill Climbing and Gradient Descent

A form of supervised or reinforcement learning which is frequently used is *hill climbing*. Hill climbing is a form of iteration in machine learning which aims to improve a candidate solution by advancing it some amount in the direction which (at least in the very short term) leads towards a better solution. Hill climbing can either be done *explicitly*, when the direction of the best step is determined, or *implicitly*, through the impact of stochastic learning processes (as in simulated annealing). All of the representations discussed above are a form of either implicit or explicit hill climbing. In the remainder of this section explicit hill climbing will be discussed more fully.

By conceptualising the current solution as having some position in some space (given different names in different contexts but typically with its dimensional structure given by the parameters to the solution) hill climbing can be performed. This space will have *maxima* and *minima* (or *peaks* and *hollows*) which reflect the quality of a solution at this position in the space. For the purpose of clear discussion in this report the peaks are assumed to be better, although the opposite assumption is also frequently made use of when it makes more sense to talk about the objective of an AI application as the minimisation of some cost. In an iteration involving a hill climbing algorithm the direction in this space towards a better solution is at least roughly calculated. The need to calculate this direction is why hill climbing cannot be carried out in an unsupervised fashion. The candidate solution is then moved some distance towards this peak. The distance moved is normally determined by some function which estimates how beneficial a move in this direction will be. If it the estimate indicates it will be very dramatic improvement a larger move is normally safer. This movement can also be scaled by some value and it can be adjusted by past movements (i.e. if momentum is modeled). Modeling momentum helps a hill climbing algorithm avoid becoming trapped at the smaller local peaks, although it also increases the chance that a candidate solution will oscillate about the global peak. Whether or not momentum is modeled, one weakness of hill climbing algorithms is that it can over-step a peak if the function which estimates how far it should move over-estimates. This sort of situation can be seen in figure 2.2. In addition it is important to remember that hill climbing algorithms are not guaranteed to find the global optimum, and as the space in which they climb becomes craggier they are more and more likely to become trapped at a sub-optimal, local peak.

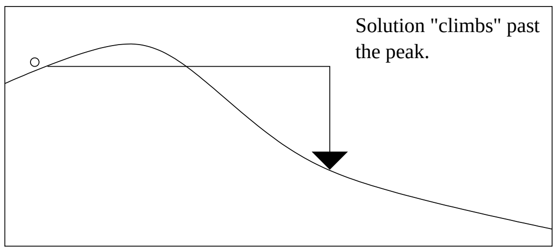


Figure 2.2: How too-large an adjustment when hill climbing can worsen a solution.

Gradient descent is a specific kind of of hill climbing which relies on a continuous cost function. As a result of this requirement the direction and extent to which a solution can be improved can be calculated much more precisely. However this precision does not protect gradient descent algorithms from the problems of over-fitting or oscillation about some peak in the landscape which are also present in hill climbing algorithms. The same problems faced by hill climbing algorithms are also necessarily a problem for gradient descent algorithms. In addition, due to the greater accuracy with which gradient descent algorithms improve a solution in the apparent direction of the global optima they are at least as vulnerable to being trapped at local optima.

2.1.5 Types of Evaluation

The types and ways in which evaluation can be carried out is discussed in section 2.4.

2.2 Multi-class Classification Problems

Only a very brief further discussion of multi-class classification tasks will be given in this section. A reasonably full discussion of multi-class classification was given in chapter 1 and the aspects of a multi-class classification task that make it hard are discussed in chapter 5 and especially in section 5.4. In this section, the value of this research will be highlighted through a more in-depth discussion of the types and kinds of multi-class classification tasks which exist.

Image classification tasks are one kind of classification task which are frequently a multi-class classification task. Possible image classification tasks include the classification of medical imagery (x-rays, MRI scans or digital photos of cells) to aid in the diagnosis of a variety of injuries and diseases. Similarly, analysis of maps to determine the proportion of various covers (asphalt, trees, soil, roofing) may be necessary. Classification of certain kinds of objects in images which are primarily ″background″ is also often necessary. An example of this kind of problem is the classification of ships in synthetic aperture radar imagery [18]. Optical character recognition of digits and letters, either handwritten or typed, represent a third important category of multi-class image classification. A fourth important kind of image classification task is the recognition of a number of possible faces (perhaps people who need to be identified and monitored for security reasons) from a crowd. Such a situation may arise at an airport where cameras may programmatically scan faces, checking for the presence of wanted criminals.

The classification of medical and biological data represents another important kind of multi-class classification task. An example of this kind is the classification of cells as either non-cancerous or as one of several types of cancer through examination of the gene expression profile [34]. Another example is the classification of the structure of protein folds in various proteins. Understanding the ways in which proteins fold is important, because although it "isn't likely to create a revolution the way Watson and Crick's discovery of the structure of DNA transformed biology, or that Einstein's theory of relativity transformed physics. [...] it will provide a profound new insight into life's basic units, and the evolutionary process that produced them" [7].

A third important type of multi-class classification problem is the classification of text documents. This is done to, for example, improve the semantic accuracy of document searches by reducing the number of false-positives. [40] and [41] provide a good summary of the kinds of ways in which text documents can be classified and the features that are used.

Another kind of problem closely related to multi-class classification problems are problems which involve multiple, inter-dependent outputs. This kind of problem has not been considered in this research, although we hypothesise the methodologies developed will be easily applicable to it and we expect them to have a similar degree of success. Investment decisions represent an example of this kind of problem [27, 6]. The risk tolerance of any individual investment depends both on the decision made regarding its absolute size and also its size relative to the total portfolio. A decision regarding the degree to which risk (negative events) will be tolerated should not be made without also considering the absolute size of the investment. Similarly, the absolute size of the investment should not be decided upon without considering the risk which will be tolerated. Generating answers to these two questions should be done in an interdependent manner, as this manner will probably be both more efficient and almost certainly more accurate (if it were not then the decisions are not truly interdependent).

2.3 Genetic Programming

Genetic Programming (GP) [23, 1] is a promising approach for building reliable classification programs quickly and automatically, given only a set of examples on which a program can evaluate. In this introductory section I will first briefly explain how genetic programming works and the important elements of a genetic programming configuration for a particular problem. I will then explain the nature of traditional tree-based genetic programming (TGP) [23].

GP uses ideas analogous to biological evolution to search the space of possible programs to evolve a good program for a particular task. A large number of random programs are generated using one of the methods outlined in section 2.3.2. Each program is then evaluated against each *fitness case* in the training set $S_{training}$. Each program will have some *fitness* as a result of performing to a certain extent on the fitness cases in the training set, which is assumed to be representative. Based on this level of performance, individuals are selected for *evolutionary operations*, which create offspring who are evaluated in the next generation. After some maximum number of generations or when some level of fitness, ϵ, is reached evolution is terminated. This creates a beam search on a particular problem that is resistant to local minima and often finds very good solutions quickly.

2.3.1 Program Representation

In TGP each program is a function tree, as shown by the example in figure 2.3. The leaves are members of a fitness case's feature vector (every fitness case is required to take the same format) or ephemeral random constants (R), introduced by Koza [23]. Internal nodes represent functions whose semantic meaning is defined as part of the configuration. `if<0`, for example, is a function usually defined as having the value of its second argument if its first is less than zero, and of having the value of its third argument otherwise. When a TGP program is evaluated the leaves which are not random constants are replaced by the values of the appropriate member of the feature vector and the values are calculated in a recursive manner. Each tree produces one output. This is almost always a real-numbered value, although the evolutionary method does not require this and research into *strongly-typed genetic programming* (STGP) [32] illustrates other possibilities. This output is interpreted for each fitness case and some fitness level is calculated. Techniques for achieving this in multi-class classification tasks and the problems they have are discussed in sections 2.3.6 and 4.2.

```
(* (- (+ (/ f1 -0.268213)
         (/ -0.828695 f6))
      (/ (/ f7 f6)
         (+ -0.828695 f5)))
   (* (- (if<0 f3 f1 f5)
         (/ f5 f6))
      (+ (- f4 -0.828695)
         (+ f1 f2)))
)
```

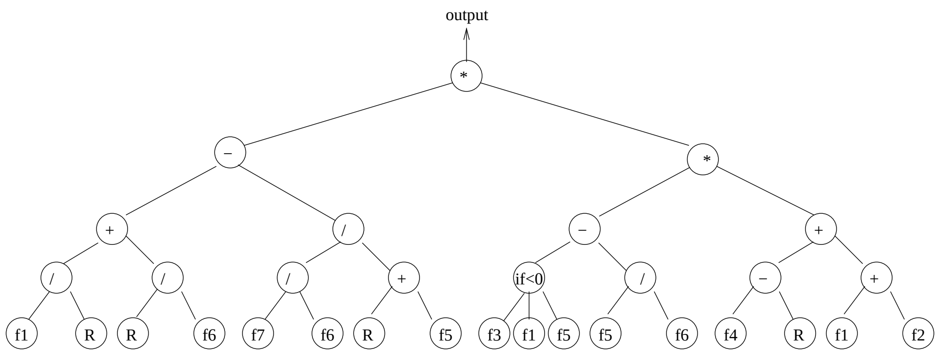


Figure 2.3: A sample program for a 10-class problem evolved by TGP. Different values of R (ephemeral random constants) are not shown due to space constraints.

2.3.2 Program Generation

Programs are initially generated in one of three ways. They can be *grown*, where a random function or terminal is selected as the root of the tree and then a random terminal or function is selected for each child of that node. This process is repeated recursively until the maximum depth is reached where a terminal is randomly selected from the set of terminals. Alternatively trees can be *full*. When a tree is built to depth n functions are randomly selected at the root and for each child until nodes at depth n are being selected, when terminals from the terminal set are randomly selected. The third approach is to generate half of the programs using the full method and half using the growth method. This is known as *ramped half-and-half* and was used by Koza [23]. Ramped half-and-half has been used in all TGP configurations in this research.

2.3.3 Selection Algorithms

Selection of individuals to participate in evolutionary operations is typically achieved through one of two approaches. *Proportional selection* [23] involves assigning to each individual program p_i some fitness less than 1 and proportional to the sum fitness of all programs in the population, as shown in equation 2.1. Each parent individual in an evolutionary operation is selected in a roulette-wheel fashion with probability $f_{proportional}(p_i)$.

$$f_{proportional}(p_i) = f_{raw}(p_i) / \sum_{p} f_{raw}(p) \tag{2.1}$$

Alternatively, *tournament selection* is often used. The tournament size is specified in the genetic programming configuration (the typical size used is in the range 4–6) and this many individuals are selected in a uniformly random manner from the population. The fittest individual in this group is then chosen for the operation. If more than one individual is needed for some operation then multiple independent tournaments with replacement are usually conducted. Size-4 tournament selection has been used in this research.

2.3.4 Evolutionary Operators

Individual programs are normally selected for one of three evolutionary operations. Evolutionary operations can either be assigned (10% of all operations will be X, where X is some operator) or probabilistically selected (there is a 10% chance of this operation being X, where X is some operator). In this research the former method was used.

Reproduction

Reproduction involves copying, unchanged, the selected program to the next generation. This operation is usually implemented as *elitism*. Elitism automatically selects the fittest k% (e.g. 10%) of the population and copies them unchanged to the next generation. This is a minimal form of local optimisation which ensures the evolution operates in a ratchet fashion [53]. A ratchet is a kind of evolutionary situation in which a population can only improve or stay the same from generation-to-generation, it cannot get worse.

Mutation

Individuals selected for a *mutation* operation have some randomly selected sub-tree replaced with a new randomly generated tree but are otherwise copied unchanged into the next generation. The use of this evolutionary operation helps ensure that a certain degree of diversity exists in the population, helping prevent any selection pressure for local optimisation from limiting the quality of the solution and (consequentially) causing poor results [23].

Crossover

Crossover is an evolutionary operation which involves two individuals. Two parents are selected and a sub-tree of each parent is uniformly randomly selected[1]. The sub-trees are then swapped from one parent to the other and the changed first parent is copied into the next generation. If desired, the changed second parent can also be copied into the next generation (this reduces the number of evolutionary operations which need to be performed). An example TGP crossover is shown in figure 2.4. If one of the resulting trees is either too deep or too shallow the typical response is simply to instead copy the first parent, unchanged, into the next generation.

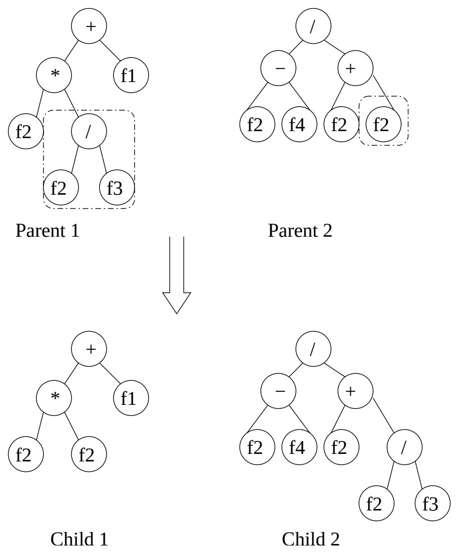


Figure 2.4: An example of a TGP crossover. Highlighted sub-trees are swapped from one parent to the other to produce the offspring.

2.3.5 Genetic Programming and Hill Climbing/Gradient Descent

Tree-based GP (TGP) and hill climbing algorithms have been combined in two ways in past research.

One approach is to adjust the *ephemeral random constants* [23] which served as numeric terminals in order to slightly adjust a program's output (developed in 2003, described in [48]) so that its classification accuracy was improved. The amount by which the numeric terminals are adjusted is scaled by a parameter, η.

On several tasks of mid-to-easy difficulty this algorithm achieved improvements in both learning time and accuracy. Unfortunately no consistent value of η was found to give the best results, although all values of η lead to more accurate results when compared to the basic TGP configuration.

A second approach was developed by William Smart in 2004 and described in [60]. In this approach the basic TGP representation is modified. Individual programs are still expressed as trees but each link between nodes in such a tree has a weight associated with it. This entails some changes to the evolutionary operations used. In addition, when evaluating a program the value of a sub-tree is multiplied by the weight on the link which is the last link on the path leading from the root of the tree to that node. This means that if a sub-tree consistently causes a program to generate an output that is either too large or too

[1]i.e. each node has the same chance of being selected as the root of the sub-tree

low its weight will be adjusted so that the program's output is closer to the ideal output. Initial weights for each link are 1.0. This means that each tree is initially identical to a normal TGP program with no hill climbing. The results of this research, presented in [48] and [60], were very similar to those obtained using the first algorithm and show that the previous algorithm may be a special case of this seemingly quite-different algorithm [48].

For both algorithms the learning can be carried out in either an online or offline manner. If online learning is used the weights or numeric terminals are updated after evaluating each fitness case in the training set. On the other hand, in offline learning, all fitness cases in the training set are presented and a program's error is summed across these fitness cases and the hill climbing algorithm is applied to the summed error. How error can be summed is discussed briefly in section 2.4.

2.3.6 Genetic Programming for Multi-class Classification

Although some brief work [4] comparing the performance of linear genetic programming in multi-class classification problems to feed-forward neural networks has been done this work was only carried out on either binary or quite easy 3-class classification tasks. In addition, the methodology used only one register when interpreting the output of the program. In most respects this research is therefore very similar to existing TGP research into multi-class classification.

In this section I will describe a number of approaches used to apply TGP (in which programs produce only one output) to multi-class classification. The first three all present different ways of partitioning the number line and thus different ways of interpreting the output of a TGP program. The fourth modifies the structure of a TGP program. A fifth approach, used in [25], is to develop $n - 1$ programs to classify each of the classes in an n class problem. This method is computationally very expensive and makes it very difficult to take interdependencies between classes into account. Because of these weaknesses it is not discussed in detail in this literature survey.

Static Class Boundary Determination

As shown in figure 2.5, when using *static class boundary determination* predefined boundaries are established on the number line on which the output of a TGP program is interpreted [29]. The class of a particular fitness case or example is determined by which region on the number line the output falls into.

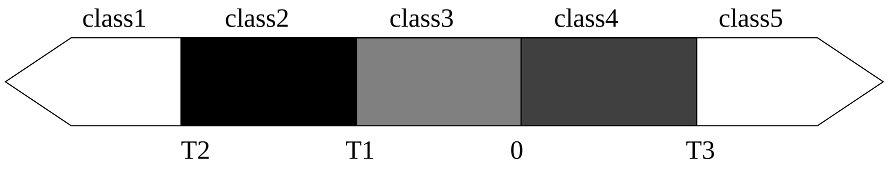


Figure 2.5: Static class boundary determination. This images shows how the numberline is divided up into regions, one per class, for a five-class problem.

Slotted Dynamic Class Boundary Determination

In *slotted dynamic class boundary determination* [59] the real number line is divided into 200 *slots*. A slot is assigned to represent whichever class makes up the largest proportion of examples which fall into it. During the evolutionary process the class a slot represents may change. Because of this mutual adaptation the output interpretation algorithm and programs could be seen to be involved in a form of co-evolution.

Centred Dynamic Class Boundary Determination

When using the *centred dynamic class boundary determination* methodology [59] to interpret the output of a tree-based GP (TGP) program the boundaries are determined dynamically. The arithmetic mean of all results for each class is calculated, giving the *centre* of that class. The boundaries of this class's region is then midway between the centre of this class and the next class in each direction on the number line. If there is no next class in one of the directions on the number line then this class's region is assumed to extend to infinity. As with slotted dynamic class boundary determination, a co-evolutionary description of this algorithm could be given.

Modi-GP

Static class boundary determination, slotted dynamic class boundary determination, and centred dynamic class boundary determination all work to apply TGP to multi-class classification problems by providing schema which change the way a TGP program's output is interpreted. Modi-GP is a different approach, developed by Yun Zhang [61] in 2004, which instead changes the program structure to make it more suitable for multi-class classification.

Modi-GP is an attempt to more accurately replicate the many-to-many structure which can be found in feed forward neural networks [45]. A Modi-GP program is still structurally equivalent to a normal TGP program but the program structure has been augmented so that it simulates a subset of the possible directed acyclic graphs (DAGs) a multiple-roots program tree could represent.

The program structure is modified by associating an output vector (with one element for each class) with each program. This output vector is virtual and does not really exist [61]. Each function node in a program tree has a certain probability of being linked to one of the elements of the output vector (the element it is linked to, if any, is selected in a uniformly random manner). Such a node is referred to as a *modi node*. See figure 2.6 for an example. Prior to evaluating a Modi-GP program the output vector is zeroed. The program tree is then evaluated in the standard manner except that when a modi node is encountered the value of the operation is added to the existing value at the element of the output vector it links to. The value of its right child is then passed to its parent as the value of this function. As a result, when a Modi-GP program has been executed one floating point value is produced for each class. A winner-takes-all algorithm is used to interpret the program's output and the class represented by the element of the vector with the largest value is interpreted as the program's output.

This method does lead to improved results across a range of data sets. In addition, some have claimed [61] that this method might make understanding why and how a TGP program works slightly easier. Possible future work involving this method may include the use of *Skinnerian creatures* - that is, the addition of a translation step from genotype to phenotype, rather than the interpretation just of the genotype. Similar steps remain to be considered for LGP.

2.4 Performance Evaluation

A number of ways of measuring performance (fitness) in multi-class classifications have been developed, although the approach used to develop these methodologies has often been

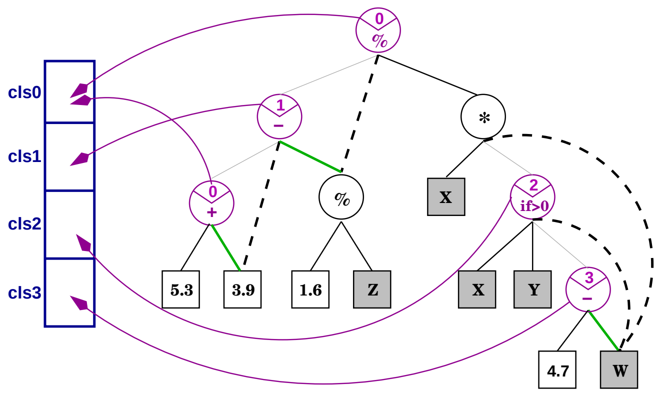


Figure 2.6: A Modi-GP program. This figure shows an example of a Modi-GP program [61].

pre-theoretical. A more in-depth and theoretical analysis of the function of the fitness function is given in chapter 5. In this section we will consider a number of previously used evaluation methods which will be referred to in this research as the *traditional fitness functions*. Note that a number of these are identical to each other when localisation is not being considered.

Each of these evaluation measures can also be combined in different ways when the results from several cases need to be considered simultaneously. *Total sum squared error* (or *TSS*) involves summing the square of the error on each case to give the total error across all of the cases. *Mean sum squared error* is the arithmetic mean of TSS. The *root mean squared error* for a problem with n examples in m classes is calculated as described in equation 2.2.

$$RMSE = \sqrt{\frac{2 \times TSS}{n \times m}} \tag{2.2}$$

Accuracy

Accuracy is a measurement of the percentage of classifications which are correct. A higher accuracy is better. The formula for the calculation of a system's accuracy is as follows:

$$Accuracy = \frac{N_{correctly\ classified}}{N_{total}} \times 100\% \tag{2.3}$$

The accuracy can be calculated on a per-class basis or across all classes being considered. In the latter case it can be a simple or weighted average of the accuracy of the solution for each of the classes.

Error Rate

The error rate is very similar to the accuracy, except that a lower error rate is fitter than a higher one. The formula for the error rate is as follows:

$$Error\ Rate = 100\% - Accuracy = \frac{N_{total} - N_{correctly\ classified}}{N_{total}} \times 100\% \tag{2.4}$$

True Positive Fraction

The *true positive fraction* (TPF) is a measurement of fitness which can be calculated on a per-class basis and then combined to give an overall measurement of fitness in the same way the accuracy or error rate can be combined. How the TPF is calculated for some class c of size $|c|$ is described in equation 2.5.

$$TPF_c = \frac{N_{correctly\ classified\ as\ c}}{|c|} \tag{2.5}$$

False Positive Fraction

Given some class c, the *false positive fraction* (FPF) is the fraction of examples which are not members of c, but which are mistakenly classified as members of c. More formally, how the FPF is calculated for some class c of size $|c|$ is described in equation 2.6.

$$FPF_c = \frac{N_{incorrectly\ classified\ as\ c}}{N_{total} - |c|} \tag{2.6}$$

Detection Rate

In multi-class classification object problems which do not include localisation, the *detection rate* is fundamentally identical to the accuracy. Hence this performance measure will not be discussed in this background.

False Alarm Rate

In multi-class classification, as with the detection rate, when localisation is not being considered (and any false positive is therefore also a false negative of another class) the *false alarm rate* (FAR) is identical to the error rate. Aside from noting several points that the FAR highlights it will not be discussed in detail here. Firstly, it should be noted that the FAR provides a more discriminative measure of the extent to which a system is ”over-eager“ when localising. This discriminative capacity is taken advantage of in the extended ROC curve [54]. In addition, note that when localisation is being considered seeking to improve the detection rate also leads to an increase in the false alarm rate. This factor is analysed in more depth in [31]. Finally, an extension of the false alarm rate developed to give better results especially for localisation is the *false alarm area*. See [26] for more detail on this method of performance evaluation.

Chapter 3

Data Sets

A range of data sets, of varying difficulty and difficult in different ways, were used in this research on linear genetic programming in multi-class image classification tasks. Each of the three types of data set used is outlined below and the general characteristics of the type of data set are described as well. The data sets have been presented in a rough ascending order of difficulty but the different ways they are difficult makes a strict ordering difficult or impossible and hence this has not been attempted.

3.1 Shape

Only one data set of type *shape* was used. For convenience it has been given and is referred to by the same name, *shape*.

3.1.1 Description

This data set was defined to give well-defined objects against a relatively clean background. The pixels of the objects were produced using a Gaussian generator with different means and variances for each class and the background. Four classes of 600 small objects (150 of each class) make up the training set. 600 more such objects make up the test set. The four classes are:

- Dark circles — *DC* or *class1*.
- Light circles — *LC* or *class2*.
- Dark squares — *DS* or *class3*.
- Light squares — *LS* or *class4*.

The mean and standard deviation of the brightness of each part of each image (with white being 255 and black being 0) is specified in table 3.1.

Example members of each class are presented in figure 3.1. Note that class 1 and class 3, and class 2 and class 4 objects are very similar in their overall mean brightness. This makes the problem reasonably difficult. The features used have been selected in order to maximise this difficulty and exacerbate the *hurdle problem* (discussed in chapter 7) as much as is reasonable, in order to provide a good data set for experimentation regarding ways in which the hurdle problem can be addressed.

Table 3.1: Mean (μ) and standard deviation (σ) of class pixels in *shape*.

Class	Component	μ Brightness	σ Brightness
	Background	140	50
DC	Circle	20	160
	Background	140	50
LC	Circle	180	160
	Background	140	50
DS	Square	60	160
	Background	140	50
LS	Square	220	160

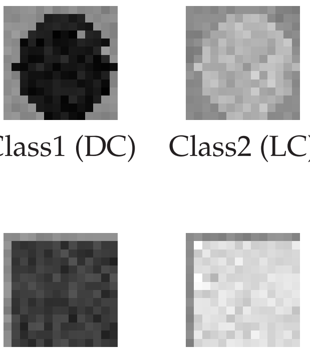
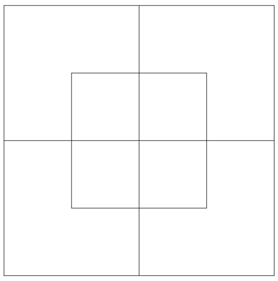


Figure 3.1: Examples of each class in *shape*.

3.1.2 Features

Eight features extracted from the *shape* data set and ephemeral random constants (R) [23] were used as the terminal set. The eight features are shown in figure 3.2. The LGP methodology is described in chapter 4 but preluding that note that feature n is mapped to feature register `cf`$[n-1]$.



Feature	LGP Index	Description
f1	`cf[0]`	mean brightness of the entire object
f2	`cf[1]`	mean of the top left quadrant
f3	`cf[2]`	mean of the top right quadrant
f4	`cf[3]`	mean of the bottom left quadrant
f5	`cf[4]`	mean of the bottom right quadrant
f6	`cf[5]`	mean of the centre quadrant
f7	`cf[6]`	standard deviation of the whole object
f8	`cf[7]`	standard deviation of centre quadrant

Figure 3.2: Terminal set for the *shape* data set. The location of the feature positions described on the right are shown in the diagram on the left.

3.2 Digits

3.2.1 Digits15

Description

Each digits image is a 7×7 binary image of one of the ten digits 0–9. Each image has 15% of its pixels flipped due to noise. See figure 3.3(a) for an example member of each class in *digits15*. Notice that human eyes cannot distinguish the majority of the patterns, particularly 6, 8 and 3, even 1 and 5. The problem is made even more difficult by the high number of classes.

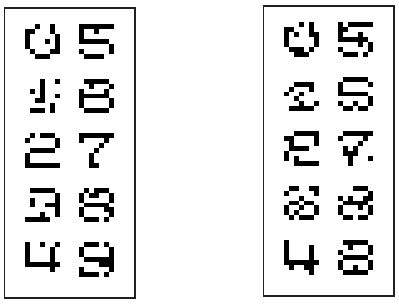


Figure 3.3: Digits data (a) *digits15*; (b) *digits30*.

Features

For simplicities sake and to ensure the problem retained a high level of difficulty the 49 raw pixels were used as the features. In addition, ephemeral random constants [23], R, were available. This resulted in the creation of a very large *feature space* (see section 5.3.1) with a high degree of noise. This created a problem in which it was particularly difficult to improve beyond a certain base level of fitness.

3.2.2 Digits30

This data set is identical to the *digits15* data set, except that each image has 30% of the pixels flipped due to noise. The same features were used for *digits30* as for *digits15*. See figure 3.3(b) for an example member of each class in *digits30*. Notice that human eyes cannot distinguish the majority of the patterns, especially 8 and 9, 3, 5 and 6, and even 1, 2 and 0. The problem is made even more difficult (as with *digits15*) by the high number of classes.

3.3 Faces

Data sets of the *faces* type investigated the classification of images of faces. Each image of a face was taken in different lighting conditions and from the same pose as all other images of that face.

3.3.1 Yale Face Database B

All faces data sets were based on the Yale Face Database B [14]. This database (hereafter just referred to as the Yale DB) consists of 5760 images of 10 faces in 576 different viewing conditions, namely all combinations of one of 9 poses and one of 64 lighting levels. In addition there was one photograph taken with ambient lighting of each face (not of each pose). Of these, only those photos taken from the directly-onwards angle (which include the ambient photos) were used in this project and there were thus 65 members of each of the ten classes. What differentiated members of one class from another were the different faces in each image. What differentiated one member in a class from another in that class was the angle and brightness of the lighting at the time the photo was taken. 3 examples from 3 classes (nine examples in all) have been provided in figure 3.4. These example images highlight the difficulty of this data set, due to the inherent similarity some members of each class have to members of *other* classes, rather than to members of their own. Data sets based on the Yale DB are also made more difficult by the fact that only 65 (often very different) members of each class exist and that the test and training set must be made up of independent subsets of these 65. Typically a training or test set consists of several hundred objects (depending on the difficulty of the problem). The reader is referred to [13] for more information on the Yale DB.

3.3.2 Faces1

Description

Five randomly selected classes from the Yale DB were chosen for the *faces1* data set, including the first and second in figure 3.4. All 65 members of each class were used. Ten-fold cross-validation was used in any experiment which used this data set, as the number of possible training and test examples was very low. This data set was the easiest *faces* dataset and is referred to as *faces1* for convenience.

Features

A 330 $\times$ 432 cutout of each face was used and 11 pixel statistics extracted from this were used in experiments involving *faces1*. These features are deliberately not optimal, as it was important that the problem remain difficult. Through the use of pixel statistics situations are deliberately created in which (in a Euclidean space) members of one class are nearer to members of other classes than they are to members of their own class. Despite this the features selected were more than sufficient to allow members of one class to be distinguished from members of another by focussing on the distinguishing areas of the face (the eyes and nose) and the general skin colour in different regions. Ephemeral random constants (R) [23] were also used in the terminal set. The 11 features are shown in figure 3.5.

3.3.3 Faces2

Description

faces2 uses each of the ten classes in the Yale DB (see section 3.3.1) but uses only the 40 most similar members of each class in the training and test sets. While this does reduce the maximum possible size of the test and training set it also significantly reduces the difficulty of the problem by removing those members of the classes most similar to members of another class rather than to members of their class. Hence, for example, example (a) of each class in figure 3.4 was not used in *faces2*. As with *faces1*, ten-fold cross-validation was used in all experiments involving this data set.

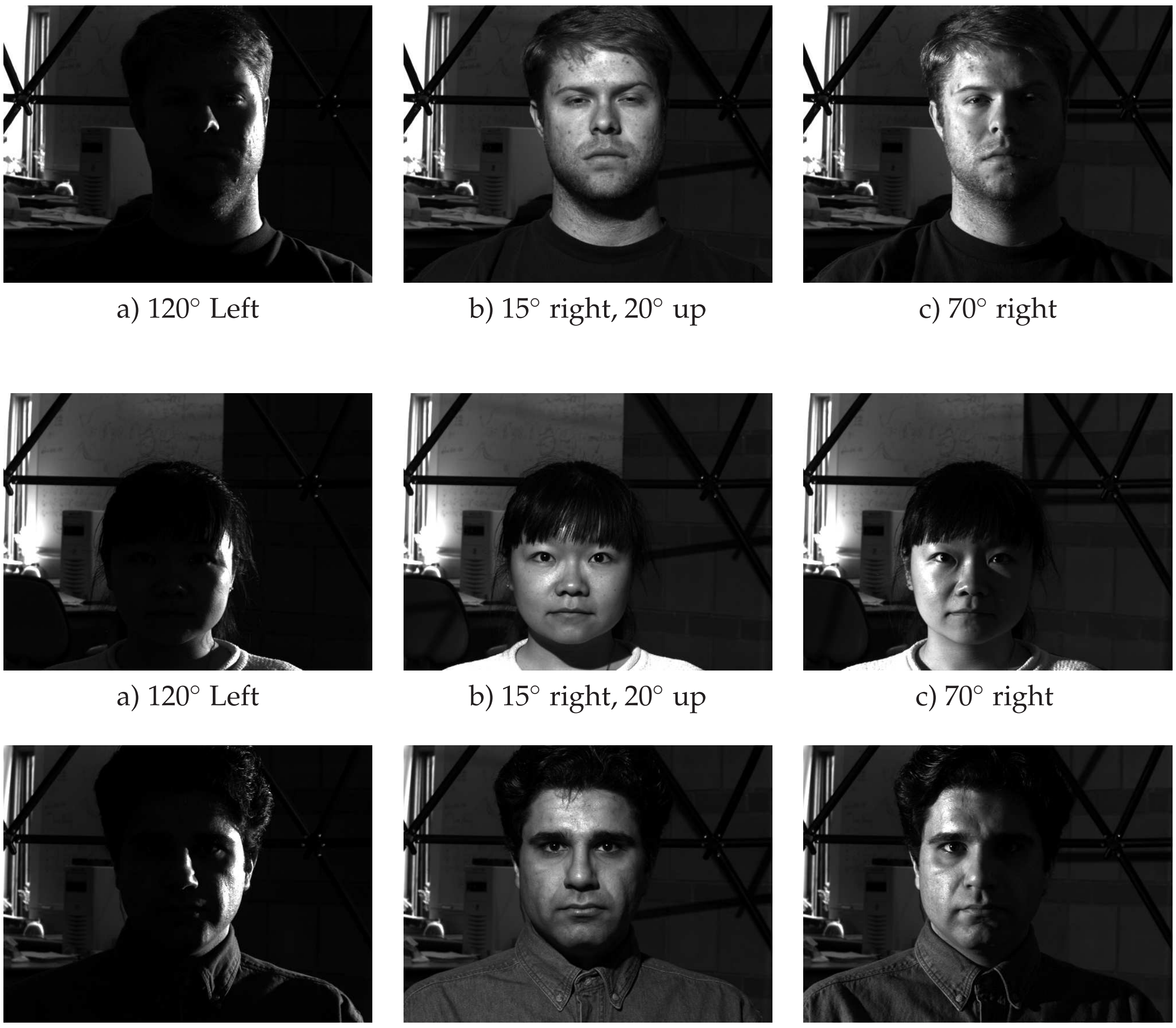


Figure 3.4: 9 examples from 3 classes in the Yale DB. Class 1 examples are given in the first row, class 5 examples in the second, class 8 examples in the third. Captions below each image describe the angle the lighting was at when the photo was taken. Vertical triplets highlight how similar the pixel statistics of different classes are under certain lighting conditions. The individuals in each class were not named.

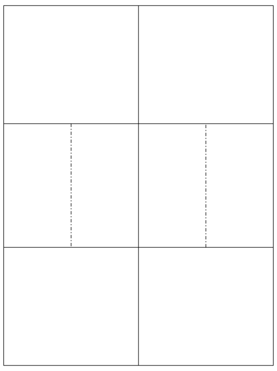
Features

The features used in the *faces2* data set are identical to those used in *faces1* (see section 3.3.2).

3.3.4 Faces3

Description

This data set is the most difficult data set used in the research reported in this paper. It uses all 65 of the images considered for each of the 10 classes. The resulting problem is extremely difficult: there are 10 classes, very few training and testing examples and the feature set makes the feature space patchy, meaning it is difficult because members of one class can be nearer to members of other classes than their own, when distance is measured in a Euclidean



Feature	LGP Index	Description
f1	`cf[0]`	mean brightness of the entire cut out
f2	`cf[1]`	mean of the top-left hexant
f3	`cf[2]`	mean of the top-right hexant
f4	`cf[3]`	mean of the middle-left hexant
f5	`cf[4]`	mean of the middle-right hexant
f6	`cf[5]`	mean of the bottom-left hexant
f7	`cf[6]`	mean of the bottom-right hexant
f8	`cf[7]`	standard deviation of the left-most semi-hexant
f9	`cf[8]`	standard deviation of the centre-left semi-hexant
f10	`cf[9]`	standard deviation of the centre-right semi-hexant
f11	`cf[10]`	standard deviation of the right-most semi-hexant

Figure 3.5: Terminal set for the *faces1* data set. The location of the feature positions described on the right are shown in the diagram on the left.

manner. As with *faces1*, ten-fold cross-validation was used in all experiments involving this data set.

Features

The features used in the *faces3* data set are identical to those used in *faces1* (see section 3.3.2).

Chapter 4

LGP for Multi-class Classification

4.1 Overview

In this chapter we seek to achieve the goal of developing a linear genetic programming methodology for multi-class object classification problems. This configuration must address the problems faced by current techniques for multi-class classification using TGP by exploiting the linear genetic programming representation. In addition, a heuristic which allows TGP and linear genetic programming configurations to be compared is developed.

No evaluation of the developed methodology is carried out in this chapter. This is done in a fuller and more complete manner in chapter 8.

The structure of this chapter is as follows. In section 4.2 problems faced by existing TGP methodologies for multi-class classification problems are analysed. The form of linear genetic programming used in this research is discussed in section 4.3. A heuristic which allows the comparison of TGP and linear genetic programming configurations of the form used in this research is developed in section 4.4 and the initial generation, output interpretation algorithm and evolutionary operators used in this linear genetic programming configuration are described in sections 4.5, 4.6 and 4.7, respectively. Finally other elements of the configuration are described in section 4.8.

4.2 Problems with Current Techniques for TGP

Two key problems impede the use of TGP in multi-class classification tasks. Both of these stem from the fact that TGP programs evolve tree-like structures [24], which map a vector of input values to a single real-valued output [50, 29, 52, 56]. For classification tasks, in a second step, this output must be mapped onto a set of class labels. For binary classification problems, there is a natural mapping of negative values to one class and positive values to the other class. For multi-class classification problems, finding the appropriate boundaries on the number line to separate the classes is very difficult. The output of a TGP program does not map naturally to a multi-class classification problem in which one output (a metaphorical ”degree of confidence“) for each class makes the most sense.

The understandability of TGP programs is already low. Consider the program in figure 2.3 (page 11). This program uses nearly every feature and the role played by each feature (even if the output interpretation algorithm is borne in mind) is unclear. A TGP program for a multi-class classification task is generally difficult to understand. This is because both the program and the output interpretation algorithm need to be kept in mind and also because the contribution of each feature to the classification of an example for each class is unclear. Is the value of `f3` important when deciding whether an example is a member of `class4`?

This two-stage calculation and interpretation process also manifests itself in the results. Ordinary TGP programs do less well on some tasks — the mean accuracy is lower and the variance in this mean is higher [61]. Several new translations have recently been developed in the interpretation of the single output value of the tree-based GP [29, 57, 58] with differing strengths in addressing different types of task. While these translations have achieved better classification performance the evolved programs are hard to interpret, particularly for more difficult problems or problems with a large number of classes.

4.3 Linear Genetic Programming Overview

In this research register-machine LGP [1] has been used. In LGP each program is a sequence of register machine instructions. This is typically expressed in human-readable form as C-style code. Other variants of linear genetic programming are discussed in [20], but they are not considered in this research.

Prior to any program being executed, the registers which it can read from or write to are zeroed. The features representing the objects to be classified are loaded into predefined registers. The program is executed in an imperative manner and represents a *directed acyclic graph* (DAG, see figure 4.1 for a simple example). This is different from tree-based GP which represents a tree. Any register's value may be used in multiple instructions during the execution of the program.

Instructions in an LGP program may also be *introns* - i.e. code whose execution has no impact on the output of the program. The existence of introns can significantly improve the evolvability of a solution by allowing good building blocks to exist non-destructively in a distributed manner across several individuals, each of which has a small part of the building block. A partial building block may have a negative impact on fitness. This impact is eliminated when it is present as an intron. As with biological evolution, introns also reduce the frequency of destructive evolutionary operations which slow the evolution of fit individuals, as defined and discussed in [36] and [35]. After the final instruction in the program has been executed the values of the registers are interpreted as the output.

```
//r[1] = r[1] / r[1];
//r[3] = cf[0] + cf[5];
//if(r[3] < 0.86539)
//r[3] = r[3] - r[1];
r[0] = 0.453012 - cf[1];
//r[3] = r[2] * cf[5];
r[1] = r[0] * 0.89811;
if(cf[6] < cf[1])
r[2] = 0.453012 - cf[3];
r[3] = cf[4] - 0.86539;
```

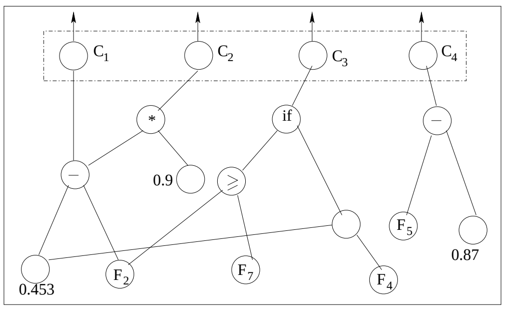


Figure 4.1: A very simple sample program evolved by LGP and a DAG representation of it. See figure 8.1 for a larger program and a fuller discussion.

4.4 Comparing TGP Depth to LGP Length

In this research it has often been necessary to compare a TGP and LGP configuration. In order to make this possible a heuristic which allows configurations to be made as similar as possible has been developed. We call this the *conversion heuristic*.

This heuristic relates the number of nodes allowed in a maximum-depth TGP program to the number of instructions in a maximum-length LGP program. An LGP instruction typically consists of one or two arguments and an operation, each of which corresponds to a node in a TGP program tree. Hence an LGP instruction might initially seem equivalent to 3 nodes. However, considering that each TGP operation might be used by its children and/or parents, an LGP instruction corresponds to roughly 1.5 tree nodes. Assuming each non-leaf node has two children (or more for some functions), we can calculate the expressive capacity of a depth-n TGP in LGP program instructions.

4.5 LGP Program Initial Generation

In this research, based on initial empirical research, the methodology was structured so that all programs were initially generated at the full length. This was motivated by three factors.

Firstly, most final solutions in TGP and LGP are of the maximum allowed length, even if they are initially generated at much less than this. It is hypothesised that this is because a longer program provides more expressiveness and evolutionary "material" for the beam search to work with. Whatever the cause, beginning evolution at this point short-circuits the part of the evolutionary process in which generations are expended evolving larger programs.

Secondly, a greater length increases the probability that any one instruction is an intron. Therefore it decreases the chance of a destructive evolutionary operation [36, 35] breaking up a building block. This is because there are more places a crossover can occur which do not break a building block. The number of crossover locations which will break one also increase (as the overall average program length increases) but this number does not increase at the same rate.

Thirdly, unlike in TGP, the chance of an entirely unproductive crossover is dramatically minimised. Crossover is discussed in section 4.7.1. This means that the costs of crossover on full length programs are negligible. Boosting the proportion of productive crossover is one of the key motivations for generating programs at less than their maximum length.

4.6 LGP Output Interpretation in Multi-class Classification

An LGP program often has only one register interpreted when determining its output [37, 1]. This configuration can be easily used for regression and binary classification problems as in TGP.

In this work, we use LGP for multi-class object classification problems. We want an LGP program to produce one output for each class. Thus, instead of using only one register as the output, each register used corresponds to one class. The winner-takes-all strategy is then applied and the class represented by the register with the largest value is considered the class of the input object by that genetic program.

This representation of each output node as one class is reflected in figure 4.1 and is very similar to a feed forward neural network classifier [44]. However, the structure of such an LGP program is more flexible than the architecture of the feed forward neural network.

4.7 LGP Evolutionary Operators

LGP evolutionary operators are very similar to TGP evolutionary operators. Reproduction (described in section 2.3.4), *linear crossover* and two forms of mutation (*micro-mutation* and *macro-mutation*) [3] have been used in this research.

4.7.1 Linear Crossover

The linear crossover operator involves two parents. A random sequence of instructions in the first parent is selected and replaced with a random sequence of instructions from the second parent. The resulting program is the offspring. If the newly produced program is longer than the maximum length allowed, then an instruction is randomly selected and removed until the program can fit into the maximum length. This is similar to two-point crossover in GAs [15], but the two sections chosen from the parents can have different lengths. An example of linear crossover is shown in figure 4.2.

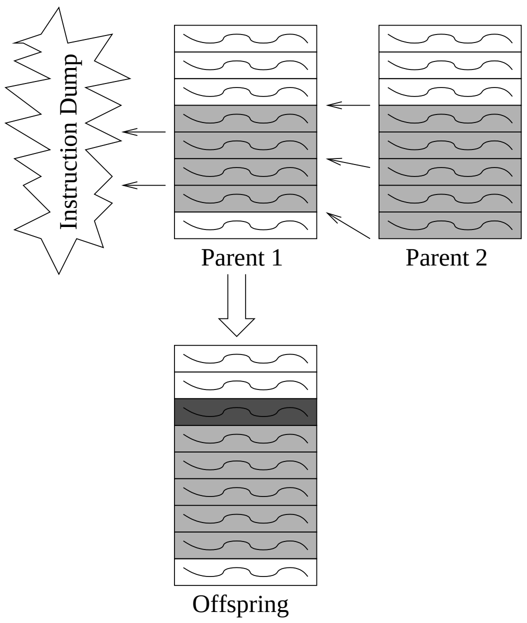


Figure 4.2: An example of an LGP crossover. This example shows how a different length segment can be selected from each program. Lightly highlighted instructions in the first parent are removed and replaced with the lightly highlighted instructions from the second parent. The darkly highlighted instruction in the offspring is the one randomly selected and culled to reduce it to the maximum allowed length of 8 instructions. In this example none of the crossed over instructions was randomly selected for culling, although they could have been.

4.7.2 Micro-mutation

Micro-mutation changes only either the destination register, a source register or the operation of one instruction. Note though that these operations can cause dramatic changes in the DAG that a program represents [5].

4.7.3 Macro-mutation

Macro-mutation replaces one randomly selected instruction in a program with a randomly generated one. The program is copied otherwise unchanged into the next generation.

4.8 Other LGP Configuration Details

4.8.1 Fitness Function

The fitness function used in this research is discussed in more detail in chapter 5, however all fitness functions used in this research were based on the error rate.

4.8.2 Function Set

The set of available functions for use in each operation was the standard arithmetic operators, protected when necessary, and one conditional. They are detailed in table 4.1. Sequential conditionals cascade to form a conjunction.

Table 4.1: Functions used in the LGP methodology developed.

Function	Description
+	Standard arithmetical addition.
-	Standard arithmetical subtraction.
*	Standard arithmetical multiplication.
/	Protected division. Returns 0 on a divide-by-zero.
if<	Skips the next assignment if the first argument is >= the second.

4.8.3 Terminal Set

The terminal set is dependent on the data set being evaluated. These have been discussed in chapter 3. In addition to problem-specific terminals the ephemeral random constant, R, was available in all terminal sets.

4.9 Chapter Summary

In this chapter the goal of developing an LGP methodology for multi-class classification problems was addressed. A new crossover operator and a heuristic which allows an LGP configuration using this methodology to be compared to a TGP configuration were also developed. These developments contribute by showing how to apply the relatively new and powerful technique of linear genetic programming [1] to multi-class classification problems. The full scope of the impact of these contributions are analysed in chapter 8.

Chapter 5

Fitness Functions in Multi-class Classification

5.1 Overview

The LGP methodology developed and described in chapter 4 is extended and improved in this chapter by addressing the goal of developing and evaluating a new fitness function which is resistant to one of the factors that can make a multi-class classification problem difficult.

This chapter is structured as follows. The role (or *telos*) of a fitness function is first analysed in a theoretical manner in section 5.2. Some definitions relevant to the fitness function which is developed are then given in a more formal manner in section 5.3. The ways in which a multi-class classification problem can be difficult are analysed in section 5.4. Given this theoretical grounding a new fitness function is developed in section 5.6 and the experiment used to evaluate it is described in section 5.7. The impact of this new fitness function is discussed in section 5.8.

5.2 The Role of the Fitness Function

The function of the fitness function in any configuration of a genetic programming run against a problem is to approximate, as closely as possible, how well any one individual program solves the problem. Because the size of the set of fitness cases against which a population is trained is finite and because the set of possible fitness cases in almost all interesting problems is infinite any individual program's fitness on a set of fitness cases is necessarily an approximation to an individual program's *true fitness*, its actual ability to correctly solve the problem.

Two factors which can effect the accuracy of a fitness function's approximation to true fitness is the degree to which the fitness cases are representative of the *feature space* (defined in section 5.3.1) and any assumptions built into the fitness function. Given a (randomly selected, statistically likely and representative) set of fitness cases these assumptions may positively or negative bias the fitness function.

In classification problems a program's true fitness is the proportion of the *feature space* it correctly classifies, perhaps weighted by the probability of a fitness case occurring at each point in this space. The feature space is the multi-dimensional space sketched out by the features and their possible values that a program can use to make a classification. A fitness function in a classification problem should serve to estimate as accurately as possible the proportion of the feature space that any one individual program can correctly classify.

Within the context of this report a *traditional fitness function* is one in which fitness decreases linearly as the number of misclassifications of any class increases. In other words, the 150th misclassification of a class has just as negative an impact on fitness as the 1st — no more and no less. The detection rate (DR), false-alarm rate (FAR) [55], true positive fraction (TPF), false positive fraction (FPF), accuracy, error rate or any combination of these in a multi-objective fitness function are traditional fitness functions [52, 57, 58].

5.3 Salient Definitions

Prior to discussing the fitness function in any further detail it is valuable to tangent briefly and define a number of concepts and terms whose use will be necessary later in this chapter. In these definitions, for the sake of avoiding confusion, the statistical aspect of many of them will be sketched out in full. In discussion using the terms this wordiness will generally be avoided for clarity's sake. When some statistic or stochastic process is spoken of in a definite or certain sense though it is important to remember that any consequences being referred to are likely or probabilistic and represent some sort of average or expected outcome.

5.3.1 Feature Space

Given these caveats the feature space does not need to be defined more strongly than it was above. Instead it is important simply to note the size of this space. If a problem uses 9 features and those 9 features are discrete and each has only 256 possible values (say we are looking at the raw pixels of a 3 by 3 PGM image) then the possible number of different fitness cases is $256^9 = 4,722,366,482,869,645,213,696 \approx 4.7 \times 10^{21}$! While it is not true that this feature space is infinite there is no way every possible fitness case can be used, and the fitness function must be an approximation of some kind.

5.3.2 Program Space

Another important and related abstract space is the *program space*. This concept is analogous to the biological concept of the *fitness landscape*, as discussed in [17] and [21]. Within the context of genetic programming, each possible program (on which evolutionary operations can be carried out) occupies some place in the program space, a very high-dimensional space. Each position in this space therefore has a fitness associated with it.

The program space can be conceptualised in a number of ways. One approach is to view each possible instruction as a dimension. An individual program's position on this (discrete) dimension is then determined by whether or not it possesses this instruction and if it does, at what position (index) in the program. Using this conceptualisation, the 9th instruction in the program would have a value of 9 in the dimension in the program space representing that instruction. Another approach is instead to create a set of binary dimensions for each pair of possible instructions and possible indexes - if a program has a particular instruction at a particular index then its value in that specific dimension is 1. For all other dimensions for that instruction it is 0. In both approaches each ephemeral constant (R) generated in a population can simply be treated as another register index (and thus another dimension).

To give an example of the number of dimensions that the program space has, consider an extremely simplified program structure. There are only two instructions in each program. For each instruction, there are four possible operators (+, -, $\times$, /), and each of these uses three registers - two arguments, one destination. Each of those three registers can be only one of two possible registers. An example program is below:

```
void VUWLGP::Program::Execute(std::vector<double>& r) const {
        r[1] = r[0] + r[0];
        r[0] = r[1] / r[0];
}
```

Using the second conceptualisation of the dimensions and given the above description of the possible instructions we can conclude that there are 4 operators $\times$ 2 registers $\times$ 3 arguments $= 24$ possible instructions. There are two possible positions for an instruction, and thus 48 dimensions per instruction, or 96 in total for a program. These dimensions are not completely orthogonal — two different instructions cannot both have the same index and therefore some positions in some pairs of dimensions cannot both be occupied at the same time, but this is a contingent fact and does not reduce the dimensionality of the program space. The dimensionality of the program space in a more realistic configuration would obviously be *substantially* larger — consider one where the maximum program length was 40, with five possible operations (each with three arguments) and 90 or so possible registers or constants.

The program space is obviously a complex concept. Why is it important? The answer to this question first requires understanding what the program space actually *is*. This is relatively simple - it is the landscape across which evolution takes place and in which the evolution happens. Each point in this landscape represents a mapping of classes to spaces in the feature space. Hence each point in the program space has a fitness (a height) and the program space is in fact a meta-feature map. Individual members of the population climb as a result of evolutionary operations in this program space. The nature of the program space and the way levels of fitness are distributed across it obviously affects the difficulty of the problem a configuration is tasked with, and for that reason the program space is important.

5.3.3 Monotonic Program Space

Thirdly I would like to define what it means for a program space to be a *monotonic program space*. A monotonic program space is one in which (it is statistically probable that) the relationship between the fitness of any two individuals will also exist in their offspring. Thus, if one is fitter than the other, we would statistically expect the offspring to (on average) retain this difference in fitness. While the highly probable increase in both average and best fitness from generation-to-generation, towards perfect fitness, means that the relationship will not be linear or even proportional (asymptotically hyperbolic perhaps?) for a program space to be monotonic this relationship, in whatever form it takes, must exist. For example, if two offspring are more likely to have the fitness of the others' parent(s) than their own parent(s) the space is decidedly non-monotonic!

5.3.4 Average Evolutionary Operation

Fourthly and finally I'd like to define the *average evolutionary operation*. This will not be defined precisely and mathematically (although it could be) as this is not necessary in the way I will be using it. For any given program in some population there are a finite number of possible evolutionary operations which change some of the instructions in that program. All of the other possible positions (programs) in the program space thus have some probability of being the "destination" of an evolutionary operation. An average operation is roughly analogous to the expected (or average, or probabilistic) distance in the program space the offspring will be from the parent(s) as a result of an evolutionary operation.

5.4 What Makes A Problem Difficult

A multiclass classification problem can be difficult in at least three ways.

5.4.1 The First Difficulty

Firstly, a problem may be difficult because at some particular point(s) within its feature space members of multiple classes may exist. At such points no individual program can successfully discriminate between members of the different classes which occur there, and thus no individual program can have perfect true fitness, in the form of a 100% accuracy rate.

The claim that for some problems 100% fitness is an impossibility has seemed overly strong to some people, and so I'd now like to take some time just to sketch out more fully why this is the case. There are three causal reasons objects from different classes can exist at the same point in the feature space. The reason the existence of multiple classes at the same point in the feature space precludes the evolution of perfect accuracy I take to be obvious.

The first cause of this first difficulty is that it may be the case that manifestly different fitness cases in fact have identical features. Consider the artificial and manufactured example of a four-class problem in figure 5.1. If only one feature is used and that feature is the mean brightness of the entire object then there is no way to distinguish between a member of class 1, class 2, class 3 and class 4.

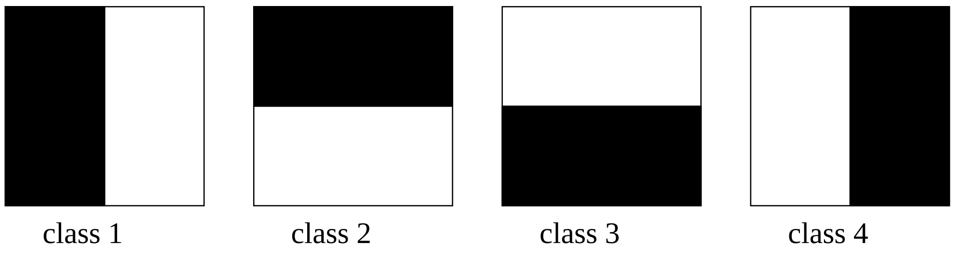


Figure 5.1: An example of a four class classification task, vulnerable to the first problem.

Secondly, it may also be the case that for some problems all information — or, more realistically, all knowable information — is simply insufficient to discriminate between members of different classes. The stereotypical binary-class "bank manager" problem is a good example of this: barring exceptional fortune and without appealing to the impractical and impossible philosopher's super-physicist there is simply no way a bank manager can decide with 100% reliability whether or not to loan somebody money. They will make mistakes and loans they approved will be defaulted on, just as they will refuse to loan money to people who it turns out would have been able to repay them.

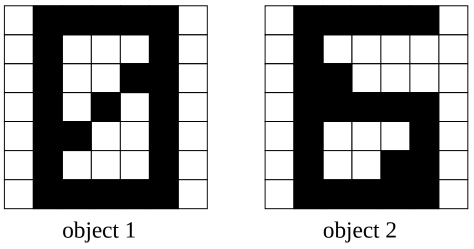


Figure 5.2: Two objects in class 8.

Thirdly, if examples which need to be classified are strongly corrupted by noise, relative to the degree of difference between members of the class, then the classes will simply begin to ″blur“ together. Consider the two example digits given in figure 5.2. Each digit in this figure is a member of the class ″8“ from the *digits15* problem. Notice how object 1 looks exactly like a ″0“, and how object 2 looks a lot like a noise-free ″6“. Despite this, both are in fact still members of class ″8“ and should be considered a member of this class by the evolutionary process. For a more complete discussion of this topic I strongly refer the reader to [9].

Of these, the first and third cause of this difficulty — the selection of an imperfect feature space and noise — are far more common. However, this does not mean that the problem of noise or of having many classes map to one point in the feature space can be trivially avoided simply by selecting the feature set carefully or by somehow reducing noise. Noise is, by definition, entropy and cannot be programmatically reduced. Similarly, predicting what kinds of values members of each class will have in advance is not trivial, except for overly simple examples where multiple deliberately overlapping features are used. If the kinds of values members of each class might have for each of the features cannot be predicted though then this cause of a many-to-one mapping of classes to points in the feature must be considered.

5.4.2 The Second Difficulty

Secondly, a problem can be difficult because the feature space is huge, with many dimensions, or because there are a large number of classes which any one fitness case might be a member of. This problem, in both of its forms, occurred in the *digits* data sets.

5.4.3 The Third Difficulty

Thirdly, a program space can be difficult to learn in if it is strongly non-monotonic. This leads to what is called the *hurdle problem*, due to the way it manifests itself: if a generation-by-generation log of best and average individual fitness is reviewed in a run in which the non-monotonicity of the program space is an issue the best program’s fitness will appear to hit some level of fitness from which it will not improve for several generations. However during this time the average fitness will continue to improve. In essence, the population is stalled against a barrier level of fitness which it is more difficult to improve beyond than other levels of fitness. Indeed, once the barrier is broken improvement is almost always exceptionally rapid.

While the details of the hurdle problem will be discussed below it’s valuable to note here the similarities of the hurdle problem to the problem of local peaks or minima when a hill climbing or gradient descent algorithm is being used (e.g. the back propagation algorithm [45]). It is similar in three important ways. Firstly, the extent to which non-monotonicity is a problem in any particular run is highly contingent on the initial random generation of programs and the random selection of evolutionary operations. Due to this it manifests itself as a meta-element of the results and is not explicitly visible in the results of any one run — instead, due to the delay in jumping the hurdle, a GP configuration may simply look like it is unsuitable for that problem. Secondly, some problems are less monotonic (and thus more problematic) than others. Thirdly, while it is less irrevocable for GP than it is for hill climbing, the hurdle is sometimes contingently un-jumpable and a population of programs may be permanently trapped at what is precisely analogous to a local peak. Evolution will then be terminated only when the maximum number of generations has elapsed. Ascertaining exactly what, if any, differences exist between the hurdle problem and the problems posed by local peaks or minima requires further research.

5.5 What Causes The Hurdle Problem

The hurdle problem occurs in a program space which forms a jagged or discontinuous surface and is structured in such a way that an easily found local peak exists from which no (or very few) average evolutionary operations will result in the offspring's fitness being better than their parents. This is a non-monotonic feature space. In this situation an increasing proportion of the population will tend to become trapped at this peak. It is only through a fortuitous and exceptional evolutionary operation that an offspring can escape the peak. Due to the shape of the program space, selection will tend to draw lineages of programs and their offspring not at the peak towards it. Just as genetic programming is analogous to explicitly hill climbing algorithms in some ways (as described above, in section 5.4.3), it is important not to forget that genetic programming is also analogous to simulated annealing [30] in other ways. Like simulated annealing it is a form of implicit hill climbing (through the selection pressure the fitness function exerts) and — like simulated annealing — stochastic processes reduce the probability of an entire population being trapped by local peaks.

The conditions which can lead to such a program space being a jagged or discontinuous (and thus non-monotonic) surface are complex and this is one area further research needs to be carried out. Despite this lack of knowledge the fact that it occurs most frequently in any configuration which addresses the *shape* data set allows a number of preliminary conclusions to be drawn.

The *shape* data set was designed to be as difficult a dataset as it could possibly be *without* being difficult due to blurriness or an overly large feature space. Instead it was designed to be difficult due to classes having complex boundaries in the feature space. This means that it is very difficult for an evolved program to distinguish any one fitness case as belonging to a single class based on only one feature - at least two are needed. For example, consider the feature vectors given in figure 5.3 for each class in *shape*.

```
               cf[0]  cf[1]  cf[2]  cf[3]  cf[4]  cf[5]  cf[6]  cf[7]
---------------------------------------------------------------------
Obj1 (class1): 0.2720 0.3139 0.2748 0.2905 0.2087 0.1063 0.4658 0.2174
Obj2 (class2): 0.6425 0.6253 0.6469 0.6394 0.6583 0.7013 0.1743 0.0912
Obj3 (class3): 0.2727 0.2968 0.2865 0.2750 0.2325 0.2227 0.2296 0.1043
Obj4 (class4): 0.8124 0.7730 0.8057 0.8137 0.8571 0.8568 0.2296 0.1049
```

Figure 5.3: Example feature vectors for each class in the *shape* data set. The `cf` registers contains the 8 features described in section 3.1 in registers `cf[0]` to `cf[7]`.

These classes form (in the feature space) spaces with complex boundaries. Figure 5.4 illustrates a simplified two-dimensional feature space situation analogous to one in which the hurdle problem could occur. Note that because of the figure's low-dimensional simplicity I am not convinced the actual feature space shown is sufficiently non-monotonic for the hurdle problem to ever be a noticeable problem, however the situation would be different if it were a five- or six-dimensional space (one obviously not representable in two-dimensional form) and just as jagged.

In the situation shown in figure 5.4 class 1 and class 2 represent the two sets of fitness cases between which the hurdle problem occurs. In the example shown there are only two sets and they constitute all fitness cases. The hurdle problem could occur if there were more than two sets of fitness cases and these sets could also be a sub-set of the set of all training fitness cases, but for explanatory simplicity it has been illustrated this way.

Correctly classifying members of more than one of the sets (in this case, both sets) is dif-

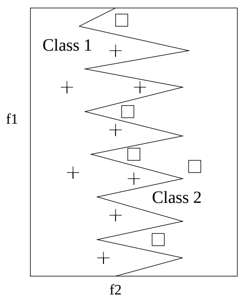


Figure 5.4: An example of the hurdle problem. This diagram shows the feature space and the set of fitness cases that a program is evolved against. Regions represent where members of class 1 and class2 can be found. +'s are fitness cases in class 1 and □'s are fitness cases in class 2. In this hypothetical example it is assumed that objects are more likely to occur near the boundary between the classes and that the set of fitness cases shown is therefore representative.

ficult because it is difficult for any evolutionary operation to accurately model the boundary in true fitness space (described by the zig-zagging line). Instead almost all evolutionary operations (and therefore the average evolutionary operation) will simply move a much simpler line, one with two or three segments to it perhaps, around a bit — adding another segment here or there and shifting it slightly to the left or slightly to the right.

Because this boundary is so difficult to learn the easiest hill for any individual program to climb will be the the hill which leads to it correctly classifying 100% of one class and (because this is such a simple example) perhaps 1 or 2 of the other class. Once this situation has occurred the boundary between the classes must be accurately learnt and, as discussed above, this is a very difficult task. It is likely that all attempts to achieve this will result in the offspring having worse fitness than their parent(s) and while the best individual's fitness will not get any worse (due to elitism) it also will not improve until an exceptionally un-average evolutionary operation takes place. A related cause of the hurdle problem with similar characteristics is a patchy feature space, illustrated in figure 5.5.

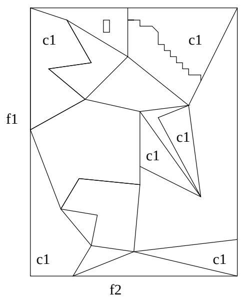


Figure 5.5: A patchy feature space for a three class problem. This diagram shows the feature space and contiguous regions that members of a class can be found in. Only regions in which members of class c1 are found have been marked. Other classes have been left unmarked for clarity.

Let us now summarise the causes and impact of the hurdle problem. It is caused by a strongly non-monotonic program space. Such a program space is probably (but not certainly) caused by a jagged (diagrammed in figure 5.4) or discontinuous (i.e. patchy) feature space (diagrammed in figure 5.5). In such a situation a traditional fitness function will not accurately estimate the proportion of the feature space correctly classified. This inaccurate relationship between the value given by the fitness function and an individual's true fitness leads to inefficient evolution. There is strong selection pressure *against* crossing the hurdle as most evolutionary operations which begin the crossing lower an individuals overall fitness, and the hurdle thus represents something analogous to a local peak in hill climbing, escapable only because of the stochastic nature of genetic programming. The impact of this on the fitness of the best individual in a run of a genetic programming configuration against a problem is visible in a generation-by-generation log of fitness, as some level of fitness at which improvement stalls at for several generations. Consequently, mean training time is lengthened and mean accuracy is decreased. Mean variation in training time may be decreased (if the hurdle is particularly high) or increased (if it is often, but not always, jumped).

5.6 Addressing the Hurdle Problem — New Decay Curve Fitness Function

The hurdle problem can be addressed by assuming that an individual which correctly classifies objects from more classes is fitter than an individual which correctly classifies the same number or slightly more individuals but from fewer classes. This assumption is manifested by penalising proportionally more strongly later misclassifications of the same class than earlier misclassifications. The first misclassification of a class might attract a penalty of 0.5 and the second might attract a penalty of 1.0. Thus the program diagrammed in figure 5.6(b) will attract a penalty of 1.5. The program diagrammed in figure 5.6(c) will only have a sum penalty of 0.5 + 0.5 = 1.0 and thus has a better fitness. This is appropriate as it also more accurately models the feature space. From generation to generation the population's ability to ″jump the hurdle“ and accurately model the feature space will grow as programs like (c) are preferentially selected.

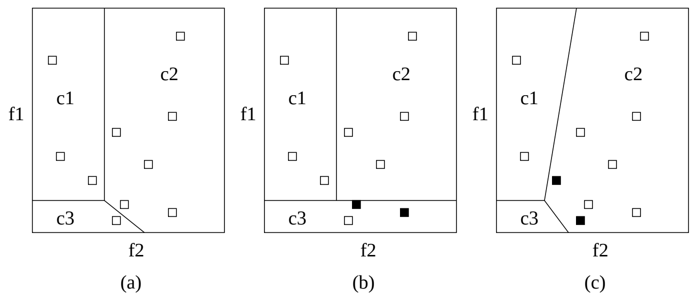


Figure 5.6: Another example of the hurdle problem. Each diagram shows the feature space and the set of fitness cases that a program is evolved against. The regions marked in (a) show the true regions. The regions marked in (b) and (c) show how two individuals classify the feature space. Incorrect classifications in (b) and (c) are represented by filled boxes. If a traditional fitness function is used then (c) is not preferred to (b) despite classifying more of the feature space correctly and therefore having a higher true fitness than (b).

It is theoretically true that in some situations an individual which misclassifies two members of class c2 will have a better true fitness than an individual which misclassifies one from c1 and one from c3. However, if a good distribution of fitness cases is used it is only in extremely rare situations that this is the case. Hence the decay curve fitness function (to be described below) will more accurately measure how well an individual program classifies the feature space almost all of the time.

Note though that we cannot just reduce the penalty of the first n misclassifications of a class, or increase the penalty of the last k misclassifications. Such an approach does not remove the hurdle problem - it simply moves it, to the $k + 1$'th or $n + 1$'th member of each class. Thus we introduced a new fitness function, f_{decay}, with an increasing penalty for each of the M_c misclassifications of some class c. An individual program's fitness in a task with N fitness cases can be calculated as described by equation 5.1.

$$f_{decay} = \sum_{c} \sum_{i=0}^{M_c} \alpha^{\beta i}/N \tag{5.1}$$

The values of α and β can be determined through empirical search. Larger values are generally better, although this depends on the extent to which classes are similar. The f_{decay} presented and discussed above is the function used in this paper and is an error rate-based decay curve fitness function. Any other function which has this increasing penalty will also probably avoid the hurdle problem.

Obviously as α approaches 1 f_{decay} becomes more and more like a traditional fitness functions. Similarly as β approaches 0 the curve becomes progressively flatter and the fitness function becomes more traditional. Old fitness functions have α = 1 and β = 0. Fitness functions with values of α smaller than 1 or with values of β smaller than 0 are either mathematically nonsensical or exacerbate the hurdle problem by creating a line which slopes the wrong way.

Finally, as discussed above, the hurdle problem almost always occurs at the boundary between two classes. Generally all fitness cases for all but one class can be correctly classified relatively easily. For this final class no fitness cases are correctly classified. In situations in which the feature space is patchy (for example, figure 5.5) some but not all of the fitness cases for that class may be correctly classified, resulting in an intra-class hurdle. Due to the exponential nature of f_{decay} an intra-class hurdle is as easily jumped as an inter-class one.

5.7 Experimental Design

To investigate whether the new fitness function is helpful in addressing the hurdle problem the *shape* data set was used and the performance of the new and old fitness function (error rate) has been compared.

5.7.1 LGP Configuration

In these experiments a reduced maximum program size has been used for LGP, with the maximum number of instructions reduced to 10 (the standard maximum length for this problem, based on the heuristic developed in chapter 6, is 16). This emphasises the impact of the hurdle problem.

By applying the LGP-to-TGP conversion heuristic (section 4.4) we see that the appropriate depth range for a TGP program compared to a length-16 LGP program is 3–5. While it is reasonable to reduce this depth to 2–4 (in the same way we have reduced the depth of the LGP program to maximise the occurrence of the hurdle problem) this has not been done. This is because the TGP configuration of depth 3–5 is shallow enough, relative to the difficulty of the problem, that the hurdle problem is already a significant factor.

The other parameters for TGP and LGP are presented in table 5.1. The two fitness functions used were the error rate, a traditional fitness function, and f_{decay}, described in section 5.6.

Table 5.1: The LGP configuration for the fitness function experiments.

	LGP	TGP
pop_size	500	500
base_program_size	16	3–5
elitism_rate	10%	10%
crossover_rate	30%	30%
macromutation_rate	30%	30%
micromutation_rate	30%	30%
tournament_size	4	4
α	1.15	1.15
β	0.18	0.18

5.8 Results and Discussion

The mean and standard deviations of 50 runs of the LGP and TGP configurations against the *shape* data set are presented in table 5.2.

Table 5.2: A comparison of the two fitness functions on the shape data set.

Method	Fitness Function	Training Accuracy ($\mu \pm \sigma$)	Test Accuracy ($\mu \pm \sigma$)
TGP	old	77.31% $\pm$ 6.74%	77.14% $\pm$ 6.68%
	new	85.04% $\pm$ 16.49%	84.41% $\pm$ 17.17%
LGP	old	98.90% $\pm$ 4.98%	98.76% $\pm$ 5.04%
	new	99.97% $\pm$ 0.11%	99.60% $\pm$ 0.25%

For the TGP configuration the new fitness function led to a very significant improvement on both the training set and the test set. For the LGP method, the classification accuracy was also improved using the new fitness function, but the improvement was not as significant as for the TGP method. This was mainly because the LGP method with the traditional fitness function already performed quite well (test accuracy of 98.76%) due to the strength of LGP on this problem.

Further inspection of the results using the TGP method on the *shape* data set shows that only 6 of the 50 runs using the traditional fitness function had a test or training accuracy greater than 75%. When those 6 runs are excluded the μ and σ becomes 74.95% $\pm$ 0.0019% on the training set and 74.86% $\pm$ 0.0024% on the test set. These figures indicate how solid the hurdle actually is in situations where the problem is at the limit of a GP configuration's expressiveness. By using the new decay curve fitness function 36 of the 50 TGP runs finished with test and training accuracies greater than 75%

5.9 Chapter Summary

In this chapter the goal of developing a fitness function which gives better performance on hard multi-class classification problems has been achieved. Along with the detailed analysis of the role played by the fitness function in LGP and TGP this contribution greatly strengthens the methodology developed in chapter 4. The importance of carefully analysing a problem when selecting a fitness function has also been highlighted.

Chapter 6

Program Length and Extra Registers

6.1 Overview

In this chapter the results of two experiments are presented and discussed. Both experiments concern important aspects regarding the configuration of LGP for multi-class classification problems. These experiments address the third goal given in section 1.2 and serve to further extend the LGP methodology developed in chapters 4 and 5.

In the first experiment the relationship between the difficulty of a multi-class classification problem and the program length needed for a solution to be reliably and quickly found was investigated. In the second experiment the value of extra registers not involved in the interpretation of the output was investigated. Prior to discussing each experiment the data sets used in both experiments and the common aspects of the configuration is described.

6.2 Data Sets

The *shape*, *digits15* and *faces1* data sets were used in these experiments. These data sets span a range of difficulties.

The *shape* data set was selected to provide an indication of how well the various configurations tested performed on a relatively easy four-class classification task. In addition the effect of the various configurations on the *hurdle problem* (in the mean and especially the standard deviation of the test and training accuracy) have also been investigated. The impact of the hurdle problem is discussed more fully in chapter 5.

digits15 is a mid-difficulty ten-class classification task in which a reasonably high degree of accuracy can be obtained. One of the key contributors to the difficulty of this data set is the size of the *feature space* (defined in section 5.3.1). In the other two data sets this factor is not an important contributor to difficulty.

faces1 is a five-class classification task which has a high level of difficulty. Of the three data sets it is also the most representative of a typical multi-class classification problem: it has a reasonable number of features and the hurdle problem is not a major issue. The task is difficult primarily because any given member of one class is often more similar to members of another class than members to of its own class (assuming nearness in the feature space is measured in a Euclidean manner).

6.3 Common Configuration

In this experiment the decay curve fitness function f_{decay} was used. This fitness function is described in chapter 5. All other details of the configuration are provided in table 6.1.

Table 6.1: The LGP configuration for the registers and length experiments.

	shape	*digits15*	*faces1*
pop_size	500	500	500
base_program_length	16	40	20
base_num_registers	4	10	5
elitism_rate	10%	10%	10%
crossover_rate	30%	30%	30%
macromutation_rate	30%	30%	30%
micromutation_rate	30%	30%	30%
tournament_size	4	4	4
α	1.20	1.15	1.15
β	0.24	0.18	0.18

6.4 Registers Experiment

6.4.1 Motivation

Koza [23] originally put forward the idea of improving the understandability and performance of evolved programs through the use of *automatically defined functions* (ADFs).

When using ADFs, TGP programs are changed to take the form of a forest of trees. One of these trees (typically the last) is defined as the output of the program. All of the other trees represent automatically defined functions and can be used (as terminals or functions) in later trees or the output tree. Koza suggested [23] that this approach would make more likely the evolution of subsumption architectures with better performance and more easily comprehensible structure. As a result he recommended that 2–3 ADFs be used.

One way of applying this approach to LGP is to increase the number of registers beyond those interpreted as part of the output. Assignments to these registers can never directly contribute to the accuracy of an object's classification. Instead, they can serve as building blocks which can be used in the calculation of the final value of multiple registers, each of which represents one class. While this process can take place to a limited extent even with no extra registers (if the final value of a register for one class is used as part of an operation for another, or if a register is used in this manner prior to it being used to calculate the final value for the class it represents) this is unlikely. Instead the typical program calculates each output register largely independently of the others. This experiment evaluates the value of extra registers which allow interdependent calculation of final register values more easily.

6.4.2 Configuration

Each data set has a certain number of classes. Between 1 and 10 extra registers not used in the interpretation of a program's output were evaluated. The results of a run in which no extra registers were available have also been provided for comparative purposes. All other configuration parameters were as provided above in section 6.3 and table 6.1. Each experiment was run 50 times and the mean and standard deviation of the results are given in the next sub-section.

6.4.3 Results and Discussion

The results for the *shape*, *digits15* and *faces1* data set have been presented in tables 6.2, 6.3 and 6.4, respectively. The mean and standard deviation of the number of generations needed for the *shape* data set has also been included. All runs on *digits15* and *faces1* used 50 generations.

Table 6.2: The impact of extra registers on the *shape* data set.

Extra Registers	Generations ($\mu \pm \sigma$)	Training Acc. % ($\mu \pm \sigma$)	Test Acc. % ($\mu \pm \sigma$)
0	18.28 ± 11.80	99.90% ± 0.68%	99.84% ± 0.80%
1	20.02 ± 11.17	99.69% ± 1.75%	99.65% ± 1.74%
2	23.22 ± 13.47	98.79% ± 5.70%	98.68% ± 6.09%
3	22.62 ± 13.07	99.28% ± 3.62%	99.31% ± 3.56%
4	25.22 ± 12.15	99.38% ± 4.06%	99.29% ± 4.10%
5	26.54 ± 14.15	98.15% ± 6.07%	98.14% ± 6.05%
6	22.24 ± 12.89	99.47% ± 2.46%	99.36% ± 2.66%
7	31.44 ± 14.42	96.98% ± 8.71%	96.98% ± 8.66%
8	23.96 ± 13.41	98.92% ± 4.95%	98.89% ± 4.94%
9	21.64 ± 14.35	98.05% ± 6.22%	97.97% ± 6.32%
10	36.48 ± 13.68	95.61% ± 9.09%	95.42% ± 9.37%

Table 6.3: The impact of extra registers on the *digits15* data set.

Extra Registers	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
0	66.26% ± 4.62%	62.07% ± 5.81%
1	65.89% ± 3.87%	61.58% ± 4.27%
2	64.97% ± 4.51%	61.39% ± 5.54%
3	66.38% ± 4.39%	61.68% ± 5.75%
4	66.45% ± 4.19%	62.37% ± 4.52%
5	65.61% ± 4.29%	61.36% ± 5.49%
6	66.52% ± 3.89%	63.04% ± 3.98%
7	66.38% ± 4.24%	63.06% ± 4.54%
8	64.96% ± 4.69%	60.24% ± 5.52%
9	64.94% ± 4.20%	60.92% ± 5.50%
10	66.29% ± 4.01%	62.34% ± 5.18%

The results from the *shape* data set show a clearly and dramatically increasing learning time for this task as the number of extra registers is increased. In addition, the mean test and training accuracy decreased and the variation increased. For the *shape* data set, extra registers show no value.

Similar results can be observed for the *faces1* task. All runs used the maximum 50 generations (because no run achieved perfect fitness on the training set) and so the difference in learning time can not be directly observed. Despite this, the negative impact of the extra registers is clearly visible in the steadily decreasing mean accuracy on the test and training set. It is hypothesised that this problem occurred for both *shape* and *faces1* because of the relative increase in the size of the program space. A run against the *faces1* problem with 15 registers involves a *program space* (section 5.3.2) at least three times as large as a run against

Table 6.4: The impact of extra registers on the *faces1* data set.

Extra Registers	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
0	57.34% ± 5.03%	50.85% ± 9.35%
1	55.23% ± 5.61%	49.30% ± 10.19%
2	53.40% ± 5.30%	49.14% ± 10.48%
3	52.69% ± 5.60%	48.00% ± 9.96%
4	53.73% ± 5.62%	48.54% ± 9.84%
5	54.79% ± 5.72%	48.32% ± 11.66%
6	53.31% ± 5.62%	47.14% ± 10.19%
7	54.09% ± 4.91%	47.78% ± 10.84%
8	52.28% ± 5.31%	45.51% ± 10.52%
9	52.50% ± 5.14%	45.41% ± 10.74%
10	53.36% ± 4.65%	45.14% ± 9.77%

it using only 5 registers.

The results for *digits15* are different. The mean results of the runs with 6, 7 and 10 extra registers seem to buck a relatively clear trend, however this trend does not show the decreasing performance visible in the results for *faces1* and *shape*. Instead performance seems relatively constant regardless of the number of extra registers. No firm conclusions as to the reasons for this anomaly have been drawn.

Across the three data sets, inspection of the resulting programs reveals that the extra registers are rarely used. This seems to indicate that no, or at most one or two, extra registers is the ideal configuration. Along with the research carried out in [33], this suggests that Koza's claims regarding the utility of ADFs may be more limited than initially thought.

Rather than increasing the number of registers, one alternative implementation of ADFs in LGP would be to implement them as they are in TGP, by having multiple programs with earlier programs becoming features or functions for later programs. However this approach would multiplicatively increase the run time and would seem to bring little benefit. This is because TGP ADFs give it a DAG-like nature, allow sub-programs to be reused and increase the understandability of the final program. All of these aspects are already present to the same extent in basic LGP.

6.5 Length Experiment

6.5.1 Motivation

Although no strongly consistent conclusions have been reached for TGP, research into the necessary and useful depth of tree which leads to solutions being found for a variety of problems has been conducted [23, 1]. This research suggests that the initial minimum and maximum depths of the programs be in the range 3–15, depending on problem difficulty. If early results are poor then increasing these values by three or four is recommended.

Similar research has unfortunately not been carried out for LGP and hence this problem has been briefly investigated (in the context of multi-class classification) in this research and the results presented here.

6.5.2 Configuration

All experiments were run with 0 extra registers. Excluding the maximum allowed program length, which varied as shown in the results tables (tables 6.5, 6.6 and 6.7) all other configuration parameters were as provided above in section 6.3.

Program length was varied relative to the number of classes, and ranged from one fewer instructions than there were classes to 10 times as many instructions as there were classes. One fewer instructions can be used because the register not assigned to will have the value of 0 and would be the largest if all the other registers were assigned negative values. The entry in each table corresponding to the number of classes has been highlighted to facilitate reading. Each experiment was run 50 times and the mean and standard deviation of the results are given in the next sub-section.

6.5.3 Results and Discussion

Results for the three data sets (*shape*, *digits15* and *faces1*) are presented in tables 6.5, 6.6 and 6.7. As with the register experiments the mean and standard deviation of the generations used in evolution is presented only for *shape* as all runs on *digits15* and *faces1* used 50 generations.

Table 6.5: The impact of program length on the *shape* data set.

Program Length	Generations ($\mu \pm \sigma$)	Training Acc. % ($\mu \pm \sigma$)	Test Acc. % ($\mu \pm \sigma$)
3	36.26 ± 16.32	89.55% ± 11.62%	89.46% ± 11.76%
4	41.26 ± 14.04	88.76% ± 11.83%	88.89% ± 11.64%
5	26.02 ± 14.51	98.15% ± 5.85%	98.16% ± 5.83%
6	25.70 ± 14.96	97.51% ± 6.27%	97.33% ± 6.42%
8	24.24 ± 14.63	98.13% ± 6.14%	97.98% ± 6.33%
10	26.26 ± 14.33	98.87% ± 5.14%	98.85% ± 5.20%
12	23.52 ± 13.80	98.96% ± 4.24%	98.98% ± 4.14%
16	24.10 ± 13.85	99.80% ± 0.79%	99.73% ± 0.96%
20	23.30 ± 13.81	99.61% ± 2.50%	99.55% ± 2.61%
25	19.72 ± 12.23	99.33% ± 3.71%	99.31% ± 3.63%
30	21.78 ± 13.00	99.71% ± 1.19%	99.67% ± 1.30%
40	19.46 ± 11.56	99.49% ± 3.53%	99.42% ± 3.53%

The results for *faces1* and *shape* indicate that increasing length generally leads to increasing performance, although the high degree of variance in the results make this trend a noisy one. Despite this, for both data sets, improvements in performance seem to tail off once there are roughly 3.5–4 times as many registers as classes. This suggests that the maximum allowed program length should be 3.5–4 times the number of classes. Call this heuristic the *class-length heuristic*. This heuristic is an important and significant result because the *shape* and *faces1* data sets are of very different degrees of difficulty. This therefore seems to indicate that the program length is at least somewhat independent of the problem difficulty for LGP classification of multi-class problems. If this is the case this is an important strength of LGP in multi-class classification problems.

As in the previous experiment (section 6.4) *digits15* has produced somewhat exceptional results. Again, no clear explanation of these results emerges: it seems surprising that the best results in a problem with 10 classes and 49 features were consistently achieved when

Table 6.6: The impact of program length on the *digits15* data set.

Program Length	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
9	64.44% ± 3.76%	60.51% ± 5.35%
10	65.82% ± 3.89%	61.28% ± 5.29%
11	67.23% ± 4.08%	63.21% ± 4.34%
12	66.49% ± 4.71%	63.20% ± 5.05%
14	66.62% ± 3.38%	63.28% ± 5.20%
16	65.88% ± 4.00%	62.09% ± 4.95%
20	66.72% ± 3.67%	62.30% ± 4.42%
30	66.78% ± 3.61%	63.11% ± 4.53%
40	66.26% ± 4.62%	62.07% ± 5.81%
55	65.10% ± 3.45%	61.10% ± 4.36%
70	63.53% ± 4.58%	59.50% ± 5.20%
85	63.40% ± 4.14%	59.46% ± 4.81%
100	63.34% ± 5.01%	59.50% ± 5.55%

Table 6.7: The impact of program length on the *faces1* data set.

Program Length	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
4	48.63% ± 5.17%	43.30% ± 9.32%
5	51.49% ± 5.28%	44.43% ± 9.30%
6	51.53% ± 5.73%	46.32% ± 10.05%
8	53.78% ± 5.25%	48.49% ± 9.85%
10	54.49% ± 6.57%	47.78% ± 10.72%
12	54.94% ± 5.18%	48.05% ± 8.57%
15	55.36% ± 5.72%	50.22% ± 9.54%
20	57.34% ± 5.03%	50.85% ± 9.35%
25	55.21% ± 5.11%	49.14% ± 10.21%
30	56.13% ± 5.48%	50.00% ± 12.16%
40	55.49% ± 6.17%	48.43% ± 10.92%
50	54.90% ± 6.18%	48.59% ± 9.32%

there were only roughly as many instructions as classes, and therefore when the register for each class would only be assigned to once or at most twice. No good explanation for these unusual results has been found, although it is strongly suspected that the feature set (the 49 binary pixels) is a contributing factor.

6.6 Chapter Summary

In this chapter the LGP methodology for multi-class classification developed in chapters 4 and 5 was analysed. The goal of analysing the relationship between good program length and the number of classes in a problem has been achieved, and the value of extra registers has also been considered. Important heuristics which guide the development of the initial configuration for a multi-class classification problem have been developed. Because these heuristics provide initial guidance on what kinds of configuration will work well for a particular problem they are a key contribution in the use of LGP for multi-class classification problems.

Chapter 7

Hill Climbing in Linear Genetic Programming

7.1 Overview

Hill climbing is a kind of machine learning iteration, discussed in section 2.1.4 of chapter 2. In this chapter the fourth goal described in section 1.2 is addressed. An algorithm which combines hill climbing and an evolutionary beam search is developed and its effect on performance is analysed.

The structure of this chapter is as follows. The motivations for combining the evolutionary beam search and hill climbing are discussed first in section 7.2. In section 7.3 the hill climbing algorithm which was created is described and the way in which the hill climbing and evolutionary beam search are integrated is justified. The experimental design used to evaluate this algorithm is discussed in section 7.4 and the effect of integrating hill climbing and the LGP evolutionary beam search are presented and discussed in section 7.5.

7.2 Hill Climbing - Motivations

In LGP multi-class classification problems the landscape evolution and hill climbing can happen on is the program space, as discussed in section 5.3.2. Hill climbing and the beam search of evolutionary computing have different advantages and disadvantages. Hill climbing provides some guarantee of local optimisation. This local optimisation can be a global optimisation in a sufficiently simple space. Evolutionary approaches, such as linear genetic programming, can not take advantage of this form of reinforcement learning [51] but are much less likely to become trapped at a local maxima, for two reasons. Firstly, a population of programs is used and it is therefore highly likely that at least one individual will not become trapped in this way, except in limited circumstances (see section 5.4.3). Secondly, the stochastic nature of the search in evolutionary computing means that short-term sub-optimal changes to a lineage of individuals can occur (although with reduced chance of persistence from generation-to-generation). Therefore, even if a lineage of programs does become trapped at a local maxima it can escape.

The motivation for seeking to combine hill climbing and linear genetic programming is to investigate the ways in which the strengths of the two approaches can be combined. Past work [48, 60] has successfully applied hill climbing (gradient descent) to TGP and achieved improvements, indicating that this combination may be possible. This research assumes that the degree to which a task is successfully addressed can be measured on some real scale in a continuous and scalar manner.

When neural networks — a key example of a successful application of gradient descent — are considered two important similarities with LGP can be noted. The underlying representation of both methods is a DAG, and the interpretation of their output in a multi-class classification (winner-takes-all) is the same. It is hoped that hill climbing will augment the evolutionary beam search, enabling the final solution to be fine tuned and locally optimised in the program space. We expect this to improve the accuracy (fitness) of the best individuals.

7.3 LGP Hill Climbing Algorithm

7.3.1 New Program Structure

So that hill climbing can be performed more easily the program structure has been modified. Each instruction now has a weight associated with it. These weights are initialised to some random value between -1.0 and 1.0 but the hill climbing algorithm can adjust them to any value. The results of an operation are multiplied by this weight prior to being assigned to the destination register. When the LGP instructions are converted to human-readable C-style code they now have the form:

```
r[d] = (r[s1] + r[s2]) * w;
```

Where + can be any function in the function set.

7.3.2 Weight Adjustment Algorithm

This hill climbing algorithm is applied as directly as possible to the program space. Thus it is the instructions themselves (i.e. entities which define part of a program's position in the program space) which have weights and on which hill climbing is performed. A simple example program, converted to human-readable C++, will illustrate this most clearly:

```
void VUWLGP::Program::Execute(std::vector<double>& r) const {
        r[0] = (r[1] + cf2[1]) * 0.6783 // instr1
        r[2] = (0.8734 + 0.1925) * 0.9821 // instr2
        r[0] = (r[1] * 0.8734) * 0.3689; // instr3
        if(r[0] < r[2]) // instr4
        r[1] = (r[2] - r[2]) * 0.0836; // instr5
}
```

The floating point value which trails the first, second, third and fifth instructions is the weight, w_i, on which hill climbing is performed. In this algorithm the idea of applying a weight to an if< instruction is meaningless. How conditional statements are incorporated into this algorithm will be discussed below.

Figure 7.1 shows how the example LGP program above can be represented as a DAG of instructions. This graph illustrates a number of important elements implicit in the hill climbing algorithm (described below) but which might otherwise go unnoticed. Because of this, a less formal description of the algorithm and these elements will be given first.

- `instr1` is a *structural intron*, and so it is not present in the graph. `instr5` could be a *semantic intron*, but this cannot be discovered efficiently through static analysis and so it is included in the graph. However if it turns out to be a semantic intron the algorithm will not adjust its weight.

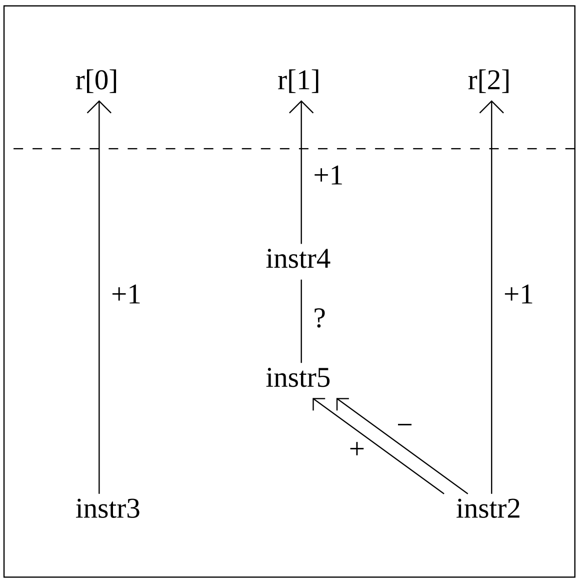


Figure 7.1: A DAG of program instructions and their links, built from the relatively simple program described above.

A structural intron is an instruction which can never have an impact on the final output, because the value it calculates is overwritten by another instruction before that value can contribute to the final output of the program in any way. [2] provides an algorithm which can be used to find these instructions in $O(l)$ time, where l is the length of the program.

A semantic intron is an instruction following a conditional which is never executed because the conditional is never true. Determining whether or not a particular conditional will ever be true depends on the possible values of the features involved and the calculations performed on them prior to the conditional being reached. [2] notes that semantic introns may not be completely detectable.

- The graph is comprised of links between instructions. A link is added to this graph whenever the register value an instruction results in is used in a later instruction. In addition, the final instruction to write to each register also has an ”output link”, represented by the link to a register in the top portion of the graph and with a sign of +1.
- All links have a sign. The sign of a link between two instructions is determined by the way in which the value of the destination register of the instruction which the link starts at is used in the instruction the link ends at. How these signs are determined is given in table 7.1. The sign of a link is determined entirely by the operation the destination register is used in and its role (first or second argument) in that operation. If it is used in both two links will be generated, one for each argument.
- The exception to the backwards direction of propagation of links are conditional statements. With all other operations used in this approach to LGP multi-class classification, instruction i's contribution to the program's error is determined entirely by what other instructions do with the value instruction i writes to a register and the final values of the registers whose value it contributes to. However conditional statements (which precede i in the list of instructions) may mean i is never even executed and thus had no contribution — positive or negative — to a program's error and accuracy.

Table 7.1: Instruction/argument position to link sign mapping.

Operation	Argument 1's Sign	Argument 2's Sign
output	+1	
+	+1	+1
-	+1	-1
*	+1	+1
/	+1	-1
if< signs are determined case-by-case — 1 if it was true, 0 otherwise		

It is for this reason that there is a link from `instr5` to `instr4` in figure 7.1 and it is for this reason that `instr5`'s sign depends on `instr4`'s sign.

Unlike all other operations, `instr4`'s sign cannot be statically analysed. Thus, when updating each instruction's contribution to error after evaluating a fitness case, information about whether or not a conditional was true needs to be retained. If the conditional *was* true then its sign will be +1. If it was false its sign will be 0. Given this, the contribution to error of the assignment instruction which followed it is either passed through unchanged (by the sign of +1) or set to 0 (by the sign of 0). This approach works even if several conditional statements follow each other sequentially and also ensures that an instruction's weight is not adjusted for error it did not create.

- Final register values have a ″sign“ (error value) too. This is because output links are defined to have a sign of +1 (as noted above). The *final register value* an output link ″links“ to may either be too large or too small. If the final register value is too large then the ″instruction sign“ of this final register value will be negative, and vice-versa if the final register value is too small. This value should be passed through unchanged to whichever instruction is at the other end of the output link. Calculation of the error in a final register value and its sign are described below, in equation 7.1.
- As with the final register values, all instructions have a sign. An instruction's sign is determined entirely by the signs of its links and the signs of the instructions it links to. Each link's sign is multiplied with the sign of the instruction it links to in order to provide a *net link sign*. These net link signs are then interpreted as shown in table 7.2 to calculate the sign of the instruction.

Table 7.2: Determining an instruction's sign. Net link signs with a value of 0 are not counted.

Net link signs are...	Instruction sign is...
Majority positive	+1
Equal positive, negative	0
Majority negative	-1

For example, consider `instr2` as shown in figure 7.1. If `instr4` is true and `r[2]` and `r[2]` both had an error of +0.5 then both `instr4` and `instr5` will have an instruction sign of +1 because `instr4`'s only net link sign is 0.5×1 (positive) and because `instr5`'s only net link sign is also therefore positive. `instr2` has one link to the output (which has a net link sign of 1×0.5), one positive link to the sign of `instr5` and one negative link to the sign of `instr5`. Therefore `instr2`'s net link signs are 0.5, 0.5 and -1, respectively. Based on these net link signs, `instr2` has an instruction sign of +1 as the majority of its net link signs are positive. Note that the instruction sign of `instr2` is never used in this graph. However this example was selected for discussion in this paragraph because it illustrated how the signs are determined in a more complex situation.

Given this informal but relatively complete discussion of the algorithm and its subtler elements the algorithm will now be presented more formally. Four key sub-algorithms and functions are used. They are:

1. The link building algorithm.
2. The desired output function.
3. The calculation of δ_i and Δ_i.
4. The weight adjustment algorithm.

The reader may need to refer table 7.1 and table 7.2.

Link Building Algorithm

Expressed in pseudo code, and assuming a length l program with instructions which have k arguments, an $O(lk)$ algorithm which builds the DAG of links between instructions is given in figure 7.2. This algorithm could also be trivially modified to incorporate Brameier's algorithm for finding structural introns [2]. By beginning at the output links and following the links backwards instructions can be marked as "not a structural intron". Once all such links have been traversed all the unmarked instructions are structural introns.

Desired Output Register Values

Desired output register values have been arbitrarily set at 1.0 for the register which represents the correct classification class of the fitness case and 0.0 for the registers which represent other classes. If the register value of the correct class is greater than 1 or the register value of the incorrect classes are less than 0 we do not want to lessen the amount by which the program gave the correct answer. Therefore, our determination of the error in the final register value, $e(r)$, for register r is given in equation 7.1.

$$e(r) = \begin{cases} 1 - min(\texttt{register[r]}, 1) & \text{if } r \text{ represents the correct class} \\ -max(\texttt{register[r]}, 0) & \text{if } r \text{ represents an incorrect class} \end{cases} \tag{7.1}$$

Calculation of δ_i

The contribution of instruction i to the program's error for one fitness case but across all registers is given by δ_i, and is calculated as follows:

$$\delta_i = \Sigma_{\text{for each of i's links, l}} \begin{cases} e(r) & \text{if } l \text{ is an output link (sign = +1) to register r} \\ \delta_j \times s \times PropScale & \text{if } l \text{ is a link to instruction } j \text{ with net link sign } s \end{cases} \tag{7.2}$$

PropScale is a constant which geometrically reduces the contribution to error (δ_i) of the instruction a link starts at as the instruction occurs at a greater remove from the instruction which writes the final register value. This is because any instruction's contribution to error in the final register value of r_1 is expected to be relatively less than its contribution to the

```
// dict[r] maps from register r to the i'th instruction
create a dictionary dict, leave it blank

// Build the internal links:
foreach instruction i, starting from the first
   if i is a conditional and is is not the last instruction then
      add a back-link from instruction i + 1 to i
      set the sign of this link to '?'
   else if i is not a conditional instruction
      foreach register r used in an argument to i's operation
         if dict[r] is set then
            add a link from instruction dict[r] to i
            set this link's sign as per table 4.2
         end if
      end foreach

      set dict[i.destinationIndex] = i
   end if
end foreach

// Build the output links for the final register values:
foreach output register r which represents a class
   if dict[r] is set then
      add an output link from the instruction dict[r] to register r
      set the sign of this link to 1
   end if
end foreach
```

Figure 7.2: The link building algorithm, used in the hill climbing algorithm.

error in the final register value of r_2 if its output is "filtered" through fewer instructions when determining the final value of r_2 than when determining the final value of r_1. Because the graph is acyclic though the value of δ_j will never depend on the value of δ_i if the value of δ_i depends on the value of δ_j.

Calculation of Δ_i

Δ_i is referred to colloquially as "the mean δ of instruction i" and is calculated for instruction i as the mean of δ over all fitness cases in the training set. More formally:

$$\Delta_i = \overline{\delta}_i \text{ over all fitness cases} \tag{7.3}$$

Weight Adjustment Algorithm for instruction i, given some Δ_i

For some instruction i, given its Δ_i we adjust its weight w_i through the use of a tan-sigmoid function f_{tan} scaled by some constant η (equation 7.4). η serves simply to moderate the size of the steps taken when the hill climbing is done and is identical in function to the role played by η in standard feed-forward neural network back propagation [45]. f_{tan} is bi-polar (see figure 7.3, which compares tan-sigmoid and log-sigmoid, the traditional activation function in neural networks [45]). In a continuous fashion and as with η it serves to scale Δw_i to reasonable values, but in a non-linear manner and regardless of the value of Δ_i. As can be seen from equation 7.5, tan-sigmoid also takes a second parameter, γ. The ideal value of γ depends on the desired rate at which f_{tan} should approach its maxima (1.0) and minima (-1.0). From empirical search a value of 2.0 was found to work well with the values of Δ that occurred. All information on tan-sigmoid has been sourced from the online lecture notes at [10].

$$\Delta w_i = \eta \times f_{tan}(\Delta_i, \gamma) \tag{7.4}$$

$$f_{tan}(\Delta_i, \gamma) = \frac{2}{1 + e^{-\gamma \Delta_i}} - 1 \tag{7.5}$$

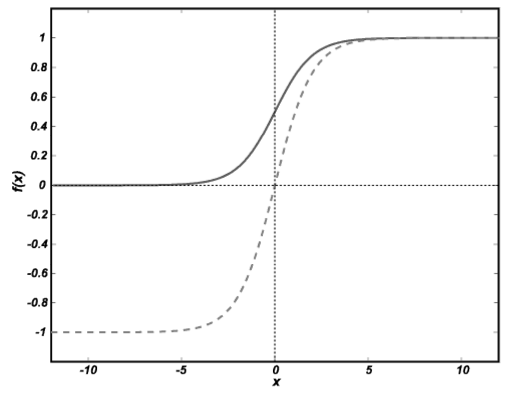


Figure 7.3: The log-sigmoid (top) and tan-sigmoid functions.

7.3.3 Evolutionary Integration

Explicit hill climbing can be applied to any members of any population at any time and there are therefore obviously a large number of possible ways of integrating hill climbing and an evolutionary beam search. Four possible approaches were investigated in this research:

1. Apply hill climbing to every member of the population after each full evaluation of the training set but before the performance of any evolutionary operations (*evolutionary offline hill climbing*). Whether or not the hill climbing leads to improved performance is not considered - all adjustments to each instruction's weights are retained. Whether or not it improves performance is not considered because this would require re-evaluating the entire training set for each individual, which would double the number of instructions which needed to be executed in any one generation. This would be too inefficient.

2. Apply hill climbing to every member of the population prior to any evolutionary operations taking place, i.e. just after the initial generation has been evaluated against the training set. Evaluate the population again before performing evolutionary operations (to update the fitness values). It is hoped that this approach (analogous to biological, Larmarckian evolution) would improve performance by accelerating the creation of useful building blocks.

3. Apply hill climbing after all evolution has terminated and before evaluating the best individual against the test set. Hill climbing is only applied if the best individual does not attain 100% accuracy on the training set. This approach is precisely analogous to

standard biological learning, in which an evolutionary beam search conducts the gross search and neural learning ("hill-climbing") locally optimises the results.

4. Apply hill climbing prior to any evolutionary operations (as with (2)) and also following all evolutionary operations if perfect fitness is not attained by the best individual on the training set (as with (3)). A biological analogy to this approach would be Larmarckian evolution with intra-lifetime skill acquisition also considered. It is hoped that this approach will improve performance by both precipitating the evolution of useful building blocks early on and by then locally optimising the final result.

Of these approaches (1) was found to be clearly inferior, leading to worse and more variable performance. This is because many of the evolutionary operations — particularly on highly fit individual programs — did not improve fitness and occasionally worsened it. In addition, this integration was also the most computationally expensive. The results for (2), (3) and (4) are presented in table 7.3. The configuration of the LGP runs for the results presented in this table was identical to that presented in table 7.4. All runs were done on the mid-level difficulty data set *digits15* and with an η and *PropScale* of 0.25.

Table 7.3: Effect of hill climbing at different times on *digits15* data set.

Method	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
Before evolution (2)	64.11% $\pm$ 4.78%	59.50% $\pm$ 6.03%
After evolution (3)	65.00% $\pm$ 4.89%	61.08% $\pm$ 6.66%
Before and after evolution (4)	65.83% $\pm$ 5.57%	61.84% $\pm$ 5.94%

The results presented in table 7.3 suggest that learning both before and after all evolution is the better method for the *digits15* data set. This method had the best training fitness, the best test fitness and lowest test variation. However, this integration of hill climbing will almost never speed up the evolution of a good solution in most cases. This is because — except in the rare circumstances a perfect solution to the test set can be hill climbed prior to any evolution — further hill climbing will only be done after all evolution has taken place and only if the maximum number of generations has been used and a perfect solution has not been found. Despite this, method (4) was used for all data sets in the experiment whose results are outlined in section 7.4 and section 7.5 below.

7.4 Experimental Configuration

The hill climbing algorithm given above in section 7.3 was evaluated 50 times with each configuration against three data sets — *faces1*, *shape* and *digits15*. Nine pairs of values of η and *PropScale* were trialled and the results are presented below. Excluding the different values of η and *PropScale* the same LGP configuration was used for each data set. For the purposes of comparison a set of 50 runs with no hill climbing but with an otherwise identical configuration was used for each data set. The use of additional, non-representative registers has not been explicitly considered in this algorithm. This is because these extra registers were found to generally have no or a negative impact on performance (see section 6.4). However, no modification of the link building algorithm would be necessary to incorporate them. The other configuration details for each data set are presented in table 7.4. The pairs of values of η and *PropScale* are presented in the results tables, tables 7.5, 7.6 and 7.7.

Table 7.4: The LGP configuration for the hill climbing experiments.

	shape	*digit15*	*faces1*
pop_size	500	500	500
max_program_length	16	40	20
elitism_rate	10%	10%	10%
crossover_rate	30%	30%	30%
macromutation_rate	30%	30%	30%
micromutation_rate	30%	30%	30%
tournament_size	4	4	4
α	1.15	1.15	1.15
β	0.18	0.18	0.18

7.5 Results and Discussion

The results of applying this hill climbing algorithm to each of the three data sets evaluated are presented along side results found which used no hill climbing to facilitate an evaluation of the hill climbing algorithm.

Table 7.5 shows the results of running the hill climbing algorithm against the *shape* data set. Of the three data sets evaluated statistics regarding the learning time (number of generations used) are presented only for this data set. This is because each run for the other two data sets always used the maximum number of generations.

Table 7.5: Hill climbing results for the *shape* data set.

PropScale	η	Generations ($\mu \pm \sigma$)	Training Acc. % ($\mu \pm \sigma$)	Test Acc. % ($\mu \pm \sigma$)
0.25	0.25	27.16 ± 13.86	97.94% ± 6.62%	97.78% ± 6.78%
0.25	0.50	27.64 ± 14.14	98.74% ± 4.60%	98.78% ± 4.34%
0.25	0.75	28.84 ± 14.94	98.79% ± 4.97%	98.71% ± 4.97%
0.50	0.25	25.60 ± 14.73	99.38% ± 3.56%	99.38% ± 3.57%
0.50	0.50	24.12 ± 13.18	99.89% ± 0.53%	99.81% ± 0.53%
0.50	0.75	20.96 ± 15.05	99.86% ± 0.72%	99.75% ± 0.94%
0.75	0.25	26.56 ± 16.88	97.27% ± 7.36%	97.26% ± 7.34%
0.75	0.50	25.20 ± 14.59	99.64% ± 1.45%	99.66% ± 1.12%
0.75	0.75	24.20 ± 14.75	98.85% ± 5.74%	98.68% ± 6.08%
No HC		20.36 ± 12.11	99.65% ± 2.43%	99.58% ± 2.66%

Table 7.6 shows the results of running the hill climbing algorithm on the *digits15* problem. Table 7.7 shows the results of running the hill climbing algorithm on the *faces1* problem.

The results of the application of this hill climbing algorithm to *shape*, *digits15* and *faces1* show that the best parameters give improved accuracy (by approximately 2%, uniformly across each of the data sets) for all data sets and at the cost of only a slight increase in variance. These improved results show that *PropScale* = 0.50 and $\eta = 0.75$ is a good starting point for the parameters to the algorithm.

Interestingly there might not have been any improvement in the number of generations needed for a population to learn *shape*. This result is confusing as earlier results (table 7.3) showed that the Larmarckian learning with intra-lifetime skill acquisition was better. Con-

Table 7.6: Hill climbing results for the *digits15* data set.

PropScale	η	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
0.25	0.25	65.83% ± 5.57%	61.84% ± 5.94%
0.25	0.50	64.18% ± 4.29%	59.44% ± 5.33%
0.25	0.75	64.76% ± 3.71%	60.56% ± 5.30%
0.50	0.25	66.06% ± 4.36%	62.02% ± 4.88%
0.50	0.50	65.56% ± 4.15%	61.32% ± 4.81%
0.50	0.75	66.00% ± 4.03%	62.66% ± 5.39%
0.75	0.25	64.60% ± 5.43%	60.44% ± 5.90%
0.75	0.50	65.92% ± 5.34%	61.67% ± 6.02%
0.75	0.75	65.19% ± 4.17%	60.90% ± 5.62%
No HC		64.96% ± 4.99%	60.53% ± 5.87%

Table 7.7: Hill climbing results for the *faces1* data set.

PropScale	η	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
0.25	0.25	55.42% ± 5.25%	48.48% ± 9.60%
0.25	0.50	56.21% ± 5.13%	48.85% ± 8.91%
0.25	0.75	56.32% ± 5.57%	48.97% ± 10.41%
0.50	0.25	56.96% ± 6.07%	50.85% ± 10.61%
0.50	0.50	55.61% ± 5.46%	47.70% ± 9.60%
0.50	0.75	57.17% ± 5.96%	52.48% ± 10.77%
0.75	0.25	56.12% ± 5.44%	52.36% ± 10.60%
0.75	0.50	57.15% ± 5.46%	50.85% ± 10.32%
0.75	0.75	57.09% ± 6.21%	49.88% ± 10.99%
No HC		57.34% ± 5.03%	50.85% ± 9.35%

firming whether or not this is the case will require a generation-by-generation analysis of fitness. It may be though that the hill climbing algorithm can only help in the construction of building blocks on data sets with certain characteristics. The improvements to the mean accuracy observed in the results may only be due to the result of hill climbing on individuals with imperfect training fitness (although this still leaves the integration results in table 7.3 unexplained).

The similar and improved relationship (when compared with the no-hill climbing results) between training and test accuracy for *digits15* and *faces1* is also significant. These results indicate that the hill climbing algorithm actually seems to reduce any overfitting that does occur. Comparing the hill climbing and non-hill climbing results for *shape* do not show this same trend, however the fact that the accuracy is already so close to perfect makes basing any conclusions on this relation in this data set very difficult.

In addition, the form of evolutionary integration adopted used the hill climbing algorithm sparingly (at the beginning and at the end only if perfect fitness on the training set was not attained). This means that the integration of the hill climbing algorithm imposed no significant penalties on the time taken to perform the evolution.

7.6 Chapter Summary

In this chapter the fourth research goal described in section 1.2 has been achieved. The problem of evolutionary beam search results not *necessarily* being locally optimised has been addressed through the development of a hill climbing algorithm for LGP and an evaluation of its value for three problems with different levels of difficulty.

This algorithm represents an important contribution in two ways. Firstly, it improves the LGP methodology developed in chapters 4, 5, 6 and analysed in chapter 8. Importantly, it achieves this at little computational cost. Secondly, the research carried out on evolutionary integration and the example this algorithm provides illustrate some of the other ways in which an evolutionary beam search and a hill climbing algorithm can be integrated. These offer important and exciting areas of future research and suggest further improvement over the excellent results achieved is possible.

Chapter 8

A Comparison of LGP and TGP

8.1 Overview

In this chapter the goal of analysing and comparing the performance of the LGP methodology in multi-class classification problems is achieved. A configuration developed using the methodology described in chapters 4, 6 and 5 (but excluding the hill climbing algorithm developed in chapter 7) is compared to a standard TGP configuration for a number of multi-class classification problems.

The structure of this chapter is as follows. First an LGP configuration is developed for each of the data sets being considered. Then, using the heuristic given in section 4.4 a comparable TGP configuration is developed. The maximum TGP tree depth is rounded up so that the comparison between LGP and TGP is biased slightly towards TGP. Following that the experimental configuration and results will be presented and discussed. As part of this discussion the understandability of an LGP program is compared to the understandability of a TGP program of the same fitness.

8.2 Data Sets

The comparison between LGP and TGP was done using the following data sets, described in chapter 3:

- *shape*
- *digits15*
- *digits30*
- *faces1*
- *faces2*
- *faces3*

Ten-fold cross validation [22] was used on the three *faces* data sets. The results presented in table 8.3 are the mean and standard deviation of 50 runs of a maximum of 50 generations against each data set.

8.3 LGP Configuration

The configuration used for each LGP run against each data set is presented in table 8.1. The evolutionary operators discussed in section 4.7 were used in these experiments, as was the fitness function developed in chapter 5. The program length was set according to the class-length heuristic discussed in section 6.5.3.

Table 8.1: The LGP Configuration for the comparison experiments.

	shape	*digit15*	*digits30*	*faces1*	*faces2*	*faces3*
pop_size	500	500	500	500	500	500
max_program_length	16	40	40	20	20	20
elitism_rate	10%	10%	10%	10%	10%	10%
crossover_rate	30%	30%	30%	30%	30%	30%
macromutation_rate	30%	30%	30%	30%	30%	30%
micromutation_rate	30%	30%	30%	30%	30%	30%
tournament_size	4	4	4	4	4	4
α	1.15	1.15	1.15	1.15	1.15	1.15
β	0.18	0.18	0.18	0.18	0.18	0.18

8.4 TGP Configuration

The LGP approach developed in this work is compared to the basic TGP approach [24]. The ramped half-and-half method was used for initial generation of the TGP programs [1] and the conversion heuristic was used to help find comparable configurations. The tournament selection mechanism and the reproduction, crossover and mutation operators [24] were used in the learning and evolutionary process. As with LGP, the mix of evolutionary operators which gave the best results are presented here. The program output was translated into a class label according to the static class boundary determination method [29]. The TGP system used the same terminal sets, function sets, fitness function, population size and termination criteria for the three data sets as the LGP approach. Other configuration details are specified in table 8.2.

Table 8.2: The TGP configuration for the comparison experiments.

	shape	*digit15*	*digits30*	*faces1*	*faces2*	*faces3*
pop_size	500	500	500	500	500	500
program_depth	3–5	4–6	4–6	3–5	4–6	4–6
elitism_rate	10%	10%	10%	10%	10%	10%
crossover_rate	60%	60%	60%	60%	60%	60%
mutation_rate	30%	30%	30%	30%	30%	30%
tournament_size	4	4	4	4	4	4
α	1.15	1.15	1.15	1.15	1.15	1.15
β	0.18	0.18	0.18	0.18	0.18	0.18

8.5 Experimental Results and Discussion

The results of these experiments are presented in table 8.3 and discussed in this section.

Table 8.3: The results from comparing equivalent TGP and LGP configurations.

Data Set	Method	Generations	Training Accuracy % ($\mu \pm \sigma$)	Test Accuracy % ($\mu \pm \sigma$)
shape	TGP	41.92 ± 14.48	85.04% ± 16.49%	84.41% ± 17.17%
	LGP	24.10 ± 13.85	99.80% ± 0.79%	99.73% ± 0.96%
digits15	TGP	50	53.57% ± 5.96%	47.84% ± 5.63%
	LGP	50	66.26% ± 4.62%	62.07% ± 5.81%
digits30	TGP	50	41.29% ± 4.16%	33.71% ± 4.79%
	LGP	50	55.93% ± 6.33%	50.46% ± 6.96%
faces1	TGP	50	59.17% ± 6.59%	50.76% ± 11.18%
	LGP	50	57.39% ± 5.17%	51.62% ± 10.13%
faces2	TGP	50	34.31% ± 5.17%	25.65% ± 8.05%
	LGP	50	37.47% ± 2.87%	29.75% ± 6.79%
faces3	TGP	50	27.94% ± 4.44%	21.82% ± 5.36%
	LGP	50	29.96% ± 2.68%	25.85% ± 5.99%

8.5.1 Performance

Two key results emerge from the performance documented in table 8.3. Firstly, it is important to note the much better performance of the LGP configuration than the TGP configuration on the *shape* and *digits* problems. The LGP configuration seems to use roughly half as many generations on the *shape* data set and the mean final accuracy is markedly better on this data set and the *digits15* and *digits30* data set.

The performance on the three *faces* data sets is interesting because the difference in performance is less significant. The LGP configuration still performs significantly better than the TGP configuration does but the difference is not as large, especially on the *faces1* task. This could be for a number of reasons. It may be the case that neither the TGP nor the LGP configuration is well suited to these tasks. It may however be the case though that the feature set constrains how good the solution can be. Given that the feature set was selected to be deliberately difficult this is a possibility which cannot be discounted.

Specifically referring to the *faces1* problem, it is hypothesised that the difference is less due to the smaller number of classes (which makes it more suitable for the TGP method using static class boundary determination) and due to the features not being the ideal feature set, a choice made to ensure the problem retained its difficulty.

It is thought that the better overall performance of LGP when compared to TGP is due to the greater suitability of the LGP methodology for multi-class classification. This methodology means that only a much simpler output interpretation algorithm (winner-takes-all) needs to be learnt, and it uses a representation that is probably much more appropriate.

8.5.2 Understandability

In this section the relative understandability of a TGP and LGP program for the *shape* problem will be examined and their understandability compared. Two typical programs with identical fitness are shown in this section.

An LGP Program

An LGP program with perfect fitness on the shape problem is provided in figure 8.1. Due to the simple nature of this task each class can be classified in a reasonably independent manner, although it is important to note the interdependence and sub-program reuse which does exist and also the multiple reuse of many of the same features (rather than the use of more different features).

Structural introns, discussed in section 4.3, are commented out using `//`. The DAG representation of the program is shown in figure 8.1 (right) after the introns are removed. It is important to note the presence and distribution of the introns in this program and how they reduce the chance of a destructive evolutionary operation [36, 35].

```
r[2] = r[1] - cf[1];
//if(r[1] < r[1])
//r[1] = r[2] / r[1];
//r[3] = r[2] - r[1];
r[1] = r[2] - cf[1];
//if(0.574554 < cf[5])
//r[3] = r[0] * r[1];
r[3] = cf[3] * r[1];
//r[2] = r[1] + cf[7];
r[1] = r[3] + 0.8399;
//r[3] = cf[3] + cf[1];
r[0] = cf[2] / cf[5];
r[0] = r[0] + 0.617964;
r[3] = cf[5] + cf[5];
r[2] = 0.714758 / cf[1];
r[1] = r[3] / r[1];
```

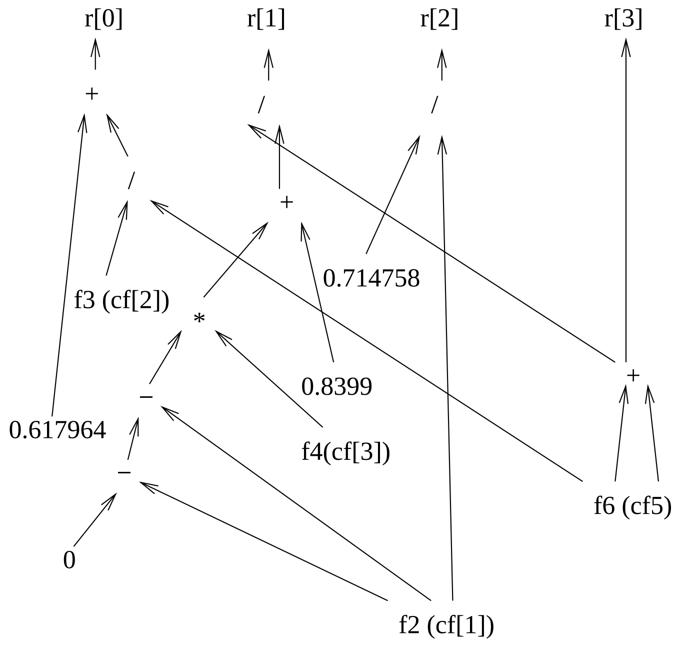


Figure 8.1: An LGP program with perfect fitness on the *shape* data set. Structural introns are commented out and not shown in the DAG representation (right).

The role played by a limited sub-set of the features is made clear, especially by the DAG representation (figure 8.1, right). Note that only four of the features are used to discriminate the classes in this LGP program: the mean brightness of three of the quadrants and the mean brightness of the centre quadrant. This suggests that the LGP approach can automatically select features relevant to a particular task. The DAG representation of the program (figure 8.1, right) shows that the LGP approach can co-evolve sub-programs together each for a particular class and that some terminals and functions can be reused by different sub-programs. In addition, the process used to calculate the ″number of votes“ for each of the classes is fairly straightforward and understandable to a human, as is visible in the DAG in figure 8.1.

An example feature vector for each class is provided in figure 8.2. In this configuration the 8 feature registers are represented by the array `cf` in the indexes `cf[0]` to `cf[7]`, inclusive. The registers `r[0]`, `r[1]`, `r[2]`, and `r[3]` represent `class1`, `class2`, `class3`, and

```
                cf[0]  cf[1]  cf[2]  cf[3]  cf[4]  cf[5]  cf[6]  cf[7]
----------------------------------------------------------------------
Obj1 (class1): 0.2720 0.3139 0.2748 0.2905 0.2087 0.1063 0.4658 0.2174
Obj2 (class2): 0.6425 0.6253 0.6469 0.6394 0.6583 0.7013 0.1743 0.0912
Obj3 (class3): 0.2727 0.2968 0.2865 0.2750 0.2325 0.2227 0.2296 0.1043
Obj4 (class4): 0.8124 0.7730 0.8057 0.8137 0.8571 0.8568 0.2296 0.1049
```

Figure 8.2: Example feature vectors of each class in the *shape* data set. The `cf` registers contain the 8 features described in section 3.1 in registers `cf[0]` to `cf[7]`.

`class4` respectively. After evaluating the LGP program the final values of the four registers are shown in table 8.4. These results show how the results for a TGP program are not just understandable to a human but also easily calculated and manually verified.

Table 8.4: The final register values of the LGP program in figure 8.1. This table shows the final register values of the LGP program given in figure 8.1 after classifying the examples given in figure 8.2. The register values with the largest values and representing the classified class (through the use of a winner-takes-all algorithm) are bolded.

Object	True-Class	`r[0]`	`r[1]`	`r[2]`	`r[3]`	Classified-Class
Obj1	class1	**3.2009**	1.5404	1.9045	1.5583	class1
Obj2	class2	0.3236	**34.7961**	0.6582	-4.0985	class2
Obj3	class3	2.2771	1.1431	**2.4079**	0.9246	class3
Obj4	class4	0.2127	1.4026	0.4454	**1.7136**	class4

A TGP Program

On the other hand, a program evolved by the TGP approach can only produce a single value, which must be translated/interpreted into a set of class labels. A TGP program with perfect fitness on the shape problem is provided in figure 8.3. Note how it uses every feature. How each feature contributes to the classification of an object as any particular class is unclear. In addition, human verification of the results is extremely difficult: calculating the final value of the tree is difficult and error prone and having done so one also needs to bear the output interpretation algorithm in mind. Using one of the often more accurate approaches (e.g. [57, 58, 61]) may not make the problem less difficult as these methods have a much more complex output interpretation algorithm than the basic static class boundary determination method.

```
(ifltz (ifltz (ifltz (+ f8 -0.276997) (- f3 f6) (+ f1 f5))
              (/     (- 0.825937 f6) f4)
              (*     (- f1 f7) (+ f1 -0.566267)))
       (/     (*     (ifltz f6 f7 f2) (/ f4 f3))
              (/     (ifltz f1 f2 f4) (+ f1 f5)))
       (-     (ifltz (/ f3 f4) (/ f3 f3) (/ f6 0.430889))
              (+     (/ 0.384887 0.425971) (/ f3 f2)))
)
```

Figure 8.3: An example TGP program with perfect fitness on the *shape* data set.

8.6 Chapter Summary

In this chapter the key contributions made in the previous chapters are made clear. The new LGP methodology outperformed a standard TGP methodology on all of the problems considered and significantly outperformed it on five of the problems considered. This chapter also highlights how much more easily LGP programs can be understood by a human than a TGP program of comparable performance. This understandability is another very important contribution, particularly in the fields of data mining and knowledge discovery.

To summarise, the results from the comparison of LGP to TGP in this chapter highlight how strongly the first, second and third goals described in section 1.2 have been addressed and how the problems outlined in section 1.1 have been resolved.

Chapter 9

Conclusions

In this chapter the various results presented and discussed in chapters 4–8 are synthesised and summarised into a number of overall conclusions and recommendations.

In this research a number of the problems faced by standard TGP for multi-class classification have been considered. A new methodology and configuration for LGP designed to work well for multi-class classification has been developed which addresses these. As part of developing this methodology an output interpretation algorithm and a crossover operator were created.

The strength of this methodology has been assessed on a range of multi-class object classification problems of varying difficulty. A theoretical analysis of fitness functions and a new fitness function designed to work well with tasks which have a complex feature space has also been developed and analysed. A hill climbing algorithm which ensures local optimisation and two heuristics which aid in creating TGP and LGP configurations in a number of situations have been created.

9.1 An LGP Methodology for Multi-class Classification

A methodology for using LGP to solve multi-class classification problems was developed. This methodology was applied to six multi-class image classification problems of differing difficulty. These problems had greatly varying difficulty, numbers of features and numbers of classes and are possibly representative of image classification tasks. Six tasks are used to examine the applicability of LGP to multi-class image classification. An output interpretation algorithm using the winner-takes-all methodology and a crossover operator based on GA two-point crossover [15] were developed.

A program length heuristic was developed to guide the selection of the initial maximum program length in an LGP configuration. This heuristic is interesting because the difficulty of the problem is not directly considered in the heuristic. Only the number of classes a problem has is a factor in determining the suggested length of the program.

In addition, research was conducted on the optimal number of registers, relative to the number of classes. This research suggests that having no extra registers did not bring any penalty and often led to slightly better results.

A methodology which generated all programs at the maximum length allowed was created. This included the development of a crossover operator which ensured that a greater proportion of the crossover operations were productive. Only if exactly and only the instructions swapped over were randomly culled would the crossover regress to a reproduction.

The results showed that the LGP methodology out-performed the TGP methodology on all six of the problems considered. In addition, the programs evolved were much more understandable and LGP performed significantly better than TGP on five of the problems.

9.2 Fitness Functions in Multi-class Classification

A theoretical analysis of the role played by a fitness function was undertaken. Three ways in which a multi-class classification problem can be difficult were discussed. Inspired by the theoretical analysis, one of these — the hurdle problem, which occurs when a feature space is particularly complex — was successfully addressed through the development of a new fitness function, f_{decay}.

This new fitness function leads to much more accurate results for both TGP and LGP and also much more consistent results for LGP. This research suggests careful analysis and tailoring of the fitness function to the kind of problem being addressed is a valuable aspect of the process of developing a GP configuration for a problem.

9.3 LGP Compared to TGP

The LGP methodology and configuration was compared to the standard TGP methodology and configuration for multi-class classification. The new LGP methodology and configuration was found to be better on all six problems evaluated. The difference was small only for a problem which was very difficult but which had few classes (meaning a TGP approach was less inappropriate). Even in this situation LGP still performed better than TGP.

A conversion heuristic was developed to ensure LGP and TGP configurations had comparable expressivity. By doing this other elements such as the relative suitability and rate of learning of LGP and TGP for particular problems could be compared.

This research has suggested that LGP is generally better for multi-class classification than TGP in a wide range of domains, particularly when there are a large number of classes in the problem.

9.4 LGP and Hill Climbing

A new program structure and hill climbing algorithm was developed for the LGP configuration and methodology earlier research suggested and discussed above in sections 9.1 and 9.2. This algorithm was found to give the expected improvements in the results. Due to the way in which the hill climbing and evolutionary beam search were integrated the hill climbing could be done with little extra computation and was relatively inexpensive.

This algorithm could be applied to any LGP problem and in some other types of problems (such as function regression) it may show even greater benefits. It is hypothesised that different evolutionary integrations would be more appropriate for other types of problem.

9.5 Future Work

In this section the areas of possible future work will be considered. They will be presented in the same structure as the conclusions were in sections 9.1–9.4. No further work regarding the comparison of TGP and LGP will be detailed as the ways in which this can be carried out are relatively obvious and also highly problem-dependent.

9.5.1 Methodology

There are a number of areas where the methodology developed could be extended.

The class-length heuristic could be investigated in more detail, especially the reasons why the multiple of 3.5–4 times the length was most appropriate. It is hypothesised that as the length gets longer the proportion of introns rises and that there is therefore a decreasing marginal benefit [6] in increasing the maximum program size allowed.

Similarly, the generation of programs at their initial maximum length could also be investigated further. The cursory empirical results which motivated this decision could be supported by further theoretical and empirical research.

The value of allowing the evolution to proceed over more generations needs to be investigated more formally. Observation of the generation-by-generation records of best and average fitness show a clearly decreasing benefit in allowing evolution to continue from generation-to-generation, however past research [12] has suggested that longer evolutionary periods may be valuable in some situations.

The multiple-output (*multi-out*) methodology developed could also be extended in a number of other ways. It could be applied to other kinds of problems which require multiple inter-dependent answers, such as the stock prediction problem discussed in section 2.2.

A further extension of this multi-out structure would be a generalisation which allowed recurrent graphs to be evolved. This could be particularly valuable for prediction problems with a temporal component.

9.5.2 Fitness Functions in Multiclass Classification

A fuller theoretical analysis of the fitness function has been given in this work. This analysis of the two types of space (the feature space and the program space) could be generalised and applied to other kinds of problems to aid in developing more appropriate fitness functions for problems in other domains. Further examination of the differences and similarities between the problems caused by local peaks and minima and the problems caused by a hurdle still needs to be done. Likewise, ways the first and second difficulties described in sections 5.4.1 and 5.4.2 can be addressed also need to be considered explicitly.

In addition, the key idea of differentially valuing different levels and kinds of success, used in f_{decay}, could be particularly applicable to problems such as the Santa Fe ant trail problem [23].

9.5.3 Hill Climbing

Extra registers which are not involved in the interpretation of an LGP program's output were not explicitly considered when the hill climbing algorithm was developed. Although the existing algorithm would not need to be changed in order to allow them to be incorporated the impact of these additional registers on the value of hill climbing and the results which can be obtained still need to be investigated.

In addition, the best way in which to integrate the evolutionary beam search and the hill climbing needs to be considered in more detail and for a wider range of problems. The current results are confusing and unclear. To achieve this clarification the impact of various evolutionary integrations of hill climbing on the rate at which a population can learn needs to be investigated in more detail with a generation-by-generation analysis of fitness being carried out.

Finally, different program structures and approaches to hill climbing for LGP also need to be investigated.

Bibliography

[1] 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, dpunkt.verlag, Jan. 1998.

[2] BRAMEIER, M. *On Linear Genetic Programming*. PhD thesis, University of Dortmund, Dortmund, Germany, February 2004.

[3] BRAMEIER, M., AND BANZHAF, W. A comparison of genetic programming and neural networks in medical data analysis. Reihe CI 43/98, SFB 531, Dortmund University, Germany, 1998.

[4] BRAMEIER, M., AND BANZHAF, W. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Trans. Evolutionary Computation 5*, 1 (2001), 17–26.

[5] BRAMEIER, M., AND BANZHAF, W. Effective linear genetic programming. Tech. rep., Department of Computer Science, University of Dortmund, 44221 Dortmund, Germany, 2001.

[6] BREALEY, R. A., AND MYERS, S. C. *Principles of Corporate Finance*. McGraw-Hill Higher Education, 2000.

[7] BROWN, DAVID (MD). Deciphering the message of life's assembly. http://wsrv.clas.virginia.edu/ rjh9u/protfold.html, Oct. 2005.

[8] CRISTIANINI, N., AND SHAWE-TAYLOR, J. *An introduction to support vector machines*. Cambridge University Press, 2000.

[9] DENNETT, D. C. Real patterns. *Journal of Philosophy 88* (1991), 27–51.

[10] EE490, D. C. U. Sigmoid function. http://wsrv.clas.virginia.edu/ rjh9u/protfold.html, Oct. 2005.

[11] EGGERMONT, J., EIBEN, A. E., AND VAN HEMERT, J. I. A comparison of genetic programming variants for data classification. In *Proceedings of the Third Symposium on Intelligent Data Analysis (IDA-99), LNCS 1642* (1999), Dpringer-Verlag.

[12] GATHERCOLE, C., AND ROSS, P. Small populations over many generations can beat large populations over few generations in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (Stanford University, CA, USA, 13-16 July 1997), J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, Eds., Morgan Kaufmann, pp. 111–118.

[13] GEORGHIADES, A. S. Yale face database. http://cvc.yale.edu/projects/yalefacesB/yalefacesB.html, Oct. 2005.

[14] Georghiades, A. S., Belhumeur, P. N., and Kriegman, D. J. From few to many: Illumination cone models for face recognition under variable lighting and pose. *IEEE Trans. Pattern Anal. Mach. Intelligence 23*, 6 (2001), 643–660.

[15] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison–Wesley, Reading, MA, 1989.

[16] Gray, H. Genetic programming for classification of medical data. In *Late Breaking Papers at the 1997 Genetic Programming Conference* (1997), J. R. Koza, Ed., Standford University, pp. 291–297.

[17] Hawai, L. A. B2.7.2 nk fitness landscapes.

[18] Howard, D., Roberts, S. C., and Brankin, R. Target detection in sar imagery by genetic programming. *Advances in Engineering Software 30* (1999), 303–311.

[19] Howard, D., Roberts, S. C., and Ryan, C. The boru data crawler for object detection tasks in machine vision. In *Applications of Evolutionary Computing, Proceedings of EvoWorkshops2002: EvoCOP, EvoIASP, EvoSTim* (Kinsale, Ireland, 3-4 Apr. 2002), S. Cagnoni, J. Gottlieb, E. Hart, M. Middendorf, and G. Raidl, Eds., vol. 2279 of *LNCS*, Springer-Verlag, pp. 220–230.

[20] Kantschik, W., and Banzhaf, W. Linear-tree gp and its comparison with other gp structures. *Proceedings of 4th EuroGP Conference, Como 2001, Springer, Berlin* (2001), 302–312.

[21] Kinnear, Jr., K. E. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence* (Orlando, Florida, USA, 27-29 1994), vol. 1, IEEE Press, pp. 142–147.

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

[23] Koza, J. R. *Genetic Programming*. MIT Press, Campridge, Massachusetts, 1992.

[24] Koza, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge, Mass. : MIT Press, London, England, 1994.

[25] Lee, J.-S., and Oh, I.-S. Binary classification trees for multi-class classification problems. In *ICDAR* (2003), pp. 770–774.

[26] Lett, M., and Zhang, M. New fitness functions in genetic programming for object detection. In *Proceedings of Image and Vision Computing International Conference* (2004), pp. 441–446.

[27] Li, J., and Tsang, E. P. K. Investment decision making using FGP: A case study. In *Proceedings of the Congress on Evolutionary Computation* (Mayflower Hotel, Washington D.C., USA, 6-9 July 1999), P. J. Angeline, Z. Michalewicz, M. Schoenauer, X. Yao, and A. Zalzala, Eds., vol. 2, IEEE Press, pp. 1253–1259.

[28] Liu, J. J., Cutler, G., Li, W., Pan, Z., Peng, S., Hoey, T., Chen, L., and Ling, X. B. Multiclass cancer classification and biomarker discovery using ga-based algorithms. *Bioinformatics 21*, 11 (June 2005), 2691–2697.

[29] 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.

[30] METROPOLIS, N. A., ROSENBLUTH, M., TELLER, A., AND TELLER, E. Equation of state calculations by fast computing machines. *Journal of Chemical Physics 21* (1956), 1087–1092.

[31] METZ, C. E. Roc methodology in radiologic imaging. *Investigative Radiology 21* (1986), 720–732.

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

[33] NANDURI, D. T., AND CIESIELSKI, V. Comparison of the effectiveness of decimation and automatically defined functions. In *KES (3)* (2005), pp. 540–546.

[34] NGUYEN, D. V., AND ROCKE, D. M. Multi-class cancer classification via partial least squares with gene expression profiles. *Bioinformatics 18*, 9 (2002), 1216–1226.

[35] NORDIN, P., BANZHAF, W., AND FRANCONE, F. Introns in nature and in simulated structure evolution. In *Bio-Computation and Emergent Computation* (Skovde, Sweden, 1-2 Sept. 1997), D. Lundh, B. Olsson, and A. Narayanan, Eds., World Scientific Publishing.

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

[37] OLTEAN, M., GROSAN, C., AND OLTEAN, M. Encoding multiple solutions in a linear genetic programming chromosome. In *Computational Science - ICCS 2004: 4th International Conference, Part III* (Krakow, Poland,, 6-9 June 2004), M. Bubak, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds., vol. 3038 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1281–1288.

[38] OOI, C. H., AND TAN, P. Genetic algorithms applied to multi-class prediction for the analysis of gene expression data. *Bioinformatics 19*, 1 (2003), 37–44.

[39] POLI, R. Genetic programming for image analysis. In *Genetic Programming 1996: Proceedings of the First Annual Conference* (Stanford University, CA, USA, 28–31 July 1996), J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, Eds., MIT Press, pp. 363–368.

[40] RENNIE, J. D. M. Improving multi-class text classification with naive bayes. Master's thesis, Massachusetts Institute of Technology, 2001.

[41] RENNIE, J. D. M., AND RIFKIN, R. Improving multiclass text classification with the Support Vector Machine. Tech. Rep. AIM-2001-026, Massachusetts Insititute of Technology, Artificial Intelligence Laboratory, 2001.

[42] RIDLEY, M. *The red queen: Sex and the evolution of human nature*. Macmillan, New York, 1995.

[43] ROUNDS, S. A. Development of a neural network model for dissolved oxygen in the Tualatin river, Oregon. In *Second Federal Interagency Hydrologic Modeling Conference* (7-8 2002).

[44] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation. In *Parallel distributed Processing, Explorations in the Microstructure of Cognition, Volume 1: Foundations*, D. E. Rumelhart, J. L. McClelland, and the PDP research group, Eds. The MIT Press, Cambridge, Massachusetts, London, England, 1986, ch. 8.

[45] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323*, 9 (1986), 533–536.

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

[47] Smart, W. Genetic programming for multi-class classification. Technical report, School of Mathematics, Statistics and Computer Science, VUW, Wellington, 2003.

[48] Smart, W., and Zhang, M. Continuously evolving programs in genetic programming using gradient descent. *Proceedings of 2004 Asia-Pacific Workshop on Genetic Programming* (2004).

[49] Smart, W., and Zhang, M. Probability based genetic programming for multiclass object classification. In *PRICAI 2004: Trends in Artificial Intelligence, Proceedings of the 8th Pacific Rim International Conference on Artificial Intelligence. Lecture Notes in Computer Science.* (2004), vol. 3157, pp. 251–261.

[50] Song, A., Loveard, T., and Ciesielski, V. Towards genetic programming for texture classification. In *Proceedings of the 14th Australian Joint Conference on Artificial Intelligence* (2001), Springer Verlag, pp. 461–472.

[51] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, Cambridge, Massachusetts, London, England, 1998.

[52] Tackett, W. A. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, Faculty of the Graduate School, University of Southern California, Canoga Park, California, USA, April 1994.

[53] Tomasello, M., Kruger, A. C., and Ratner, H. H. Cultural learning. *Behavioural and Brain Sciences 16* (1993), 495–552.

[54] Zhang, M. *A Domain Independent Approach to 2D Object Detection Based on the Neural and Genetic Paradigms*. PhD thesis, Department of Computer Science, RMIT University, Melbourne, Australia, 2000.

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

[56] Zhang, M., and Ciesielski, V. Genetic programming for multiple class object detection. In *Proceedings of the 12th Australian Joint Conference on Artificial Intelligence (AI'99)* (Sydney, Australia, December 1999), N. Foo, Ed., Springer-Verlag Berlin Heidelberg, pp. 180–192. Lecture Notes in Artificial Intelligence (LNAI Volume 1747).

[57] ZHANG, M., CIESIELSKI, V., AND ANDREAE, P. Cs-tr-02-4: A independent approach to multiclass object detection using genetic programming. Tech. rep., MSCS, Victoria University of Wellington, 2002.

[58] ZHANG, M., AND SMART, W. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC* (Coimbra, Portugal, 5-7 Apr. 2004), G. R. Raidl, S. Cagnoni, J. Branke, D. W. Corne, R. Drechsler, Y. Jin, C. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, Eds., vol. 3005 of *LNCS*, Springer Verlag, pp. 369–378.

[59] ZHANG, M., AND SMART, W. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT, EvoIASP, EvoMUSART, EvoSTOC* (Coimbra, Portugal, 5-7 Apr. 2004), G. R. Raidl, S. Cagnoni, J. Branke, D. W. Corne, R. Drechsler, Y. Jin, C. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, Eds., vol. 3005 of *LNCS*, Springer Verlag, pp. 369–378.

[60] ZHANG, M., AND SMART, W. Learning weights in gp using gradient descent for object classification. *Lecture Notes in Computer Science 3449* (2005), 417–427.

[61] ZHANG, Y., AND ZHANG, M. A multiple-output program tree structure in genetic programming. *Proceedings of 2004 Asia-Pacific Workshop on Genetic Programming* (2004).

Appendix A

VUWLGP

The research described in this report was carried out using a package developed by Christopher Fogelberg in March, April and May of 2005. The architecture and usage of this package is described in this appendix.

A.1 Code Overview

VUWLGP was developed in ANSI STL C++ and has a configurable architecture. Programs are represented by the `Program` class, and instructions are represented by the `Instruction` structure. A population of programs is represented and managed by one instance of the `Population` class. The `FitnessCase` class represents fitness cases and the `Fitness` class is responsible for organising, loading and managing the test, training and — if used — validation sets of fitness cases. Each of these classes will now be discussed briefly.

The `Instruction` structure encapsulates whatever components comprise an individual instruction and provides a number of methods. The output of these methods is (in normal usage) interpreted by methods of the `Program` class. The methods include those which return the operator, the index and type of register of any arguments and the destination register the result of calculating the operation the instruction represents should be stored in. There are also methods which convert an `Instruction` into a C-style string.

Any one program is an instance of the `Program` class. This class stores the sequence of instructions which comprise a program and also some associated data such as the individual's fitness, and its fitness on a per-class basis (which assists in investigating the hurdle problem, see chapter 5). The DAG of instructions and links necessary for the hill-climbing algorithm and which varies by class is also stored in an instance of this class.

Similarly, the `Program` class provides methods associated with this data and also methods associated with the execution and evolution of a program. These methods include those which zero, update and return the program's fitness, and also those which execute it against a particular `FitnessCase`, cross it over with another `Program` and apply micro- or macro-mutation operators.

The `Program::Execute` method is the area of tightest cohesion with the `Instruction` structure. In this method the values returned by the methods in `Instruction`, such as those identifying the operator or the type of array being used, are given their semantic meaning by the impact they have on the registers passed to this method and initialised with the features of a `FitnessCase`.

Each individual fitness case is represented by one instance of the `FitnessCase` class. This class stores the feature values of the fitness case it represents and also the correct outputs (final register values). In addition, it stores the relative importance of each of the final register values. This is valuable should VUWLGP be configured to run against a problem which has multiple outputs in which one of the outputs is less important and should therefore make a much smaller negative contribution to fitness than the others. The `Fitness` class simply represents a training, test and validation set of fitness cases.

Finally, the `Population` class serves as the general evolutionary interface with methods that evaluate a population of programs against a set of fitness cases (through the use of a `Fitness` objects) and which builds the next generation of `Program`s based on the current generation's fitnesses. This class also includes the methods which orchestrate the selection of individual programs from the population for use in evolutionary operations.

An application which uses VUWLGP must do the following steps in the following order:

1. Create a `Fitness` object, e.g. `f`. Tell it in which files it can find the fitness cases for test, training and validation.
2. Call `f.InitFitness`. This method will load the sets of fitness cases and ensure that the information specified on the command line or set as defaults in the `Config` class is correct.
3. Create an instance of the `Population` class, e.g. `p`. This class must be instantiated after the call to `Fitness::InitFitness` has made any corrections to the configuration that need to be made.
4. Until the termination criteria (perfect fitness on the training set or the maximum number of generations) is met evaluate the current population against the training set and evolve the next generation.
5. Evaluate the best individual against the test set and output the statistics of this run.

A number of other programmatic elements touched on already encapsulate other functionality. The generation of random numbers is managed by code in the namespace `Rand`. This code is relatively simple and will not be discussed further here. The way in which any one application of VUWLGP is configured is managed by the `Config` class, which is discussed in more detail in section A.2. How another operator — for example an if> operator — could be added is discussed in section A.3. Finally, the applications which manage and generate fitness cases from raw data are discussed in section A.4 and the text format of a fitness case is described in section A.5.

A.2 Configuration

The class `Config` represents most of the configuration details of any VUWLGP run against a problem and also contains the functions which parse these arguments if they are presented on the command line. Variables in this class define the following aspects of the configuration:

- The number of registers, features and classes.
- The maximum and minimum length of a program and the maximum length at the time of the initial generation.
- The size of the population and the maximum number of generations which can be used.
- The percentage of each type of each evolutionary operation.

In addition, a number of other minor details (such as the log path) are defined, and elements such as *PropScale*, α, and β (see chapter 7 and chapter 5) are also defined in this class if used.

A.3 Adding an Operator

Due to the division of some responsibilities amongst several classes in VUWLGP the addition of a new operator requires modifying 2 classes. This configuration was chosen (over the alternative of having the semantics stored in subclasses of `Instruction`) in order to create code which is as efficient as possible.

`Instruction::ExtractOpString` in the file `Instruction.h` needs to be modified, and the `ToString` methods (in the file `Instruction.cpp`) may also need to be modified.

In addition, the semantics of the new operation need to be represented in `Program::Execute`. The variable `NumOps` in this method needs to be incremented and a case for the number of the new operator needs to be added to the switch statement in this method.

A.4 Fitness Case Generation

The way in which fitness cases are generated for each type of problem are different, however a number of common features are encapsulated in the class `Pattern` and then processed by the code in the application defined in `Generator.cpp`. See or extend this code in order to understand in more detail how to generate patterns for other problems.

The `Pattern` class is encapsulated entirely in the file `Pattern.h`. An instance of this class can be constructed either from an existing text representation (described below) or by passing its constructor a number of parameters. These parameters are:

- The image from which the pattern came, or else some other human-readable name-type `std::string`. This string cannot contain any white space.
- The x and y coordinate of the image the pattern came from. Alternatively, any two unsigned integers which have some semantic meaning for the problem can be used.
- A `std::string` with no white space which is the name of the class this pattern is a member of.
- An `unsigned int` representing the number of the class this pattern belongs to.
- An `unsigned int` specifying the number of features in this fitness case/pattern.
- An array of `double` containing each of the features in this fitness case.

A.5 Fitness Case Text Format

The string-format of a pattern (fitness case) is as follows:

[image file or name] [x-position] [y-position] [class-number] [class-name] [f1] [f2] ... [fn]

Appendix B

Paper Accepted to the 18th Joint Australian Conference on Artificial Intelligence

This appendix contains a full copy of the paper submitted to the 18th Australian Joint Conference on Artificial Intelligence and accepted as a full paper. It will be published in the *Lecture Notes in Artificial Intelligence*.

Linear Genetic Programming for Multi-class Object Classification

Christopher Fogelberg and Mengjie Zhang

School of Mathematics, Statistics and Computer Sciences
Victoria University of Wellington, P. O. Box 600, Wellington, New Zealand
{fogelbchri,mengjie}@mcs.vuw.ac.nz

Abstract. Multi-class object classification is an important field of research in computer vision. In this paper basic linear genetic programming is modified to be more suitable for multi-class classification and its performance is then compared to tree-based genetic programming. The directed acyclic graph nature of linear genetic programming is exploited. The existing fitness function is modified to more accurately approximate the true feature space. The results show that the new linear genetic programming approach outperforms the basic tree-based genetic programming approach on all the tasks investigated here and that the new fitness function leads to better and more consistent results. The genetic programs evolved by the new linear genetic programming system are also more comprehensible than those evolved by the tree-based system.

1 Introduction

Image classification tasks occur in a wide variety of problem domains. While human experts can frequently accurately classify the data manually, such experts are typically rare or too expensive. Thus computer based solutions to many of these problems are very desirable.

Genetic Programming (GP) [1, 2] is a promising approach for building reliable classification programs quickly and automatically, given only a set of examples on which a program can be evaluated. GP uses ideas analogous to biological evolution to search the space of possible programs to evolve a good program for a particular task.

While showing promise, current GP techniques frequently do not give satisfactory results on difficult classification tasks, particularly multi-class classification (tasks with more than two classes). There are at least two limitations in currently used GP *program structures* and *fitness functions* that prevent GP from finding acceptable programs in a reasonable time.

The programs that GP evolves are typically tree-like structures [3], which map a vector of input values to a single real-valued output[4–6]. For classification tasks, this output must be mapped into a set of class labels. For binary classification problems, there is a natural mapping of negative values to one class and positive values to the other class. For multi-class classification problems, finding the appropriate boundaries on the number line to separate the classes is very difficult. Several new translations have recently been developed in the interpretation of the single output value of the tree-based GP [4, 7, 8], with differing strengths in addressing different types of problem. While these translations have achieved better classification performance, the evolution is still slow and the evolved programs are hard to interpret, particularly for more difficult problems or problems with a large number of classes.

In solving classification problems, GP typically uses the classification accuracy, error rate or a similar measure as the fitness function [5, 7, 8], which approximates the true fitness of an individual program. Given that the training set size is often highly limited, such an approximation frequently fails to accurately estimate the classification of the true feature space.

1.1 Goals

To address the problems above, this paper aims to investigate an approach to the use of linear genetic programming (LGP) and a new fitness function for multi-class object classification problems. This approach will be compared with the basic tree-based GP (TGP) approach on three image classification tasks of increasing difficulty. Specifically, we are interested in:

- Whether the LGP approach outperforms the basic TGP approach on these object classification problems in terms of classification performance.
- Whether the genetic programs evolved by LGP are more comprehensible.
- Whether the new fitness function improves the classification performance over the existing fitness function.

2 LGP for Multi-class Object Classification

2.1 LGP Overview

This work used register machine LGP (hereafter just LGP) [2], where an individual program is represented by a sequence of register machine instructions, typically expressed in human-readable form as C-style code.

Prior to any program being executed, the registers which it can read from or write to are zeroed. The features representing the objects to be classified are loaded into predefined registers. The program is executed in an imperative manner and represents a *directed acyclic graph* (DAG). This is different from tree-based GP which represents a tree. Any register's value may be used in multiple instructions during the execution of the program.

2.2 Multi-class Output Interpretation

An LGP program often has only one register interpreted in determining its output [9, 2]. This configuration can be easily used for regression and binary classification problems as in the tree-based GP.

In this work, we use LGP for multi-class object classification problems. We want an LGP program to produce one output for each class. Thus, instead of using only one register as the output, we use multiple registers each corresponding to one class. The winner-takes-all strategy is then used and the class represented by the register with the largest value is considered the class of the input object by that genetic program.

This program output representation for the different classes is very similar to a feed forward neural network classifier [10]. However, the structure of such an LGP program is more flexible than that of the feed forward neural network.

2.3 Evolutionary Operators

We used reproduction, crossover and mutation as genetic operators. In reproduction, the best programs in the current generation are copied into the next generation without any change.

We used two different forms of mutation [11] in this work. *Macromutation* involves the replacement of an entire instruction with a randomly generated one. *Micromutation* changes only either the destination register, a source register or the operation. These operations can cause dramatic changes in the DAG that a program represents [12].

In the crossover operator, we randomly choose a section from each of the two parents, then swap them to produce offspring. If a newly produced program is longer than the maximum length allowed, then an instruction is randomly selected and removed until the program can fit into the maximum length. This is similar to two-point crossover in GAs[13], but the two sections chosen from the parents can have different lengths here.

2.4 The Old Fitness Function and the Hurdle Problem

Given that the size of the training set must be finite, any fitness function can only be an approximation to an program's true fitness. This can lead to problems such as overfitting, where a program's true fitness is sacrificed for fitness on the training set. In a multi-class object classification problem, a program's true fitness is the fraction of the feature space it can correctly classify. A good fitness function is one which accurately estimates this fraction.

A typical fitness function for classification problem is the *error rate* of a program classifier. This was also used in our early experiments. While it performed reasonably well, this fitness function frequently fails to accurately estimate the fraction of the feature space correctly classified by a program.

Figure 1 shows a simple classification problem with two features. Figure 1(a) shows the true feature space — feature vectors of class c1 objects always appear in the fraction of the feature space denoted "c1", and similarly for the fractions denoted "c2" and "c3". Figure 1(b) shows that `program1` misclassifies two objects of c2 as c3. This program has an error rate of 18% (2/11). Figure 1(c) shows that `program2` misclassifies one object from class c3 and one object from

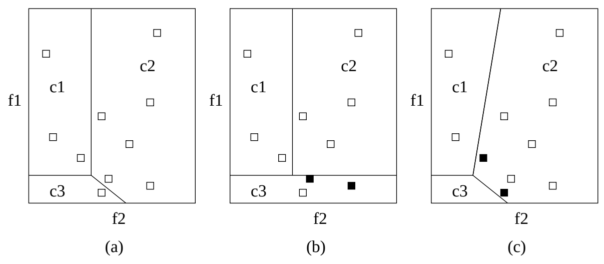


Fig. 1. The hurdle problem. Solid objects are misclassified.

class c1 as c2. This program also has an error rate of 18% and will be treated the same as the `program1`. `program2` actually approximated the true fitness more accurately than `program1`, but the fitness function cannot accurately reflect this difference.

We call this problem *the hurdle problem.* It occurs when (any two) classes have a very complex boundary in the feature space. In such a situation, it is easy to classify the bulk of fitness cases for one class correctly, but learning to recognise the other class often initially comes only at an equal or greater loss of accuracy in classifying the first class. This creates a strong selection pressure against making the classification boundary in the feature space more complex and GP with such a fitness function often cannot surmount the hurdle.

2.5 The Decay Curve Fitness Function

To address the hurdle problem, we introduced a new fitness function, the *decay curve fitness function* to estimate true fitness more accurately. The new fitness function uses an *increasing penalty* for each of the M_c misclassifications of some class c, as shown in equation 1.

$$f_{decay} = \sum_{c} \sum_{i=0}^{M_c} \alpha^{\beta i}/N \tag{1}$$

The values of α and β are determined through empirical search. We used a fitness function with $\alpha > 1$ to approximate the true fitness so that the penalty of later misclassifications increased exponentially. N is the number of training examples.

Obviously, as α approaches 1.0 and β approaches 0.0, the curve becomes progressively flatter and more similar to a traditional fitness function (error rate in this case).

3 Experiment Design and Configuration

3.1 Data Sets

Experiments were conducted on three different image data sets providing object classification problems of increasing difficulty. Examples are shown in figure 2.

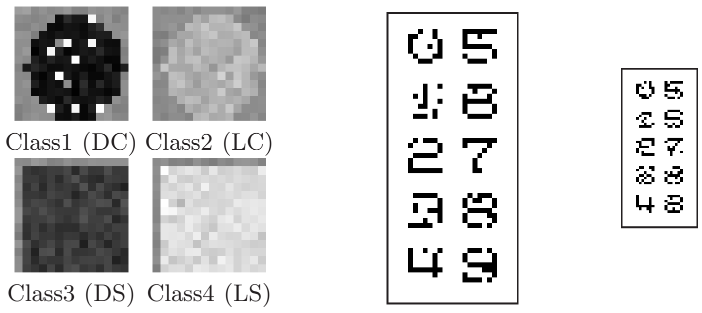


Fig. 2. Image data sets. (a) shape; (b) digit15; (c) digit30.

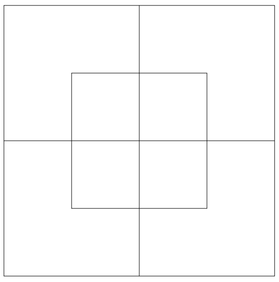
The first data set (figure 2a) was generated to give well defined objects against a relatively clean background. The pixels of the objects were produced using a Gaussian generator with different means and variances for each class. Four classes of 600 small objects (150 for each class) were used to form the classification data set. The four classes are: dark circles (*class1*), light circles (*class2*), dark squares (*class3*), and light squares (*class4*). This data set is referred to as *shape*. The objects in class1 and class3, and in class2 and class4 are very similar in the average values of pixel intensities, which makes the problem reasonably difficult.

The second and third data sets are two digit recognition tasks, each consisting of 1000 digit examples. Each digit is represented by a 7×7 bitmap image. In the two tasks, the goal is to automatically recognise which of the 10 classes (0, 1, 2, ..., 9) each bitmap belongs to. Note that all the digit patterns have been corrupted by noise. In the two tasks (figure 2 (b) and (c)), 15% and 30% of pixels, chosen at random, have been flipped. In data set 2 (*digit15*), while some patterns can be clearly recognised by human eyes such as “0”, “2”, “5”, “7”, and possibly “4”, it is not easy to distinguish between “6”, “8”and “3”. The task in data set 3 (*digit30*) is even more difficult — human eyes cannot recognise majority of the patterns, particularly “8”, “9” and “3”, “5” and “6”, and even “1”, “2” and “0”. In addition, the number of classes is much greater than that in task 1, making the two tasks even more difficult.

3.2 Primitive Sets

Terminals. In the *shape* data set, we used eight features extracted from the objects and an random number as the terminal set. The eight features are shown in figure 3.

For the two digit data sets, we used the raw pixels as the terminal sets, meaning that the feature vector of each object has 49 values. The large number of terminals makes these tasks more difficult, but we expect that the GP evolutionary process can automatically select those highly relevant to each recognition problem.



Feature	LGP Index	Description
f1	`cf[0]`	mean brightness of the entire object
f2	`cf[1]`	mean of the top left quadrant
f3	`cf[2]`	mean of the top right quadrant
f4	`cf[3]`	mean of the bottom left quadrant
f5	`cf[4]`	mean of the bottom right quadrant
f6	`cf[5]`	mean of the centre quadrant
f7	`cf[6]`	standard deviation of the whole object
f8	`cf[7]`	standard deviation of centre quadrant

Fig. 3. Terminal set for the *shape* data set.

Functions. The function set for all the three data sets was {`+, -, *, /, if`}. Division (/) was protected to return 0 on a divide-by-zero. `if` executes the next statement if the condition is true.

3.3 Parameters and Termination Criteria

The parameter values used for the LGP system for the three data sets are shown in table 1. Evolution is terminated at generation 50 unless a successful solution is found, in which case the evolution is terminated early.

Table 1. Parameter values for the LGP system for the three data sets.

parameter name	shape	digit15	digit30	parameter name	shape	digit15	digit30
pop_size	500	500	500	macromutation_rate	30%	30%	30%
max_program_length	15	35	35	micromutation_rate	30%	30%	30%
reproduction_rate	10%	10%	10%	α	1.2	1.2	1.2
crossover_rate	30%	30%	30%	β	0.24	0.24	0.24

3.4 TGP Configuration

The LGP approach developed in this work was compared to the basic TGP approach [3]. In TGP, the ramped half-and-half method was used for initial generation and mutation[2]. The proportional selection mechanism and the reproduction, crossover and mutation operators [3] were used in the learning and evolutionary process. The program output was translated into a class label according to the static range selection method [4].

The TGP system used the same terminal sets, function sets, fitness function, population size and termination criteria for the three data sets as the LGP approach. The reproduction, mutation, and crossover rates used were 10%, 30%, and 60%, respectively. The program depth was 3–5 for the shape data set, and

4–6 for the two digit data sets. All single experiments were repeated 50 times. The average results are presented in the next section.

The program depths above in TGP were derived from the LGP program lengths based on a heuristic. An LGP instruction typically consists of one or two arguments and an operation, each corresponding to a node in a TGP program tree. Considering that each TGP operation might be used by its children and/or parents, an LGP instruction roughly corresponds to 1.5 tree nodes. Assuming each non-leaf node has two children (or more for some functions), we can calculate the expressive capacity of a depth-n TGP in LGP program instructions.

4 Results and Discussion

4.1 Classification Performance

Classification Accuracy. Table 2 shows a comparison between the LGP approach developed in this work and the standard TGP approach for the three object classification problems.

Table 2. Classification accuracy of the LGP and TGP on the three data sets.

Data set	Method	Training Set Accuracy % ($\mu \pm \sigma$)	Test Set Accuracy % ($\mu \pm \sigma$)
shape	LGP	100.00 ± 0.00	99.91 ± 0.17
	TGP	85.04 ± 16.49	84.41 ± 17.17
digit15	LGP	68.02% ± 4.16%	62.48% ± 5.03%
	TGP	52.60% ± 6.65%	51.80% ± 6.85%
digit30	LGP	55.22% ± 3.49%	51.04% ± 4.26%
	TGP	41.15% ± 5.03%	35.00% ± 6.17%

On the shape data set, our LGP approach always generated a genetic program which successfully classified all objects in the training set. These 50 program classifiers also achieved almost perfect classification performance on the unseen objects in the test set. On the other hand, the TGP approach only achieved about 85.04% and 84.41% accuracy on the training and the test sets, respectively. In addition, the LGP approach resulted in a much smaller standard deviation than the TGP approach. This shows that the LGP method is more stable and more reliable than the TGP approach on this problem. These results suggest that the LGP approach greatly outperforms the TGP approach on this data set in terms of the classification accuracy.

The classification results on the two digit data sets show a similar pattern to those on the shape data set. In both cases, the LGP approach achieved a higher average value and a lower standard deviation of the classification accuracy on the test set than the corresponding TGP approach. The improvements are quite considerable, suggesting that the LGP approach is better than the TGP approach for these multi-class object classification problems.

Training Efficiency. Inspection of the number of generations used reveals that the LGP approach is more efficient than the TGP approach in finding a good genetic program classifier for these object classification problems. For example, in the shape data set, the $\mu \pm \sigma$ of the number of generations for the LGP approach was 16.46 ± 10.22, which was much smaller than the corresponding number for the TGP approach (41.22 ± 14.11).

4.2 Comprehensibility of the Evolved Genetic Programs

To check whether the genetic programs evolved by the LGP approach are easy to interpret or not, we use a typical evolved program which perfectly classified all objects for the shape data set as an example. The code of the evolved genetic program is shown in figure 4 (left). Note that structural introns are commented using `//`. The DAG representation of the simplified program is shown in figure 4 (right) after the introns are removed.

```
//r[1] = r[1] / r[1];
//r[3] = cf[0] + cf[5];
//if(r[3] < 0.86539)
//r[3] = r[3] - r[1];
r[0] = 0.453012 - cf[1];
//r[3] = r[2] * cf[5];
r[1] = r[0] * 0.89811;
if(cf[6] < cf[1])
r[2] = 0.453012 - cf[3];
r[3] = cf[4] - 0.86539;
```

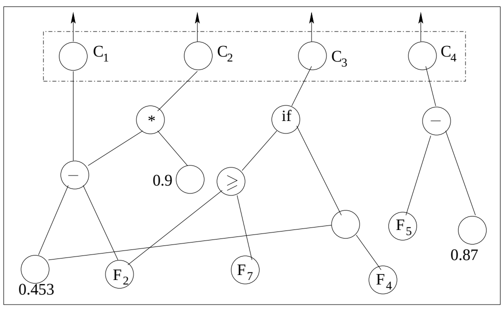


Fig. 4. A sample program evolved by LGP.

In this program, the array `cf (cf[0] to cf[7])` are the eight feature terminals (`f1, ..., f8`) as described in figure 3 and the register array `r (r[0] to r[3])` correspond to the four class labels (`class1, class2, class3, class4`). Given an object, the feature values and the register values can be easily calculated and the class of the object can be simply determined by taking the register with the largest value. For example, given the following four objects with different feature values:

```
                 cf[0]  cf[1]  cf[2]  cf[3]  cf[4]  cf[5]  cf[6]  cf[7]
-----------------------------------------------------------------------
Obj1 (class1): 0.3056 0.3458 0.2917 0.2796 0.3052 0.1754 0.5432 0.5422
Obj2 (class2): 0.6449 0.6239 0.6452 0.6423 0.6682 0.7075 0.1716 0.1009
Obj3 (class3): 0.2783 0.3194 0.2784 0.2770 0.2383 0.2331 0.2349 0.0958
Obj4 (class4): 0.8238 0.7910 0.8176 0.8198 0.8666 0.8689 0.2410 0.1021
```

we can obtain the following register values and classification `r[]` for each object example.

Object	True-Class	`r[0]`	`r[1]`	`r[2]`	`r[3]`	Classified-Class
Obj1	class1	**0.1474**	0.13240	0.0000	-0.5602	class1
Obj2	class2	-0.1919	**-0.1723**	-0.1893	-0.1972	class2
Obj3	class3	0.1747	0.1569	**0.1760**	-0.6271	class3
Obj4	class4	-0.3708	-0.3330	-0.3668	**0.0012**	class4

As can be seen from the results, this genetic program classified all the four object examples correctly. Examining the program and features used suggests that the genetic programs evolved by LGP are quite comprehensible.

Further inspection of this program reveals that only four of eight features were selected from the terminal set. This suggests that the LGP approach can automatically select features relevant to a particular task. The DAG representation of the program (figure 4b) shows that the LGP approach can co-evolve sub-programs together each for a particular class and that some terminals and functions can be reused by different sub-programs.

On the other hand, a program evolved by the TGP approach can only produce a single value, which must be translated/interpreted into a set of class labels. A typical genetic program evolved by the TGP approach is:

```
(* (- (+ (/ f1 -0.268213) (/ -0.828695 f6))
      (/ (/ f7 f6) (+ -0.828695 f5)))
   (* (- (/ f1 f5) (/ f5 f6))
      (+ (- f4 -0.828695) (+ f1 f2)))
)
```

This program used almost all the features and it is not clear how it does the classification. Such programs are more difficult to interpret for multi-class classification problems.

4.3 Impact of the New Fitness Function

To investigate whether the new fitness function is helpful in reducing the hurdle problem, we used the shape data set as an example to compare the classification performance between the new fitness function and the old fitness function (error rate).

When doing experiments, we used a slightly different setting in program size. Notice that the frequency of the hurdle problem will drop as the program size is increased, although it is not eliminated. Hence the LGP programs in the assessment of the new decay curve fitness function use a program length 10, which is still long enough to express a solution to the problem — solutions have been found when the maximum length is 5. In TGP the tree depths are left at 3–5. These limits are likely to be representative of the situation when a much more difficult problem is being addressed. In such tasks, the maximum depth which is computationally tractable with existing hardware may also be so short relative to the problem's difficulty that the hurdle problem is a major issue.

Table 3 shows the classification results of the two fitness functions using both the TGP and the LGP methods for the shape data set. For the TGP method, the new fitness function led to a very significant improvement on both the training set and the test set. For the LGP method, the classification accuracy was also improved using the new fitness function, but the improvement was not as significant. This was mainly because the LGP method with the old fitness function already performed quite well (98.76%) due to the power of LGP. When using either the old or the new fitness functions, the LGP method always outperformed the TGP method. This is consistent with our previous observation.

Table 3. A comparison of the two fitness functions on the shape data set.

Method	Fitness Function	Training Accuracy ($\mu \pm \sigma$)	Test Accuracy ($\mu \pm \sigma$)
TGP	old	77.31% ± 6.74%	77.14% ± 6.68%
	new	85.04% ± 16.49%	84.41% ± 17.17%
LGP	old	98.90% ± 4.98%	98.76% ± 5.04%
	new	99.97% ± 0.11%	99.90% ± 0.25%

Further inspection of the results using the TGP method on the shape data set shows that only 6 of the 50 runs using the old fitness function had a test or training accuracy greater than 75%. When those 6 runs are excluded, the μ and σ becomes 74.95% ± 0.0019% on the training set and 74.86% ± 0.0024% on the test set. These figures indicate how solid the hurdle actually is in situations where the problem is at the limit of a GP configuration's expressiveness. By using the new decay curve fitness function, 36 of the 50 runs finished with test and training accuracies greater than 75%.

5 Conclusions

The goal of this paper was to investigate an approach to the use of LGP and a new fitness function for multi-class object classification problems. This approach was compared with the basic TGP approach on three image data sets providing object classification problems of increasing difficulty. The results suggest that the LGP approach outperformed the TGP approach on all tasks in terms of classification accuracy and evolvability.

Inspection of the evolved genetic programs reveals that the programs evolved by the LGP approach are relatively easy to interpret for these problems. The results suggest that the LGP approach can automatically select features relevant to a particular task, that the programs evolved by LGP can be represented as a DAG, and that the LGP approach can simultaneously sub-programs together, each for a particular class.

A comparison between the old fitness function and the new fitness function has also highlighted the nature of the fitness function as an approximation to the true fitness of a problem. The results show that the new fitness function, with either the TGP approach or the LGP approach, can bring better and more consistently accurate results than the old fitness function.

Although developed for multi-class object classification problems, we expect that this approach can be applied to other multi-class classification problems.

References

1. Koza, J.R.: Genetic Programming. MIT Press, Campridge, Massachusetts (1992)
2. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications. Morgan Kaufmann, dpunkt.verlag (1998)
3. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable P rograms. Cambridge, Mass. : MIT Press, London, England (1994)
4. Loveard, T., Ciesielski, V.: Representing classification problems in genetic programming. In: Proceedings of the Congress on Evolutionary Computation. Volume 2., IEEE Press (2001) 1070–1077
5. Tackett, W.A.: Recombination, Selection, and the Genetic Construction of Computer Programs. PhD thesis, Faculty of the Graduate School, University of Southern C alifornia, Canoga Park, California, USA (1994)
6. Zhang, M., Ciesielski, V.: Genetic programming for multiple class object detection. In Proceedings of the 12th Australian Joint Conference o n Artificial Intelligence, Springer-Verlag (1999) 180–192 (LNAI Volume 1747).
7. Zhang, M., Ciesielski, V., Andreae, P.: A domain independent window-approach to multiclass object detection using genetic programming. EURASIP Journal on Signal Processing **2003** (2003) 841–859
8. Zhang, M., Smart, W.: Multiclass object classification using genetic programming. In Applications of Evolutionary Computing, EvoWorkshops2004. Volume 3005 of LNCS., Springer Verlag (2004) 369–378
9. Oltean, M., Grosan, C., Oltean, M.: Encoding multiple solutions in a linear genetic programming chromosome. In Proceedings of 4th International Conference on Computational Science, Part III. Springer-Verlag (2004) 1281–1288
10. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In Parallel distributed Processing, Explorations in the Microstructure of Cognition, Volume 1: Foundations. The MIT Press (1986)
11. Brameier, M., Banzhaf, W.: A comparison of genetic programming and neural networks in medical data analysis. Reihe CI 43/98, Dortmund University (1998)
12. Brameier, M., Banzhaf, W.: Effective linear genetic programming. Technical report, Department of Computer Science, University of Dortmund, Germany (2001)
13. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning. Addison–Wesley, Reading, MA (1989)