Explorations in Parallel Linear Genetic Programming

by

Carlton Downey

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

Victoria University of Wellington
2011

Abstract

Linear Genetic Programming (LGP) is a powerful problem-solving technique, but one with several significant weaknesses. LGP programs consist of a linear sequence of instructions, where each instruction may reuse previously computed results. This structure makes LGP programs compact and powerful, however it also introduces the problem of instruction dependencies. The notion of instruction dependencies expresses the concept that certain instructions rely on other instructions. Instruction dependencies are often disrupted during crossover or mutation when one or more instructions undergo modification. This disruption can cause disproportionately large changes in program output resulting in non-viable offspring and poor algorithm performance.

Motivated by biological inspiration and the issue of code disruption, we develop a new form of LGP called *Parallel LGP (PLGP)*. PLGP programs consist of n lists of instructions. These lists are executed in parallel, and the resulting vectors are summed to produce the overall program output. PLGP limits the disruptive effects of crossover and mutation, which allows PLGP to significantly outperform regular LGP.

We examine the PLGP architecture and determine that large PLGP programs can be slow to converge. To improve the convergence time of large PLGP programs we develop a new form of PLGP called Cooperative Coevolution PLGP (CC PLGP). CC PLGP adapts the concept of cooperative coevolution to the PLGP architecture. CC PLGP optimizes all program components in parallel, allowing CC PLGP to converge significantly faster than conventional PLGP.

We examine the CC PLGP architecture and determine that performance

is compromised by poor fitness estimates. To alleviate this problem we develop an extension of CC PLGP called Blueprint Search PLGP (BS PLGP). BS PLGP uses Particle Swarm Optimization (PSO) to search a specially constructed search space for good fitness estimates. BS PLGP significantly outperforms both PLGP and CC PLGP.

The applicability of all LGP algorithms is severely compromised by poor efficiency. Many problem domains have strict time constraints. Algorithms which cannot produce an acceptable solution within these time constraints cannot be applied to these problems. LGP algorithms are well known for their extensive run times, severely limiting applicability. To improve the applicability of our new algorithms we develop a number of complementary caching techniques. In all cases we present both theoretical and empirical results to confirm the effectiveness of our new caching algorithms.

We develop the execution trace caching algorithm for LGP to serve as a baseline estimate as well as a standalone improvement. We show that execution trace caching can decrease the execution time of LGP programs by up to 50%.

We develop a new caching algorithm for PLGP. We show that caching for PLGP can decrease the execution time of PLGP by up to an order of magnitude.

We develop a new caching algorithm for CC PLGP and BS PLGP. We show that caching for CC PLGP and BS PLGP can decrease the execution time of CC PLGP and BS PLGP by up to an order of magnitude.

Acknowledgments

First and foremost I would like to thank my supervisor, Mengjie Zhang, for the time and effort he has invested in teaching me how to perform research. Mengjie has always found the time to help me, despite his myriad commitments, and for his friendship and support I will always be grateful.

I would also like to thank my family, my father Rod, my mother Kristin, and my brother Alex for their continued love and support throughout the highs and lows of my masters. No matter the circumstance they have always been there for me.

I would like to thank my friends for understanding my unusual hours and helping to keep me sane. You were always there to distract me whenever my experiments failed to work as planned.

Finally I would like to thank the members of the VUW Evolutionary Computation Research Group for acting as a source of both inspiration and criticism.

This work was supported in part by Victoria University of Wellington (VUW Masters Scholarship) and the Royal Society of New Zealand Marsden Fund (Grant VUW0806).

Contents

Chapter 1

Introduction

In this chapter we will introduce some general concepts of LGP, identify a number of current problems with LGP, and form several research questions which this thesis will aim to answer. We will also summarise both the major contributions made by this thesis, and the overall organisation of this thesis.

1.1 Linear Genetic Programming

Genetic Programming (GP) is a method of automatically generating computer programs which solve a user defined task, inspired by the principles of biological evolution [46]. Linear genetic programming (LGP) [12, 8] is a GP variant where each program is a linear sequence of instructions in some imperative programming language. LGP begins with an initial group of randomly generated programs called the *population*. Program performance is calculated via a *fitness function*, which uses a *training set* of problem examples to determine a numerical representation of program quality known as the program *fitness*. Fitness values are used to *select* individuals as a basis for the next program *generation*. The *crossover*, *mutation*, and *elitism* genetic operators are applied to the selected programs to create a new population of programs. Recombination exchanges code between

programs; mutation randomly modifies part of a program; and elitism retains the best programs in the population. The process of creating a new population by selecting high quality individuals and applying the genetic operators, is repeated until certain user-defined termination criteria are met. Algorithm output is typically the "best" program found during the entirety of evolution. LGP can be viewed as a genetic beam search through the space of all possible programs.

GP in all forms is an emerging field in the area of evolutionary computation and machine learning. GP has been successfully applied to a wide variety of tasks [46, 47], including image analysis [69], object detection [95], symbolic regression [46], and actuator control for robotics [14]. LGP in particular has seen great success, with algorithms based on the LGP architecture often outperforming alternative GP approaches [12, 26, 97].

1.2 Issues in LGP

1.2.1 Instruction Dependencies

Instruction dependencies are a fundamental problem in LGP. LGP programs consist of a sequence of instructions to be executed in order. Instruction dependencies occur when instructions interact; in other words, the output of one instruction forms the input for another instruction. Instruction dependencies allow LGP programs to be concise, yet powerful, as results computed early in the program can be reused many times. Unfortunately they also represent a significant barrier to effective evolution.

Instruction dependencies are often disrupted during evolution, resulting in low quality offspring. Instruction dependencies express the notion that certain instructions "depend" on other instructions for intput. Evolution, in the form of crossover or mutation, modifies a randomly selected instruction sequence. Modified instructions will produce different output, disrupting those instructions which depended on the modified instruc-

tions for input. Disrupted instructions will produce random output, resulting in offspring likely to have poor performance.

Decreasing the number of instruction dependencies disrupted during evolution is an important step towards improving LGP. The performance of current LGP algorithms is severly compromised by evolution producing low quality offspring, due to disrupted instruction dependencies. If the number of instruction dependencies disrupted during evolution is decreased, the number of high quality offspring produced can be expected to increase. This would increase the overall algorithm performance.

1.2.2 Fitness Evaluation

Fitness evaluation is the most computationally intensive procedure in GP [30]. In each generation all programs typically need to be evaluated for fitness. In many problem domains this can mean evaluating each program on hundreds, or even thousands, of training examples. The cost of fitness evaluation dwarfs that of all other algorithm components, and is the primary reason behind the extensive execution times of GP algorithms.

Minimising the cost of fitness evaluation improves the efficiency and applicability of any GP variant. The flexibility and applicability of search algorithms such as GP is directly related to how long they take to run. Many problems have rigid time constraints which prevent algorithms which execute too slowly from being applied. Hence algorithms which execute more rapidly can be applied to a wider range of problems.

1.3 Thesis Objectives

The overall goal of this thesis is to improve the effectiveness, and increase the efficiency and applicability of Linear Genetic Programming. This overall goal encompasses three complementary subgoals. The first subgoal is to design and develop a new LGP architecture where fewer depeden-

cies are disrupted during evolution. Disrupted dependencies result in low quality offspring and overall poor algorithm performance. The second subgoal is to explore new directions in LGP suggested by such an architecture to improve system effectiveness. By reducing the number of instruction dependencies disrupted during evolution we grant access to many novel algorithms. The third subgoal is to use caching to decrease program execution time for each of these new algorithms. Large program execution times significantly reduce algorithm applicability.

In order to achieve these goals, this thesis will focus on answering the following research questions.

1. *How do we develop a LGP architecture where fewer dependencies are disrupted during evolution?* Instruction dependencies compromise the performance of conventional LGP architectures by reducing the number of viable offspring. Evolutionary operators such as crossover and mutation disrupt instruction dependencies, causing large, and undesirable changes in program output. In this thesis, we will develop a new LGP architecture, based on the concept of independently executed code sequences, in which fewer instruction dependencies are disrupted during evolution. We expect that the use of this LGP architecture will give improved performance over conventional LGP.

2. *What novel algorithms are suggested by our new LGP architecture?* By adopting a new LGP architecture, we have laid the groundwork for developing novel LGP algorithms. Existing LGP algorithms are structured to exploit the strengths of the conventional LGP architecture. Our new architecture will posses different strengths to those of the conventional LGP architecture, offering up new algorithm opportunities. In this thesis we will develop new LGP algorithms based on our novel LGP architecture, with the aim of further improving performance over that of conventional LGP.

3. *How can caching best be applied to each new algorithm?* It is important

that our new algorithms are fast. Producing high fitness solutions is an important aspect of any algorithm, however if the algorithm does not run within a reasonable time frame it will rarely be deployed. This is particularly true in the case of population based search algorithms such as LGP, which are well known for their extensive run times. In this thesis, we will develop caching techniques for each new architecture. We expect these techniques to significantly reduce program execution time.

1.4 Major Contributions

This thesis makes the following contributions towards the field of LGP.

- **Parallel Linear Genetic Programming**
  We have developed a new LGP architecture called Parallel LGP (PLGP) where each program consists of multiple independently executed factors. PLGP programs have fewer instruction dependencies which means fewer programs disrupted during evolution. This work shows how a straightforward change in program structure can give excellent results. Our results show that PLGP gives significantly superior performance on a range of classification problems, particularly when large programs are used. In addition, the PLGP program structure provides a versatile base from which to develop powerful new LGP algorithms.

  Part of this work has been published in:

  - Carlton Downey, Mengjie Zhang. *"Parallel Linear Genetic Programming"*. Proceedings of the 14th European Conference on Genetic Programming. Lecture Notes in Computer Science. Vol. 6621. Springer. Torino, Italy 2011. pp. 178-189. *(Nominated for the Best Paper Award)*

- **Novel Algorithms based on the PLGP architecture**
  We have developed two novel algorithms based on the PLGP architecture. These algorithms exploit the parallel structure of the PLGP architecture to evolve solutions in ways not previous possible. We combined the highly successful concept of cooperative coevolution with the PLGP program structure to develop the Cooperative Coevolution PLGP (CC PLPG) algorithm. Our results show that CC PLGP significantly outperforms PLGP during initial generations. We extended the CC PLGP algorithm by introducing the notion of a structured solution space together with a Particle Swarm Optimization based search to develop the Blueprint Search PLGP (BS PLGP) algorithm. Our results show that BS PLGP significantly outperforms both CC PLGP and PLGP.

- **Caching Techniques**
  We have developed three novel caching techniques which significantly improve algorithm efficiency. Firstly, we developed the execution trace caching technique for LGP as both a baseline indicator, and as a standalone improvement to LGP. We provided theoretical and empirical results which show that execution trace caching can decrease the execution time of LGP programs by up to 50%. Secondly, we developed a novel caching technique for PLGP which exploits the parallel PLGP architecture. We provided theoretical and empirical resluts which show caching can decrease PLGP program execution time by an order of magnitude. Thirdly we developed a novel caching technique for CC PLGP and BS PLGP which exploits the dual population architecture used by both of these algorithms. Once again we provided theoretical and empirical results which show that caching can reduce execution time of both CC PLGP and BS PLGP by an order of magnitude.

  Part of this work has been published in:

- Carlton Downey and Mengjie Zhang. *"Execution Trace Caching for Linear Genetic Programming"*. Proceeding of the 2011 IEEE Congress on Evolutionary Computation. IEEE Press. New Orleans, USA. June 5-8, 2011. pp. 1191-1198.
- Carlton Downey, Mengjie Zhang. *"Caching for Parallel Linear Genetic Programming"*. Proceedings of Genetic and Evolutionary Computation Conference (GECCO'11), ACM Press. pp 201-202.

1.5 Structure

The remainder of this thesis is structured as follows.

- Chapter 2 provides a survey of relevant background concepts together with a detailed discussion of work related to this thesis.
- Chapter 3 describes the data sets, settings, and parameters used in experiments throughout this thesis.
- Chapter 4 investigates our first research question. It presents our new LGP architecture which reduces the number of instruction dependencies disrupted during evolution. We perform experiments comparing the performance of our new architecture to that of conventional LGP.
- Chapters 5 and 6 investigate our second research question. Chapter 5 presents a new algorithm which combines the concept of cooperative coevolution and the architecture developed in chapter 4. We perform experiments comparing the performance of our new algorithm to that of vanilla PLGP. Chapter 6 presents an extension of the algorithm developed in chapter 5. We perform experiments comparing the performance of this extension to that of the original algorithm developed in chapter 5.

- Chapters 7 and 8 investigate our third research question. Chapter 7 presents a new caching technique for LGP which can act as a baseline for future work. We perform a theoretical analysis of this caching algorithm as well as experiments to obtain empirical results. Chapter 8 presents new caching techniques for the algorithms introduced in chapters 4, 5, and 6. We perform a theoretical analysis for each of these caching algorithms, as well as experiments using various classification problems to obtain empirical results.

- Finally chapter 9 presents the major conclusions of the work presented in this thesis, together with potential future work directions.

Chapter 2

Literature Survey

This section covers some necessary background material vital to understanding the work presented in this thesis. Our intention is to present only a brief overview of this material, sufficient only to familiarize the reader with the broadest outline of these concepts. For an in-depth study of this material we refer the reader to the citations provided.

2.1 Overview of Machine Learning

Machine Learning (ML) [59, 56, 2, 10] is a major sub-field of Artificial Intelligence (AI) which concerns the design and development of algorithms which enable computers to "learn". ML is primarily focused on using empirical data to infer characteristics of the underlying probability distribution for the purposes of predicting future behavior. Common ML applications include medical diagnosis, handwriting recognition, actuator control, financial analysis, etc.

ML algorithms are categorized according to their *Learning Strategy* and their *Learning Paradigm*. The learning Strategy corresponds to the way in which data is presented to the algorithm. The *Learning Paradigm* corresponds to the inspiration behind the algorithm; in other words, the way in which inputs are mapped to outputs.

2.1.1 Learning Strategies

ML techniques are separated into a number of techniques based on assumptions about how learning occurs. These include Supervised Learning, Unsupervised Learning, Reinforcement Learning, Semi-Supervised Learning, etc. The work presented in this thesis is concerned with Supervised Learning, however we briefly outline the three major approaches.

- **Supervised Learning**
  Supervised learning algorithms [10] use a set of *labeled* training instances to infer a function which maps inputs to desired outputs. The labeled data is manually specified by an expert in the area. In other words, supervised learning can be viewed as learning to mimic the behavior of a human expert. The work presented in this thesis belongs to the area of supervised learning.

  The difficulty with supervised learning lies with the limited number of training examples. Problem domains often contain infinitely many possible input combinations, while the training set is limited to some finite subset of input combinations. The aim of supervised learning is to produce a learner which can correctly predict the output of previously unseen instances based solely on the limited number of training examples provided.

- **Unsupervised Learning**
  Unsupervised learning algorithms [39] seek to discover hidden structure in *unlabeled* data. Unlike supervised learning the data instances are not labeled with a desired output, so there is no "correct" answer.

- **Reinforcement Learning**
  Reinforcement learning algorithms [85] interactively learn within the environment. The learner is not provided with a fixed training set of data. Instead, the learner generates its own training data by taking actions and receiving rewards. Rewards are real numbers which

indicate how well the learner is performing, and are used to select future actions.

- **Hybrid Learning**
  Hybrid learning algorithms use any combination of supervised learning, unsupervised learning, and reinforcement learning.

2.1.2 Learning Paradigms

There are generally four major paradigms in machine learning: Evolutionary Paradigm, Connectionist Paradigm, Case-Based Learning Paradigm, and Inductive Learning Paradigm [79].

- **Evolutionary Paradigm**
  Evolutionary computation methods evolve a population of potential solutions through the repeated application of evolutionary operators. This learning paradigm was inspired by Darwinian evolution in biological systems. Evolutionary computation is central to the work presented in this thesis, and is covered in detail in section 2.2.

- **Connectionist Paradigm**
  Connectionist methods represent a solution as a network of connected nodes. This learning paradigm was inspired by the structure of biological neural networks within the human brain. Learning is achieved by optimizing the network parameter values until each input produces the correct output. Connectionist methods include Artificial Neural Networks (ANNs) [78, 93] and Parallel Distributed Processing Systems (PDPs) [55].

- **Case-Based Learning Paradigm**
  Case-Based methods directly compare each new example to the entire training set. This approach is attractive in its simplicity, however it has the significant disadvantage that execution time is pro-

portional to the size of the training set. An example of Case-Based learning is the Nearest Neighbor (NN) algorithm [5].

- **Inductive Learning Paradigm**
  Inductive methods derive explicit rules from the training data, and apply these rules to the test data. These methods are distinct because each rule is a standalone entity, unlike a connectionist approach where implicit rules are contained within a single large network. Inductive approaches include Decision Trees [74].

2.1.3 Classification

Classification problems involve determining the type or *class* of an object instance based on some limited information or *features*. Formally the goal of classification is to take an input vector x and to assign it to one of K discrete classes C_k where $k = 1, ..., K$ [10]. Solving classification problems involves learning a classifier, a program which can automatically perform classification on an object with unknown class. The classifier is a model encoding a set of criteria that allows a data instance to be assigned to a particular class depending on the value of certain variables. A classification algorithm is a method for constructing a classifier. Classification problems form the basis of empirical testing in this paper.

Classification problems are categorized based on the number of classes which must be distinguished between. Binary classification problems are those which require distinguishing between two classes. In contrast, multiclass classification problems require distinguishing between more than two classes. Multiclass classification problems are often extremely challenging for GP based classifiers [26, 99].

2.1.4 Data Sets

Supervised learning methods, such as the algorithms developed in this thesis, require a labeled data set. A data set consists of sample instances from the problem domain. In the case of labeled data, each instance consists of several inputs, together with a desired output. In the case of unlabeled data, outputs are not provided.

Training, Test, and Validation Sets

In supervised learning the data set is typically partitioned into three subsets. The training set is used to train the learner by optimizing parameter values. The validation set provides an estimate of test set performance. The test set is used as a measure of performance on unseen data.

The validation set is a mechanism used to prevent overfitting [1, 87]. Overfitting occurs when the learner fails to generalize, resulting in high training set performance but low test set performance. Overfitting can be seen as a model with an excess of parameters for a particular problem [56]. When a validation set is not used, other mechanisms must be put in place to avoid overfitting.

A graphical representation of overfitting is shown in figure 2.1. The green line shows a model with good generalization which will make reasonable predictions for unseen x values. The red line shows an overfitting model. This model performs well on the training set, but will make outrageously bad predictions for any unseen data points.

2.2 Overview of Evolutionary Computation

Evolutionary Computation (EC) [38, 22, 25] is concerned with developing solutions to problems through algorithms inspired by the process of biological evolution [83]. EC algorithms maintain a population of individuals, each of which is a potential solution to the problem. High quality

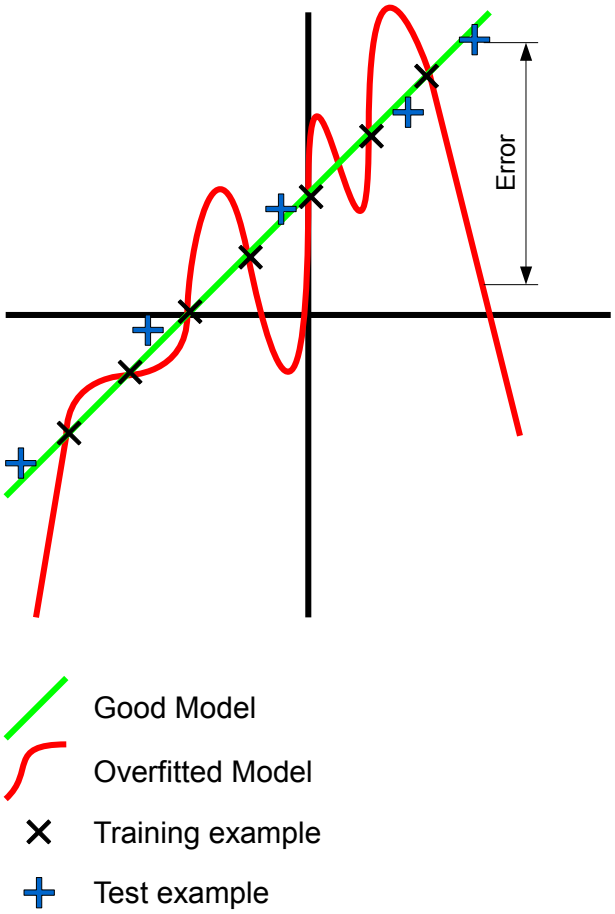


Figure 2.1: A simple example of overfitting

individuals are stochastically selected from the population and modified via algorithm specific genetic operators. These operators vary from algorithm to algorithm, but often have their roots in biological evolution.

There are a wide variety of algorithms which fall under the umbrella of EC. Some of these include:

- Evolutionary Algorithms

  - Genetic Algorithms (GA) [58]
  - Evolutionary Programming (EP) [22]
  - Evolution Strategies (ES) [9]
  - Genetic Programming (GP) [46]

- Swarm Intelligence
  - Ant Colony Optimization (ACO) [19]
  - Particle Swarm Optimization (PSO) [45]

- Others
  - Differential Evolution (DE) [84]
  - Artificial Immune Systems (AIS) [15]
  - Learning Classifier Systems (LCS) [13]

2.2.1 Basic Structure

EC algorithms differ in a variety of ways, however they all follow the same underlying procedure: A population of individuals is initialized. Individual quality is evaluated, and good solutions are selected as a basis for a new population. Finally the selected individuals undergo modification to produce a population of new solutions. These three steps are discussed in more detail below.

- **Initialization**
  Each algorithm begins by initializing a collection, or *population* of individuals. Each individual is a potential solution to the problem, and can be viewed as a single point within the search space of all possible solutions. There is ongoing research into the best way to generate the individuals in the initialization [4], however many algorithms simply generate individuals at random.

- **Selection**
  Each algorithm iteratively generates a new population by stochastically sampling the individuals in the previous generation. Selection is biased towards higher quality solutions, resulting in an overall increase in solution quality within the population. In order to compare solution quality, algorithms possess a *fitness function* which provides a quantitative assessment of an individual with regards to its quality as a solution to the problem, called the *fitness*. It is important to note that individuals can be selected more than once, and that solutions with higher fitness will contribute more to later generations.

- **Reproduction** [1]
  Each algorithm uses genetic operators to modify the individuals selected, with the aim of discovering new, high fitness solutions. The exact form of the genetic operators used varies from algorithm to algorithm, but the majority fit three categories:

  - Mutation operators randomly modify part of a single individual. Mutation acts to maintain genetic diversity within the population, as well as being a form of local search in some cases.
  - Recombination operators combine the information from two parents into a single offspring. Recombination acts to combine existing genetic material in new ways, with the aim of producing offspring bearing the strengths of both parents.
  - Elitism operators directly copy a single individual. Elitism acts to preserve high quality solutions to ensure that population fitness does not drop.

[1] Note that the term reproduction is sometimes used to refer solely to elitism. In this instance we use the word reproduction to mean the generation of new individuals through the application of any genetic operator. However, for the remainder of this thesis we do not include elitism in the set of genetic operators as it does not modify the program in any way.

The effect of the Mutation, Recombination, and Elitism operators is illustrated in figure 2.2.

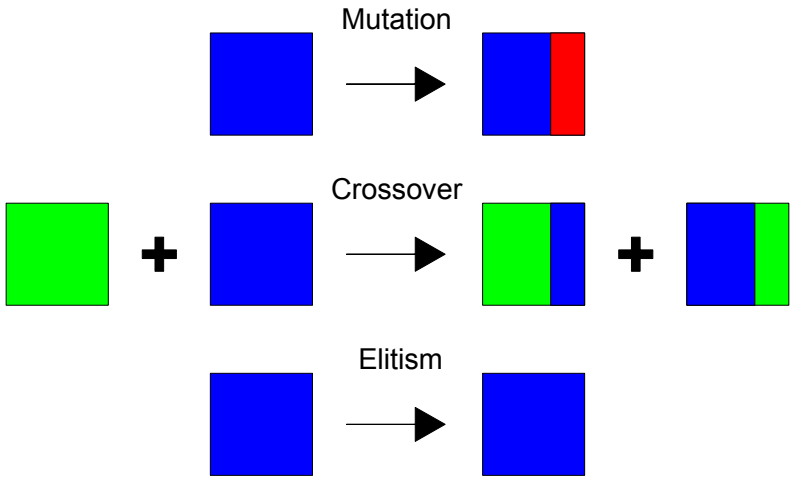


Figure 2.2: The three types of genetic operators

2.2.2 Representation

Different EC methods use different representations for the individuals in the population. Many representations exist, including bitstrings, vectors of real valued numbers, trees, and graphs. The choice of representation is extremely important as it dictates many other aspects of the algorithm, such as the form of the genetic operators.

Different representations possess different strengths and weaknesses and are appropriate for different problems. The representation controls the size and shape of the search space. For example the search space of all possible bit strings of length n is both finite and countable, whereas the search space of all possible graphs is infinite and continuous. In addition,

some search spaces are easier to search than others, something which must be considered when selecting a representation.

2.2.3 Selection

EC algorithms use fitness values to stochastically select individuals for use in a new population. Selection requires that individuals with high fitness are chosen more often than individuals with low fitness. There are several methods used to select individuals including proportional selection, tournament selection, and rank selection [58]:

- **Proportional Selection**
  Proportional Selection samples individuals with probability proportional to their fitness [6]. The probability of any single individual being sampled is calculated as $f_x / \sum_{i=0}^{n} f_i$. This type of selection is also known as "roulette wheel selection" as it can be viewed as spinning a roulette wheel, where each individual has a segment on the wheel proportional in size to its fitness.

  Unfortunately there are several problems with proportional selection. In particular, proportional selection can easily result in homogenous populations, where a small number of individuals are oversampled [51]. If there is large disparity between the fitness of individuals within the population then there will be also be a large disparity in how many times each individual is sampled. This is extremely problematic if a small number of individuals have much higher fitness than the rest of the population. The high fitness individuals will be selected too often, resulting in the population converging to a single genotype within a small number of generations. Premature population convergence prevents effective search resulting in poor algorithm performance.

- **Tournament selection**
  Tournament Selection samples individuals according to the results

of a tournament [57]. n individuals are sampled uniformly at random from the population to compete against each other, where n is a user-defined parameter. The individual with the best fitness automatically wins the tournament, and is selected.

Tournament selection has the advantage of preventing convergence to a single homogenous population [51]. The disparity in fitness is irrelevant because individuals are sampled uniformly at random to participate in tournaments. High fitness individuals will be sampled more than low fitness individuals, as they will win the tournaments they participate in. However the number of times any single individual can be selected is controlled by the number of tournaments that individual participates in. With this in mind population convergence rates are controlled by the size of the tournament, which in turn is controlled by the user defined parameter n. n acts to adjust the selection pressure. It is more difficult to win larger tournaments, therefore larger tournaments favour high fitness solutions leading to population convergence.

- **Rank Selection**
  Rank Selection samples individuals with probability proportional to their rank [34]. The entire population of individuals is ranked according to their fitness values. The probability of any single individual being selected is a function of that individual's rank within the population.

  Rank selection also has the advantage of preventing convergence to a single homogenous population. The disparity in fitness is irrelevant because the probability of any single individual being selected is a function of that individual's rank within the population. High quality solutions will be sampled more often than low quality solutions as they will possess higher rank. However the difference in fitness is irrelevant when determining rank.

2.2.4 Parameters

When deploying an EC algorithm there are a number of parameters which need to be specified by the user. It is important that good values are chosen for these parameters, as a poor choice of parameters results in slow algorithm convergence and low quality final solutions. The particular parameters pertaining to the algorithms used in this thesis are covered in chapter 3.

Note that these parameters are typically static, however several methods exist which dynamically adjust the GP parameters during evolution [54, 82]. Note that dynamic values are not used in this thesis as this is not the goal of this work.

2.3 Overview of Genetic Programming

Genetic Programming (GP) [71, 51] is an EC method where the individuals being evolved are simple computer programs. GP was derived from genetic algorithms [40], and popularized by Koza in 1992 [46].

GP algorithms are categorized based on the type of programs used. Major categories include Tree GP, Linear GP, Graph GP [70], and Grammatical GP [88, 89]. This thesis only reviews Tree GP (the conventional and most commonly used form of GP) and Linear GP (which is used in the work presented in this thesis).

2.3.1 Tree GP

The original, and most widely used form of GP is called tree-based GP (TGP). In TGP the programs are LISP-S expressions stored in a tree structure. An example of a TGP program is shown in figure 2.3.

Each TGP program consists of a single tree. The internal nodes of the tree, called *non-terminal* nodes, are nested functions. These functions use

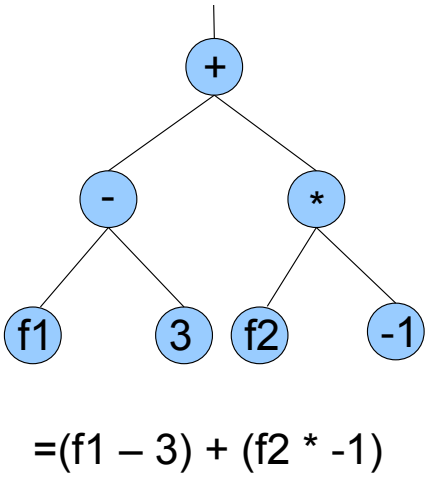


Figure 2.3: An example of a TGP program

the output of their children as inputs. The leaf nodes of the tree, called *terminal* nodes, are constants or features instantiated by the training instance.

Strongly Typed GP

Strongly Typed GP (STGP) [60] is an extension of the basic TGP approach, which does not require closure. An example of a STGP program is shown in figure 2.4.

In conventional TGP all variables, constants, features, and values returned by features must be of the same data type, typically real numbers. This is known as the closure property. Closure has the advantage of ensuring any possible program produced during reproduction is valid. Unfortunately requiring closure has the disadvantage of greatly restricting the function set.

In STGP we allow data values of different types to occur in the same program. For instance, functions which take real numbers as input and produce a single Boolean value as output. STGP has the advantage of

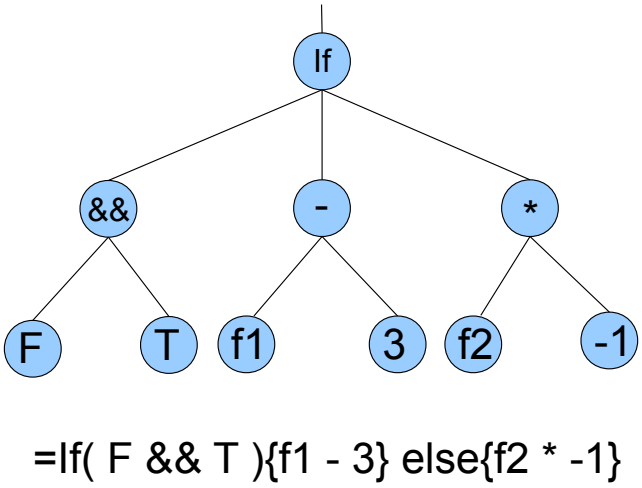


Figure 2.4: An example of a STGP program

a more flexible program structure which allows a wider range of functions. Unfortunately reproduction of STGP programs can often result in non-viable offspring. These are programs where the function types do not match, preventing program execution.

2.3.2 Linear GP

In Linear GP (LGP) [66, 23, 7, 64, 11] the individuals in the population are programs in some imperative programming language. Each program consists of a number of lines of code, to be executed in sequence. The LGP used in this paper follows the ideas of register machine LGP [**?**]. In register machine LGP each individual program is represented by a sequence of register machine instructions, typically expressed in human-readable form as C-style code. Each instruction has three components: an operator, 2 arguments and a destination register. To execute the instruction, the operator is applied to the two arguments and the resulting value is stored in the destination register. The operators can be simple standard arithmetic

operators or complex specific functions predefined for a particular task. The arguments can be constants, registers, or features from the current instance. An example of a LGP program is shown in figure 2.5.

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

Figure 2.5: An example of a LGP program

After a LGP program has been executed the registers will each hold a real valued number. For presentation convenience, the state of the registers after execution is represented by a vector of reals `r`. These numbers are the outputs of the LGP program and can be interpreted appropriately depending on the problem at hand. A step by step example of LGP program execution can be found in figure 2.6.

Genetic Operators

LGP algorithms use three genetic operators adapted to the LGP architecture [**?**]:

- **Elitism:** Elitism makes a perfect copy of the selected LGP program.
- **Mutation:** Mutation replaces a randomly selected instruction sequence with a randomly generated instruction sequence.
- **Crossover:** Crossover exchanges two randomly selected instruction sequences.

The operation of these three operators is illustrated in figure 2.7.

Program

Inputs

f1	f2	f3
0.1	3.0	1.0

(a)

Program Execution

Program		Registers		
index	Instruction	r[1]	r[2]	r[3]
0	-	0	0	0
1	r[1] = 3.1 + f1;	3.2	0	0
2	r[3] = f2 / r[1];	3.2	0	0.94
3	r[2] = r[1] * r[1];	3.2	10.24	0.94
4	r[1] = f1 - f1;	0	10.24	0.94
5	r[1] = r[1] - 1.5;	-1.5	10.24	0.94
6	r[2] = r[2] + r[1];	-1.5	8.74	0.94

(b)

Program

Outputs

r[1]	r[2]	r[3]
-1.5	8.74	0.94

(c)

Figure 2.6: Example of LGP program execution on a specific training example.

Classification using LGP

LGP is particularly well suited to solving multiclass classification problems [21, 26, 97, 67, 20]. The number of outputs from a LGP program is determined by the number of registers, and the number of registers can be arbitrarily large. Hence we can map each class to a particular output

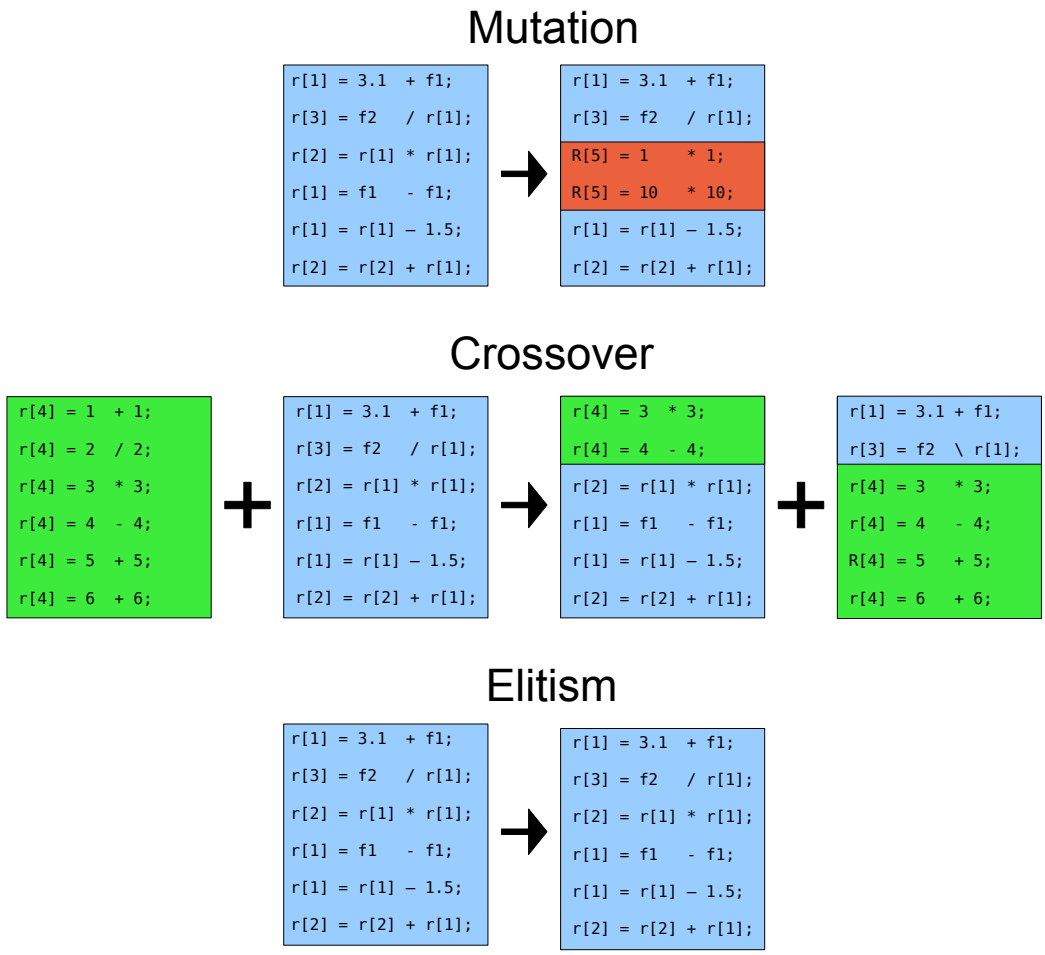


Figure 2.7: Three common LGP genetic operators

in the form of a single register. Classification then proceeds by selecting the register with the largest final value and classifying the instance as the associated class. For example, if registers (r1, r2, r3) held the values (-1.5, 8.74, 0.94) then the object would be classified as class 2, since register 2 has the largest final value (8.74).

This thesis will use multiclass classification tasks as example data sets to examine new algorithms and structure/representations.

2.4 Overview of Particle Swarm Optimization

Particle Swarm Optimization (PSO) [44] is a population based search technique inspired by the social behavior of various organisms known as "Swarm Intelligence". PSO uses a population of solutions called "particles" with a position and velocity in the search space, and updates these particles based on a combination of local and global optima. PSO can be envisioned as a swarm of particles moving around the search space, over time converging on the optimal values discovered by the swarm as a whole.

Let S be the number of particles in the swarm, each having a position $x_i \in \mathbb{R}^n$ and a velocity $v_i \in \mathbb{R}^n$. Let p_i be the best known position of particle i and let g be the best known position of the entire swarm. Let b_{lo} and b_{hi} be the lower and upper bounds of the search space. Finally let $f()$ be the fitness function which takes a particle and returns its fitness value. A basic PSO algorithm is shown in algorithm 1.

PSO has a number of important strengths:

- PSO does not use the gradient of the search space being explored, allowing PSO to solve problems which are not differentiable, are noisy, or change over time.
- PSO can produce a population of distinct, high fitness solutions.
- PSO offers rapid particle convergence when compared to evolutionary computation algorithms.
- PSO is easy to implement and there are few parameters to adjust.
- PSO is computationally inexpensive both in terms of memory and CPU.

An example of PSO is shown in figure 2.8. In this example PSO is searching a discrete 2-dimensional search space, so each PSO particle will have a discrete two-dimensional position, and a two-dimensional velocity.

Algorithm 1 Generalized PSO Algorithm

for *(each particle i = 1, ..., S)* **do**
  Initialize the particle's position with a uniformly distributed random vector $x_i := U(b_{lo}, b_{hi})$;
  Initialize the particle's best known position to its initial position: $p_i := x_i$;
  if *(* $f(p_i) < f(g)$ *)* **then**
    update the swarm's best known position: $g := p_i$;
  Initialize the particle's velocity: $v_i := U(-|b_{up} - b_{lo}|, |b_{up}, b_{lo}|)$;

while *(generations < max)* **do**
  for *(each particle i = 1,...,S)* **do**
    Pick random numbers r_p, $r_g \in U(0, 1)$;
    Update the particle's velocity: $v_i := \omega v_i + \varphi_p r_p(p_i - x_i) + \varphi_g r_g(g - x)$;
    Update the particle's position: $x_i := x_i + v_i$;
    if *(* $f(x_i < f(p_i)$ *)* **then**
      Update the particle's best known position: $p_i := x_i$;
      if *(* $f(p_i) < f(g)$ *)* **then**
        Update the swarm's best known position: $g := p_i$;

Each cross represents the position of a particle, and each arrow shows its velocity. The particles will converge on high fitness solutions resulting in a clustering of particles in areas of high fitness. An example of a converged PSO population is shown in figure 2.9.

2.5 Overview of Cooperative Coevolution

Cooperative Coevolution (CC) [72, 73, 92] is a recently popularized EC framework where each individual is a *partial solution*. Conventional evolutionary algorithms have a single population containing complete solutions to the problem. In CC there are n populations, called sub-populations,

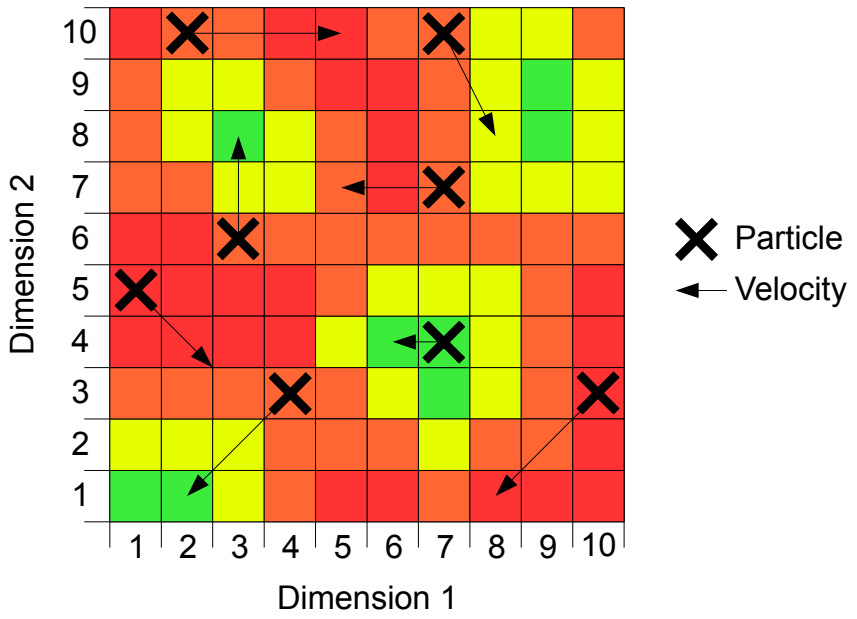


Figure 2.8: PSO searching the Blueprint Space

each of which contains partial solutions to the problem. A complete solution consists of n partial solutions, one from each population.

In a CC model each individual aims to optimize one part of the solution through cooperation with other individuals. The underlying concept of CC is to produce several homogeneous populations of individuals, each of which optimizes one aspect of the final solution. A complete solution to the problem can then be obtained by combining partial solutions from the various populations. Evolving these populations in parallel is equivalent to performing several parallel searches for different pieces of the solution. It is hypothesized that in many situations this is more efficient than a single search for the entire solution [62].

Through its use of multiple populations CC also encourages the nich-

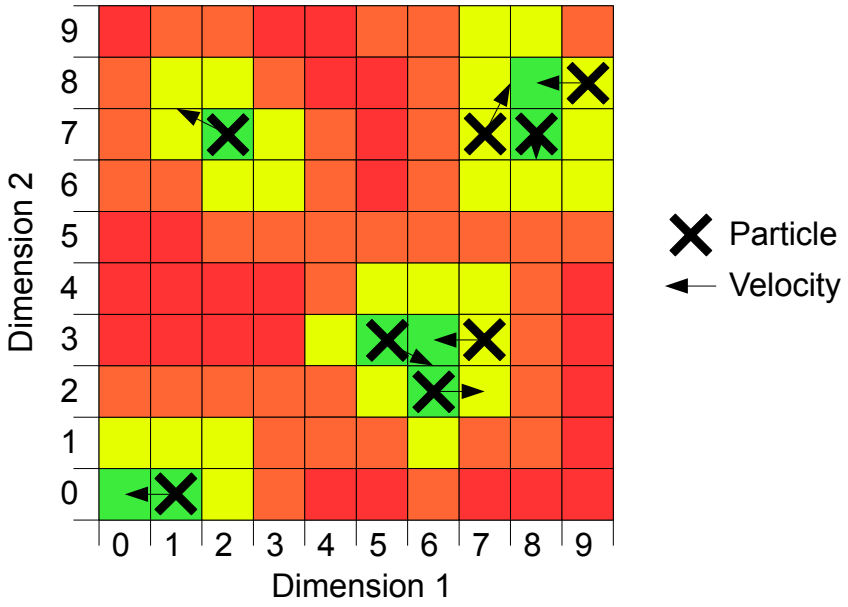


Figure 2.9: Converged PSO

ing of components [62]. Components which vary too widely in their genotype often produce non-viable offspring when mated. This is particularly a problem with single population evolutionary techniques. Good solutions often require a range of diverse genetic components, and evolving these components within a single population often gives poor results. By allowing multiple populations, components are free to evolve within a *niche* consisting solely of similar individuals.

2.5.1 SANE

SANE [62, 61, 63] is a CC algorithm for evolving neural networks first described by Moriarty and Miikkulainen in 1997 [62]. The distinguishing

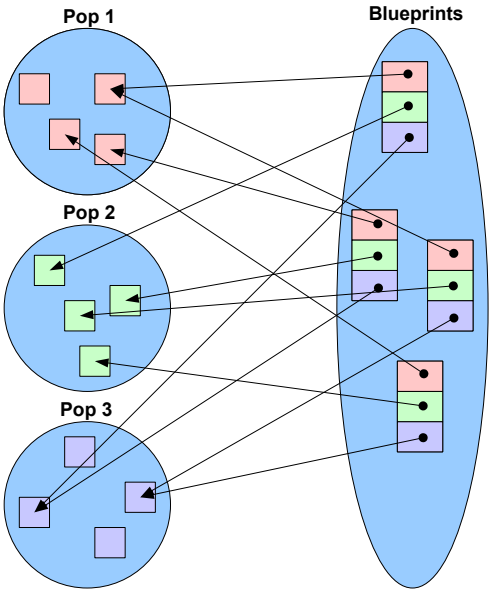


Figure 2.10: Architecture of the SANE algorithm.

feature of SANE is its two level population structure. SANE features both low level populations of neurons, and high level populations of neuron combinations known as blueprints. The SANE architecture is shown in figure 2.10.

The blueprints act as a mechanism for discovering and remembering high quality combinations of neurons. Both populations undergo evolution, but in slightly different ways. Evolution of the neurons proceeds as in a conventional EA, with crossover and mutation between genetic information. Evolution of the blueprints has no effect on the genetic material itself, it simply changes which combinations of genetic material are evaluated together. SANE gives excellent performance on a number of important problems [62].

2.6 Overview of Distance Metrics

A *distance metric* is a function which defines a *distance* between elements of a set. A distance is a real valued number representing how far apart objects are. Distances can also be used to represent how similar two objects are. Similar objects have a small distance between them, while dissimilar objects have a large distance between them.

In order to qualify as a distance metric a function needs to satisfy our intuitive notions about the concept of distance. For example, the distance from x to y should be same as the distance from y to x. Formally, a distance metric on a set X is a function $d : X \times X \rightarrow \mathbb{R}$ which satisfies the following four conditions:

1. $d(x, y) \geq 0$. (non-negativity)

2. $d(x, y) = 0$ if and only if $x = y$. (identity of indiscernibles)

3. $d(x, y) = d(y, x)$. (symmetry)

4. $d(x, z) \leq d(x, y) + d(y, z)$. (triangle inequality)

There exist a number of useful distance metrics. Some of these include:

- **Discrete Distance**
  If $x = y$, $d(x, y) = 0$; otherwise $d(x, y)$ = 1. This is the simplest possible distance metric. The discrete distance encodes the concept that all points are isolated from each other [42]. For example, $d((3, 1, 2), (1, 1, 1)) = 1$.

- **Euclidean Distance**
  $d(x, y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2}$. The euclidean distance is the "ordinary" distance between two points that one would measure with a ruler [17]. For example, $d((3, 1, 2), (1, 1, 1)) = 2.24(2dp)$.

- **Taxicab Distance**
  $d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$. Also known as the Manhattan Distance. The taxicab distance is best envisioned as the distance which must be traveled when moving along the lines of a grid [48]. For example, $d((3, 1, 2), (1, 1, 1)) = 3$.

- **Hamming Distance**
  $d(x, y)$ is the number of places in which x and y differ [36]. For example, $d((3, 1, 2), (1, 1, 1)) = 2$.

Distance metrics are often used in machine learning to calculate the similarity between potential solutions. This can be useful for a number of purposes, including measuring and maintaining diversity, and aiding with local search.

2.7 Related Work

In this section we present a review of the most relevant literature with regards to the contributions presented in this thesis.

2.7.1 GP with Caching

The Problem

GP algorithms often require a long time to produce an acceptable solution, severely limiting algorithm applicability.

Many machine learning problem domains include tasks with strict time constraints. These tasks require algorithms which can produce useful solutions *quickly*. Algorithms which cannot produce useful solutions quickly cannot be applied to these tasks.

GP algorithms are often computationally expensive. Fitness evaluation requires each potential solution to be executed on a large number of training examples. While a single fitness evaluation is usually fast, there are

often hundreds of individuals or thousands of training examples, resulting in significant running times.

In short, the extensive execution time of many GP algorithms negatively impacts their applicability. When designing and developing new algorithms it is vital to maximize algorithm efficiency. In this way algorithm applicability is also maximized, increasing the number of tasks the algorithm is suited to solve.

There exist a number of techniques for reducing the execution time of GP algorithms. One technique is to reduce the number of training examples by carefully selecting a representative subset [30, 29, 94]. Another is to improve the fitness evaluation procedure directly, typically by parallelizing the fitness evaluations and the use of Graphics Processing Units [16, 77]. A third approach, and the approach focused on in this thesis, is to use caching. In the remainder of this section we explore related work on the use of caching for reducing the execution time of GP algorithms.

Caching for GP

Caching is one approach to reducing the execution time of GP algorithms. Caching stores partially computed results in memory and uses them to increase the efficiency of later computations. In this way caching trades a cost in memory for a saving in execution time.

Caching is a general technique successfully applied in virtually every computer science domain. Caches can be found in operating systems, web servers, hard drives, data bases, and search engines - to name a few examples. Caching can be successfully applied to any system where the same partial results are computed multiple times. Caching is a natural fit for GP given the repeated code in many GP programs.

A number of caching techniques have been proposed for decreasing the execution time of GP, with varying degrees of success.

Roberts [75] develops an inter-generational caching technique for image segmentation using Strongly Typed GP. Commonly occurring sub-

trees, together with their output, are cached using a combination of memory and hard disk. The paper shows that this technique can decrease the number of evaluations needed by up to 66%, resulting in a decrease in elapsed time of up to 52%.

Handley [37] develops an inter-generational caching technique for TGP by using a Directed Acyclic Graph (DAG) to represent the population of individuals. This approach conserves space by not duplicating identical subtrees. In addition, values computed by each subtree for each fitness evaluation is cached. This saves computation time by using a single execution to evaluate all instances of a repeated subtree. The paper shows that this approach can result in 30 fold reduction in the number of nodes executed. This result highlights the degree of repetition present in GP programs, and the potential savings if redundant executions can be avoided.

Langdon [50] uses caching to reduce the runtime of his GP algorithm. Despite the presence of side effects which severely limit the effectiveness of caching, runtime is still reduced by 32%.

Keijzer [43] examines a number of subtree caching mechanisms for TGP that are capable of adapting during the course of a run while maintaining a fixed size cache of already evaluated subtrees. One approach presented in this paper is to represent the population using a DAG. A second is to store a cache of subtrees represented as a DAG. Both approaches are shown to greatly reduce run time. In addition the results show large benefits for the use of even very small subtree caches.

Wong [91] develops a caching algorithm for TGP called SCHEME, based on subtree caching. TGP subtrees are hashed using a hashing algorithm developed by the author. This hash is used to position subtrees within the cache and determine subtree equivalence. SCHEME has the advantage of recognizing functionally equivalent subtrees. Subtrees which are functionally equivalent despite structural differences will hash to the same value, greatly increasing cache efficiency.

It should be noted that these techniques are all for TGP. To the author's

knowledge, there are no caching techniques specifically developed for the LGP architecture.

2.7.2 GP for Classification

Classification is an important problem domain, and one which forms the basis of empirical testing in this thesis. Furthermore, the classification tasks used in this thesis are all multiclass classification problems. Therefore in this section we briefly review related work on GP for classification, with a focus on multiclass classification.

There has been an enormous quantity of work in the area of GP for classification, and it is not the intent of this thesis to act as a comprehensive survey paper. To this end we limit ourselves to a representative sampling of important papers. Esperjo *et al* [24] provides an excellent survey of GP for classification for the interested reader.

GP can be applied to classification tasks in a variety of ways. The three major application categories *Preprocessing* [65], *Model Extraction*, and *Ensemble Classifiers*. Most of the papers published related to GP and classification focus on the application of GP to model extraction, that is, the induction of classifiers [24]. As this thesis focuses on *Model Extraction* via *Discriminant Functions* in the form of LGP programs, we will limit our review appropriately.

TGP for Classification

We begin by reviewing papers related to TGP for classification. While not the focus of this thesis, TGP is the conventional form of GP, used in the vast majority of literature. In addition, the use of LGP for classification is typically motivated by the drawbacks associated with using TGP for classification. Hence it is important give a brief overview of relevant literature related to TGP for classification.

TGP has been widely applied to various classification problems [49,

52, 68, 86, 96, 98, 100]. TGP has proven to be a highly effective approach for solving binary classification problems. Unfortunately TGP is not well suited to solving multiclass classification problems [99]. TGP programs output a single real number, which is difficult to interpret effectively as a class label in the context of multiple classes. Hence TGP classifiers often perform poorly on multiclass classification problems.

Several attempts have been made to improve the performance of TGP on multiclass classification problems. The majority of these approaches focus on new mapping functions for converting the single program output into a class label.

Loveard [53] applies GP to several binary and multiclass medical data sets. The methods of binary decomposition, static range selection, dynamic range selection, class enumeration and evidence accumulation are described. Dynamic range selection was found to be the method with the best mix of speed and accuracy.

Zhang and Smart [99, 81] introduce two mapping functions: centered dynamic range selection and slotted dynamic range selection. These two approaches outperform the basic static range selection on more difficult multiclass classification problems with many classes.

Smart and Zhang [80] introduce a probability based mapping function which uses a parameterized Gaussian to map tree output to a class label. This approach gives significantly improved performance on a number of multiclass classification problems, however performance suffers when the underlying output distribution is not a Gaussian.

LGP for Classification

In this subsubsection we review papers related to LGP for classification. While LGP is an effective technique for solving binary classification problems [**?**] it has particularly come into its own as a method for solving multiclass classification problems. With this in mind this thesis uses multiclass classification problems as the basis of empirical testing. Hence this

subsection will review LGP for classification with a focus on multiclass classification.

LGP is particularly well suited to solving multiclass classification problems. It has been demonstrated that LGP has significantly superior performance to conventional LGP on many multiclass classification problems [26, 67]. Despite this, relatively little work has been done in the area of linear genetic programming for multiclass classification.

Olague *et al* [67] develop a LGP approach for image classification which simultaneously solves the region selection and feature extraction tasks. The method searches for optimal regions of interest, using texture information as its feature space and classification accuracy as the fitness function. The paper shows that this LGP based approach gives superior performance to previous methods.

Fogelberg and Zhang [97] apply LGP to a number of multiclass image recognition problems with favorable results. The paper demonstrates that a LGP based approach significantly outperforms the basic TGP based approach. It also provides heuristic guidelines for initially setting system parameters.

Downey and Zhang [20] introduce a new mutation operator called selective mutation. Selective mutation identifies "bad"[2] instructions within the program and focuses mutation on these instructions. The paper shows that selective mutation significantly outperforms conventional mutation on several multiclass classification problems.

Downey, Zhang and Browne [21] introduce two crossover operators: *Class Graph Crossover*, and *Selective Crossover*. These operators are specifically designed for multiclass classification problems. Class graph crossover examines program struture to determine which instructions are responsible for each output. Selective crossover examines program performance on the training set, and attempts to select the most useful program code from each parent during crossover. The paper shows that these opera-

[2]bad instructions are those which contribute to incorrect output

tors significantly outperform conventional crossover on several multiclass classification problems.

Chapter 3

Data Sets

This chapter describes the data sets, GP parameters, implementation, and hardware used throughout the experiments in this thesis.

3.1 Data Sets

The algorithms presented in this thesis are evaluated on three tasks from the important problem domain of object classification. Object classification problems are ideal for comparing and contrasting LGP algorithm performance. They occur naturally in a wide range of applications,

The three tasks chosen are *Hand Written Digits*, a problem in classifying hand written digits; *Artificial Characters*, a problem in classifying artificial characters; and *Yeast*, a problem in predicting protein localization sites in yeast. These tasks are highly challenging due to a high number of attributes, a high number of classes, and noise in some sets. All three tasks are sourced from the UCI machine learning database [27].

3.1.1 Hand Written Digits

This data set consists of hand written digits with added noise. Each instance consists of an 8x8 square pixel array where each element is an in-

teger in the range 0 to 16. Each fitness case was generated by scanning a printed form to produce a 32x32 bitmap. This bitmap was divided into nonoverlapping 4x4 blocks, and the number of pixels in each block was summed to produce an integer in the range 0 to 16. Further details about the *Hand Written Digits* data set can be found in table 3.1.

Data Type	Multivariate
Task	Classification
Attribute Types	Real
Instances	3750
Attributes	64
Classes	10

Table 3.1: *Hand Written Digits* dataset information

3.1.2 Artificial Characters

This data set consists of vector representations of 10 capital letters from the English language. The capital letters represented are the following: A, C, D, E, F, G, H, L, P, R. Each instance is described by a set of segments (lines) which resemble the way an automatic program would segment an image. The data set has been artificially generated by using a first order theory which describes the letter structure together with a random choice theorem prover which accounts for heterogeneity in the instances. Further details about the *Artificial Characters* data set can be found in table 3.2.

3.1.3 Yeast

This data set consists of information about yeast cells, together with their protein localization site. Each yeast instance consists of 8 real numbered attributes giving the results of various tests performed on the yeast cells. These attributes are listed in table 3.3.

Data Type	Multivariate
Task	Classification
Attribute Types	Real
Instances	5000
Attributes	56
Classes	10

Table 3.2: *Artificial Characters* dataset information

Attribute	**Value**	**Type**
1	McGeoch's method for signal sequence recognition	Real
2	von Heijne's method for signal sequence recognition	Real
3	Score of the ALOM membrane spanning region prediction program	Real
4	Score of discriminant analysis of the amino acid content of the N-terminal region (20 residues long) of mitochondrial and non-mitochondrial proteins	Real
5	Presence of "HDEL" substring (thought to act as a signal for retention in the endoplasmic reticulum lumen)	Binary
6	Peroxisomal targeting signal in the C-terminus	Real
7	Score of discriminant analysis of the amino acid content of vacuolar and extracellular proteins	Real
8	Score of discriminant analysis of nuclear localization signals of nuclear and non-nuclear proteins	Real

Table 3.3: *Yeast* attribute information

Further details about the *Yeast* data set can be found in table 3.4.

Data Type	Multivariate
Task	Classification
Attribute Types	Real, Binary
Instances	1484
Attributes	8
Classes	10

Table 3.4: *Yeast* dataset information

3.2 GP Settings

In this section we describe the common parameters used in a typical LGP system.

- **Population Size:** This parameter controls the number of individuals in each generation. A typical setting for this parameter may be 500-1000 programs per generation.

- **Generations:** This parameter controls the number of times the evolutionary process is iterated before the LGP system is halted and a solution is given. A typical setting for this parameter may be 50-400 generations.

- **Instructions:** This parameter controls the number of instructions present in each program. In some GP systems the number of instructions is allowed to vary between programs, however in this thesis all programs have the same number of instructions. Program size is a problem dependent parameter, and a typical setting for this parameter may be 20-600 instructions.

- **Registers:** This parameter controls the number of registers available for use.

- **Terminal Set:** The terminal set consists of variables, features, and constants. In LGP the variables are the set of available registers. The features are problem dependent, and represent the set of all possible inputs. The constants are randomly generated floating point numbers in a user determined range.

- **Function Set:** The function set consists of functions available for use in solution programs. The function set is heavily problem dependent, but commonly used functions include $\{ +, -, \times, \div, if \}$. The if function returns 1 if the first argument is smaller than the second argument, and returns 0 otherwise. Note that $\div$ is the protected division operator which returns 1 if when the denominator is 0. This function set is used for all experiments present in this thesis.

- **Selection:** This controls the method used to sample programs from the population. All experiments in this thesis used tournament selection. Tournament selection has an addition parameter which controls the number of individuals which participate in each tournament. All experiments in this thesis use a tournament size of 5.

- **Crossover, Mutation, and Elitism Rates:** These three complementary parameters control what proportion of the program population in each generation will be formed using Crossover, Mutation, or Elitism. Each of these parameters is a fraction in the range [0,1], and all three parameters must sum to 1.0. Common parameter values are crossover = 0.6, mutation = 0.3, and elitism = 0.1. These are the parameter values used in this thesis.

Comparing TGP Depth to LGP Length

For those readers more familiar with TGP systems we briefly contrast the TGP depth parameter to the LGP length parameter. Both parameters serve to limit the maximum program size. The LGP length controls the size of each LGP program by specifying the maximum number of instructions present in each program. The TGP depth parameter controls the size of each TGP program by specifying the maximum depth of each tree.

In order to facilitate the comparison of LGP length and TGP depth we use a heuristic which relates the number of nodes in a maximum depth TGP program to the number of instructions in a maximum length LGP program. Each TGP program can be expressed in terms of LGP instructions. Therefore one way to relate TGP depth and LGP length is to calculate the number of LGP instructions required to encode a TGP tree.

Each TGP operator node must be represented by a single LGP instruction. Each TGP argument node will be contained within one of the operator node instructions. Therefore we require as many LGP instructions as there are operator nodes in the TGP tree. Assuming each TGP node must have 2 children, $1/2$ of all nodes are operator nodes. Therefore we require $1/2$ as many LGP instructions as there are TGP nodes. Hence each LGP instruction is equivalent to 2 TGP nodes.

Table 3.5 lists various TGP tree depth values, together with the number of instructions in an equivalent size LGP program. Note that numbers are rounded to the nearest integer as it is not possible to use half an instruction.

TGP typically uses trees of depth 4-10 (some rare cases use up to 17) [46] . Table 3.5 shows that this is equivalent to using LGP programs with 15-512 instructions. This is consistent with the number of instructions used in experiments in this thesis.

TGP	LGP
Tree Depth	Instructions
1	1
2	1
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512

Table 3.5: Comparison of LGP instructions and TGP tree depth

3.3 Implementation and Hardware

The LGP system that we use in our experiments was written by the author for the purposes of this thesis. JVUWLGP is a PLGP library written in the java programming language, designed for high execution speed over modularity.

All timed experiments were performed on a single machine with constant hardware. The machine used was a *Dell Optiplex GX745* with the following specifications:

- **Company:** Dell
- **Model:** Optiplex 745
- **CPU:** Pentium D 2.8GHz
- **Chipset:** Intel Q965 (ICH8) Express Chipset
- **Ram:** 2048MB DDR SDRAM

- **Network:** Broadcom 5754 Gigabit Ethernet LAN
- **Hard Disk:** 80GB Serial ATA 7200rpm disk
- **OS:** NetBSD

Experiments which did not require timing measurements were executed in parallel on a large number of machines (50-100) with similar hardware configurations (grids).

Chapter 4

Parallel Linear Genetic Programming

4.1 Introduction

4.1.1 Motivation

LGP programs can be viewed as a tangled web of dependencies. The input to instruction n is the state of the registers after the first $n-1$ instructions have been executed. Therefore the output of instruction n depends on the output of the first $n-1$ instructions. As a consequence, instructions cannot operate independently, and they only have meaning when executed together and in the correct sequence.

Specifically each instruction uses up to 2 registers as input, and writes to a single register as output. The value held in each input register is the output of some number of prior registers. We say that there is an *Instruction Dependency* between instruction a and instruction b if $a > b$ and b is responsible for the value held in a register which a takes as input.

An example of a LGP program with dependencies marked is shown in figure 4.1. In this example we represent each instruction dependency by an arrow between two instructions. For instance instruction 2 depends on

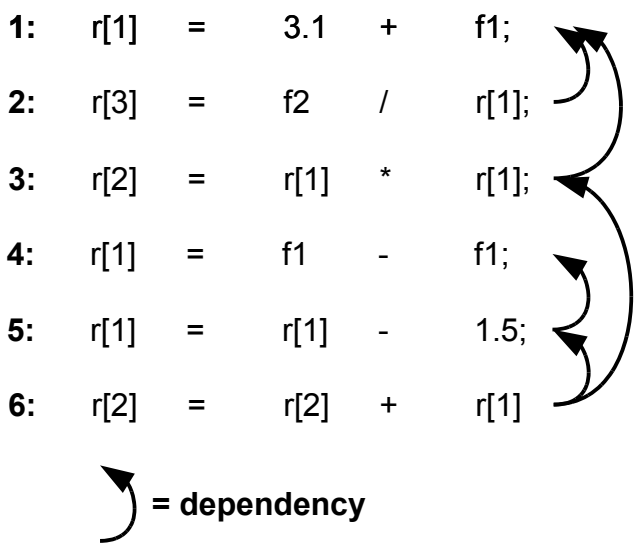


Figure 4.1: An example LGP program with the dependencies marked.

instruction 1, while instruction 6 depends on instruction 3 and instruction 5.

The power of the LGP paradigm is tightly linked to the concept of instruction dependencies. Results computed early in a LGP program can be reused many times by later instruction sequences. This makes it possible to express complicated solutions compactly with a short sequence of LGP instructions. At the same time each time we reuse a result we introduce a new instruction dependecy. The upshot of this is that the number of dependencies is directly proportional to the complexity of the solution.

Dependencies allow LGP programs to be compact and powerful, however they can make it extremely difficult to evolve good solutions. Dependencies often interfere with evolution by causing large, random changes in program output when the program is modified in any way. Each instruction directly or transitively affects the output of many other instruc-

tions. Hence modifying any single instruction can affect the output of a large number of subsequent instructions. This ultimately results in massive changes to the final register values, and hence program output. An example of how dependencies can disrupt LGP programs during evolution is shown in figure 4.2.

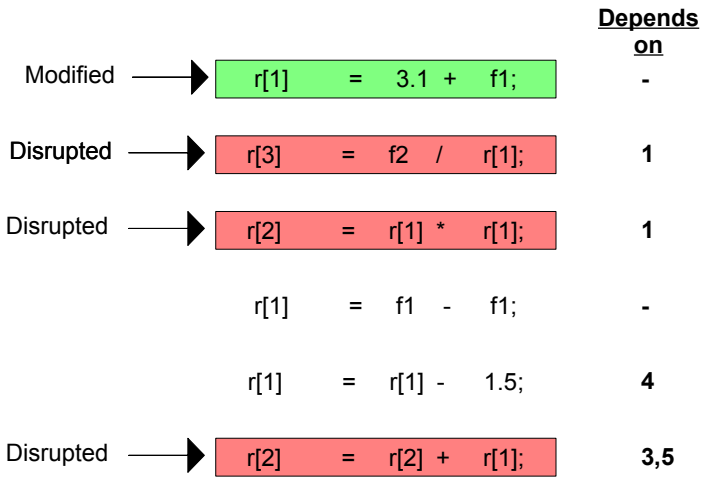


Figure 4.2: An example of program disruption. Modifying the first instruction disrupts the execution of three other instructions.

Large, random changes to program output are detrimental to evolutionary computation because they interfere with spatial locality. Spatial locality is the concept that similar solutions should have similar output, and hence similar fitness. As a direct consequence of spatial locality, small changes to the program should result in small changes to program output. This property allows us to perform local search by taking existing good solutions and making small changes to them. In other words we can exploit known good solutions to find other good solutions by investigating similar individuals.

Instruction dependencies make it difficult to perform local search on

LGP programs, since there is no longer a link between programs with similar structure, and programs with similar output. Local search relies on finding programs with similar outputs by examining programs with similar instructions. Instruction dependencies mean small changes to the program can result in massive changes to program output. Hence instruction dependencies prevent effective local search in the space of LGP programs.

By preventing effective local search, instruction dependencies interfere with the evolution of LGP programs in a number of ways. We now discuss some of these problems in detail.

Mutation

The primary goal of mutation is to maintain diversity in the population. However in many systems mutation also acts as a local search operator. Mutation takes programs stochastically selected for their high fitness and makes small random changes to the program code. According to the theory of spatial locality, mutation should result in programs with similar output and similar fitness.

Unfortunately, mutation becomes an ineffective local search operator in a LGP system due to the presence of instruction dependencies. In LGP programs the small changes caused by mutation often result in disrupted dependencies. Disrupted dependencies cause large changes in program output, invalidating the assumptions we used to motivate mutation. In summary, mutation often results in low fitness offspring due to large changes in program output, preventing effective local search.

Crossover

Crossover becomes an ineffective operator in a LGP system due to the presence of instruction dependencies [18]. Crossover takes two programs selected for their high fitness and exchanges code between them. Crossover relies on the concept of building blocks: short, high fitness instruction se-

quences which can be exchanged between programs. Unfortunately the concept of building blocks is fundamentally flawed in a LGP system, because code cannot be exchanged intact between programs. Instruction sequences in LGP programs consist of two parts: the code itself, and the dependencies which that code relies on. When code is exchanged between programs, dependencies are disrupted, meaning that any exchanged instruction sequence will execute differently in crossover offspring. In summary, code exchanged between LGP programs performs differently in donor and offspring due to disrupted instruction dependencies. This makes crossover in LGP more of a macromutation operator, rather than an effective method of combining the strengths of two programs [3, **?**].

Large Programs

Instruction dependencies are a particular problem for large LGP programs. As the size of the LGP programs increases, the number of chained and interwoven dependencies also increases. Each instruction may influence many subsequent instructions through instruction dependencies. In long programs there are a greater number of potential instruction dependencies. As a direct consequence of this, instructions positioned early in large LGP programs have a disproportionately large effect on the final program output. Furthermore, if these instructions are modified, a large number of instruction dependencies will be broken, causing a large change in program output.

Due to a larger number of instruction dependencies, the disruptive effect of crossover and mutation is particularly pronounced in large LGP programs. Unfortunately, it is often necessary to use large LGP programs. Large programs often have more power, as they are able to explore a larger solution space. To solve complicated problems we often require complex solutions which can only be expressed by large programs.

In summary, LGP programs do not scale well. It is difficult to evolve large LGP programs because crossover and mutation are ineffective. Each

time a large LGP program is modified, many instruction dependencies are broken. Broken dependencies cause large changes in program output, preventing local search and negating the advantages of maintaining a population of individuals. To solve this problem it appears necessary to devise a form of LGP where few instruction dependencies are disrupted during evolution.

4.1.2 Chapter Goals

In this chapter, we aim to develop a new program structure for LGP which limits the number of instruction dependencies while still allowing compact, powerful programs. By limiting the number of instruction dependencies disrupted during evolution, we aim to increase the usefulness of crossover and mutation, and hence improve the overall performance of LGP. Specifically, this chapter has the following research objectives:

- To isolate and identify the features of conventional LGP program structure which prevent effective crossover and mutation.
- To develop a new program structure for LGP which limits these problems while preserving the power of conventional LGP.
- To compare the performance of the new program structure with the performance of a conventional program structure over a range of parameter settings on several classification problems.

4.2 Parallel Linear Genetic Programming

The simplest way to ensure there are no dependencies between any two given instructions is to execute them independently. This idea is the driving force behind Parallel Linear Genetic Programming (PLGP). Suppose we separate each LGP program into a number of short instruction sequences, and execute these instruction sequences independently. Then

there will be no instruction dependencies between any two instructions in different sequences, as no instruction in one sequence can affect the output of any instruction in any other sequence. Based on this idea we have developed a new program structure, called Parallel Linear Genetic Programming (PLGP).

4.2.1 Program Structure

Parallel LGP (PLGP) is a LGP system where each LGP program consists of n independent instruction sequences called *Factors*. An example PLGP program is shown in figure 4.3. A PLGP program consists of n factors which are evaluated independently, to give n results vectors. These vectors are then summed to produce a single results vector. Formally let V_i be the ith results vector and let S be the summed results vectors. Then $S = \sum_{i=1}^{n} V_i$.

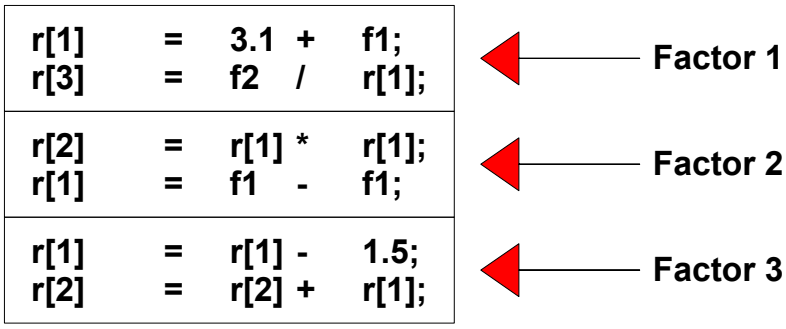


Figure 4.3: An example of a PLGP program with three factors, each with two instructions.

An example of PLGP program execution is shown in figure 4.4 and contrasted to an example of LGP program execution. In LGP all instructions

are executed in sequence using a single set of registers, to produce a single results vector as output. In PLGP the instructions in *each program factor* are executed on *their own set of registers* to produce n (in this case 3) results vectors. In this example the program factors are separated by horizontal lines, so our PLGP program consists of 3 factors. These results vectors are then summed to produce the final program output. Notice how our LGP program and our PLGP program have *the same instructions* but produce *different outputs*. This is because in our LGP program the results of earlier computations are stored in the registers and can be reused by later computations, while in PLGP each factor begins execution with all registers initialized to zero.

PLGP programs may consist of a large number of instructions, but each instruction is executed as if it was part of a very short program. Each factor is executed independently, so it is impossible for any instruction to influence the output of an instruction in a different factor. Hence there are no instruction dependencies between instructions in different factors. In other words the maximum number of chained instruction dependencies is now limited by the number of instructions per factor, rather than the number of instructions per program. Furthermore modifying the instructions in one factor cannot disrupt the instructions in any other factor as no inter-factor instruction dependencies exist.

4.2.2 Evolution of PLGP Programs

Our initial motivation for developing the PLGP program structure was based on the ineffectiveness of crossover and mutation when applied to conventional LGP programs. We have already outlined a form of LGP which minimizes the number of instruction dependencies disrupted during evolution. We now consider how best to adapt the crossover and mutation operators to work with our new PLGP program structure.

LGP program execution

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

$$= \begin{bmatrix} -1.5 \\ 8.74 \\ 0.94 \end{bmatrix}$$

PLGP program execution

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

$$= \begin{bmatrix} 3.2 \\ 0 \\ 0.94 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ -1.5 \\ 0 \end{bmatrix} = \begin{bmatrix} 1.7 \\ -1.5 \\ 0.94 \end{bmatrix}$$

Figure 4.4: Contrasting PLGP program execution with LGP program execution

Factors

Each PLGP program consists of a number of factors, therefore it is important to decide how many of these factors will participate in any single evolution. As discussed in the motivation, it is important that offspring typically generate similar output to their parents. This is difficult to ensure with conventional LGP programs, but simple to achieve with PLGP programs. As each factor is executed independently, any factor which is not modified during evolution will continue to produce identical output, regardless of how other factors are modified. The similarity in output between parents and offspring will be directly related to the number of fac-

tors which are left unchanged during evolution.

Therefore we specify that each evolution should only affect a single program factor. In this way we maximize the number of unchanged factors. This ensures that program offspring will produce similar output to their parents regardless of the changes which occur during evolution.

Mutation

The mutation operator for PLGP is simply the mutation operator for LGP applied to a single factor. Although this may appear to be a simple change, in reality it has far reaching consequences. By limiting mutation to a single PLGP program factor we ensure that regardless of the mutation, the offspring's output will be similar to that of its parent. This is a significant advance, as it allows the mutation operator to be an effective local search operator, in addition to its role in maintaining population diversity. PLGP offspring produced by mutation will have similar output, and hence are likely to be of similar fitness. In contrast, LGP offspring produced by mutation often have very different output, and hence have random fitness.

Crossover

The crossover operator for PLGP exists in two forms, each with their own distinct advantages.

The first approach to crossover in PLGP is to exchange entire *factors* between PLGP *programs*. Factors are a natural level of granularity for crossover because they are entirely self contained in their execution. Each factor is executed independently, so each factor will produce identical output in both parent and offspring. This is extremely important as it allows crossover to exchange program code with the guarantee that no instruction dependencies will be disrupted. The only downside of factor crossover is that it severely limits the ways in which code can be recombined.

The second approach to crossover in PLGP is to exchange *instructions* between *factors*. The idea is to select two factors, and perform crossover between them as if they were LGP programs. This approach allows us to exchange code between PLGP programs at a lower level of granularity. The downside of this approach is that crossover can now disrupt instruction dependencies. Instruction dependencies still exist between instructions in the same PLGP program, therefore exchanging code between factors may cause a large change in factor output. On the other hand, this effect is countered by the face that all other factors continue to produce identical output.

When exchanging instructions between factors it is important to carefully consider whether the factors in PLGP programs have an ordering. While program output will remain constant regardless of the order in which the factors are executed, it can be useful to define an explicit first factor, second factor etc. If program factors are not ordered, then we can view each PLGP program as a *set* of factors. Conversely if the program factors are ordered, then we can view each PLGP program as an ordered list of factors. These two approaches to PLGP have a large impact on how we perform crossover, regardless of whether we perform crossover between instructions or factors. Furthermore each approach determines how we view the population gene pool.

If each PLGP program is a set of factors, then we are forced to perform crossover between *randomly selected* factors. Selecting factors at random allows program code to be exchanged between any two factors in any two programs. If we allow random crossover in this way, then we can view all genetic code as belonging to a single 'genetic population' where any instruction can be exchanged with any other instruction.

Conversely, if each PLGP program is an ordered list of factors, then we can limit which factors we exchange code between. Specifically, we can limit crossover to factors with the same position, which we will refer to as *equivalent factors*. The concept of equivalent factors is illustrated in

figure 4.5. If we restrict crossover based on factor number then we are effectively creating a number of genetic sub-populations where there is no inter-population genetic flow. In other words genetic material can be exchanged within each sub population, but there is no transfer of genetic material *between* sub populations. We borrow some terminology from the area of Cooperative Coevolution and term this kind of crossover as crossover with *enforced sub populations (ESP)* [31].

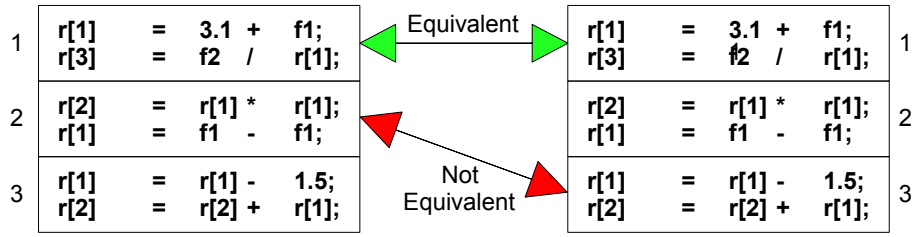


Figure 4.5: An example of a PLGP program with three factors, each with two instructions.

Using ESP crossover has been shown to give improved classification accuracy when applied in the area of cooperative coevolution. The theory is that by limiting the exchange of genetic material to within sub populations we encourage speciation. This in turn increases the likelihood of crossover between compatible segments, and hence improves the likelihood of a favorable crossover outcome. This theory is supported by preliminary results, which demonstrate that PLGP using ESP significantly outperforms constraint free crossover.

4.2.3 PLGP Program Topologies

It is important to note that there are many different ways of arranging the instructions in a PLGP program. For instance a PLGP program with 10 in-

structions could consist of either 2 factors of 5 instructions, or 5 factors of 2 instructions. We refer to these different arrangements as *Program Topologies*. By choosing between different program topologies we control the average number of instruction dependencies present. Program topologies with a large number of small factors have few instruction dependencies. Conversely, topologies with a small number of large factors have many instruction dependencies.

The relationship between program topologies and instruction dependencies is particularly clear if we examine the extreme cases. On the one hand, program topologies where each factor consists of a single instruction have no instruction dependencies. On the other hand, topologies consisting of only a single factor containing all of the instructions are equivalent to conventional LGP programs, and have equally many instruction dependencies.

It is important to choose an appropriate program topology. The topology controls the number of instruction dependencies, which in turn controls the solution complexity. If too many factors are used, instructions cannot interact, and we limit ourselves to overly simplistic solutions. In contrast, if too few factors are used, then there are too many instruction dependencies interfering with evolution.

In addition, the number of factors controls the number of unique components available to solve the problem. Many problems can be naturally decomposed into a number of subproblems. In this case a good solution should consist of a solution to each of the subproblems. The key to solving such a problem is to use an appropriate number of factors, and hence an appropriate number of subpopulations.

4.3 Experimental Setup

We compare the performance of PLGP to that of conventional LGP. This section outlines the experiments and parameters used to facilitate this com-

parison.

4.3.1 Data Sets

The three data sets described in chapter 3 will form the basis for our experiments.

4.3.2 Program Topologies

Preliminary results show that a large number of reasonable topologies give near optimal results. The program topologies used in our experiments have been previously determined to be optimal to within some small tolerance.

4.3.3 Parameter Configurations

The parameters in table 4.1 are *constant parameters*. These are the parameters which will remain constant throughout all experiments. These parameters are either experimentally determined optima, or common values whose reasonableness is well established in literature [46].

Table 4.1: Experimental Parameters

Parameter	Value
Population	1000
Generations	400
Mutation	30%
Elitism	10%
Crossover	60%
Tournament Size	5
Registers	10

We allow terminal constants in the range [-1,1], and a function set containing Addition, Subtraction, Multiplication, Protected Division, and If. The data set is divided equally into a training set, validation set, and test set, and results are averaged over 30 runs. All reported results are for performance on the test set. The fitness function used is the number of misclassified training examples. Finally all initial programs in the population consist of randomly chosen instructions.

4.3.4 Experiments

The parameters in table 4.2 are the *experiment specific parameters*. Each column of the table corresponds to the parameter settings for a specific experiment. Each experiment has two components, a LGP stage and a PLGP stage. In the LGP stage we determine the classification accuracy of a LGP system using programs of the specified length. In the PLGP stage we repeat our experiment but we use PLGP programs of equivalent length. Note that we repeat each experiment 30 times and average the results.

Table 4.2: Experiments

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Total Instructions	10	20	35	50	100	400
# PLGP factors	2	4	5	5	10	20
PLGP factor Size	5	5	7	10	10	20

4.4 Results

In this section we compare the effectiveness of LPG and PLGP. We present the results of the experiments detailed in section 4.3 together with discussion.

The following graphs compare the performance of LGP with PLGP as a classification technique on the three data sets described in chapter 4.3. Figure 4.6 compares performance on the Hand Written Digits data set, Figure 4.7 compares performance on the Artificial Characters data set and Figure 4.8 compares performance on the Yeast data set. Each line corresponds to an experiment with programs of a certain fixed length. Program lengths vary from very short (10 instructions) to very long (400 instructions).

We examine the statistical significance of these results by performing a students t-test on the fitness at generation 400. Figure 4.9a tests the significance of the results in figure 4.6. Figure 4.9b tests the significance of the results in figure 4.7. Figure 4.9c tests the significance of the results in figure 4.8. In these results n is the number of trials, SD is the standard deviation, and p is the p-value resulting from the t-test. Note that by convention a p value smaller than 0.05 is considered significant.

4.4.1 Discussion

These results are entirely in line with our expectations.

Performance with small programs

LGP performs at least as well as PLGP when small programs are used. In the Hand Written Digits data set LGP significantly outperforms PLGP for programs of length 10-20. In the Artificial Characters data set the performance of LGP and PLGP is comparable for programs of length 10. In the Yeast data set the performance of LGP and PLGP is comparable for programs of length 10-35.

Short programs have few instruction dependencies and short instruction dependency chains. This means crossover and mutation are less disruptive when applied to short LGP programs. Hence minimising the number of instruction dependencies by dividing each program into numerous factors is of negligible benefit for short programs.

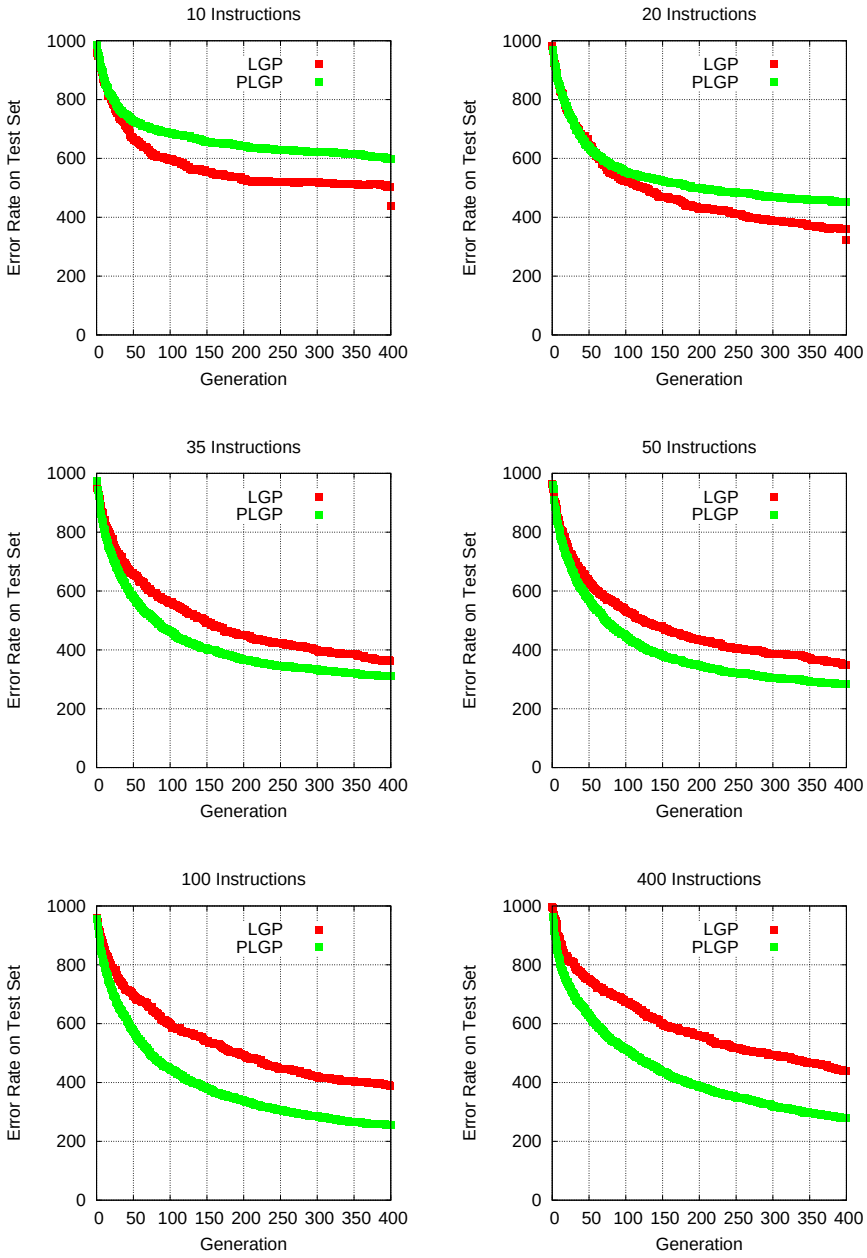


Figure 4.6: LGP vs. PLGP on the *Hand Written Digits* data set

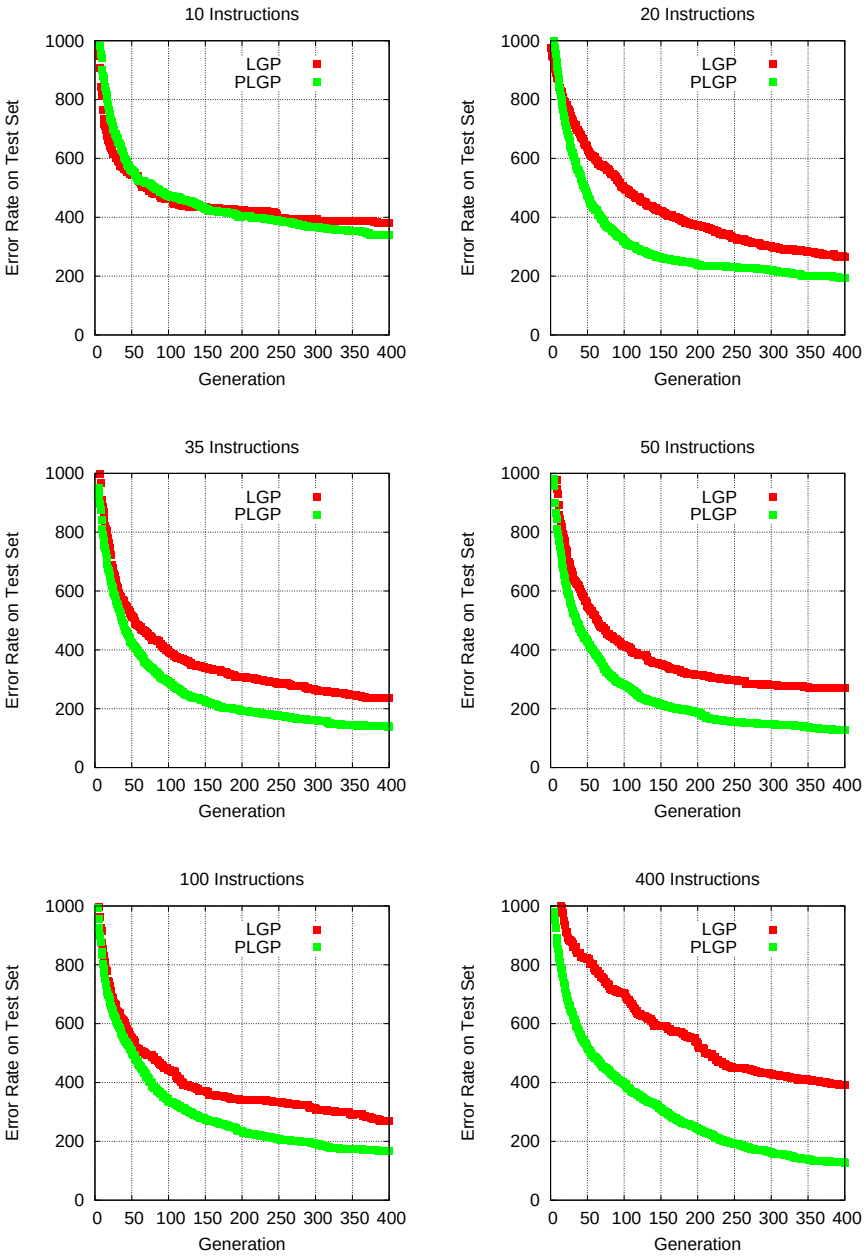


Figure 4.7: LGP vs. PLGP on the *Artificial Characters* data set

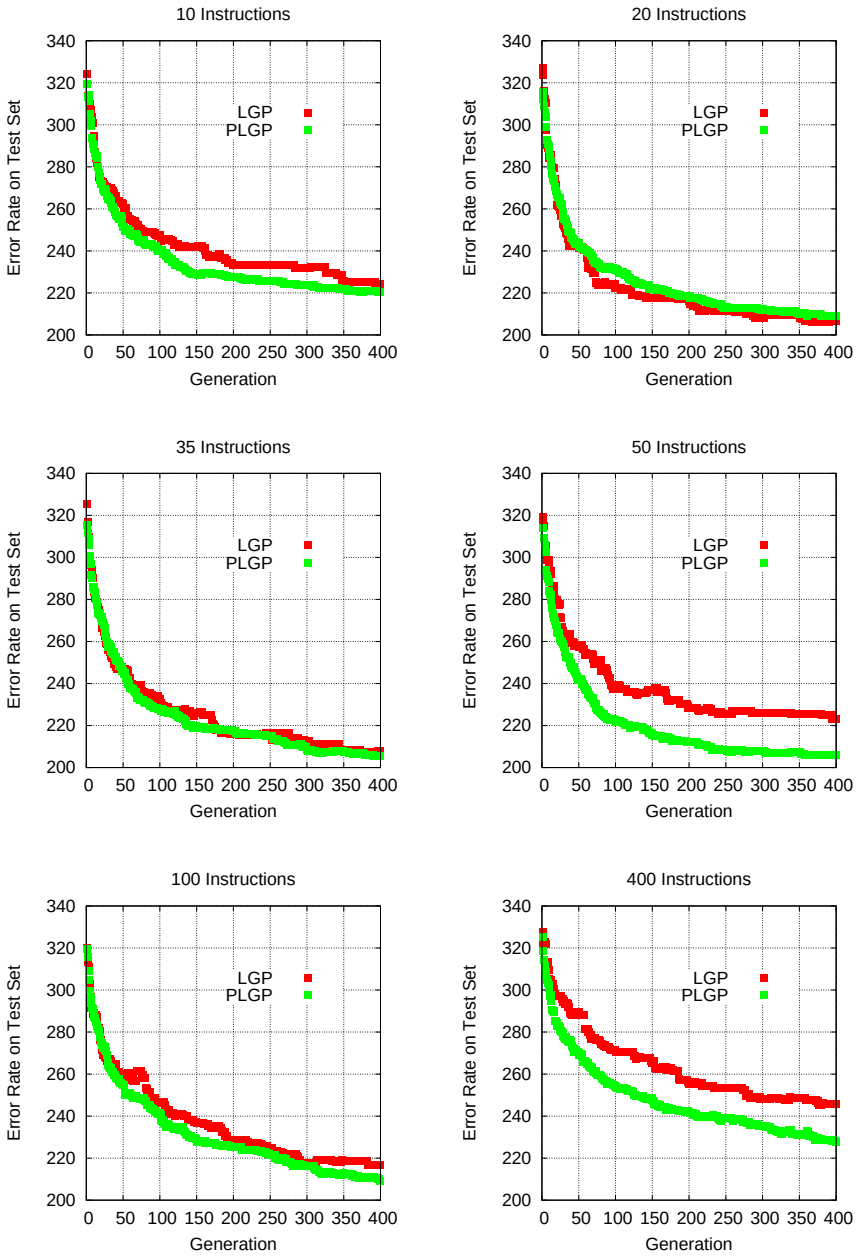


Figure 4.8: LGP vs. PLGP on the *Yeast* data set

Figure 4.9: Significance of Results: LGP vs. PLGP

		LGP		PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	10	503.3	31.86	597.2	35.81	30	0.0001	YES (-)
Exp2	20	361.4	36.88	451.63	47.78	30	0.0001	YES (-)
Exp3	35	363.7	65.30	309.6	47.57	30	0.0005	YES (+)
Exp4	50	348.4	45.62	284.73	53.69	30	0.0001	YES (+)
Exp5	100	387.2	61.83	256.23	42.44	30	0.0001	YES (+)
Exp6	400	437.5	36.82	278.96	52.79	30	0.0001	YES (+)

(a) *Hand Written Digits*

		LGP		PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	10	379.1	86.88	340.24	93.60	30	0.0971	NO
Exp2	20	264.7	53.36	192.44	53.22	30	0.0001	YES (+)
Exp3	35	234.93	102.84	139.6	73.49	30	0.0001	YES (+)
Exp4	50	269.8	92.68	127.28	79.98	30	0.0001	YES (+)
Exp5	100	269.7	100.08	166.52	78.88	30	0.0001	YES (+)
Exp6	400	390.6	153.93	118.68	46.75	30	0.0001	YES (+)

(b) *Artificial Characters*

		LGP		PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	10	224.3	12.48	220.56	11.95	30	0.1836	NO
Exp2	20	206.7	8.1	209.13	10.1	30	0.3082	NO
Exp3	35	208	15.5	205.5	15.2	30	0.4522	NO
Exp4	50	223	15.2	205.9	11.31	30	0.0001	YES (+)
Exp5	100	216.7	18.1	208.9	11.0	30	0.0001	YES (+)
Exp6	400	245.7	14.9	227.56	13.41	30	0.0001	YES (+)

(c) *Yeast*

Furthermore, short PLGP programs often have insufficient power to solve interesting (and difficult) problems. Instruction dependencies cause disruption during evolution, however their primary purpose is to allow results to be reused. If we remove too many instruction dependencies, then we severely limit the power of our LGP programs. This is the case with short PLGP programs. Factors in short PLGP programs are small, preventing dependency chains forming, and limiting short PLGP programs to expressing simplistic solutions.

However the flip side of this result is that short LGP programs are not what we are really interested in. Typically, short programs are not powerful enough for most interesting applications. On difficult problems neither LGP or PLGP can achieve good performance using short programs.

Performance with large programs

PLGP significantly outperforms LGP for large programs. In all three data sets PLGP significantly outperforms LGP whenever program length exceeds some minimum. For the Hand Written Digits this holds for all programs larger than 20 instructions. For the Artificial Characters data set this holds for all programs larger than 10 instructions. For the yeast data set this holds for all programs larger than 35 instructions.

Large LGP programs have many instruction dependencies and long instruction dependency chains. These dependencies result in major disruptions occurring during program evolution. In contrast long PLGP programs have fewer instruction dependencies, and the length of each instruction dependency chain is strictly limited. This means crossover and mutation can be applied to PLGP programs with minimal disruption occurring. Hence evolution of PLGP programs results in a larger number of viable offspring than evolution of LGP programs, greatly improving algorithm convergence time.

Furthermore, large PLGP programs are usually powerful and expressive despite the limited instruction dependencies. In large PLGP programs

it is possible to divide each program into factors of moderate size. Dependency chains can form within the factors, allowing for complicated, yet compact solutions. Hence large PLGP programs are both expressive and robust.

This result is important because many interesting problems require the expressive power of large programs. Traditionally it has been difficult to solve these problems using LGP. We hypothesize that this is because large LGP programs are disrupted during evolution. With this in mind it is clear that PLGP is better suited to solving problems which require complicated solutions.

Optimal Size

There is an optimal size for programs. Short programs are insufficiently expressive: it is not possible to easily *express* good solutions using only a very small number of instructions. Long programs are overly expressive: while it is possible to express good solutions the search space is too large, making it inefficient/difficult to *find* good solutions. This optimal size is problem dependent, as more complicated problems typically require more complicated solutions. Hence there is an optimal program size for each problem, a balance between expressive power and search space complexity.

The optimal size for PLGP programs is significantly larger that the optimal size for LGP programs. In our three experiments the optimal size for LGP programs occurs between 20-50 instructions. However in these same three experiments the optimal size for PLGP programs occurs between 50-400 instructions.

Instruction dependencies in large LGP programs cause major disruptions during evolution, negatively impacting algorithm performance. Conversely the PLGP program structure alleviates this problem, allowing large PLGP programs to give good performance. Hence the optimal size of LGP programs is smaller than the optimal size of PLGP programs.

Optimal Performance

It is clear that PLGP gives rise to better fitness solutions than LGP. This is intrinsically linked to the optimal program size for each of these two methods: PLGP has a higher optimal program size than LGP. Both program representations are equally able to *express* powerful solutions, however only PLGP is able to actually *find* these powerful solutions.

It is difficult to find good solutions when using large LGP programs because evolution is often disruptive. Large LGP programs may be able to express many high fitness solutions, however it is difficult to locate these solutions within the search space. Hence LGP often achieves the best performance using shorter programs, not because these are the best possible solutions, but rather because these are the only solutions it is possible to find.

It is possible to find good solutions when using large PLGP programs because evolution is typically less disruptive. Evolutionary search using crossover and mutation is effective for finding large PLGP programs with high fitness. Hence PLGP often achieves the best performance using longer programs, representing more complicated solutions.

In summary, when using LGP we are forced to limit our search to simpler solutions, whereas when using PLGP we can broaden our search to include more complicated solutions. Hence PLGP can effectively explore a broader range of potential solutions, allowing the discovery of higher fitness solutions.

Optimal Parameter Range

One of the most time consuming steps in deploying a computational machine learning system is determining good parameter values. If poor parameter values are chosen, then the system will give poor results. Hence choosing good parameter values is critical to the successful deployment of a machine learning system. Unfortunately, the optimal parameter values

are strongly problem dependent, and often the only option is to determine them experimentally. This involves performing numerous trial runs with different parameter settings until a good setting is found. If the number of good parameter settings is very small, many experiments may be required, and parameter determination can be extremely costly. Conversely if the number of good parameter settings is large, few experiments will be required, and parameter determination can be accomplished cheaply. Therefore the time required to find good parameter values is tightly bound to the number of good parameter settings, and *a system with a large number of good parameter settings is highly desirable*.

It is significantly easier to determine a good value for the program length parameter in PLGP than LPG. In LGP there are only a small number of program lengths which give good performance. Both small programs and large programs give poor performance, leaving only a small number of acceptable settings. This means experimentally determining a good value for the program length parameter is expensive in LGP. In PLGP there are a large number of program lengths which give good performance. Small programs give poor performance, but in contrast to LGP, large programs still give good performance. This is a significant advantage since it greatly simplifies the task of determining a good value for the program length parameter.

4.5 Chapter Summary

PLGP is a technique designed to minimize building block and program disruption by parallelizing the standard LGP architecture. Longer LGP programs are easily disrupted during evolution due to large instruction dependency chains. These chains mean that small changes to program structure can often result in massive changes to program output. To reduce the number of instruction dependencies disrupted during evolution and limit the length of instruction dependency chains we introduced the

PLGP architecture. PLGP programs consist of n short instruction sequences called factors which are executed independently. The final program output is calculated by summing all program factors to produce a single results vector.

PLGP programs are better suited to evolutionary search. Because each factor is executed independently there are no inter-factor instruction dependencies. While intra-factor instruction dependencies still exist, the length of instruction dependency chains is limited by the factor size. Hence crossover and mutation can be applied to PLGP programs without significant program disruption occurring. This allows PLGP to effectively exploit larger programs for significantly superior results. Our empirical tests support this: long PLGP programs significantly outperform long LGP programs on all data sets. In addition our results show that by exploiting the ability of PLGP to utilize large programs it is possible to obtain a significant overall improvement in performance. Both theory and results clearly demonstrate the benefits of this new parallel architecture.

4.5.1 Next Step

PLGP is an important improvement over conventional LGP, however there remain several unsolved issues. In particular, large PLGP programs with many factors are slow to converge, because evolution is limited to a single factor in each generation. This issue will be addressed in chapter 5.

Chapter 5

Cooperative Coevolution for PLGP

5.1 Introduction

One major problem with PLGP is its slow initial convergence speed. It is important to be able to iteratively improve good existing solutions through small changes, however it is equally important to be able to rapidly improve the initial population through large scale changes. PLGP is highly successful at improving existing solutions through small changes and local search. Because evolution is limited to a single factor in each generation, evolution is constrained to small changes in program output. The downside of this is that it is not possible to rapidly improve bad PLGP programs.

Each PLGP program is separated into n factors which together produce the final program output. In order for the program to produce the correct output, each of these factors needs to be "correct" [1]. During each algorithm iteration a single factor is modified during evolution. Furthermore, a factor will often require several iterations of the evolutionary operators

[1] A correct factor is one which cooperates with the other factors to produce correct output.

before it becomes correct. Therefore to arrive at a program consisting entirely of correct factors it is necessary to evolve each factor many times. In other words it will typically take many iterations to evolve programs which produce the desirable output.

Slow PLGP program convergence is particularly pronounced in large PLGP programs. The time taken for PLGP programs to converge is directly related to the number of factors present. Each factor must be independently evolved, and PLGP can only evolve a single factor in each generation. The slow convergence of large PLGP programs is compounded by the high execution time of large programs.

If we want to improve the convergence time of PLGP we need to concurrently improve multiple factors within a single generation. Improving a single factor in each generation is an excellent approach for fine tuning high fitness solutions, however it is a slow and cumbersome method for improving an initial population of randomly generated programs.

The simplest approach is to evolve multiple factors in each program during each generation, however this would negate the hard won benefits of PLGP. Evolving multiple factors within a single program would cause massive changes in program output, preventing effective search. Furthermore, it would be difficult to identify good factors when they occurred, as both good and bad factors would concurrently evolve within the same program, giving overall poor fitness.

We turn to the method of Cooperative Coevolution (CC) to improve the convergence time of PLGP. Cooperative coevolution rapidly evolves high fitness partial solutions (building blocks), which are combined into high fitness solutions. If we can use CC to rapidly evolve high fitness factors in parallel, then we expect to combine those factors to form high fitness PLGP programs. By evolving the factors in parallel, we hypothesize that CC could greatly decrease the number of generations required to improve an initial population.

Our task then, is to adapt the CC algorithm to function effectively with

the PLGP architecture, with the goal of greatly decreasing PLGP convergence time.

5.1.1 Chapter Goals

In this chapter, we aim to develop a cooperative coevolution algorithm for PLGP. By using CC to evolve PLGP programs we aim to improve the convergence time of large PLGP programs with many factors. Specifically this chapter has the following research objectives:

- Develop a CC algorithm for PLGP.
- Develop a hybrid algorithm which exploits the strengths of both PLGP and CC PLGP.
- Empirically compare the performance and execution time of all three algorithms.

5.2 CC for PLGP

We construct a Cooperative Coevolution algorithm for PLGP, using as inspiration the ideas of the SANE [62] algorithm.

5.2.1 Program Structure

SANE is a CC framework which evolves both partial and complete solutions in parallel. If each complete solution consists of n partial solutions, then SANE makes use of n populations to evolve these partial solutions in parallel. In CC, a *Population* is a set of individuals which are allowed to exchange genetic material. If two individuals are in different populations they cannot undergo recombination. A population of partial solutions is called a *subpopulation*. SANE has a single additional population of complete solutions. Complete solutions, called *blueprints*, consist of pointers to

partial solutions, and serve as a means of remembering high fitness partial solution combinations.

To implement our own CC PLGP algorithm we construct $n + 1$ populations, where n is the number of PLGP program factors. n of these populations are subpopulations of factors. The single remaining population is a population of blueprints. A blueprint consists of a sequence of n pointers, each of which points to a factor from one of the subpopulations. Note that the first pointer points to a factor from the first subpopulation, the second pointer points to a factor from the second subpopulation, etc. In other words a blueprint consists of n pointers, each of which points to a single subpopulation in a predetermined order, with all blueprints possessing identical pointer ordering. The only variation between blueprints is which individual in each subpopulation is pointed to. The architecture of our CC PLGP algorithm is shown in figure 5.1.

In this example we have three subpopulations each consisting of four factors, and one population with five blueprints. Each blueprint consists of three pointers, one pointer for each subpopulation. Each program factor consists of three instructions. Therefore each blueprint has a total of 9 instructions.

Note that the mapping from blueprints to factors is neither injective nor surjective. The number of blueprints each factor participates in varies from factor to factor. Some factors may participate in many blueprints, while others may participate in none. Due to selection pressures, factors which perform well will participate in more blueprints. However mutation acts to maintain diversity in the blueprint population by modifying blueprints with factors selected uniformly at random. Hence assuming a reasonable number of blueprints, statistical arguments show that in general each factor participates in a reasonable number of blueprints.

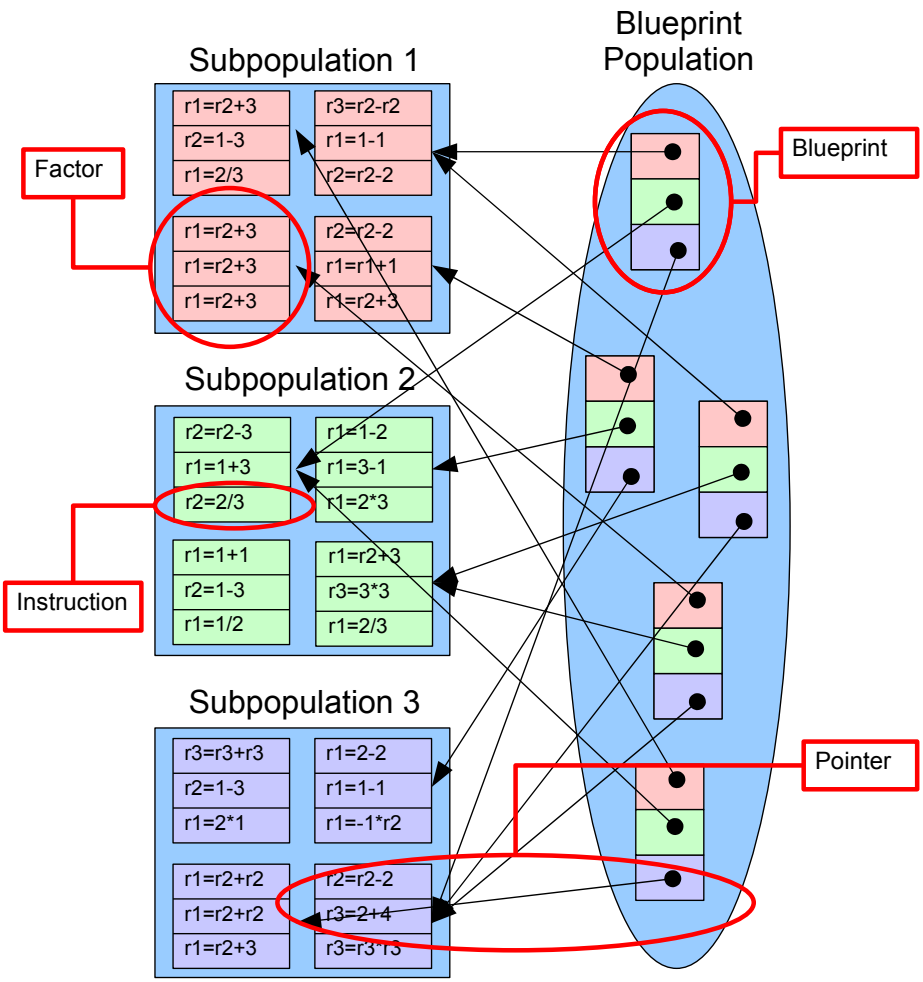


Figure 5.1: Architecture of our CC PLGP algorithm

5.2.2 Evaluation

In the CC framework partial solutions must be combined into complete solutions before fitness evaluation can occur. These complete solutions are evaluated, and the resulting fitness values are used to calculate the fitness of the partial solutions. In our CC PLGP system, complete solutions take the form of blueprints. Blueprints serve as a means of remembering high

fitness complete solutions. Thus program fitness evaluation consists of two distinct stages: blueprint evaluation followed by factor evaluation.

In the first stage, the blueprints are evaluated against the training examples to determine their fitness. Blueprints are complete potential solutions, so blueprint execution is identical to PLGP program execution.

In the second stage, the fitness values for the blueprints are used to calculate fitness values for the factors. A good factor is one which is used in many high quality blueprints, and conversely a poor factor is one which participates solely in low quality blueprints.

Accordingly, we calculate the fitness of a factor as the average fitness of all blueprints to which it contributes. In other words if factor f participates in blueprints $b_1, ..., b_n$ then $fitness(f) = \frac{1}{n} \sum_{i=1}^{n} fitness(b_i)$. For example, if factor f was used in three blueprints with fitness values of 1, 2, and 3, then $fitness(f) = \frac{1}{3}(1 + 2 + 3) = 2$.

If a factor does not participate in any blueprints then we have no specific information with which to estimate its fitness. Therefore such a factor is assigned the overall average fitness of all blueprints, as this is the best fitness estimate we can make, using the information available to us.

Factor Filtering

It has been suggested by several sources (e.g. [62, 61]) that a better method would be to restrict or filter the blueprints used in factor fitness evaluation. Specifically, the fitness of factor f should be calculated as the average fitness of *the best n blueprints in which f participates*. Thus when evaluating factor fitness we ignore those blueprints which use the factor, but have poor performance.

The proponents for this approach argue that factors which produce high fitness blueprints are good, regardless of how many poor blueprints they participate in. The reasoning is that it is better to produce a few outstanding blueprints, rather than a large number of mediocre blueprints. It is argued that by averaging over all blueprints we are ignoring specialized

factors which are highly useful, but only in a select number of situations.

Unfortunately there are several flaws with this approach. First and foremost, by restricting the number of blueprints used for factor fitness estimation, we are decreasing the accuracy of these estimates. Factor fitness is calculated as an estimate, not an exact value. The accuracy of this estimate will be directly related to the number of data points, in the form of blueprints, used to form the estimate. Using more blueprints will result in fitness estimates with greater accuracy. By discarding all but the best few blueprints we greatly decrease the accuracy of our factor fitness estimates.

If the accuracy of our factor fitness estimates decreases, then algorithm performance will suffer. Factor fitness values are used to decide which factors are selected for reproduction. If bad fitness values exist, then the wrong factors will be selected for evolution. Selecting the wrong factors will greatly hamper the evolutionary process and retard the formation of good individuals. Hence it is vital that we maximize the accuracy of our factor fitness estimates.

With this in mind we have decided to use all blueprints when estimating factor fitness. This decision is supported by preliminary results, which show that restricting the blueprints used in factor fitness evaluation has a detrimental effect on algorithm performance. Hence the factor fitness is defined as the average fitness of all blueprints in which the factor participates.

5.2.3 Evolution

Once the fitness of each blueprint and factor has been determined, each population independently undergoes evolution. In the CC framework, factors are evaluated *together* but evolved *separately*. In the blueprint population, high fitness blueprints are stochastically selected and used as a basis for the next generation of blueprints. In each factor, subpopulation

high fitness factors are selected and used as a basis for the next generation of factors in that specific subpopulation. It is important to note that no genetic material is exchanged between subpopulations during evolution.

- **Factor Subpopulations.** Each subpopulation individually undergoes evolution in an identical fashion to a population of conventional LGP programs. Elitism preserves the best factors, mutation randomly modifies factors, and crossover exchanges segments of code between any two factors within the subpopulation. Note that there is **no** exchange of genetic material between individuals in different subpopulations. This follows the ideas of a SANE variant known as Enforced Sub Populations (ESP) and is shown to enhance performance [32, 33].

- **Blueprint Population.** Evolution in the blueprints occurs solely at the pointer level. Elitism preserves the best blueprints, mutation changes a pointer into a new random pointer, and crossover exchanges randomly selected pointers between two blueprints. Blueprint evolution affects which factors cooperate as solutions, not the factors themselves.

5.3 Hybrid PLGP

In this section we combine PLGP and CC PLGP to produce an algorithm which posses the strengths of both approaches.

5.3.1 Motivation

We expect that PLGP and CC PLGP will have unique strengths and weaknesses. CC PLGP was motivated by a desire to improve the convergence time of PLGP during the first stage of evolution. CC PLGP evolves all program factors in parallel, allowing rapid convergence to a population

of high fitness individuals. Therefore we expect CC PLGP to outperform PLGP during the first stage of evolution. Unfortunately we expect PLGP to outperform CC PLGP during the latter stages of evolution. PLGP excels at fine tuning existing high fitness solutions as evolution is localized to a single factor in any generation. In contrast, CC PLGP always makes large changes to program output, as all factors undergo evolution in each generation. Hence we expect CC PLGP to perform well during the first stage of evolution, and PLGP to perform well during the latter stages of evolution.

We hypothesize that these strengths are complementary. CC PLGP allows us to rapidly generate high quality solutions, while PLGP allows us to rapidly improve existing high quality solutions. Therefore it is natural to ask if we can combine these two techniques into a hybrid form which posseses the strengths of both. We expect such an algorithm to rapidly evolve high quality solutions, yet allow continued improvement throughout later generations.

5.3.2 Implementation

Hybrid PLGP is a form of PLGP which combines the strengths of PLGP and CC PLGP. The key idea is to use CC PLGP for the first x generations, then switch to PLGP for the remaining generations. Using CC PLGP during the first stage of evolution will allow us to rapidly evolve high quality individuals, while using PLGP for the later generations will allow us to further improve the high quality solutions generated by CC PLGP. In order to implement such a hybrid scheme we need to be able to convert CC PLGP blueprints into PLGP programs. This conversion must be fitness preserving, and the converted programs must have identical fitness to the original blueprints.

We note that PLGP programs and CC PLGP blueprints are closely related. Both programs and blueprints consist of n factors, each of which is a

sequence of instructions, and both programs and blueprints have identical program execution. There are two major differences:

- PLGP programs contain a private copy of each factor, while CC PLGP blueprints contain a pointer to a shared copy for each factor.
- PLGP programs undergo evolution on the instructions themselves, while CC PLGP program evolution affects only the pointers, leaving the instructions themselves unchanged.

The difference in evolution is not significant, as it is simply a manipulation of an existing program. If we can convert between program structures then changing the evolution is easily achieved. Therefore to convert from CC PLGP to PLGP we need to replace each factor pointer with an identical concrete factor. We achieve this by *replacing each factor pointer with a deep clone of the appropriate concrete factor*. A deep clone is an exact copy of the factor being cloned.

It is clear that the PLGP programs produced as a result of this conversion will have identical fitness to the original CC PLGP blueprints. They contain identical instructions, which when executed on the problem instances will give identical results. The difference is that code is *no longer shared* between individuals. This means that evolutionary changes will be local. The mutation of one individual will have no effect on the rest of the population.

To complete the conversion we switch to PLGP using the individuals generated through our conversion as the starting population. It is important to note that this conversion is extremely fast, adding a negligible amount to the running time of the algorithm.

5.4 Experimental Setup

We compare the performance of CC PLGP to that of PLGP. This section outlines the experiments and parameters used to facilitate this compari-

son.

5.4.1 Data Sets

The three data sets described in chapter 3 will form the basis for our experiments.

5.4.2 Changeover Point

The changeover point n is an important parameter required for the deployment of the Hybrid PLGP algorithm. Hybrid PLGP uses CC PLGP for the first n generations, and PLGP for all subsequent generations. An optimal value for n is the point at which PLGP first becomes more effective than CC PLGP. If n is too small we will prematurely switch from CC PLGP to PLGP, preventing rapid convergence of the initial population. If n is too large we will continue to use CC PLGP even after fitness begins to stagnate due to an inability to fine tune existing solutions.

The optimum changeover point occurs when the performance difference between CC PLGP and PLGP is maximum. To this end we examine the results presented in chapter 5. We note that for different program lengths and different problems the maximum performance difference occurs at different times. We are most interested in optimizing the performance of Hybrid PLGP for large programs, as it is under these conditions CC PLGP is most effective. Furthermore given that CC PLGP does not improve performance on the yeast data set we can likewise ignore these results. Therefore we choose the value n = 25, as this is the point at which the maximum performance difference occurs when using large programs on the *Hand Written Digits* and *Artificial Characters* data sets.

5.4.3 Parameter Configurations

The parameters in table 5.1 are *constant parameters*. These are the parameters which will remain constant throughout all experiments. These parameters are either experimentally determined optima, or common values whose reasonableness is well established in literature [46].

Table 5.1: Experimental Parameters

Parameter	Value
Population	1000
Generations	400
Mutation	30%
Elitism	10%
Crossover	60%
Tournament Size	5
Runs	30
Registers	10

We allow terminal constants in the range [-1,1], and a function set containing Addition, Subtraction, Multiplication, Protected Division, and If. The data set is divided equally into a training set, validation set, and test set, and results are averaged over 30 runs. All reported results are for performance on the test set. The fitness function used is the number of missclassified training examples. Finally all initial programs in the population consist of randomly chosen instructions.

5.4.4 Experiments

The parameters in table 5.2 are the *experiment specific parameters*. Each column of the table corresponds to the parameter settings for a specific experiment. Each experiment has two components, a PLGP stage and a CC PLGP stage. In the PLGP stage we determine the classification accuracy

of a PLGP system using programs of the specified length. In the CC PLGP stage we repeat our experiment but we use CC PLGP programs of equivalent length. Note that we repeat each experiment 30 times and average the results.

Table 5.2: Experiments

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Total Instructions	20	50	100	200	400	600
# PLGP factors	4	5	10	10	20	30
PLGP factor Size	5	10	10	20	20	20

5.5 Results

In this section we compare the effectiveness of PLGP, CC PLGP, and Hybrid PLGP. We present the results of the experiments detailed in section 5.4 together with discussion.

5.5.1 CC PLGP

Figure 5.2, figure 5.3, and figure 5.4 compare the effectiveness of PLGP with that of CC PLGP as a classification techniques on the three data sets described in chapter 3. Figure 5.2 compares performance on the *Hand Written Digits* data set, Figure 5.3 compares performance on the *Artificial Characters* data set and Figure 5.4 compares performance on the *Yeast* data set. Each line corresponds to an experiment with programs of a certain fixed length. Program lengths vary from very short (10 instructions) to very long (600 instructions).

We examine the statistical significance of these results by performing a students t-test on the fitness at generation 25. We choose generation 25 because we wish to examine relative performance during the initial stages

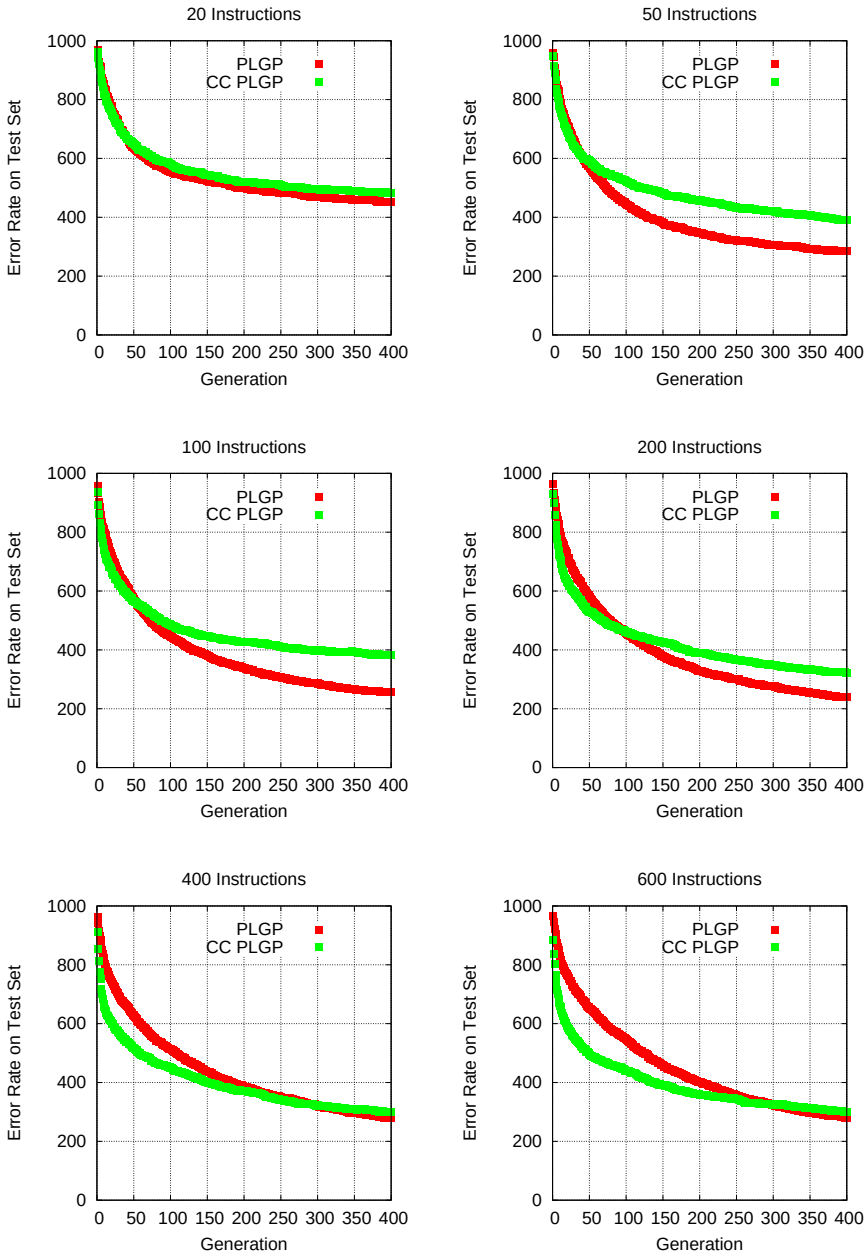


Figure 5.2: PLGP vs CC PLGP on the *Hand Written Digits* data set

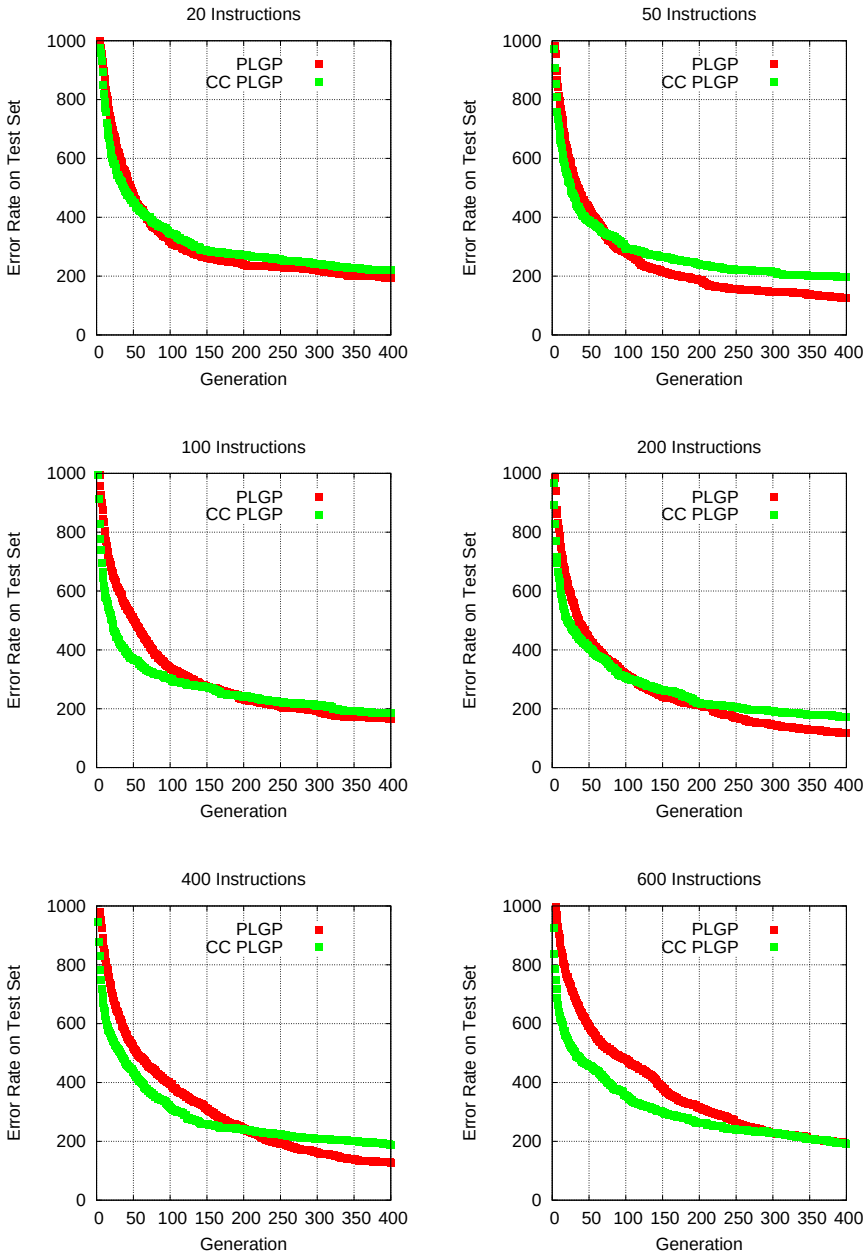


Figure 5.3: PLGP vs CC PLGP on the *Artificial Characters* data set

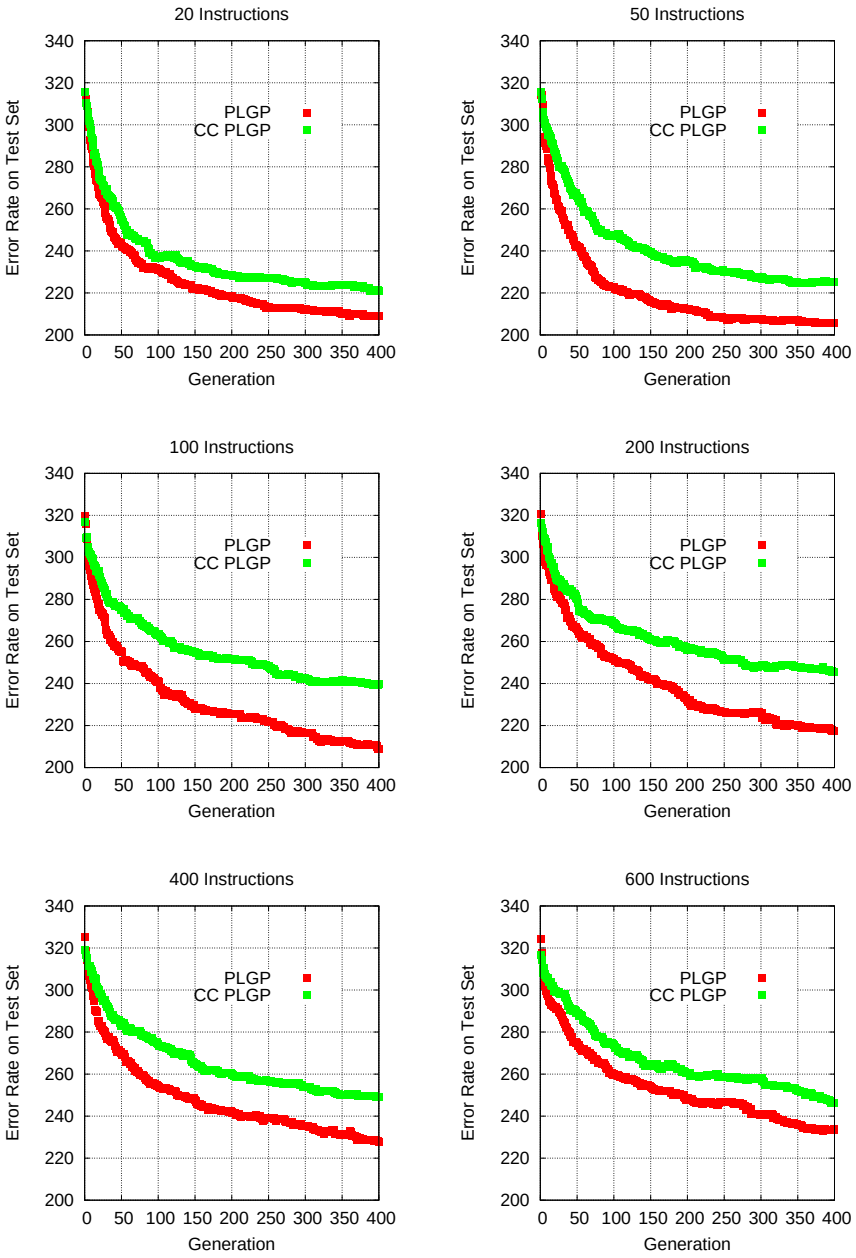


Figure 5.4: PLGP vs CC PLGP on the *Yeast* data set

of evolution. Figure 5.5a tests the significance of the results in figure 5.2. Figure 5.5b tests the significance of the results in figure 5.3. Figure 5.5c tests the significance of the results in figure 5.4. In these results n is the number of trials, SD is the standard deviation, and p is the p-value resulting from the t-test. Note that by convention a p value smaller than 0.05 is considered significant.

Discussion

The results on the *Hand Written Digits* and *Artificial Characters* data sets are consistent with our initial hypothesis, and perfectly illustrate the strengths and weaknesses of both the CC PLGP and PLGP approaches. These results are discussed in more detail below.

The results on the *Yeast* data set show that PLGP significantly outperforms CC PLGP for all program lengths. We hypothesize that this is because the *Yeast* data set is not well suited to a cooperative coevolutionary approach. Cooperative coevolution excels at solving problems which can be decomposed into multiple cooperating subcomponents. If the problem cannot be decomposed into cooperating subcomponents, then there is no advantage to using cooperative coevolution. In fact cooperative coevolution is often slower, since the multiple subcomponents in each solution will make it difficult to evolve a non-decomposed solution.

We now discuss in detail the results on the *Hand Written Digits* and *Artificial Characters* data sets. We note that these results only generalize to problems which can be decomposed into cooperating subproblems, as shown by the yeast data set.

- **Overall Performance**
  CC PLGP does well during the first stage of evolution, as it rapidly establishes a population of high fitness blueprints. Unfortunately, the performance of CC PLGP rapidly falls off in later generations as the algorithm struggles to fine tune existing high fitness solutions.

Figure 5.5: Significance of Results: PLGP vs. CC PLGP

		PLGP		CC PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	748.23	43.95	732.70	55.63	30	0.2144	NO
Exp2	50	692.13	35.91	670.20	52.93	30	0.0595	NO
Exp3	100	693.83	42.94	645.99	54.60	30	0.0003	YES (+)
Exp4	200	695.33	48.74	615.50	60.28	30	0.0001	YES (+)
Exp5	400	725.83	37.84	585.83	56.27	30	0.0001	YES (+)
Exp6	600	746.26	37.09	574.80	77.59	30	0.0001	YES (+)

(a) *Hand Written Digits*

		PLGP		CC PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	678.12	90.73	582.03	103.87	30	0.0003	YES (+)
Exp2	50	590.36	99.33	515.53	90.52	30	0.0033	YES (+)
Exp3	100	644.32	97.53	465.06	81.03	30	0.0001	YES (+)
Exp4	200	596.68	79.43	486.56	88.81	30	0.0001	YES (+)
Exp5	400	676.28	80.31	537.03	123.80	30	0.0001	YES (+)
Exp6	600	735.48	80.80	527.33	136.96	30	0.0001	YES (+)

(b) *Artificial Characters*

		PLGP		CC PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	265.36	19.10	273.43	20.78	30	0.1251	NO
Exp2	50	261.3	18.59	282.73	21.93	30	0.0002	YES (-)
Exp3	100	273.56	23.93	287.20	20.14	30	0.0147	YES (-)
Exp4	200	281.96	22.22	289.83	20.97	30	0.1460	NO
Exp5	400	282.06	18.33	296.03	19.51	30	0.0057	YES (-)
Exp6	600	291.06	18.32	298.96	16.55	30	0.1292	NO

(c) *Yeast*

In contrast PLGP has poorer fitness values during the first stage of evolution, as it struggles to produce a population of high fitness individuals from an initial random population. Fortunately, the performance of PLGP rapidly improves in later generations, as the algorithm excels at fine tuning.

These results are consistent with our initial hypothesis.

- **Fast Convergence**
  CC PLGP excels at identifying and promoting the components required for high quality solutions. By concurrently optimizing all program factors CC can significantly reduce the time required to produce high quality solutions from an initial population. In contrast PLGP can only optimize a single program factor in each generation. This means PLGP requires many more generations before high quality solutions can be produced.

- **Slow Fine Tuning**
  CC PLGP performs poorly when attempting to fine tune high quality solutions. Improving existing high quality solutions requires small steps within the search space, in other words small changes to the program code. In a CC system it is likely that many of the components within a blueprint will undergo mutation or crossover concurrently, causing massive code changes. In contrast PLGP excels at fine tuning existing solutions. The PLGP architecture was specifically designed to minimize the disruptive effects of crossover and mutation, allowing small changes to be achieved with ease.

- **Small Programs**
  Small programs can be more effectively evolved using standard PLGP.

  Because small programs have few factors it is efficient to evolve a single factor during each generation. Hence CC PLGP lacks any initial performance advantage, since both methods can efficiently evolve

an initial random population.

Furthermore CC PLGP still struggles to fine tune existing high fitness solutions. Therefore PLGP and CC PLGP have comparable performance during the first stage of evolution, however CC PLGP has distinct performance advantages during later generations.

In addition, the number of factors per program is directly related to the average number of blueprints each factor participates in. When programs have few factors, each factor participates in only a small number of blueprints. This reduces the accuracy of factor fitness estimates, retarding evolution.

Hence PLGP is the superior method for small programs.

- **Large Programs**
  Large programs can be more effectively evolved using CC PLGP.

  Because large programs have many factors it is very inefficient to evolve a single factor during each generation. Hence CC PLGP has an initial performance advantage as it can concurrently optimize all program factors. This advantage scales with the number of factors, so CC PLGP has a greater initial performance advantage on larger programs.

  At the same time the performance of PLGP will eventually exceed that of CC PLGP even on large programs. Because CC PLGP struggles to fine tune existing high fitness solutions the fitness will plateau at lower level. Hence CC PLGP is the superior technique, only if computation resources are at a premium.

5.5.2 Hybrid PLGP

Figure 5.6, figure 5.7, and figure 5.8 compare the effectiveness of PLGP with that of Hybrid PLGP as classification techniques on the three data sets described in chapter 3. Figure 5.6 compares performance on the *Hand*

Written Digits data set, Figure 5.7 compares performance on the *Artificial Characters* data set and Figure 5.8 compares performance on the *Yeast* data set. Each line corresponds to an experiment with programs of a certain fixed length. Program lengths vary from very short (10 instructions) to very long (600 instructions).

We examine the statistical significance of these results by performing a students t-test on the fitness at generation 400. Figure 5.9a tests the significance of the results in figure 5.6. Figure 5.9b tests the significance of the results in figure 5.7. Figure 5.9c tests the significance of the results in figure 5.8. In these results n is the number of trials, SD is the standard deviation, and p is the p-value resulting from the t-test. Note that by convention a p value smaller than 0.05 is considered significant.

Discussion

These results are precisely in line with our expectations.

The results on the *Hand Written Digits* and *Artificial Characters* data sets show that Hybrid PLGP performs at least as well as PLGP for all program sizes. Furthermore Hybrid PLGP converges far more rapidly than PLGP during initial generations. This is particularly true for large PLGP programs with many factors.

The results on the *Yeast* data set show that PLGP significantly outperforms Hybrid PLGP for the majority of program lengths. This it to be expected as we have already shown that CC PLGP is not well suited to this data set. PLGP significantly outperforms CC PLGP on the yeast data set, so we did not expect a hybrid PLGP/CC PLGP algorithm to outperform PLGP.

We now discuss in detail the results on the *Hand Written Digits* and *Artificial Characters* data sets. We note that these results only generalize to problems which can be decomposed into cooperating subproblems, as shown by the yeast data set.

PLGP works well when the number of factors is small and tuning one

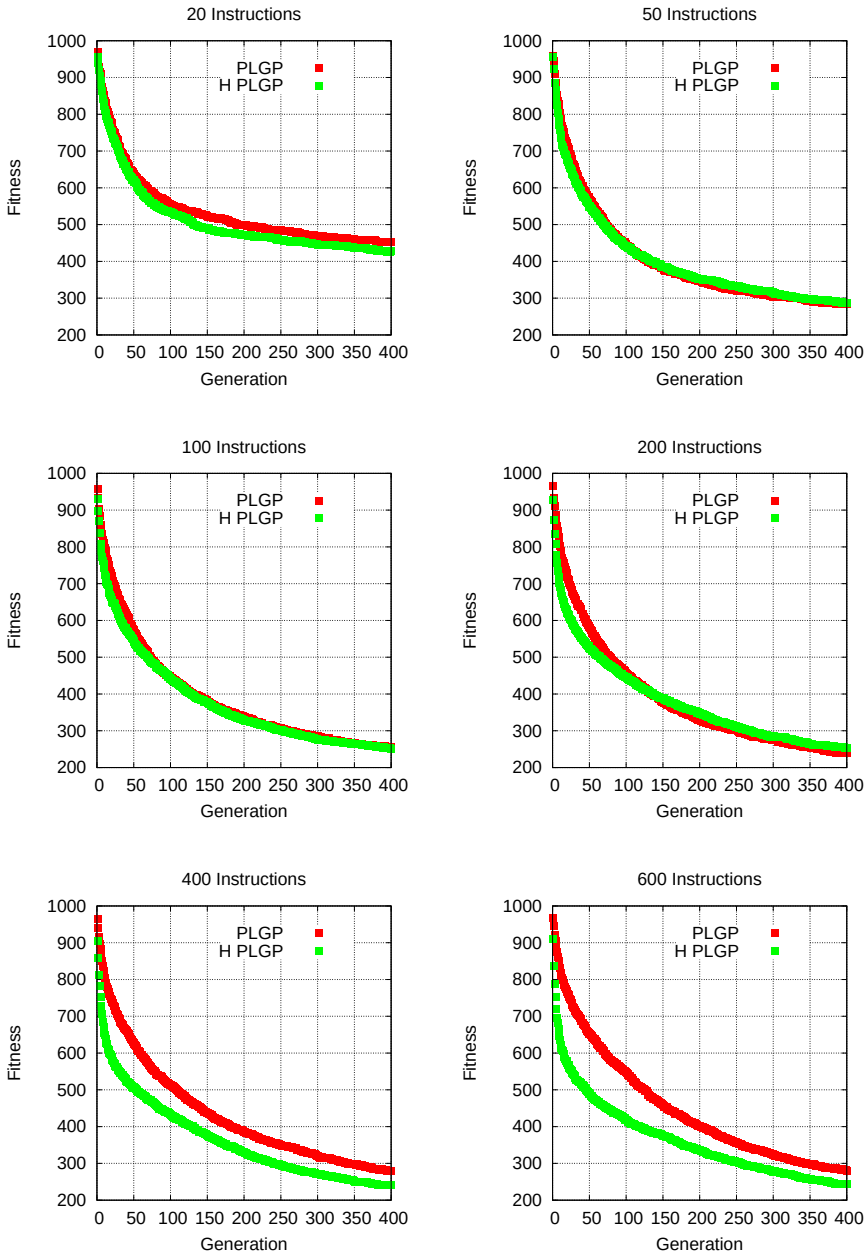


Figure 5.6: PLGP vs Hybrid PLGP on the *Hand Written Digits* data set

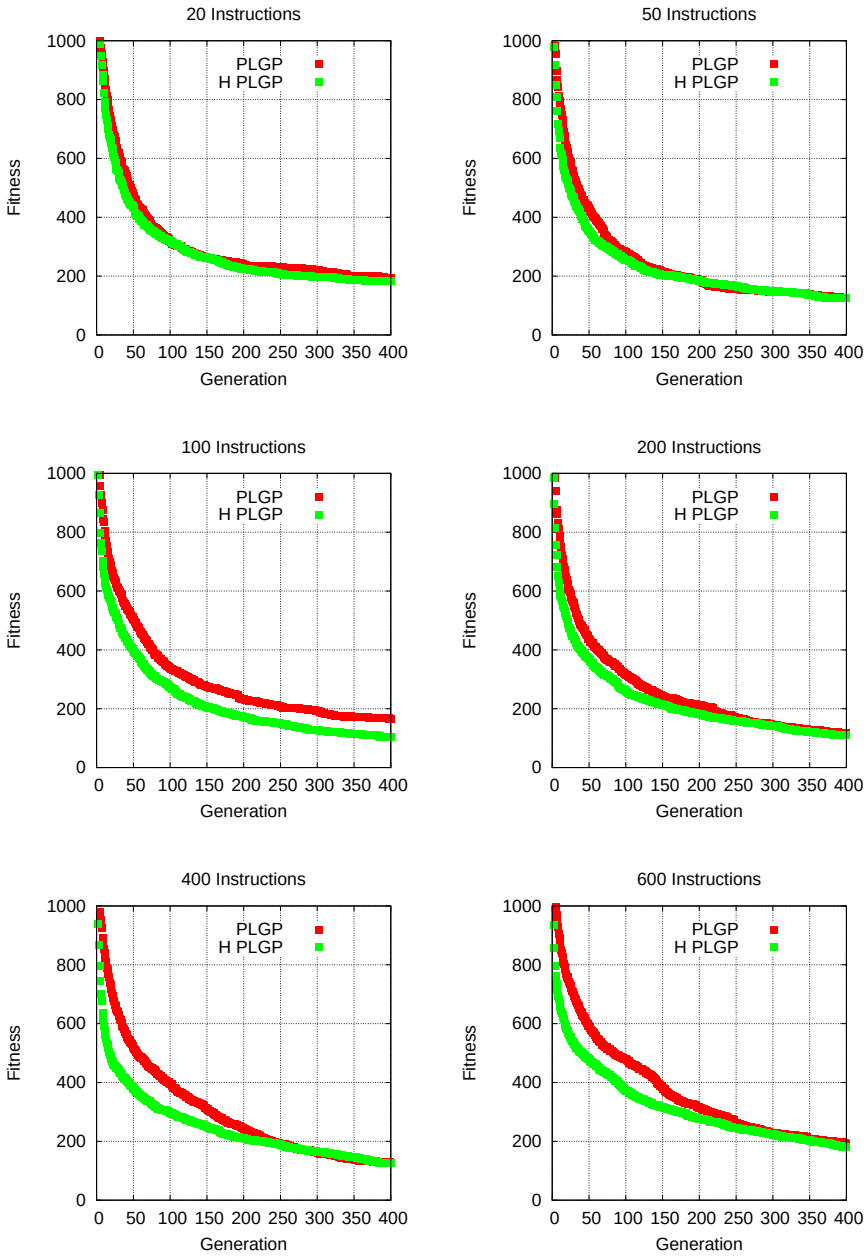


Figure 5.7: PLGP vs Hybrid PLGP on the *Artificial Characters* data set

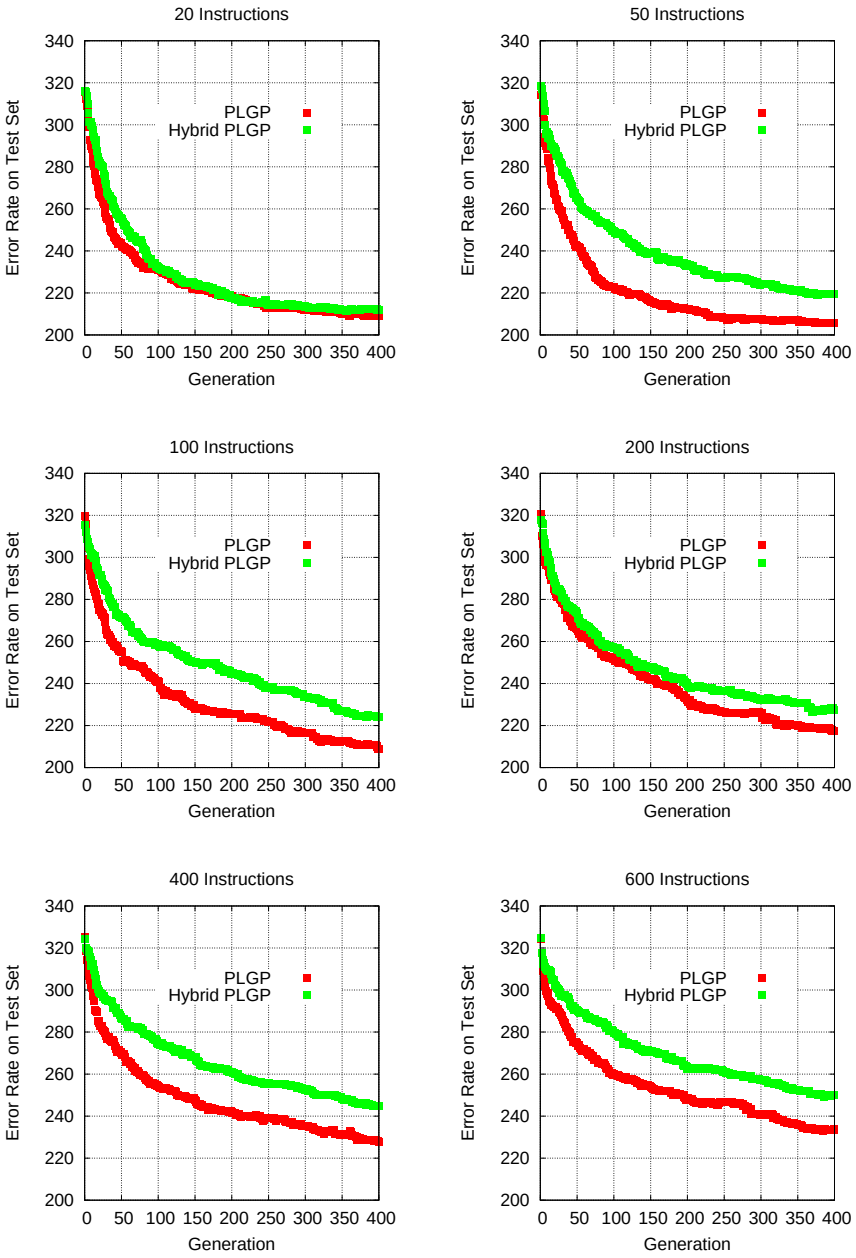


Figure 5.8: PLGP vs Hybrid PLGP on the *Yeast* data set

Figure 5.9: Significance of Results: PLGP vs. Hybrid PLGP

		PLGP		Hybrid PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	451.63	47.78	427.00	45.46	30	0.0480	YES (+)
Exp2	50	284.73	53.69	287.46	51.87	30	0.8664	NO
Exp3	100	256.23	42.43	251.70	53.88	30	0.7323	NO
Exp4	200	238.93	53.87	254.10	43.41	30	0.2346	NO
Exp5	400	278.96	52.79	240.50	36.52	30	0.0013	YES (+)
Exp6	600	280.63	51.46	277.80	52.73	30	0.8223	NO

(a) *Hand Written Digits*

		PLGP		Hybrid PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	192.44	53.21	181.90	62.04	30	0.4631	NO
Exp2	50	127.28	79.98	124.30	52.11	30	0.8627	NO
Exp3	100	166.52	78.88	103.50	59.53	30	0.0005	YES (+)
Exp4	200	118.68	46.75	109.53	85.41	30	0.6087	NO
Exp5	400	127.80	64.89	126.16	66.27	30	0.9235	NO
Exp6	600	192.60	104.05	180.23	115.09	30	0.6640	NO

(b) *Artificial Characters*

		PLGP		Hybrid PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	209.13	10.14	212.06	15.29	30	0.3658	NO
Exp2	50	205.50	11.31	218.50	18.81	30	0.0009	YES (-)
Exp3	100	208.90	21.79	224.03	17.14	30	0.0019	YES (-)
Exp4	200	217.63	19.13	227.53	21.16	30	0.0580	NO
Exp5	400	227.56	24.18	245.00	24.93	30	0.0069	YES (-)
Exp6	600	233.70	17.40	250.00	25.35	30	0.0032	YES (-)

(c) *Yeast*

factor per generation is reasonably effective. CC PLGP offers no advantage for small PLGP programs, so hybrid PLGP gives comparable performance to conventional PLGP.

PLGP performs relatively poorly when the number of factors is large and tuning one factor per generation is impractical. CC PLGP allows us to tune multiple factors in parallel, allowing for rapid fitness improvement during initial generations. This allows hybrid PLGP to significantly outperform PLGP during the first stage of evolution, after which it switches to conventional PLGP to fine tune these high quality solutions. During this stage of the algorithm both PLGP and hybrid PLGP operate in an identical fashion, however hybrid PLGP has a headstart, allowing it to converge more rapidly.

5.6 Chapter Summary

Cooperative Coevolution (CC) is a highly effective and widely used framework for evolutionary algorithms. PLGP is naturally suited to CC due to its independent factors and implicit subpopulations. We have exploited these attributes to develop a CC algorithm for PLGP based on the SANE CC architecture. We have shown that CC PLGP has significantly superior performance during the first stage of evolution, particularly on programs with a large number of factors.

We combined PLGP and CC PLGP into a hybrid algorithm possessing the strengths of both approaches. We begin evolution using CC PLGP in order to rapidly obtain high quality components, and complete execution using PLGP in order to fine tune these components. We have shown that hybrid PLGP has comparable effectiveness to PLGP but converges more rapidly.

5.6.1 Next Step

CC PLGP is most effective if we can find good factor combinations. The current approach uses blueprints to keep track of previously encountered successful factor combinations. A better approach would be to independently identify high quality factor combinations for each generation. This issue will be addressed in chapter 6.

Chapter 6

Blueprint Search for PLGP

6.1 Introduction

The effectiveness of CC PLGP evolution depends heavily on which blueprints are chosen for execution. Blueprints are used to estimate factor fitness, so if the blueprints are chosen badly then it is not possible to obtain a good fitness estimate. Poor fitness estimates result in the wrong factors being selected for evolution, greatly hampering the evolutionary process. Therefore it is vital to select good blueprints.

To estimate factor fitness values effectively we require high fitness blueprints. Unfortunately, finding high fitness blueprints is difficult. If we have p subpopulations of n blueprints then there are n^p possible blueprints, the vast majority of which will be of low fitness. Ideally we would evaluate all possible blueprints in order to find those with high fitness, however this is computationally infeasible. In practice we are forced to limit ourselves to evaluating some small subset of blueprints.

In conventional CC PLGP, described in chapter 5, we use the results from the previous generations to predict which blueprints to evaluate in the current generation. This predictive approach is used because each blueprint is extremely costly to evaluate, requiring p factors to be executed against the entire training set. It is assumed that blueprints which had high

fitness in generation n can be used as a basis for the blueprints in generation $n + 1$, despite the fact that many of the factors have changed. Hence we maintain a population of blueprints and evolve them in the same way as we evolve the factors.

Unfortunately, using a population of blueprints to evaluate the factor populations has a critical weakness. Namely, the blueprints are always several generations out of date. For each generation we "guess"' blueprints based on historical results. This means the current factors will have no influence on the blueprints used to evaluate their performance. Using out of date factors to predict which blueprints to use is clearly a critically flawed approach to factor fitness evaluation.

We wish to develop an approach to finding good blueprints which bases its decisions on the *current* factors, not an out of date estimate. We believe that by improving the quality of the blueprints used during factor evaluation we can improve the accuracy of the fitness estimates, and the effectiveness of the evolutionary process. This in turn is expected to lead to greatly improved algorithm performance.

6.1.1 Objectives

In this chapter we aim to improve the CC PLGP algorithm by developing a new approach to factor fitness evaluation. Our goal is to improve the quality of the blueprints used to evaluate the factors, and in doing so improve the accuracy of factor fitness evaluation. Specifically, this chapter has the following research objectives:

- To study the relationship between blueprints and factors.
- To determine how to use the current factors to produce high fitness blueprints.
- To combine our method for finding high fitness blueprints with the CC PLGP algorithm to produce a new PLGP algorithm.

- To compare the performance of our new algorithm to that of CC PLGP and PLGP.

6.2 Blueprint Search

We propose to improve blueprint quality by searching the space of potential blueprints. By searching for high fitness blueprints based on current factors we can be sure that any blueprints found will not be out of date.

Specifically, we propose to use an algorithm which iteratively improves a population of solutions by using the solutions from the previous iteration to select those in the current iteration. Assuming the search space has some sort of structure, then each evaluated solution gives us information about where to focus our search. This sort of algorithm can be highly efficient in terms of solution evaluations [44].

This leaves us with several unsolved problems:

- How do we formalize the problem of finding high fitness blueprints through the notion of a search space?
- How do we measure the "distance" or "similarity" between two blueprints?
- How do we construct a blueprint search space with useful "structure"?
- How do we search our blueprint search space?
- How do we use these blueprints to calculate factor fitness?

The remainder of this chapter will focus on answering these questions.

6.2.1 Formalization

In this section we formalize the problem of finding high fitness blueprints.

Blueprints as Vectors

It is important to formalize the concept of a blueprint in order to gain a better understanding of the problem. To this end we represent blueprints as vectors. Let the following assumptions hold:

- There are n subpopulations.
- Each subpopulation consists of f factors.
- Each factor in each subpopulation is associated with a unique integer label in the range 1 to f.

Then we can formally define vectors as follows:

A blueprint consists of an integer vector of length n. Each entry in the vector is an integer in the range 1 to f. We interpret the vector as consisting of n factor labels, one for each subpopulation. The first entry in the vector corresponds to a factor in the first subpopulation, the second entry in the vector corresponds to a factor in the second subpopulation, etc.

Defining the Blueprint Space

Representing blueprints as vectors allows us to formalize the notion of a blueprint search space. We represent our population of blueprints as a population of n-dimensional vectors. Therefore the problem of finding high fitness blueprints can be viewed as a search through a discrete, bounded, n-dimensional search space.

The best way to picture this is to think about a 2-dimensional problem, one where each blueprint consists of two factors. In this case all possible blueprints can be mapped into a two dimensional grid on the plane. Selecting any point on the plane gives x and y coordinates for that point, which can be interpreted as factor labels in the appropriate subpopulations. For instance suppose the point (3,6) was selected, then this corresponds to a blueprint consisting of the third factor from subpopulation 1

and the sixth factor from subpopulation 2. This idea generalizes to an n-dimensional space, where selecting any point corresponds to selecting an n-dimensional coordinate vector which can be interpreted as a blueprint.

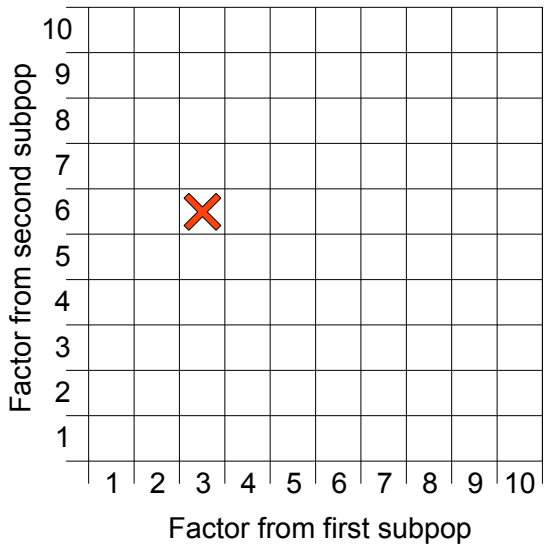


Figure 6.1: The Vectorspace of Blueprints

An example of a 2-dimensional blueprint space is shown in figure 6.1. This blueprint space has 2 subpopulations, each consisting of 10 factors. Each box in the grid corresponds to a single possible blueprint with two coordinates between 1 and 10. Therefore there are 100 different possible blueprints, corresponding to the 100 possible positions in the grid. The blueprint (3,6) consisting of the third factor from population 1, and the sixth factor from subpopulation 2 is marked on the blueprint space.

6.2.2 Spatial Locality

On the surface it appears that our n-dimensional Blueprint Space should be easy to search, however there is a significant problem which limits the effectiveness of any attempt to search this space. Namely the distance between points in our search space is unrelated to the similarity of blueprints.

In a conventional search space the similarity of potential solutions is inversely proportional to the distance between them within the search space. In other words two solutions located at opposite ends of the search space are likely to be very different, while two solutions located close to each other are likely to have similar structure. This property is known as *Spatial Locality* and is vital to the success of many search algorithms.

Search algorithms exploit spatial locality in order to find good solutions by focusing their search on regions of the search space which have been found to contain good solutions. Spatial locality ensures that solutions in these regions will have similar structure to solutions known to give good performance. It is assumed that because these solutions are similar in structure they will also have similar performance, an assumption which often holds. In this case it means that solutions which lie close to known good solutions are highly likely to give good performance. This is in contrast to a randomly chosen solution, of which the vast majority will give poor performance.

Unfortunately in our Blueprint Space the factor index has no relation to the factor structure. This means that two factors which have similar indices are may or may not have similar structure. Suppose we have three n-dimensional vectors $(1,1)$ $(1,2)$ and $(1,100)$. Intuitively speaking we would expect vectors $(1,1)$ and $(1,2)$ to be more similar than vectors $(1,1)$ and $(1,100)$. However in our formalization of the problem this is not the case. There is no reason to assume that factor 1 and factor 2 are any more similar than factor 1 and factor 100, as each is a random factor within the population. This problem is illustrated in figure 6.2.

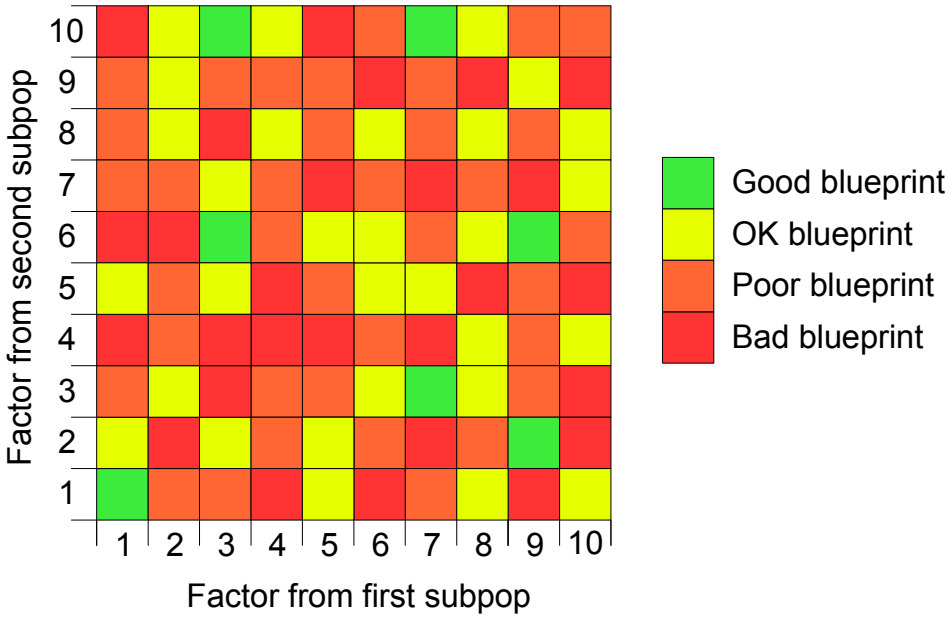


Figure 6.2: A bad Blueprint Space

Figure 6.2 clearly shows the problems with searching a space consisting of apparently randomly distributed solutions. Finding one good solution tells us nothing about the location of other good solutions. In contrast figure 6.3 illustrates the concept of a good search space. In this search space solutions with similar structure are located close to each other, which causes good solutions to also occur close to each other. Searching this space is much simpler, since we use the location of known good solutions to find other good solutions.

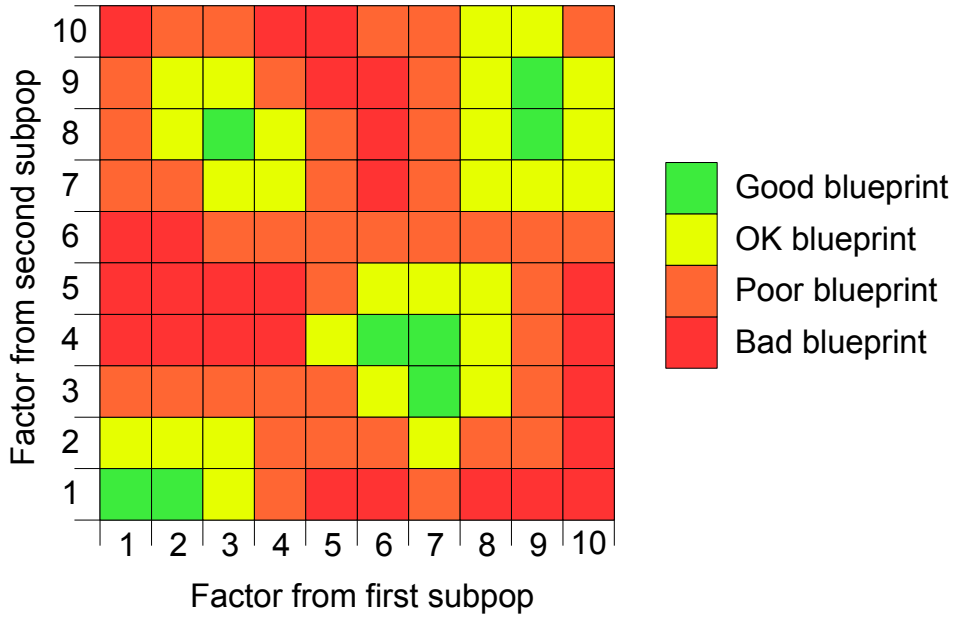


Figure 6.3: A good Blueprint Space

6.2.3 Calculating the Distance Between Factors

In this subsection we develop methods for measuring the distance between two blueprints.

In our Blueprint Space we require any two factors which have similar indices to also have similar structure. In other words the difference between factor indices should be proportional to the difference in factor structure. Since every factor must be assigned an integer index between 1 and n this is equivalent to finding an ordering of the factors which has the property of spatial locality.

In order to find an ordering of the factors which has the spatial locality property, we need to formalize the notion of "similarity" between factors through the use of a distance metric. A distance metric is a function which

outputs a single real valued number for each pair of inputs. This value represents the similarity or distance between the pair of inputs.

When comparing factors we are comparing two lists of instructions. There are several ways in which these lists can be similar, we list them in order of decreasing importance:

- The lists may be the same.
- The lists may contain many of the same instructions at the same locations.
- The lists may contain similar instructions at the same locations.
- The lists may contain similar instructions in the same order but at different locations.
- The lists may contain similar instructions in a different order at different locations.
- The lists are completely different.

We want our distance metric to give a low distance to lists which are strongly related, and a high distance to lists which have no relation. Ideally, we would devise a distance metric which examines two instruction lists in detail and takes into account the most subtle of relationships. Unfortunately devising such a metric is extremely difficult, and such metrics are generally computationally expensive. Therefore for now we restrict ourselves to a metric which only takes into account those relationships which are easy to check for. Specifically our metric will look for factors which have similar instructions at the same locations.

Converting Factors to Vectors

To calculate our distance metric we begin by converting each factor into a vector of integers. There are many existing distance metrics which can be

used to compare vectors, therefore by converting our factors into vectors we can transform the problem into one which has already been solved.

To transform our factors into vectors we first note that each factor is an ordered list of instructions, and each instruction is an ordered list of objects. Therefore we will convert each instruction into a vector and lay these vectors end to end, resulting in a single lengthy vector representing the entire factor.

Each instruction consists of 4 objects: A destination register, two arguments, and an operator. The arguments can be either constants, features, or registers. We now associate a single integer value with each of these 4 objects, allowing us to represent each instruction as an integer vector of length 4.

Each object is encoded as an integer as follows:

Register: we use the register index to encode a register. i.e. r_i is encoded as i.

Feature: we use the feature index to encode a feature. i.e. f_j is encoded as j.

Operator: if there are n operators, then we enumerate the operators from 1 to n and encode each operator as its enumerated value. i.e. if $+$ is the third operator, then all instances of $+$ are encoded as 3.

Constant: We use bins to encode the constants. If there are x bins then each constant is scaled to the range 1 to x, after which the constant is cast to an integer. In other words constant c is encoded to c' as follows. $c' = floor(c \times (x/c_{max}))$ where c_{max} is the maximum possible constant value.

Each instruction is encoded as a vector consisting of the 4 encoded instruction objects. These vectors are laid end-to-end to produce the final factor vector. If each factor has i instructions then the vector representa-

tions of each factor will have length $4i$. An example of vectorization is shown in figure 6.4.

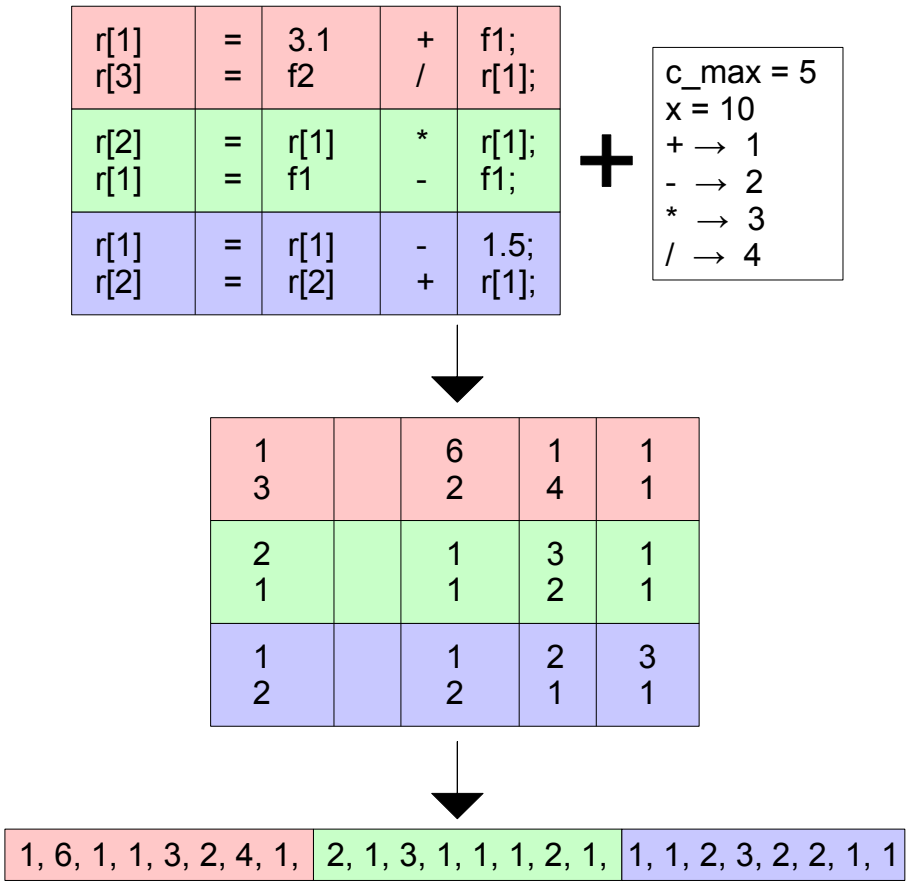


Figure 6.4: An example of how to vectorize a CC PLGP blueprint consisting of three factors

It is important to note that these vectors **will not** be unique. Two different programs can result in the same vector due to the overlap in the encoding scheme. For example feature f_4 and register r_4 both encode to 4. While it is possible to avoid this overlap by using a more advanced cod-

ing scheme, it is not necessary. The purpose of our distance metric is to provide information about the *relative* similarity of solutions, not provide a precise measurement of similarity. To achieve this, our distance metric only needs to give a rough approximation of distance between two factors. Fortunately any two factors with significant differences are virtually certain to encode to significantly different vectors. Factors often consist of tens of instructions, each of which is encoded by four integers. Therefore encoded vectors will often be hundreds of integers long. The probability of two different factors encoding to the same vector is so small as to be negligible. Hence our simple encoding scheme suffices.

Choosing a Distance Metric

A number of useful distance metrics are outlined in chapter 2, however we need to be careful when choosing a distance metric due to the way in which we have set up our vectors. In our vectors the difference between the values has no actual meaning. For instance suppose we have three vectors representing encoded instructions (1, 1, 1, 1), (1, 1, 2, 1), and (1, 1, 10, 1). Intuitively we would assume that the instructions encoded by (1, 1, 1, 1) and (1, 1, 2, 1) are more similar than the instructions encoded by (1, 1, 1, 1) and (1, 1, 10, 1). However on close inspection the only information conveyed by these vectors is that they have different operators. Specifically (1, 1, 1, 1) has the operator enumerated first, (1, 1, 2, 1) has the operator enumerated second, and (1, 1, 10, 1) has the operator enumerated tenth. In other words these three vectors are all equidistant from one other. Unfortunately many distance metrics would give undesired behavior insofar as they would show that (1, 1, 1, 1), (1, 1, 2, 1) are closer than (1, 1, 1, 1), (1, 1, 10, 1). We need a distance metric which calculates distance based only on the number of places in which our vectors differ.

Fortunately this notion of distance is formalized in the well known metric called the *Hamming Distance*. Hence we choose to use the Hamming distance to calculate the distance between factors. The Hamming

distance is defined as the number of places in which one n-dimensional vector differs from another. For example the vectors 1111 and 1142 have Hamming distance 2, since they differ in two places.

It is important to note that there are several disadvantages to using the Hamming distance as our distance metric. For example the two instructions r1 = 1 + 3 and r1 = 3 + 1 have the same effect, however these two instructions would have a hamming distance of 2, since they differ in two locations. However trying to capture these subtle relationships between instructions is extremely challenging and is left for future work.

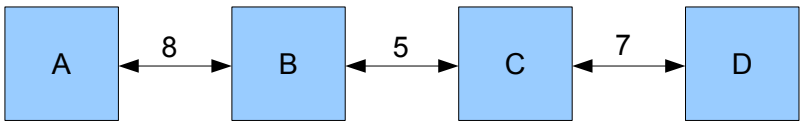
6.2.4 Constructing the Search Space

Now that we have formalized the distance between factors we can proceed with constructing a search space which has the spatial locality property.

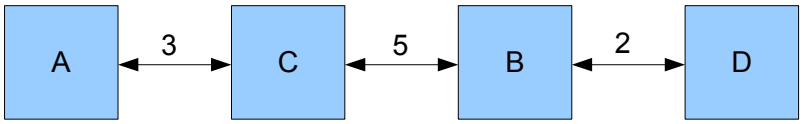
We require that any two factors which have similar indices should have similar structure. In other words we need to enumerate our factors in such a way that similar indices are assigned to factors with similar structure. Another way to view this problem is in terms of orderings. We want to order the factors so that all neighboring factors in the ordering share similar structure. Furthermore we want the difference between factor indices to be inversely related to the similarity in structure.

If factors 1 and 2 are similar, and factors 2 and 3 are similar, then according to the properties of the Hamming distance factors 2 and 3 must be similar. Therefore our new goal is to find an ordering of the factors which minimizes the distance between neighboring factors. This goal is illustrated in figure 6.5.

Unfortunately, the problem of finding a factor ordering which minimizes neighbor distance is NP-Complete. Specifically it is equivalent to the traveling salesman problem. Suppose we have a complete weighted graph with one node for each factor, and each edge is labeled with the factor distance between the adjacent nodes. Then a factor ordering is equiva-



(a) An example of a bad factor ordering

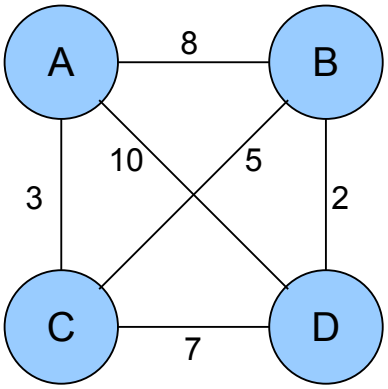


(b) The same factors rearranged into a good factor ordering

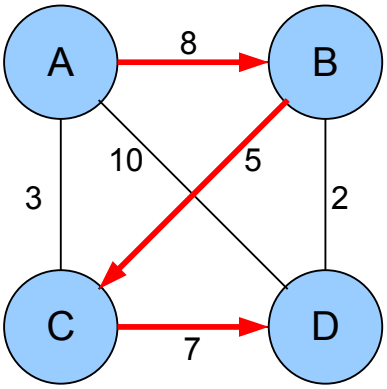
Figure 6.5: Four Factors arranged in two different orderings

lent to a path through the graph visiting each node precisely once, in other words a Hamiltonian path. We wish to find such a path with minimal total weight, in other words the traveling salesman problem. A example of the factor ordering problem expressed in terms of the travelleing salesman problem is shown in figure 6.6. Note that this is the same example used in figure 6.5.

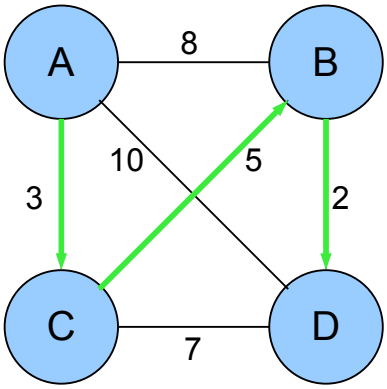
Fortunately it is not necessary for us to find the optimal factor ordering. As long as factors which are similar are close together in the ordering it is still possible to perform local search effectively. Therefore it is reasonable to use an ordering which is approximately correct. Equivalently it is reasonable to approximately solve the traveling salesman problem. The traveling salesman problem has been widely studied, and a number of approximation algorithms exist. One simple but effective approximation algorithm is *Nearest Neighbor* [35]. Note that the nearest neighbor algorithm used here differs somewhat from the nearest neighbour method for machine learning and classification.



(a) Four Factors expressed as a graph



(b) A bad path



(c) A good path

Figure 6.6: The Factor Ordering problem as a Graph

The nearest neighbor algorithm works as follows: Select an arbitrary node. Find the closest node which has not yet been visited and move to that node. Repeat until all nodes have been visited. Note that we initialize the nearest neighbour algorithm with a randomly selected factor. On

average the nearest neighbor algorithm yields a path 25% longer than the optimal path [41], although it is easy to construct special cases which give the worst possible route.

Nearest Neighbor is a simple, fast algorithm that provides a reasonable approximation to the optimal solution. These properties make nearest neighbor our algorithm of choice for determining factor orderings.

6.2.5 Searching the Blueprint Space

In this subsection we develop a method for effectively and efficiently searching the blueprint space constructed in the previous subsection.

The simplest possible search algorithm is random guessing. In a random guessing approach we would execute a large number of randomly selected blueprints and select those with the best fitness as our final population. While simple, random guessing is also an extremely poor choice of algorithm, as it has no local search aspect and completely fails to exploit information about known good solutions. However random search is useful for comparing baseline performance, and for determining the extent to which we gain from our carefully organized blueprint space.

A much more sensible alternative is to use an iterative style search algorithm. Iterative style algorithms repeatedly test a small number of potential solutions, using the solutions from the previous iteration to select those in the current iteration. These algorithms work on the assumption that each solution evaluation gives us information about where we should look for good solutions. Iterative style algorithms are typically much more efficient in terms of solution evaluations than random guessing.

The goal of this chapter is to find high fitness blueprints. So far we have reduced this problem to searching the space of n-dimensional vectors of natural numbers. The next question is *how do we search this space?*. A numerical solution is difficult due to the absence of gradient information, and an approach based on GP or GA is infeasible due to time re-

strictions. Based on these restrictions we propose to search by using the Particle Swarm Optimization (PSO) algorithm.

6.2.6 Estimating Factor Fitness

We use the PSO algorithm to locate high fitness blueprints. The PSO algorithm generates n generations of blueprints, with the average blueprint fitness increasing over the generations as the population converges. The final task is to use these blueprints to estimate the fitness of the factors. We calculate factor fitness as the average fitness of *all* blueprints in which the factor participates.

Blueprint Filtering

Many CC algorithms filter the blueprints used for factor fitness evaluation, using only the best n blueprints in which that factor participates. There are good reasons for doing this. In particular, we want to promote blueprints which are useful in forming high fitness solutions, regardless of whether they can also be used as constituents in poor solutions.

The problem with limiting factor fitness evaluation to the best n blueprints is choosing a good value for n. If n is too small we will discard blueprints which communicate important information about the fitness of n. If n is too large then limiting ourselves to n blueprints has no effect. In short, if n is chosen badly the performance of the algorithm will suffer.

Fortunately it is not necessary to explicitly limit factors fitness evaluation to the best n blueprints. PSO already rewards factors which participate in high fitness blueprints regardless of participation in low fitness blueprints.

There is an intentional and beneficial skew in how many times each factor is evaluated during PSO. The proposed new algorithm focuses blueprint search on areas of the search space which contain high fitness solutions. The blueprints in these areas will share many common factors, so a small

number of factors will participate in a large number of high fitness blueprints. In contrast areas of the search space which contain low fitness solutions will rarely be visited. In other words a large number of factors will participate in a small number of low fitness blueprints.

Factors may participate in blueprints in three possible ways: a factor may participate in only high fitness blueprints; a factor may participate in only low fitness blueprints; or a factor may participate in both high and low fitness blueprints. In the first two cases the result of averaging the blueprint fitness values is clear, therefore we turn to examine the third case.

A large number of factors participate in a small number of low fitness blueprints. On average any single factor which participates in low fitness blueprints will participate in only a *small number* of low fitness blueprints. In contrast a small number of factors participate in a large number of high fitness blueprints. On average any single factor which participates in high fitness blueprints will participate in a *large number* of high fitness blueprints. Therefore any factor which participates in both high and low fitness blueprints will participate in a far larger number of high fitness blueprints.

Another way to consider this mechanism is that it focuses computational resources where it is most important to do so. Factors which participate only in low fitness blueprints will rarely be executed, while factors which participate in high fitness blueprints will frequently be executed. This makes sense because we want to spend the minimum amount of time possible executing poor quality factors which will later be dropped from the gene pool.

6.2.7 Algorithm

We can now present our new algorithm in its entirity. We refer to this algorithm as *Blueprint Search PLGP* or *BS PLGP*.

- Initialize n subpopulations of randomly generated factors.
- For n GP generations:
  - Order the factors to form the search space.
  - Initialize a population of randomly generated blueprints.
  - For x PSO iterations:
    - Execute the blueprints.
    - Store the resulting fitness values.
    - Update blueprint positions within the search space.
  - Use blueprint fitness values to calculate factor fitness values.
  - Select and evolve factors.

6.3 Experimental Setup

We compare the performance of BS PLGP to that of PLGP and CC PLGP. This section outlines the experiments and parameters used to facilitate this comparison.

6.3.1 Data Sets

The three data sets described in chapter 3 will form the basis for our experiments.

6.3.2 Factor Ordering

The nearest neighbor factor ordering algorithm was used throughout these experiments. This is important as different algorithms result in different factor orderings of varying quality.

6.3.3 Parameter Configurations

When deploying the BS PLGP algorithm there are a large number of parameters which need to be instantiated. This is due to the complexity of BS PLGP, and the large number of existing techniques used as components in the BS PLGP algorithm. BS PLGP uses CC PLGP, PSO, and Nearest Neighbor, each of which has its own parameters which need to be instantiated to reasonable values.

GP parameters

The GP parameters used in these experiments are provided in table 6.1.

Table 6.1: GP Parameter Configurations

Parameter	Value
Population	1000
Max Gens	400
Mutation	30%
Elitism	10%
Crossover	60%
Tournament Size	5
Runs	30
Registers	10

PSO parameters

The PSO parameters used in these experiments are provided in table 6.2. These values are experimentally determined optima.

6.3.4 Nearest Neighbour Parameters

The nearest neighbour algorithm requires no parameters to be defined.

Table 6.2: PSO parameter Configurations

Parameter	Value
Iterations	10
Velocity weight	-0.3
Local Best Weight	0.5
Global Best Weight	0.1

6.3.5 Experiments

The parameters in table 6.3 are the *experiment specific parameters*. Each column of the table corresponds to the parameter settings for a specific experiment. Each experiment has three components, a PLGP stage, a CC PLGP stage, and a BS PLGP stage. In the PLGP stage we determine the classification accuracy of a PLGP system using programs of the specified length. In the CC PLGP stage we repeat our experiment but we use CC PLGP programs of equivalent length. In the BS PLGP stage we repeat our experiment a third time, but we use BS PLGP programs of equivalent length. Note that we repeat each experiment 30 times and average the results.

Table 6.3: Experiments

	Exp 1	Exp 2	Exp 3	Exp 4	Exp 5	Exp 6
Total Instructions	20	50	100	200	400	600
# PLGP factors	4	5	10	10	20	30
PLGP factor Size	5	10	10	20	20	20

6.4 Results

Figure 6.7, figure 6.8, and figure 6.9 compare the effectiveness of BS PLGP with that of PLGP and CC PLGP as classification techniques on the three data sets described in chapter 3. Figure 6.7 compares performance on the

Hand Written Digits data set, Figure 6.8 compares performance on the *Artificial Characters* data set and Figure 6.9 compares performance on the *Yeast* data set. Each line corresponds to an experiment with programs of a certain fixed length. Program lengths vary from very short (10 instructions) to very long (600 instructions).

We examine the statistical significance of these results by performing a students t-test on the fitness at generation 400. We compare BS PLGP to both PLGP and CC PLGP. Figure 6.10a and figure 6.11a test the significance of the results in figure 6.7. Figure 6.10b and figure 6.11b test the significance of the results in figure 6.8. Figure 6.10c and figure 6.11c test the significance of the results in figure 6.9. In these results n is the number of trials, SD is the standard deviation, and p is the p-value resulting from the t-test. Note that by convention a p value smaller than 0.05 is considered significant.

6.5 Discussion

These results are precisely in line with our expectations. BS PLGP significantly outperforms both PLGP and CC PLGP. We now discuss these results in detail.

- **BS PLGP vs PLGP:** BS PLGP demonstrates significantly superior performance to PLGP. BS PLGP significantly outperforms PLGP on both the *Hand Written Digits* and *Artificial Characters data* sets, and has generally comparable performance on the *Yeast* data set.

- **BS PLGP vs CC PLGP**
  BS PLGP demonstrates significantly superior performance to CC PLGP on all data sets (save experiment 1 of the yeast data set).

- **Fast Convergence**
  BS PLGP converges more rapidly than either PLGP or CC PLGP during the first stage of evolution.

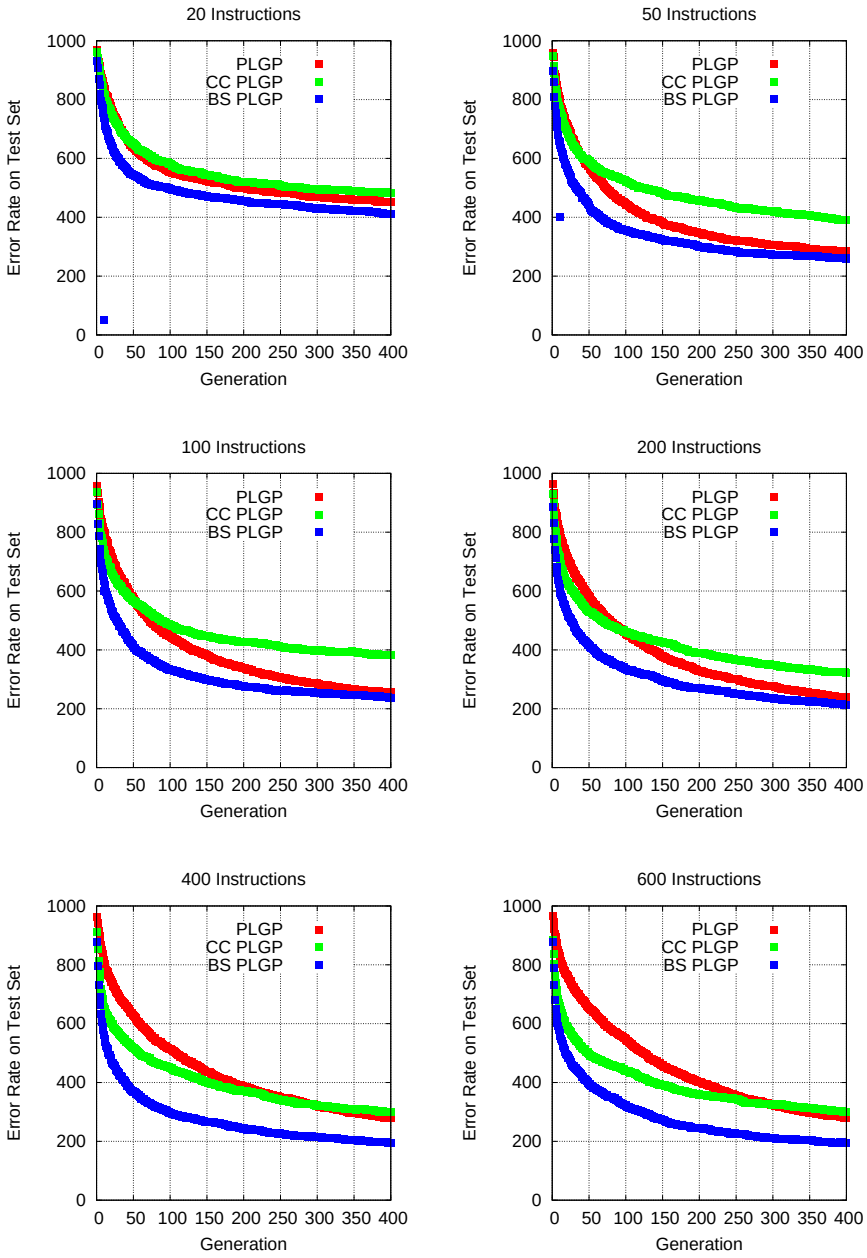


Figure 6.7: PLGP vs CC PLGP vs BS PLGP on the *Hand Written Digits* data set

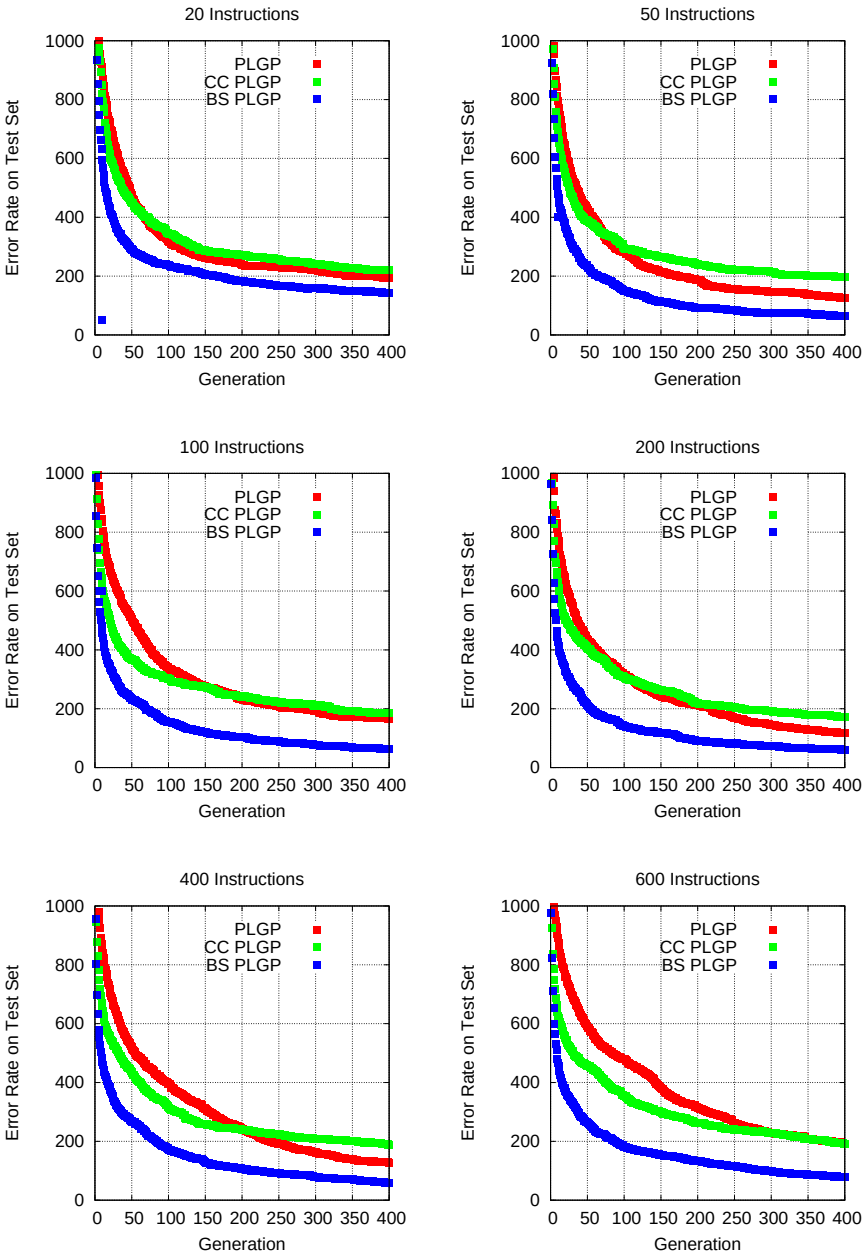


Figure 6.8: PLGP vs CC PLGP vs BS PLGP on the *Artificial Characters* data set

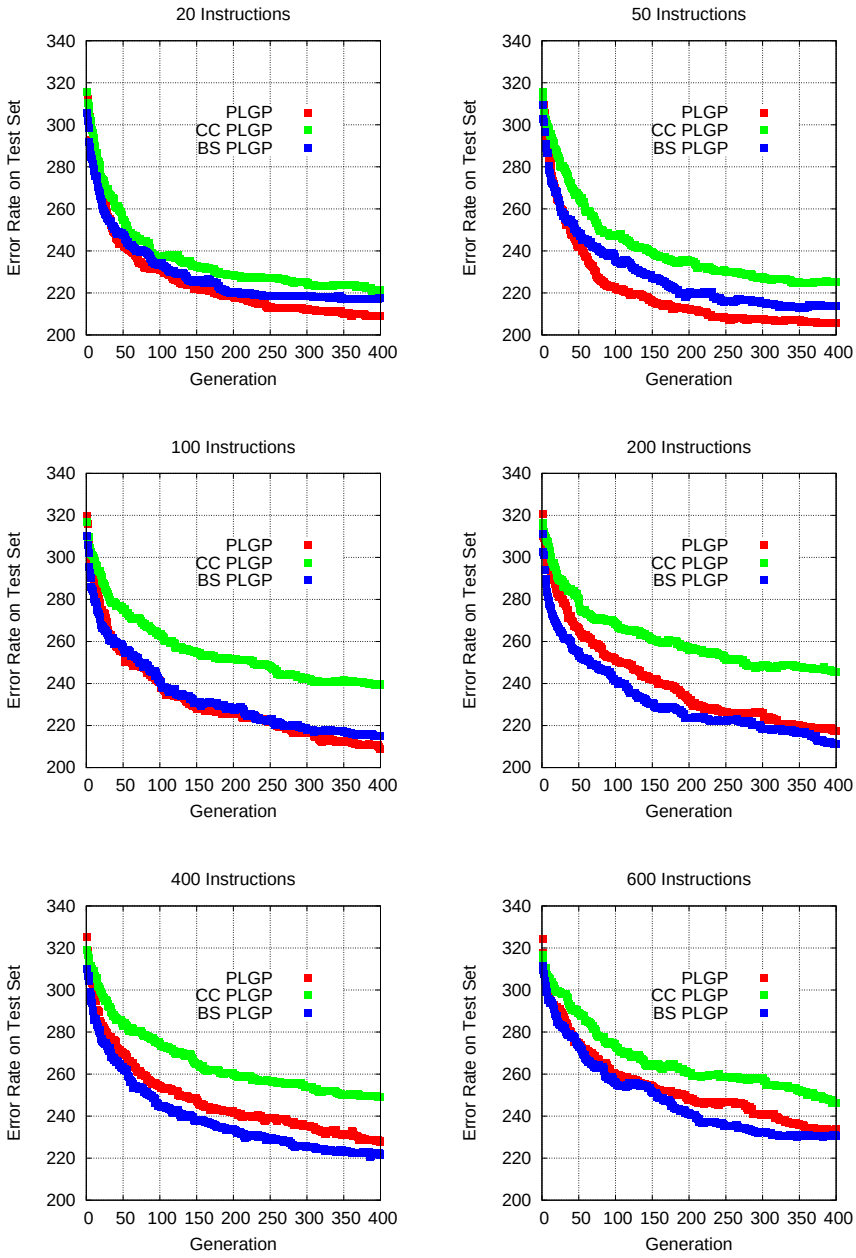


Figure 6.9: PLGP vs CC PLGP vs BS PLGP on the *Yeast* data set

Figure 6.10: Significance of Results: BS PLGP vs. PLGP

		PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	451.63	47.78	412.00	48.64	30	0.0024	YES (+)
Exp2	50	284.73	53.69	259.46	43.37	30	0.0495	YES (+)
Exp3	100	256.23	42.43	236.83	36.73	30	0.0499	YES (+)
Exp4	200	238.93	53.87	213.80	29.56	30	0.0271	YES (+)
Exp5	400	278.96	52.79	194.43	33.63	30	0.0001	YES (+)
Exp6	600	280.63	51.46	194.50	24.45	30	0.0001	YES (+)

(a) *Hand Written Digits*

		PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	192.44	53.21	140.96	32.83	30	0.0001	YES (+)
Exp2	50	127.28	79.98	65.00	34.21	30	0.0002	YES (+)
Exp3	100	166.52	78.88	62.63	34.23	30	0.0001	YES (+)
Exp4	200	118.68	46.75	60.63	34.02	30	0.0001	YES (+)
Exp5	400	127.80	64.89	58.20	38.05	30	0.0001	YES (+)
Exp6	600	192.60	104.05	78.30	30.93	30	0.0001	YES (+)

(b) *Artificial Characters*

		PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	209.13	10.14	217.43	11.83	30	0.0044	YES (-)
Exp2	50	205.50	11.31	213.90	13.89	30	0.0128	YES (-)
Exp3	100	208.90	21.79	214.93	12.20	30	0.1813	NO
Exp4	200	217.63	19.13	211.13	16.18	30	0.1591	NO
Exp5	400	227.56	24.18	221.40	18.73	30	0.3201	NO
Exp6	600	233.70	17.40	230.63	19.62	30	0.5239	NO

(c) *Yeast*

Figure 6.11: Significance of Results: BS PLGP vs. CC PLGP

		CC PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	482.90	44.64	412.00	48.64	30	0.0001	YES (+)
Exp2	50	390.66	50.20	259.46	43.37	30	0.0001	YES (+)
Exp3	100	381.66	54.14	236.83	36.73	30	0.0001	YES (+)
Exp4	200	322.33	33.58	213.80	29.56	30	0.0001	YES (+)
Exp5	400	300.00	57.20	194.43	33.63	30	0.0001	YES (+)
Exp6	600	300.53	53.85	194.50	24.45	30	0.0001	YES (+)

(a) *Hand Written Digits*

		CC PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	220.00	44.51	140.96	32.83	30	0.0001	YES (+)
Exp2	50	197.00	55.98	65.00	34.21	30	0.0001	YES (+)
Exp3	100	184.00	67.69	62.63	34.23	30	0.0001	YES (+)
Exp4	200	171.00	77.34	60.63	34.02	30	0.0001	YES (+)
Exp5	400	188.00	68.12	58.20	38.05	30	0.0001	YES (+)
Exp6	600	192.00	71.80	78.30	30.93	30	0.0001	YES (+)

(b) *Artificial Characters*

		CC PLGP		BS PLGP				
	# Ins	Mean	SD	Mean	SD	n	p	Significant?
Exp1	20	221.07	19.02	217.43	11.83	30	0.3225	NO
Exp2	50	224.97	13.47	213.90	13.89	30	0.0018	YES (+)
Exp3	100	239.93	19.97	214.93	12.20	30	0.0001	YES (+)
Exp4	200	245.70	19.74	211.13	16.18	30	0.0001	YES (+)
Exp5	400	249.07	20.12	221.40	18.73	30	0.0001	YES (+)
Exp6	600	246.23	16.49	230.63	19.62	30	0.0008	YES (+)

(c) *Yeast*

It is expected that BS PLGP will converge more rapidly than PLGP during the first stage of evolution. BS PLGP uses a cooperative coevolutionary architecture to optimize multiple factors in parallel. In contrast PLGP is limited to optimizing a single factor in each program in each generation. Hence BS PLGP will clearly converge more rapidly than PLGP during the first stage of evolution.

It is significant that BS PLGP converges more rapidly than CC PLGP during the first stage of evolution. Both CC PLGP and BS PLGP use an identical cooperative coevolution architecture to optimize multiple factors in parallel. The key difference between these two methods lies in which blueprints are selected for factor fitness evaluation. CC PLGP uses an out of date estimate for which blueprints should be selected. BS PLGP uses PSO to search for high quality blueprints based on the current population of factors. The fast convergence of BS PLGP during the first stage of evolution demonstrates the importance of selecting the right blueprints for factor fitness evaluation. By using active search to locate high quality blueprints we have greatly improved the convergence rate of the CC PLGP architecture.

- **Fast Fine Tuning**
  BS PLGP converges more rapidly than CC PLGP and PLGP during the later stages of evolution, resulting in higher fitness final solutions.

  It is significant that BS PLGP converges more rapidly than PLGP during the later stages of evolution. As shown in chapter 5 CC PLGP excels at initial convergence, but fails at fine tuning high fitness solutions. In virtually all cases it was shown that if the system was run for a significantly long time the performance of PLGP would eventually surpass that of CC PLGP. Despite using the same basic CC architecture as CC PLGP, BS PLGP still excels are fine tuning high fitness solutions. Clearly locating high fitness blueprints greatly improves

the fine tuning ability of the CC PLGP architecture.

- **Small Programs**
  The performance difference is less noticible when small programs with few factors are used. This is to be expected as small programs with few factors are not well suited to CC PLGP, and by extension BS PLGP. The reasons behind this are discussed in detail in chapter 5.

  It is significant that when small programs are used BS PLGP significantly outperforms PLGP on both the *Hand Written Digits* and *Artificial Characters* data sets. BS PLGP uses the same architecture as CC PLGP, and as discussed in chapter 5 CC PLGP performs poorly when small programs with few factors are used. Therefore it is particularly noteworthy that when small programs are used BS PLGP outperforms PLGP on two of our data sets and has comparable performance on the third.

  Finally it is important to note that short BS PLGP programs are not what we are really interested in. Typically, short programs are not powerful enough for most interesting applications. On difficult problems neither PLGP, CC PLGP, or BS PLGP can achieve good performance using short programs.

- **Large Programs**
  BS PLGP is particularly effective when large programs with many factors are used. This is to be expected as BS PLGP is an adaptation of CC PLGP, and CC PLGP is also particularly effective when large programs with many factors are used.

  PLGP performs poorly when large programs are used as each program has many factors, and PLGP can only optimize one factor at a time. CC PLGP performs well during the first stage of evolution when large programs are used, however it is difficult to find high

fitness solutions due to the difficulty of fine tuning. BS PLGP allows fast initial convergence and fine tuning of high fitness solutions. Therefore BS PLGP can produce outstanding results when large programs are used, significantly outperforming both alternative approaches.

6.6 Chapter Summary

Selecting high fitness blueprints for factor evaluation is critical to the success of the CC PLGP algorithm. Selecting blueprints with poor fitness disrupts the evolution of factors and negatively impacts algorithm performance. The SANE inspired method considered in chapter 5 where a population of blueprints is evolved in parallel with the factor subpopulations has clear disadvantages. Most importantly, the blueprints generated through this approach are not based on the current population of factors. Hence this approach leads to poor blueprints being selected for fitness evaluation, and overall poor algorithm performance.

In this chapter we have introduced a new approach to blueprint selection based on searching a carefully defined blueprint space through particle swarm optimization. This approach, called Blueprint Search PLGP (BS PLPG) bases its blueprint selections on the current factors, not an out of date estimate. BS PLGP significantly outperforms both PLGP and CC PLGP, demonstrating the importance of good blueprint selection.

6.6.1 Next Step

All of the algorithms we have developed thus far offer significant advantages in terms of classification accuracy. Unfortunately many of these algorithms have extra processing steps, introducing additional overhead and slowing fitness evaluation. This is particularly true for algorithms such as BS PLGP where PSO requires a large number of blueprint evaluations.

Therefore from this point onwards this thesis will change tack and focus on optimizing the efficiency of the algorithms already developed. In this way we will maximise the applicability of this work.

Chapter 7

Execution Trace Caching for Linear Genetic Programming

7.1 Introduction

A major problem present in all GP variants is the extensive time required to arrive at a satisfactory solution. The quality of the solutions produced by GP depends heavily on the time available. Increasing the training time of a GP system results in greater exploration of the search space, and hence the discovery of superior fitness solutions. Unfortunately real world problems often impose severe restrictions on the available training time. These time constraints limit the applicability of GP, as the majority of GP systems cannot evolve effective solutions within the available time.

In order to maximize the applicability of the algorithms presented in this thesis it is vital to minimize the time they require to find satisfactory solutions. Producing high quality solutions is meaningless if these solutions cannot be found within the time constraints of the problem. Hence for the remainder of this thesis we will focus on minimizing the training time of our new LGP algorithms. This chapter in particular will focus on minimizing the training time of conventional LGP in order to establish a baseline for this work (PLGP algorithms).

Fitness evaluation is the most computationally costly component of GP. GP requires each potential solution to be executed on a large number of training examples. While a single fitness evaluation is usually fast, there are often hundreds of individuals and thousands of training examples, resulting in significant running times.

There are a few major approaches to reducing the time required for fitness evaluation in GP. One approach is to reduce the number of training examples by carefully selecting a representative subset [30, 29, 94]. Another is to improve the fitness evaluation procedure directly, typically by parallelizing the fitness evaluations and the use of Graphics Processing Units [16, 77]. Unfortunately these two approaches have clear limitations, either needing extra work expertise for selecting good representations, or requiring additional hardware support in addition to a parallel algorithm.

Caching is a third approach which lacks the limitations of the first two methods. We choose to focus on minimizing algorithm training time through the use of caching. Caching is a commonly used technique for decreasing execution time in many domains. Caching trades a cost in memory for a saving in execution time by storing partial results, preventing the redundant reevaluation of many already computed functions. Caching is a natural fit for linear genetic programming, where we repeatedly execute similar functions many times on the same data. By caching partial programs it is possible to recycle prior results, hence decreasing execution times [43, 75, 90]. Due to crossover, many programs in the same generation will be related, sharing common program code. Caching can take advantage of this by storing common results and fetching them when required [43, 75]. Caching has been applied to TGP with a certain level of success [43, 75, 90, 28, 91, 76] but caching has not been extensively used in LGP.

Therefore this chapter will focus on minimizing the execution time of conventional LGP through the use of caching. This work will serve as both a baseline indicator for the work in later chapters, and a standalone improvement to LGP. By comparing the execution time of cached LGP

to that of cached PLGP we will gain some insight into the comparative advantages of these two architectures. Furthermore, we will be able to directly compare the optimized execution time of all algorithms presented in this thesis.

7.1.1 Chapter Goals

The overall goal of this chapter is to decrease the training time of LGP. The execution time savings presented in this chapter will serve as both a baseline for work in later chapters, and a standalone improvement. Specifically, this chapter has the following research objectives:

- Develop a new caching technique for linear genetic programming which decreases the time it takes to evaluate LGP programs.
- Derive an equation which describes the cost-benefit trade off associated with caching.
- Use this equation to optimize the caching parameters, and hence determine in which situations caching is cost effective.
- Empirically confirm the theoretical results presented in this paper.

7.2 Execution Trace Caching for LGP

In many ways LGP presents a natural environment for caching. The combination of crossover and selection ensures that the vast majority of individuals which occur during evolution will be closely related. These related individuals will share many common program components. Existing caching techniques already exploit common program components. Programs are decomposed into code segments, and the results of segment execution are cached. Therefore if several programs share common code

segments then execution can often be avoided by fetching partial results from the cache.

Existing caching techniques have a single universal cache. When a particular instruction sequence is recognized to occur frequently within the population, that instruction sequence is then cached. Each time a program is executed we consult the entire cache to determine whether part or all of the required result can be fetched.

Universal caching can be effective [90] but leaves significant room for improvement. In particular universal caching completely ignores useful information about program relationships. A more intelligent approach to caching would be to use this information to link specific programs to specific cached information.

It is possible to track inter-generational program relationships during the evolution stage of GP. Furthermore it is possible to track precisely which program code is shared between related programs. Using these two pieces of information it is possible to cache specific code segments for each program.

7.2.1 Concept

We develop a new caching algorithm based on exploiting the relationships between programs. We cache the execution of each parent in order to expedite the execution of all offspring. By keeping track of its parents, each offspring will know exactly which cached information it can use to expedite its own execution. We call this *execution trace caching*.

A LGP program is a linear sequence of instructions. Each instruction stores the results of its execution in the register collection. The input of one instruction is the output of all previous instructions, in the form of the values currently in the register collection. In other words, we can trace the execution of the program through the values in the registers after each instruction execution, generating a table like the one in figure 7.1.

Program Execution

Program						Execution Trace		
Index	Instruction					r[1]	r[2]	r[3]
0	-					0	0	0
1	r[1]	=	3.1	+	f1;	3.2	0	0
2	r[3]	=	f2	/	r[1];	3.2	0	0.94
3	r[2]	=	r[1]	*	r[1];	3.2	10.24	0.94
4	r[1]	=	f1	-	f1;	0	10.24	0.94
5	r[1]	=	r[1]	-	1.5;	-1.5	10.24	0.94
6	r[2]	=	r[2]	+	r[1];	-1.5	8.74	0.94

Figure 7.1: Example of an execution trace

It is important to note that the current state of the registers completely describes the program execution up until that point in time. In other words, as long as we are given the correct register collection state, we can begin program execution from part way through the program. Therefore if we cache the execution trace for each program applied to each training example we can re-execute a specific program on a specific training example starting from any point in the program.

7.2.2 Complete Caching

The problem is that when the genetic operators are applied to the program during evolution, the program instructions will change. This means that the execution trace will also change, and the previous cached execution trace will no longer be correct.

We assume that reproduction[1] occurs by selecting an instruction from the program uniformly at random, and modifying some number of subsequent instructions. All instructions prior to the modification point will be

[1] ignoring elitism

identical to the previous generation. Therefore the execution trace prior to the modification point will also be identical to the previous generation. This means that we can always restart execution from the first modified instruction using the cached execution trace and arrive at the correct result.

7.2.3 Approximate Caching

Unfortunately caching execution traces incurs a cost, which must be balanced against the savings it generates. Caching every single step in the execution is prohibitively expensive both in terms of memory and computation time, so an alternative approach is required.

We decrease the overhead cost of caching to manageable levels by caching only a small part of each execution trace. We choose a small number of evenly spaced instructions, and for each of these we cache the state of the registers. We use the term *cache points* to describe the number of cached instructions. This means that for the majority of program instructions no caching occurs, greatly reducing cache overheads.

Cached program execution remains the same with one minor difference: We now begin execution from the *closest cached instruction*. In our initial caching algorithm we begin execution at the first modified instruction. When using approximate caching many of the instructions are not cached. Therefore we search backwards through the program until we find a cached instruction, and begin execution from this cached value. The approximate caching algorithm is given in algorithm 3.

It is clear that approximate caching requires more instruction executions than complete caching. However approximate caching also incurs much lower overhead costs than complete caching. Hence we expect the benefits of approximate caching to greatly outweigh the necessary sacrifices, giving greatly improved performance.

The performance of the execution trace caching algorithm will be di-

Algorithm 2 PROGRAM EXECUTION WITH CACHING

```
begin
    Let I = earliest modified instruction
    index = I − I MOD num_cache
    registers = cache[index]
    for (i = index ... |instructions|) do
        if (i MOD num_cache = 0) then
            cache[i] = registers
        registers = execute(instruction[i], registers)
```

rectly related to the number of cache points. The optimum number of cache points is investigated in detail in section 7.3. For now, assume c cache points evenly spaced throughout our program of length n, so we stop every n/c steps in our execution and cache the current state of the registers. An example of caching using three, or six cache points is given in table 7.1.

Table 7.1: Caching using two, three, or six cache points

Program		c = 6			c = 3		
index	Instruction	Reg1	Reg2	Reg3	Reg1	Reg2	Reg3
0	-	-			-		
1	r[1] = 3.1 + f1;	3.2	0	0	3.2	0	0
2	r[3] = f2 / r[1];	3.2	0	0.94	-		
3	r[2] = r[1] * r[1];	3.2	10.24	0.94	3.2	10.24	0.94
4	r[1] = f1 - f1;	0	10.24	0.94	-		
5	r[1] = r[1] - 1.5;	-1.5	10.24	0.94	-1.5	10.24	0.94
6	r[2] = r[2] + r[1];	-1.5	8.74	0.94	-		

7.3 Theoretical Analysis

Now that we have established an (efficient) caching algorithm it is important to develop theoretically motivated guidelines for optimal algorithm

deployment. Caching always results in a trade off, with higher levels of caching granting increased execution time savings, but incurring larger overhead costs. Badly applied caching can actually result in a net performance loss. We wish to determine a general rule for choosing the parameter settings which give optimum performance for any given situation.

Choosing the correct number of cache points is the key step in deploying the LGP execution trace caching algorithm. The level of caching is completely controlled by this single parameter. If too many cache points are used the cache will be large, with correspondingly large overheads. If too few cache points are used then only small performance improvements are possible.

In this section we investigate how the number of cache points used affects the performance of the execution trace caching algorithm. In order to do so we require the following variables:

- p: The number of LGP programs in the population (population size).
- i: The number of instructions per program.
- c: The number of cache points per program.
- r: The number of registers.
- t: The number of training examples for a task.

7.3.1 Savings

We begin this section by formalizing the benefits of caching. We derive an equation which we can use to calculate the number of instruction executions saved by caching. We make the following assumptions:

- The cache points are evenly spaced. (A trivial information theory argument will show this is optimal).

- We always backtrack as little as possible, i.e we begin program execution using the closest possible cache point.

- The modification point occurs uniformly at random.

Suppose we modify a program at position x. Then we are required to backtrack to some cache point y such that $y < x$. Clearly it makes sense to backtrack to the *greatest* cache point y such that $y < x$, since this will result in the fewest instructions being executed. If we have c cache points, then each program is split up into $d = c + 1$ sections. Each section is equally large, so each section is of length $i/(c+1)$ and each section has probability $i/(c \times i) = 1/c$ of containing the modified instruction. If the modification occurs in the first section caching is of no benefit. If it occurs in the second section, we save $i/(c + 1)$ instruction executions, if it occurs in the third section, we save $2i/(c + 1)$ instruction executions etc.

We now calculate the *expected* savings. Let a be the index of a specific section. Let $p(a)$ be the probability of section a containing the modified instruction. Let $s(a)$ be the number of instruction executions avoided if section a contains the modified instruction. We calculate $E(s(a))$, the expected number of instruction executions saved.

$$\begin{aligned} E(s(a)) &= \sum_{a=1}^{d} p(a)s(a) \\ &= \sum_{a=1}^{d} \frac{1}{d} \times \frac{(a-1)i}{d} \\ &= \frac{i}{d^2} \sum_{a=1}^{d} \times (a-1) \\ &= \frac{i}{d^2} \times \frac{d(d-1)}{2} \\ &= \frac{i}{d} \times \frac{d-1}{2} \end{aligned} \tag{7.1}$$

$$\begin{aligned} &= \frac{i}{2} - \frac{i}{2d} \\ &= \frac{i}{2} - \frac{i}{2(c+1)} \end{aligned} \tag{7.2}$$

This result is exactly what we would expect. Firstly, the more cache points we have, the fewer instructions we are required to execute, and hence the greater our execution time savings. Secondly, as the number of cache points approaches i, the execution time savings asymptotically approaches $i/2$. In other words only half of the instructions are executed.

7.3.2 Cost

By caching execution traces we incur a cost, both in terms of memory and in terms of execution time. These costs are actually closely related, since the primary cost in time is caused by reading from and writing to this memory. We must cache r registers for each of the d cached instructions.

$$\text{cost} = r \times d \tag{7.3}$$

To begin with we need to check that the memory requirements of caching are physically practical. Note that the total cost of cahing can be found by multiplying equation 7.3 by both the number of programs and the number of training examples. Selecting some large values let:

- b = 4 (float)
- r = 20
- c = 10
- t = 1000
- p = 1000

In this case memory usage would be $420 \times 10 \times 1000 \times 1000 = 800,000,000$ bytes $=$ 800Mb. While this is a large amount of memory it is still well within practical limits.

7.3.3 Optimization

We now determine the optimum number of cache points for any given problem. We do this by combining our savings and cost equations and differentiating to find the stationary point.

$$\begin{aligned} \text{net savings} &= \text{savings} - \text{cost} \\ &= \frac{i}{2} - \frac{i}{2d} - r \times d \qquad (7.4) \\ \frac{d(\text{net savings})}{d(d)} &= \frac{i}{2d^2} - r \\ 0 &= \frac{i}{2d^2} - r \\ r &= \frac{i}{2d^2} \\ d^2 &= \frac{i}{2r} \\ d &= \sqrt{\frac{i}{2r}} \qquad (7.5) \end{aligned}$$

Optimal performance is achieved by setting the number of cache points to be proportional to the square root of the number of instructions over the number of registers. Hence the optimal number of cache points is always going to be very small relative to the number of instructions. This confirms what we already knew: we require only a few cache points in order to achieve large performance improvements. We are now in a position to calculate the net benefit of caching by substituting our optimal value for d

back into equation 7.4:

$$\begin{aligned}
\text{net savings} &= \frac{i}{2} - \frac{i}{2d} - r \times d \\
&= \frac{i}{2} - \frac{i}{2\sqrt{\frac{i}{2r}}} - r\sqrt{\frac{i}{2r}} \\
&= \frac{i}{2} - \frac{i}{2\frac{\sqrt{i}}{\sqrt{2r}}} - r\frac{\sqrt{i}}{\sqrt{2r}} \\
&= \frac{i}{2} - \frac{\sqrt{i}\sqrt{2r}}{2} - \sqrt{i}\sqrt{2r} \\
&= \frac{i - 3\sqrt{i}\sqrt{2r}}{2}
\end{aligned} \tag{7.6}$$

We want to know what *fraction* of the execution time we are saving. Our current calculation is in terms of the number of instruction executions saved. Hence we calculate the fractional savings by dividing equation 7.6 by the number of instructions.

$$\begin{aligned}
\text{fraction savings} &= \frac{i - 3\sqrt{i}\sqrt{2r}}{2i} \\
&= \frac{1 - 3\sqrt{\frac{2r}{i}}}{2}
\end{aligned} \tag{7.7}$$

We are now in a position to calculate the critical point at which caching becomes cost effective. The overhead cost of caching depends solely on the number of registers. The expected execution time savings depends on both the number of registers and the number of instructions. Hence a large instruction to register ratio (i/r) will give good caching performance, while a low instruction to register ratio (i/r) will give poor caching performance. We calculate the minimum instruction to register ratio which still

allows performance improvement from caching.

$$\begin{aligned} 0 &< \frac{i - 3\sqrt{i}\sqrt{2r}}{2} \\ 3\sqrt{i}\sqrt{2r} &< i \\ 3\sqrt{2r} &< \frac{i}{\sqrt{i}} \\ 3\sqrt{2r} &< \sqrt{i} \\ i &> 18r \end{aligned} \tag{7.8}$$

Caching is effective when the number of instructions (i) is at least 18 times larger than the number of registers (r).

It is important to note that these results rely on the assumption that the cost of copying r registers is equal to r times the cost of copying one register. This is not the case in modern programming languages where utilities exist to rapidly copy whole blocks of memory (such as r registers stored in an array). In other words, $18r$ is an upper bound on the number of instructions required before caching becomes worthwhile, and in practice it is highly likely that caching can be made cost effective for much smaller instruction to register ratios (i/r).

7.4 Experimental Design

We conduct a series of three experiments. The following parameter settings are common to all three experiments.

7.4.1 Experiments

Experiment One

In the first experiment we will compare the execution time of LGP programs with and without caching. In this experiment we will vary the size of our LGP programs between 10 and 150 while using the optimum number of cache points calculated using equation 7.6.

Table 7.2: Parameter Configurations

Parameter	Value
Population	1000
Max Gens	400
Mutation	30%
Elitism	10%
Crossover	60%
Tournament Size	5
Runs	30
Registers	2

Experiment Two

In the second experiment we will compare the execution time of LGP programs with caching with our theoretical estimates. Once again we will vary the size of our LGP programs between 10 and 150 while using the optimum number of cache points calculated using equation 7.6.

Experiment Three

In the third experiment we investigate how LGP program execution time changes when we vary the number of cache points (c). In this experiment we vary the number of cache points between 0 and 20 while keeping all other parameters constant. We wish to determine the trend in execution time caused by varying the number of cache points. This trend will be most obvious when we can cause large variation in the execution time savings. Large variation is only possible with large LGP programs. Therefore in this experiment we use large LGP programs with 200 instructions.

7.4.2 Data Set

The choice of data set for running the experiments is not important for this work. The classification accuracy will remain the same for LGP with or without caching. We are not interested in the performance of the LGP system, and the program execution time is independent of the data set used. The data set simply serves as a source of inputs to the programs, and to this end a series of random numbers would suffice equally well. The only important parameters are the number of instructions, the number of registers and the number of cache points used, although details of the other parameters can be found in Table 7.2.

Purely for convenience we chose the *Hand Written Digits* data set described in chapter 3 for the purposes of testing our algorithm. In order to limit the number of registers and to clearly present the results we reduce this problem from a 10 class problem to a two class problem.

7.5 Results

In this section we compare our theoretical estimates with empirical results. We present the results of the experiments detailed in section 7.4.1 together with discussion.

7.5.1 Caching vs. No Caching

In this section we investigate how our new caching algorithm affects the running time of the LGP algorithm. Specifically we aim to determine whether our new caching algorithm, on average, significantly decreases the time required to execute a LGP program.

There are three important results present in figure 7.2:

- Caching is detrimental for small LGP programs.
- Caching is beneficial for large LGP programs.

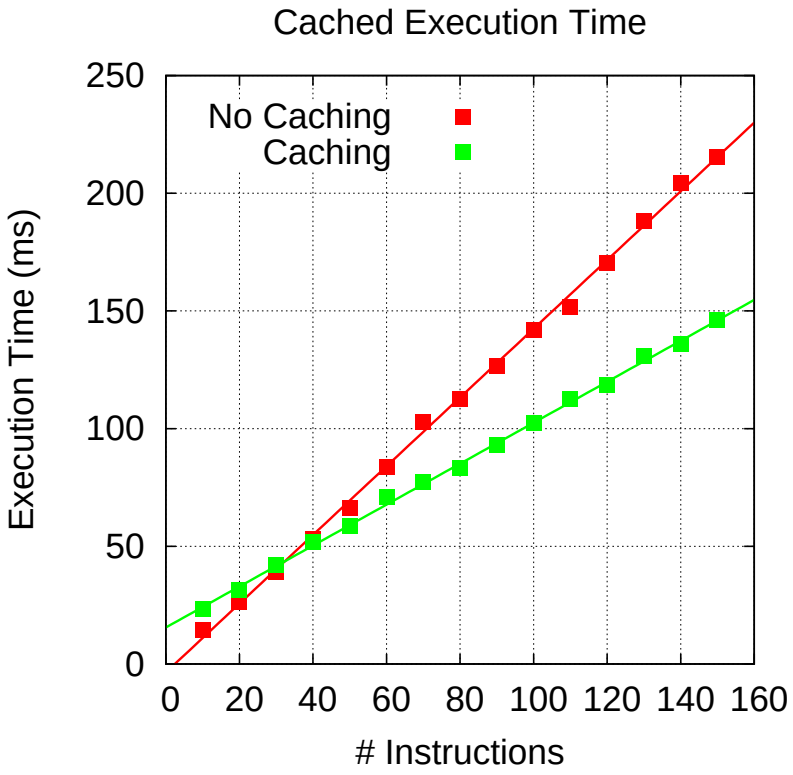


Figure 7.2: Comparing the running time of LGP programs of various lengths with and without caching

- The critical point at which caching becomes cost effective occurs at roughly 32 instructions.

The empirical results shown in figure 7.2 are precisely in line with our expectations.

Caching grants a *percentage saving* in return for a *constant overhead cost*. Caching is beneficial only if *the percentage saving exceeds the constant overhead*. Small LGP programs already execute extremely rapidly. Therefore the percentage savings of caching are outweighed by the overhead costs. Hence caching is not cost effective for small LGP programs. Conversely large LGP programs are slow to execute. Therefore the percentage sav-

ings of caching outweigh the overhead cost. Hence caching is extremely effective for large programs.

In section 7.3 we calculated that caching is cost effective when the number of instructions is at least 18 times as large as the number of registers. In this experiment our programs use 2 registers, so we expect caching to be cost effective when programs have at least 36 instructions. Our empirically determined value for the critical point is approximately 32, however as we discussed in section 7.3 our theoretical value is an upper bound. These results suggest that our theoretical predictions for the critical point are in precise agreement with our empirical results.

7.5.2 Theoretical Performance

In this section we investigate whether our theoretical predictions are borne out in practice. Specifically we aim to determine whether it is possible to achieve the theoretical results of section 7.3 in a practical implementation.

The empirical results shown in figure 7.3 are precisely in line with our theoretical predictions. For all parameter settings our concrete implementation achieves the expected execution time savings. In fact it appears that for large LGP programs our implementation slightly outperforms the suggested theoretical values. This is reasonable, since in our calculations we made worst case assumptions about the overhead cost of caching. It is likely that our implementation was highly efficient at memory manipulation, allowing for slightly faster program execution than we calculated based on our theoretical analysis.

7.5.3 Number of Cache Points

In this section we investigate how the number of cache points affects the performance of our caching algorithm. In particular we wish to empirically determine the optimum number of cache points, and confirm whether this agrees with our theoretical estimate.

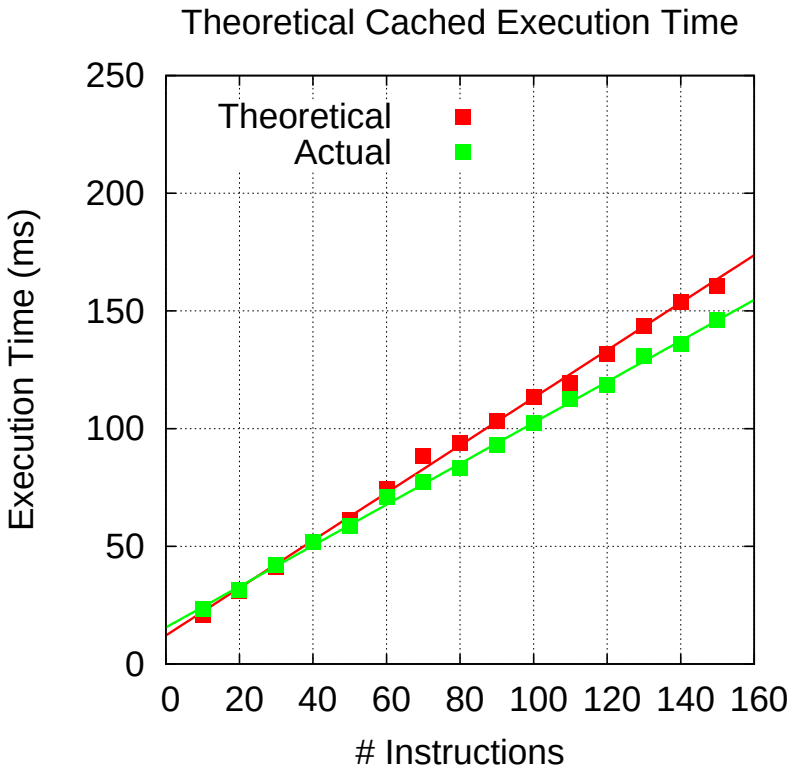


Figure 7.3: Comparing the theoretical running time calculated in section 7.3 to the empirically measured running time

The empirical results shown in figure 7.4 agree precisely with our theoretical estimates. Using equation 7.5 we estimate that the optimum number of cache points is 7. Our empirical results agree with this result, showing programs execute most rapidly when using approximately 7 cache points.

It is interesting to study the shape of the results curve. Performance improves rapidly with the addition of cache points up to the critical point, in this case 7 cache points. Further increasing the number of cache points beyond the optimum number causes programs to execute slightly less rapidly, but the difference is minor. In other words using fewer than the optimum number of cache points greatly impairs efficiency, while using

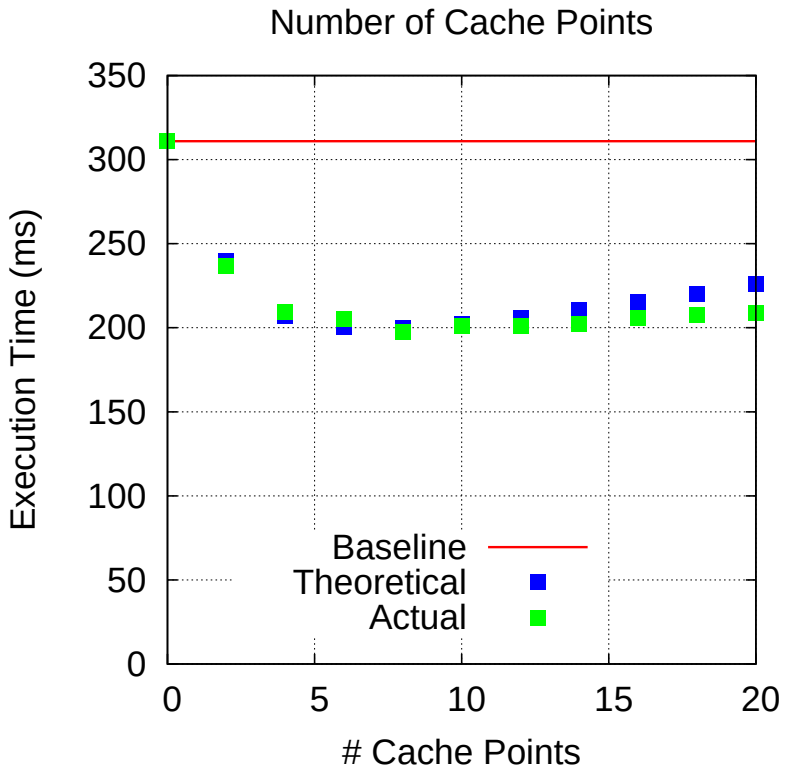


Figure 7.4: Comparing the running time of LGP programs when various numbers of cache points are used

more than the optimum number of cache points causes only slightly impairs efficiency. Considering the space cost for large numbers of cache points we suggest that when in doubt it is better to use slightly more cache points.

7.6 Chapter Summary

The goal of this chapter was to develop a caching technique for LGP which significantly decreases program execution time. This goal was successfully achieved by the development of the execution trace caching algo-

rithm. Execution trace caching exploits the inter-generational relationships between between individuals to associate specific cache material with specific programs. Programs use their parents' execution trace from the previous generation to expedite their own execution.

The theoretical results suggest that an optimal number of cache points exists, and that this optimal value is dependent on the parameter configuration. Specifically using $\sqrt{i/2r}$ cache points is optimal, and caching is cost effective whenever the inequality $i > 18r$ is satisfied. These two equations provide precise guidelines for *when* and *how* the execution trace caching algorithm should be deployed.

The empirical experimental results confirm the effectiveness of execution trace caching. LGP systems demonstrated a significant decrease in program execution time when the execution trace caching algorithm was applied. In addition our empirical results precisely match our theoretical results, strongly supporting the validity of the equations in section 7.3.

7.6.1 Next Step

This material forms a useful baseline for future work. The next chapter will focus on using caching to decrease the execution time of various PLGP algorithms. Using the results detailed in this chapter we will be able to empirically compare optimized algorithm execution time. This will grant us significant insight into the pros and cons of specific algorithms.

Chapter 8

Caching for Parallel Linear Genetic Programming

8.1 Introduction

Algorithms based on the PLGP architecture have significant performance advantages, however at present these advances come at the cost of an increase in algorithm run time. PLGP, CC PLGP, and BS PLGP all have significantly superior effectiveness over conventional LGP. Unfortunately, all three algorithms are significantly less efficient than conventional LGP, as it takes significantly longer to evaluate each individual.

As discussed in chapter 7, algorithms are most useful if they can produce high quality solutions *quickly*. If we can decrease algorithm execution time we can greatly increase algorithm usefulness by increasing the number of solvable problems. The usefulness of PLGP, CC PLGP, and BS PLGP is currently limited because all three algorithms execute slowly. Hence this chapter focuses on increasing the usefulness of PLGP, CC PLGP, and BS PLGP by decreasing the execution time of all three algorithms.

Decreasing algorithm execution time also has the effect of indirectly improving algorithm performance. For difficult problems LGP systems rarely converge to a perfect solution within the specified time constraints.

Under such circumstances the quality of the final solution produced using a LGP algorithm depends on how long it takes to complete each generation. As execution time decreases, the number of iterations increases, allowing more solutions to be explored, and a stochastically better final solution. In other words if we can decrease the execution time of the PLGP, CC PLGP, and BS PLGP algorithms, then we can increase algorithm performance for time sensitive problems.

Techniques which decrease the execution time of PLGP, CC PLGP, and BS PLGP complement the performance advantages these algorithms already possess. If performance is limited by time constraints which restrict the number of generations, then reducing algorithm execution time will enhance performance. Furthermore, the greater the degree of execution time reductions, the better the algorithm performance. As in chapter 7, we will focus on improving algorithm run time through caching. We have already shown that caching is effective in decreasing the execution time of LGP. Now we wish to extend the ideas developed in chapter 7 to the PLGP architecture.

The PLGP architecture is particularly well suited to caching due to the parallel form of its programs. Caching is of limited effectiveness for LGP programs due to interwoven instruction dependencies. These dependencies make it difficult to localize the changes which occur during evolution, compromising effective caching. In contrast PLGP programs consist of independently executed factors with no dependencies between factors. Evolutionary changes are localized to a single factor, giving the PLGP architecture the potential for enormous execution time savings through caching. If we can realize this potential then we can further improve the performance of PLGP, CC PLGP, and BS PLGP above and beyond what is possible using conventional LGP algorithms.

8.1.1 Objectives

In this chapter, we aim to develop two caching techniques which can significantly reduce algorithm execution time. One caching technique will be for PLGP, the other will be for both CC PLGP and BS PLGP. These caching techniques should exploit PLGP's parallel architecture and lack of dependencies. The primary goal is to significantly improve the system efficiency by reducing the CPU time required for each fitness evaluation. Specifically this chapter has the following research objectives:

- Develop a new caching technique for PLGP which decreases the time it takes to evaluate PLGP programs.
- Develop a new caching technique for both CC PLGP and BS PLGP which significantly decreases the time it takes to evaluate both CC PLGP programs and BS PLGP programs.
- Derive equations which describe the cost-benefit trade off associated with caching.
- Use these equations to optimize the caching parameters, and hence determine in which situations caching is cost effective.
- Empirically confirm the theoretical results presented in this chapter.

8.2 Caching for PLGP

In this section we introduce a new caching algorithm for PLGP. As we will see PLGP is naturally suited to caching due to its parallel architecture. We focus on caching between related individuals in different generations as described in chapter 2. By caching the execution of each program it is possible to speed up the execution of related programs in subsequent generations.

8.2.1 Basic Caching

In this subsection we develop a basic approach to caching PLGP programs which avoids a large number of instruction executions, but also incurs significant overhead.

In PLGP, each program consists of n sequences of instructions or *factors*. PLGP program execution proceeds by executing all program factors in parallel to produce a set of result vectors, and then summing these result vectors (see figure 4.4, page 55). When program evolution occurs, precisely one of these factors is modified (genetic operators will apply to instructions within a single factor). Hence only the instructions in the modified factor will be affected, and only a single result vector will change. In other words *all program factors which were not modified during evolution will produce identical output to the previous generation*. Suppose we cache the factor result vectors from generation to generation. We can use the cached *factor* result vectors to compute the new *program* result vector. In this way we can avoid executing the majority of the program factors, and hence the majority of program instructions.

In figure 8.1 an example of normal PLGP program execution is compared to an example of cached PLGP program execution. In this example the second factor has undergone modification. Hence the result vectors for factors one and three can be retrieved from the cache, while the result vector for factor two must be recalculated. Notice that the cache holds one result vector for each factor executed on each training example.

Basic caching avoids a large number of instruction executions. All factor result vectors are cached, and only the single modified vector needs to be executed. Hence basic caching avoids the majority of instruction executions. Furthermore the savings associated with basic caching will scale beneficially with the number of factors.

Unfortunately basic caching has a large memory footprint and significant computational overhead. It requires us to store one result vector for each factor, for each program executed, on each training example. The

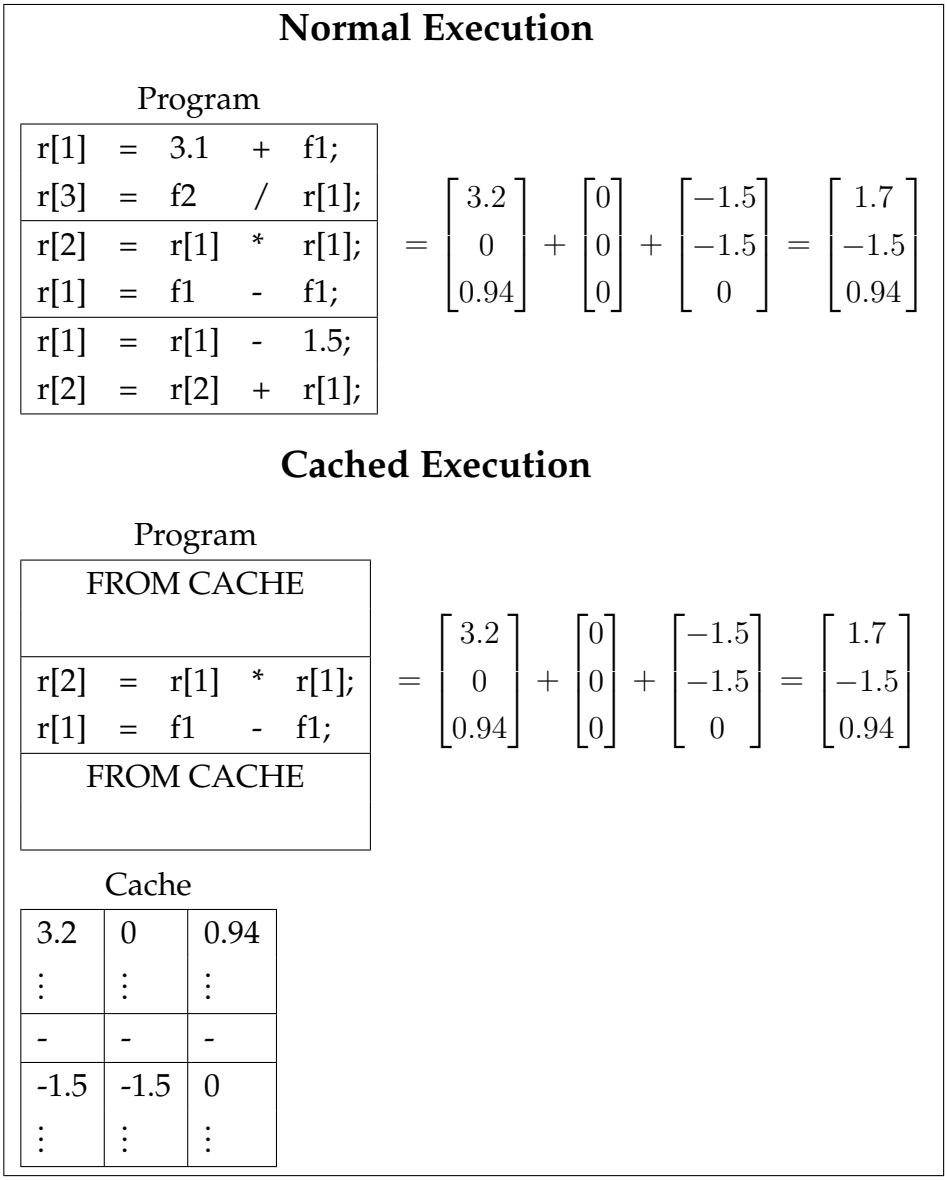


Figure 8.1: Contrasting normal PLGP program execution to basic cached program execution

high cost associated with managing such a large quantity of memory significantly offsets the intended savings. Furthermore, the memory required will increase as the number of factors increases, causing greater overheads

with larger programs.

Therefore, although basic caching offers significant savings, the associated overhead prevents this being an effective caching technique.

8.2.2 Difference Caching

In this subsection we refine our PLGP caching technique to remove the large overhead associated with caching.

We wish to cache program results in such a way that the cost of caching is *independent* of the number of program factors. In this way greater parallelization will result in greater execution time savings without greater overheads.

Caching with constant overhead is the best way to achieve large execution time savings. If execution time savings increase with program size, while overhead remains constant, then net savings will increase drastically with size. While small programs will enjoy only minor benefits from caching, these programs hardly need it, as small programs execute rapidly. On the other hand large programs which are slow to execute will experience massive benefits from caching, a highly desirable outcome.

It is possible to achieve constant overhead caching if we change the information we keep in the cache. We note that at each generation we calculate the same sum with one minor difference — the factor that underwent modification. The idea is that instead of storing one result vector for each factor we will store only the final output — the sum of these factors. To calculate the new program output we begin by retrieving the program result vector for the previous generation from the cache. From this sum we subtract the result vector for the previous version of the modified factor. Finally we add a new result vector for the current (post modification) version of the modified factor. The subtraction and addition steps serve to remove the outdated and incorrect values from the sum and replace them with corrected values. The following equations express this idea more

concisely. Let V_i^m be the output of the i'th *factor* at the m'th generation and let S^m be the *program output* at the m'th generation. Then:

$$\begin{aligned} S^m &= \sum_{i=1}^{n} V_i^m \\ S^m &= \sum_{i=1}^{n-1} V_i^m + V_n^m \\ S^{m+1} &= \sum_{i=1}^{n-1} V_i^{m+1} + V_n^{m+1} \end{aligned} \tag{8.1}$$

Without loss of generality let V_n be the factor that is modified in generation $m+1$. Then we know that factors $V_1, ..., V_{n-1}$ will evaluate to the same vector in generation $m+1$ as they did in generation m. Hence:

$$\begin{aligned} S^{m+1} &= \sum_{i=1}^{n-1} V_i^m + V_n^{m+1} \\ S^{m+1} &= S^m - V_n^m + V_n^{m+1} \end{aligned} \tag{8.2}$$

We have arrived at a recursive expression for the value of a PLGP program after the $n+1$'th generation in terms of its value at the n'th generation. This equation is the key to execution time savings for PLGP programs. It demonstrates that the value of a PLGP program can be calculated using only its output from the previous generation, plus the current and previous output of the modified program factor. Notice that this equation is independent of the number of program factors, hence very large programs can be executed very rapidly if they consist of a large number of factors.

The one component of this caching scheme we have not yet discussed is how to obtain V_n^m, the *previous value* of the *modified program factor*. One option is to cache this value for all training examples. Another option is to cache the previous factor itself (the code), and recalculate all the training examples on the fly. Caching the value of all training examples will double the size of the cache, but limit code execution to a single program factor. Caching a copy of the previous factor causes virtually no increase

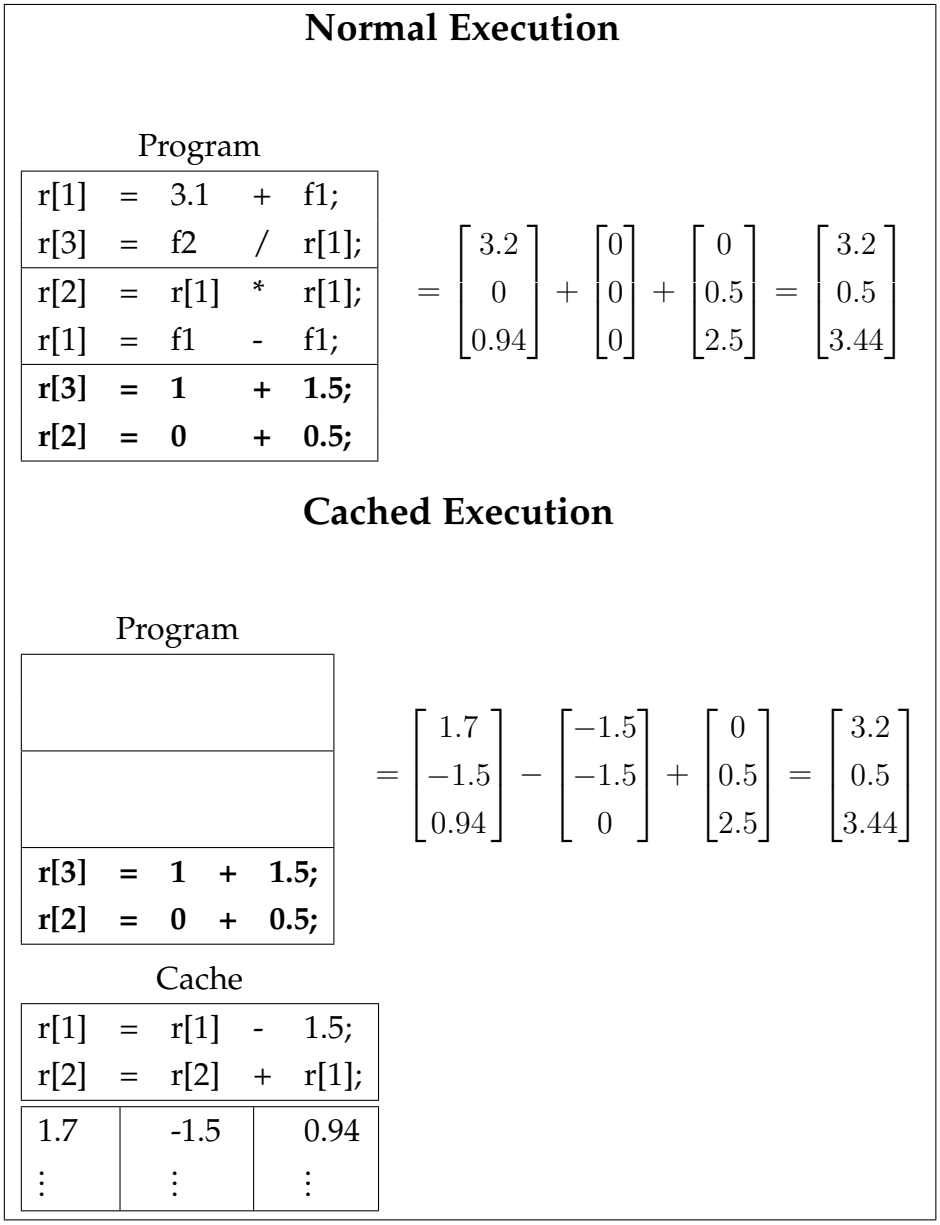


Figure 8.2: Contrasting normal PLGP program execution with advanced cached PLGP program execution. Assume the program in figure 8.1 is the parent of the program in this figure.

in the size of the cache, however it means we have to execute two factors for each training example. These two approaches are well suited to different situations. If program factors are small and contain few instructions then doubling the number of instruction executions incurs a much smaller cost than doubling the size of the cache. On the other hand if program factors are large and contain many instructions, then doubling the size of the cache may be significantly cheaper than doubling the number of instruction executions.

In this thesis we choose to minimize the cache size by storing a copy of the modified program factor. In other words, when we select a program factor for modification, we cache a copy of that factor *prior* to modification. When executing the modified program we use the cached factor, the new factor, and our cached program result vectors to calculate the new program result vector. This caching process is detailed in figure 8.2. In this example we have a PLGP program consisting of three factors, each containing two instructions. This program has three features and three registers. The cache contains a copy of the program factor prior to modification, together with the output vector for each training instance in the previous generation. To execute this program we take the output vector from the cache, recalculate and subtract the output of the cached program factor, and finally calculate and add the output of the new program factor.

We refer to this form of caching as *Difference Caching*. The program output is calculated using the prior output and the difference between program execution in two subsequent generations. Difference caching possesses the same benefits as basic caching, but achieves constant overhead costs.

8.2.3 Theoretical Analysis

Now that we have established an (efficient) caching algorithm for PLGP it is vital that we perform a rigorous theoretical analysis. This analysis will

provide a number of important results. Here we will consider:

- The *savings* associated with caching.
- The *cost* associated with caching.
- A general rule for choosing settings for the *caching parameters* which give optimum performance for any given situation.
- *Deployment guidelines* for the situations in which caching is cost effective.
- The *overall benefit* associated with caching.

In order to perform our theoretical analysis we require the following variables. Note that we assume all factors have an equal number of instructions:

- p: The number of PLGP programs in the population (population size).
- f: The number of factors per PLGP program.
- i: The number of instructions per factor.
- r: The number of registers.
- t: The number of training examples for a task.

Savings

In this subsection we formalize the benefits of caching. We derive an equation to calculate the number of instruction executions saved by caching.

Standard execution of a PLGP program requires all instructions in all factors to be executed and the resulting vectors to be summed. In other words $f \times i$ instructions are executed and $f \times r$ registers are summed.

Cached execution of a PLGP program requires a single factor to be executed and two vectors to be summed. Each vector consists of r registers. In other words i instructions are executed and $2r$ registers are summed. Hence we calculate $savings$, the savings associated with caching:

$$\begin{aligned} savings &= standard - cached \\ &= (f \times i + f \times r) - (i + 2r) \\ &= fi + fr - i - 2r \\ &= (f-1)i + (f-2)r \end{aligned} \tag{8.3}$$

This equation makes sense, as it expresses the notion that we avoid (f-1) factor evaluations, and we avoid (f-2) vector sums.

Cost

In this subsection we formalize the costs of caching. We derive an equation which we can use to calculate the additional cost resulting from caching.

Caching requires a single additional factor to be executed, and a single factor together with program output to be written to cache. Program output consists of r registers. Note that the cost of caching the factor is constant and negligible so can be ignored. The cost of executing an additional factor is i instructions, and the cost of caching program output is r. Hence we calculate $cost$, the cost associated with caching

$$cost = i + r \tag{8.4}$$

Caching Parameters

In this section we determine the caching parameters which optimize caching performance. While caching for PLGP has no explicit caching parameters, the values of f, i, and r will affect the efficiency of caching. In particular

f, the number of factors, will have a big effect on efficiency. We begin by combining our savings and cost equations to find the *net savings*.

$$\begin{aligned} \text{net savings} &= savings - cost \\ &= (f-1)i + (f-2)r - (r+i) \\ &= (f-1)i + (f-2)r - r - i \\ &= (f-2)i + (f-3)r \end{aligned} \tag{8.5}$$

According to equation 8.5 there are no optimal values for f, i, and r. Instead, increasing the number of factors will always improve the efficiency of caching, regardless of the number of registers or instructions.

Deployment Guidelines

In this subsection we formalize deployment guidelines for our PLGP caching algorithm. We derive an equation which we use to calculate the point at which caching becomes cost effective.

Caching is cost effective when the net savings is positive. We now use the net savings equation to determine the point at which caching becomes cost effective.

$$\begin{aligned} \text{net savings} &> 0 \\ (f-2)i + (f-3)r &> 0 \\ fi - 2i + fr - 3r &> 0 \\ f(i+r) &> 2i + 3r \\ f &> \frac{2i+3r}{i+r} \end{aligned} \tag{8.6}$$

This result tells us that caching is cost effective when the number of factors is larger than $\frac{2i+3r}{i+r}$. Table 8.1 gives the number of factors required

for various values of i and r. In addition, equation 8.6 asymptotically approaches 3, which tells us that caching is **always** cost effective if the number of factors is larger than 3. This makes sense according to the structure of our caching algorithm, as two factor executions and several vector sums are required to execute any program.

		i			
		5	10	15	20
r	5	2.5	2.33	2.25	2.2
	10	2.66	2.5	2.4	2.33
	15	2.75	2.6	2.5	2.43
	20	2.8	2.66	2.57	2.5

Table 8.1: The number of factors required to make caching cost effective for various numbers of registers and instructions.

The overhead cost of caching is independent of the number of factors, however execution savings is proportional to the number of factors. Increasing the number of factors while keeping factor size constant has no effect on the cost of caching, since precisely two factors will be cached. In contrast, increasing the number of factors while keeping the number of instructions constant will greatly decrease program execution time. If the number of factors increases, then the size of each factor will decrease, causing a proportional decrease in the number of instruction executions required. Hence programs with more factors give proportionally greater execution time savings, and programs with a large number of factors can be executed extremely rapidly.

Overall Benefit

So far we have obtained several important theoretical results. Caching is always cost effective when 3 or more factors are used. Increasing the number of factors always increases the efficiency of caching. In this sub-

section we study the *degree* of execution time savings. We derive an equation which we use to calculate the degree of execution time savings for a given parameter configuration.

We want to know what *fraction* of the execution time we are saving. Our current calculation is in terms of the number of instruction executions saved. Hence we calculate the fractional savings by dividing equation 8.5 by the total cost.

$$\begin{aligned} \text{fractional savings} &= \frac{\text{net savings}}{\text{total cost}} \\ &= \frac{(f-2)i + (f-3)r}{fi + fr} \\ &= \frac{fi + fr - 2i - 3r}{fi + fr} \\ &= \frac{fi + fr}{fi + fr} - \frac{2i + 3r}{fi + fr} \\ &= 1 - \frac{2i + 3r}{fi + fr} \end{aligned} \tag{8.7}$$

Equation 8.7 shows that caching for PLGP can result in enormous execution time savings. Examples for the percentage savings in execution time for various parameter combinations are shown in table 8.2.

		i			
		5	10	15	20
f	5	46.6%	50.0%	52.0%	53.3%
	10	73.3%	75.0%	76.0%	76.6%
	15	82.2%	83.3%	84.0%	84.4%
	20	86.6%	87.5%	88.0%	88.3%

Table 8.2: The fractional savings for r=10, with various instruction/factor combinations.

These results demonstrate the power of our PLGP caching technique. We see that using caching it is possible to decrease the execution time of a

reasonable sized PLGP program by almost an order of magnitude. This is vastly superior to what was achieved when applying caching to conventional LGP.

8.3 Caching for CC PLGP and BS PLGP

The CC architecture used in CC PLGP and BS PLGP provides unparalleled opportunities for execution time reductions through caching. Furthermore these two algorithms both use the CC architecture consisting of blueprints and factors. Both algorithms use the same programs executed in the same ways. Hence we developed a caching technique which can be applied to both of these algorithms in order to improve algorithm efficiency.

Blueprints are simple combinations of PLGP programs. Due to the CC architecture each program factor will be executed independently. This means any given factor will have the same result in every single blueprint it participates in. Operating under normal parameter settings each factor would be expected to participate in 10 or more blueprints. At present each blueprint is executed by executing all appropriate program factors and summing the resulting vectors, causing each factor to undergo 9 or more wasted executions.

A more intelligent approach would be to execute all program factors precisely once and cache the resulting vectors. To execute a blueprint we retrieve the cached output vector for each factor and sum these values to determine the program output. This caching algorithm is described in detail in algorithm 3.

The critical advantage of cached PLGP is that the cost of fitness evaluation depends only on the total number of program factors. In other words the cost of fitness evaluation is independent of the number of blueprints. Calculating blueprint output requires several vectors to be summed, however it requires no instruction executions. The cost of vector addition is trivial compared to the cost of instruction execution, therefore blueprint

Algorithm 3 PROGRAM EXECUTION WITH CACHING

```
begin
    for t ϵ Training Set do
        for prog ϵ Programs do
            execute prog on t;
            cache prog.result;
        for b ϵ Blueprints do
            Vector res;
            for factor ϵ b.factors do
                res = res + factor.result;
        b.result = res;
```

evaluation has almost no impact on the overall time required for fitness evaluation.

The efficiency of CC PLGP and BS PLGP is proportional to the number of factors per blueprint. Both algorithms are based on the concept of evolving subpopulations of factors which are useful in constructing high quality solutions. We determine the fitness of a factor by observing its performance in a number of blueprints. By allowing each factor to participate in a larger number of blueprints we increase the accuracy of the fitness values we assign to each factor. This in turn leads to subpopulations consisting of superior individuals, which in turn leads to solutions with better fitness.

Caching allows each factor to participate in a huge number of blueprints for an extremely low cost. This is not possible without caching, as each additional blueprint causes a large increase in program execution time. This is a huge advantage and can greatly improve algorithm performance.

8.3.1 Theoretical Analysis

Now that we have established an (efficient) caching algorithm for CC PLGP and BS PLGP it is vital that we perform a rigorous theoretical analysis. This analysis will provide a number of important results.

- The *savings* associated with caching.

- The *cost* associated with caching.
- A general rule for choosing settings for the *caching parameters* which give optimum performance for any given situation.
- *Deployment guidelines* for the situations in which caching is cost effective.
- The *overall benefit* associated with caching.

In order to perform our theoretical analysis we require the following variables. Note that we assume all factors have an equal number of instructions:

- b: The total number of blueprints.
- p: The total number of factors in *all* subpopulations.
- f: The number of factors per blueprint.
- i: The number of instructions per factor.
- r: The number of registers.

Savings

In this subsubsection we formalize the benefits of caching. We derive an equation which we can use to calculate the number of instruction executions saved by caching.

Evaluation *without* caching requires all instructions, in all factors, in all blueprints, to be executed; and the resulting vectors to be summed. In other words $i \times f \times b$ instructions must be executed, and $r \times f \times b$ vectors must be summed. Evaluation *with* caching requires all instructions, in all factors, in all subpopulations to be executed; and all registers in all factors in all blueprints to be summed. In other words $i \times p$ instructions must

be executed, and $r \times f \times b$ vectors must be summed. Hence we calculate $savings$, the savings associated with caching.

$$\begin{aligned} savings &= standard - cached \\ &= (ifb + rfb) - (ip + rfb) \\ &= ifb - ip \\ &= i(fb - p) \end{aligned} \tag{8.8}$$

This equation appeals to our intuitive understanding of the CC architecture. Evaluation *without* caching executes all instructions in all factors in all blueprints, while evaluation *with* caching executes all instructions in all factors in all subpopulations.

Cost

In this subsection we formalize the costs of caching. Caching does not have a significant computational cost associated with it. The only major cost associated with caching is the memory required to keep all of the results vectors for all factors in memory at once. Therefore the focus of this subsection is to ensure that the memory required is reasonable, for all reasonable parameter values. Selecting some large values, let $r = 20$ and $p = 5000$.

$$\begin{aligned} memory &= r \times p \times 8(bytes) \\ &= 20 \times 5000 \times 8(bytes) \\ &= 800,000bytes \end{aligned} \tag{8.9}$$

This is easily within the reasonable limits of modern computers.

Caching Parameters

There are no parameters to optimize for the CC PLGP/BS PLGP caching algorithm. Furthermore there is no computation cost associated with the caching algorithm. Therefore the net savings equation is equivalent to the savings equation.

Deployment Guidelines

In this subsection we formalize deployment guidelines for our CC PLGP/BS PLGP caching algorithm. We derive an equation which we use to calculate the point at which caching becomes cost effective.

Caching is cost effective when the net savings is positive. We now use the net savings equation to determine the point at which caching becomes cost effective.

$$\begin{aligned} \text{net savings} &> 0 \\ i(fb - p) &> 0 \\ fb - p &> 0 \\ fb &> p \end{aligned} \tag{8.10}$$

Equation 8.10 tells us that caching is cost effective when the total number of factors in all blueprints is larger than the total number of factors in all subpopulations. Table 8.3 gives the number of factors required to make caching cost effective for various values of p and b.

The results in table 8.3 show that in most cases a very small number of factors per blueprint is required to make caching cost effective. The results chapter 5 show that the best performance is achieved using programs with 10 or more factors. Hence caching is virtually always cost effective.

		p			
		250	500	750	1000
b	250	1	2	3	4
	500	0.5	1	1.5	2
	750	0.33	0.66	1	1.33
	1000	0.25	0.5	0.75	1

Table 8.3: The number of factors required to make caching cost effective for various numbers of registers and instructions.

Overall Benefit

So far we have obtained several important theoretical results. Caching is always cost effective for all reasonable parameter combinations. Increasing the number of factors increases the efficiency of caching. In this subsubsection we study the *degree* of execution time savings. We derive an equation which we can use to calculate the degree of execution time savings for a given parameter configuration.

We want to know what *fraction* of the execution time we are saving. Our current calculation is in terms of the number of instruction executions saved. Hence we calculate the fractional savings by dividing equation 8.10 by the total cost.

$$\begin{aligned} \text{fractional savings} &= \frac{\text{net savings}}{\text{total cost}} \\ &= \frac{(ifb + rfb) - (ip + rfb)}{ifb + rfb} \\ &= \frac{ifb + rfb}{ifb + rfb} - \frac{ip + rfb}{ifb + rfb} \\ &= 1 - \frac{ip + rfb}{ifb + rfb} \end{aligned} \tag{8.11}$$

Equation 8.11 shows that caching for CC PLGP can result in enormous execution time savings. Examples for the percentage savings in execution

time for various parameter combinations are shown in table 8.4.

		p			
		250	500	750	1000
b	250	60%	53.3%	46.6%	40%
	500	63.3%	60%	56.6%	53.3%
	750	64.4%	62.2%	60%	57.7%
	1000	65%	63.3%	61.6%	60%

Table 8.4: The percentage savings in execution time for i=20, r=10, f=10, for various combinations of p and b.

These results demonstrate the power of our CC PLGP/BS PLGP caching technique. We see that using caching it is possible to decrease the execution time of reasonably sized systems by up to 65%. This is vastly superior to what was achieved when applying caching to conventional LGP. Furthermore this result is extremely pessimistic as it is likely we have overestimated the cost of vector addition. Therefore we can expect our empirical results to outperform this theoretical estimate.

8.4 Experimental Setup

In this section we describe the experiments and parameter settings used in this chapter.

8.4.1 Data Set

In these experiments the data set chosen is unimportant. We are not attempting to compare PLGP and LPG program effectiveness, only program execution time. Hence the data set is simply a convenient way of instantiating features to concrete values so that program execution can proceed.

For this purpose, we use the *Hand Written Digits* data set described in chapter 3.

8.4.2 Parameter Configurations

The experimental parameter settings in table 8.5 are generic parameters, i.e. all experiments in this section will use these parameter settings. Since we are interested only in program execution time these parameters are not of particular importance and will not be considered in detail, however they must be instantiated to reasonable values for GP to proceed. These values in table 8.5 meet this requirement.

Table 8.5: Constant Parameters

Population	1000
Max Gens	400
Mutation	30%
Elitism	10%
Crossover	60%
Tournament Size	5
Runs	30
Registers	10

8.4.3 Experiments

PLGP

We will compare the execution time of LGP with that of PLGP, then the execution time of PLGP with cached PLGP, and finally the execution time of LGP with cached PLGP. We do this by determining experimentally the average program execution time for programs of various lengths. We will use equivalent LGP and PLGP programs by using the same number of

instructions for programs of both types. The number of factors in each PLGP program will be fixed throughout these experiments.

Having established that cached PLGP offers execution time savings, we will investigate how the number of factors in a PLGP program influences program execution time. Specifically, according to our theoretical analysis, we expect programs with more factors to offer greater execution time savings. Investigation proceeds by fixing the total number of instructions but varying the number of factors.

All experiments in this section will involve the parameter combinations shown in table 8.6. We will repeat these experiments twice, once with caching active and once without. Note that all results are the collated averages obtained from 30 repetitions using a specific combination of parameters.

Table 8.6: PLGP Experimental Parameter Combinations

	LGP	PLGP			
# factors	-	2	5	10	20
# Ins	Length	Factor Length			
20	20	10	4	2	1
40	40	20	8	4	2
60	60	30	12	6	3
80	80	40	16	8	4
100	100	50	20	10	5
200	200	100	40	20	10
300	300	150	60	30	15

CC PLGP/BS PLGP

We will compare the execution time of PLGP with that of cached PLGP. We do this by determining experimentally the average program execution time for programs of various lengths.

Having established that cached PLGP offers execution time savings, we will investigate how the number of factors in a CC PLGP/BS PLGP program influences program execution. Specifically, according to our theoretical analysis, we expect programs with more factors to offer greater execution time savings. Investigation proceeds by fixing the number of instructions per factor, by varying the number of factors.

All experiments in this section will involve the parameter combinations shown in table 8.7. We will repeat these experiments twice, once with caching active and once without. Note that all results are the collated averages obtained from 30 repetitions using a specific combination of parameters.

Table 8.7: CC PLGP/BS PLGP Experimental Parameter Combinations

Instructions per Factor	Factors	Ins	Factors	Ins	Factors	Ins
5	1	5	5	25	10	50
10	1	10	5	50	10	100
15	1	15	5	75	10	150
20	1	20	5	100	10	200
25	1	25	5	125	10	250
30	1	30	5	150	10	300

8.5 Results

This section presents the results of the experiments described in section 8.4 together with discussions.

8.5.1 PLGP (No Caching)

In this subsection we compare the efficiency of LGP and PLGP without caching. It is important to note that no optimizations are applied in this set of experiments. We are comparing the base running time of the two algorithms prior to any modifications. Experimental results are shown in Figure 8.3.

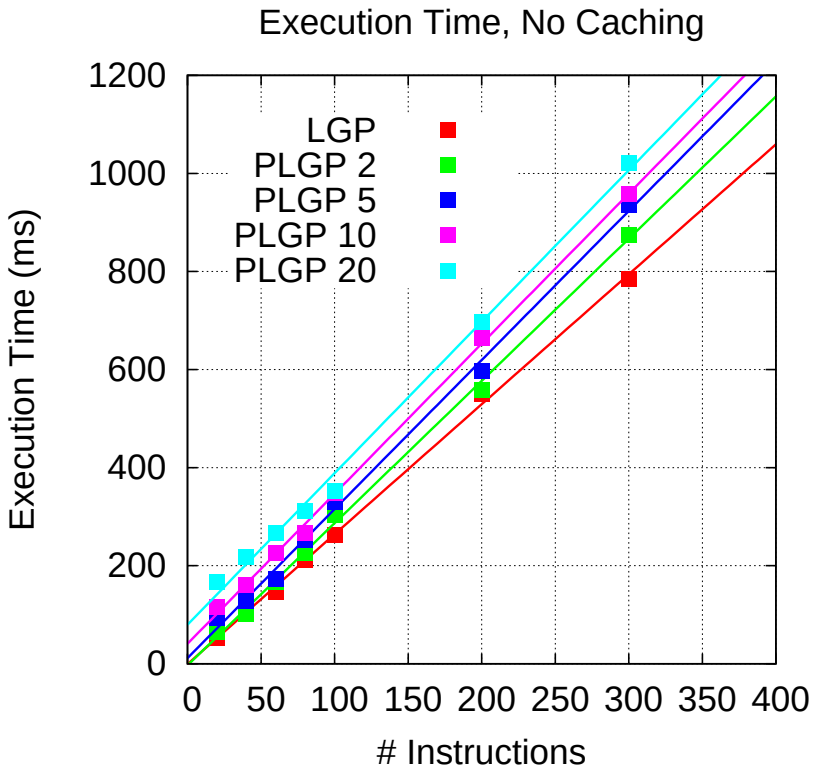


Figure 8.3: Execution time LGP vs. PLGP for various program lengths

Discussion

There are a number of important results present in figure 8.3.

- LGP Programs execute more rapidly than PLGP programs regardless

of the number of factors.

- PLGP program execution time is proportional to the number of factors: programs with more factors execute more slowly.

PLGP programs execute more slowly because there is an additional step to the execution process. To execute a *LGP program* we execute all instructions. To execute a *PLGP program* we execute all instructions *and calculate a vector sum consisting of one vector per factor*. Hence for any LGP program, a PLGP program with an equivalent number of instructions will always execute more slowly. It is important to note that the vector addition cost depends solely on the number of factors, and is *independent* of the number of instructions. We are required to perform the same vector addition irrespective of the number of instructions in the program. This is reflected by the fact that all of the trend lines in figure 8.3 are approximately *parallel*. Hence when the number of instructions is large, the difference in execution time between LGP and PLGP programs is small relative to the overall cost of program execution.

PLGP programs with more factors execute more slowly because the constant cost of the vector sum is larger. When executing a PLGP program we are required to sum all of the result vectors. Clearly the cost of this sum is proportional to the number of vectors we are summing. PLGP programs produce one result vector for each program factor. Hence programs with a larger number of factors have a larger number of result vectors, and hence a larger vector addition cost associated with program execution.

8.5.2 PLGP (Caching)

In this section we compare the efficiency of LGP and PLGP when using Difference Caching. Experimental results are shown in figure 8.4.

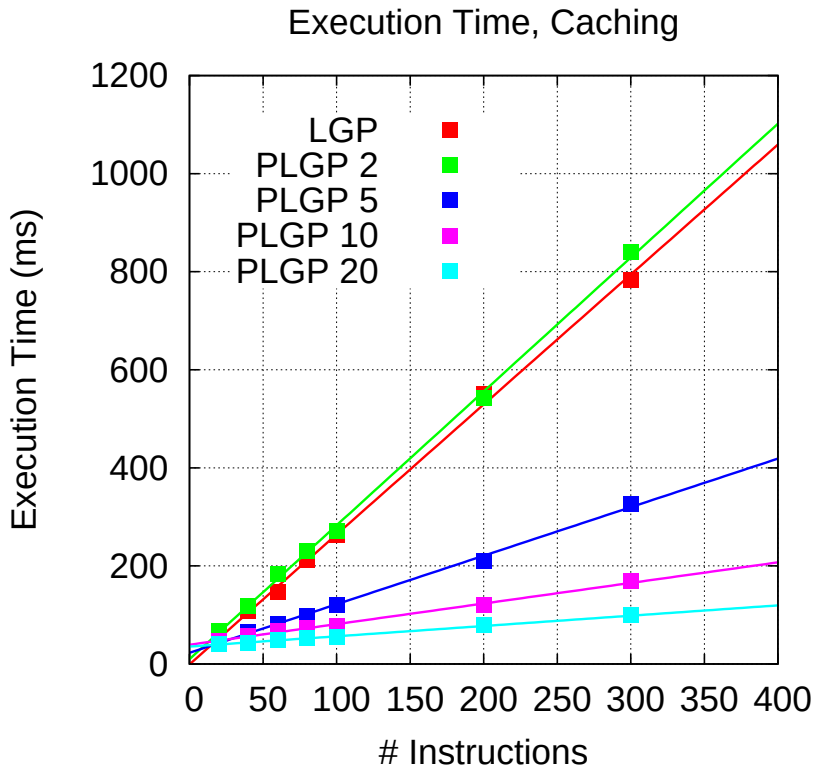


Figure 8.4: Execution time LGP vs. PLGP with caching for various program lengths

Discussion

There are a number of important results present in figure 8.4.

- Cached PLGP programs can be executed far more rapidly than equivalent size LGP programs.

- The execution time of cached PLGP programs depends heavily on the number of program factors. Programs with more factors execute more rapidly, while programs with fewer factors execute more slowly.

In table 8.8 we use the gradients of the trend lines in figure 8.4 to calculate precise values for the fractional difference in execution time. The values in this table show how many times more rapidly cached PLGP programs can be executed than equivalently sized LGP programs. Empirical results are compared to theoretically calculated values. For instance, table 8.8 shows that theoretically cached PLGP programs with 20 factors should execute 10 times as rapidly as equivalently sized LGP programs, however experimentally, such programs were found to execute 12.62 times as rapidly.

Table 8.8: Speedup Factor

	LGP	Cached PLGP			
Factors	1	2	5	10	20
Theoretical Improvement	1	1	2.5	5	10
Empirical Improvement	1	0.97	2.68	6.31	12.62

Table 8.8 shows that in all experiments we achieve execution time savings *at least as good* as the values predicted by our theoretical estimates. In some cases our empirical improvement actually exceeds the theoretical estimate. We believe this is due to intron labeling in PLGP programs. When executing PLGP programs we begin by labeling the structurally redundant instructions, as these should not be executed. In PLGP this labeling must be performed once for each program factor. However with difference caching only one of the two factors to be executed requires labeling, as the other was already labeled from the previous generation. This results in empirical efficiency which slightly exceeds our theoretical estimates.

8.5.3 CC PLGP/BS PLGP (No Caching)

In this subsection we empirically investigate the efficiency of CC PLGP/BS PLGP *without* caching. It is important to note that no optimizations are applied in this set of experiments. Experimental results are presented in

figure 8.5. We plot program execution time against the number of instructions per program factor for various program lengths. We use programs with 1, 5, and 10 factors. Note that programs with more factors will contain more instructions, since factor size is fixed. Therefore we expect programs with more factors to execute more slowly.

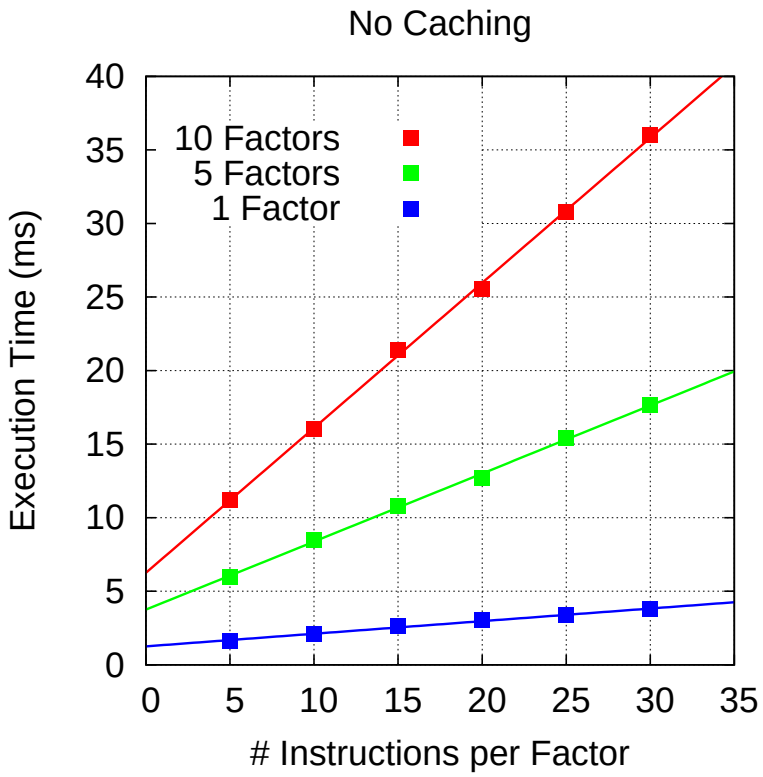


Figure 8.5: Execution time for CC PLGP *without* caching for various programs lengths

Discussion

There are a number of important results present in figure 8.5.

- Increasing the number of factors in each blueprint results in a proportional increase in the execution time.

Increasing the number of factors causes an increase in the execution time because more instructions must be executed. Each factor consists of a constant number of instructions, so blueprints with more factors have more instructions. Evaluation without caching requires every instruction in every blueprint to be executed. Therefore increasing the number of instructions in each blueprint results in a proportional increase in execution time.

8.5.4 CC PLGP/BS PLGP (Caching)

In this subsection we empirically investigate the efficiency of CC PLGP/BS PLGP *with* caching. Experimental results are presented in figure 8.6. We plot program execution time against the number of instructions per program factor for various program lengths. We use programs with 1, 5, and 10 factors. Note that programs with more factors will contain more instructions, since factor size is fixed. Therefore we expect programs with more factors to execute more slowly.

Discussion

There are a number of important results present in figure 8.6.

- Increasing the number of factors results in a negligible increase in the execution time.
- When each blueprint consists of a single factor, cached CC PLGP/BS PLGP executes more slowly than conventional CC PLGP.
- When each blueprint consists of multiple blueprints, cached CC PLGP/BS PLGP executes far more rapidly than conventional CC PLGP/BS PLGP.
- CC PLGP/BS PLGP systems with more factors per blueprint receive more benefit from caching.

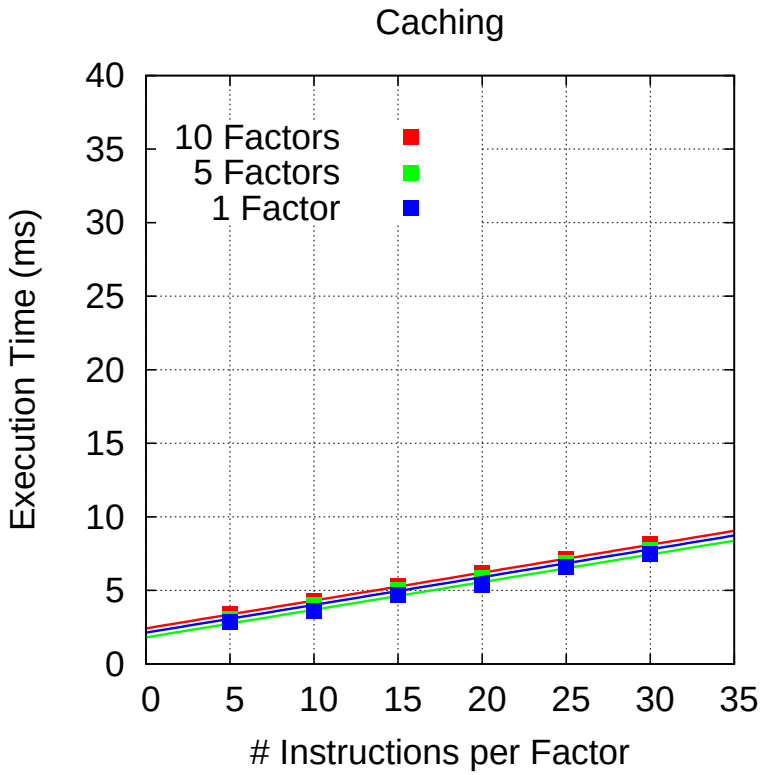


Figure 8.6: Execution time for CC PLGP *with* caching for various programs lengths

Increasing the number of factors causes a negligible increase in execution time because the number of instructions executed remains the same. Instructions are only present in factors, and all factor outputs are pre-calculated prior to blueprint evaluation. Hence the only cost associated with increasing the number of blueprints is an increase in the number of vectors which must be summed. Increasing the number of vectors summed results in a negligible increase in the overall cost. Hence increasing the number of factors causes a negligible increase in execution time.

When using a single factor caching is not cost effective. Evaluation without caching executes every factor in every blueprint. In contrast evaluation with caching executes every factor in every subpopulation. When a

single factor is used many factors will be present in some subpopulation, but will not participate in any blueprint. Hence when using a single factor evaluation without caching is more efficient than evaluation with caching. Note that this result is largely unimportant as single factor CC PLGP/BS PLGP systems should never be used.

When using multiple factors caching is extremely cost effective. Evaluation with caching executes every factor in every subpopulation. This means that each factor is only executed once. This is in contrast to evaluation without caching where each factor is executed once for each blueprint it participates in. When each blueprint consists of multiple factors, each factor will (on average) participate in multiple blueprints. Hence caching becomes cost effective.

CC PLGP/BS PLGP systems with more factors per blueprint receive more benefit from caching. Evaluation without caching executes every factor in every blueprint. If the number of factors per blueprint increases, then the cost of execution also increases. In contrast the cost of factors executed is independent of the number of factors per blueprint. Hence the savings associated with caching is proportional to the number of factors in each blueprint.

8.6 Chapter Summary

The goal of this chapter was to develop caching techniques for PLGP, CC PLGP, and BS PLGP which significantly decrease program execution time. This goal was successfully achieved by the development of two new caching algorithms.

We developed a new caching algorithm for PLGP which exploits the parallel architecture of PLGP programs. PLGP programs are composed of n independent factors. PLGP program execution consists of executing these factors and summing the resulting vectors. By caching both the vector sum and program factor which underwent execution we can cheaply

execute program offspring in the subsequent generation. By doing this for all programs in generation n the execution time of generation $n + 1$ is hugely decreased.

We presented theoretical results which show that the execution time of cached PLGP programs depends only on the size of program factors, and is independent of overall program size. Hence PLGP programs with a large number of factors can be executed extremely rapidly.

The empirical experimental results confirm the effectiveness of our PLGP caching algorithm. PLGP systems demonstrated a significant decrease in program execution time when caching was applied. In addition our empirical results matched, or slightly exceeded theoretical predictions. This strongly supports the validity of the equations in section 8.2.3.

We developed a new caching algorithm for both CC PLGP and BS PLGP which exploits the dual population representation of the SANE style architecture. Both CC PLGP and BS PLGP use a representation which distinguishes blueprints from factors. Conventional evaluation executes each factor when required to execute a blueprint. Caching executes all factors prior to the execution of any blueprints, and caches the results. These cached results are used to rapidly execute all blueprints.

We presented theoretical results which show that the evaluation time of CC PLGP/BS PLGP is independent of the number of blueprints. The major evaluation cost associated with CC PLGP/BS PLGP is the time required for factor execution. The evaluation cost without caching is proportional to the number of factors in each blueprint. In contrast the evaluation cost with caching is constant, regardless of the number of blueprints. Hence CC PLGP/BS PLGP systems with many blueprints stand to benefit greatly from caching.

The empirical experimental results confirm the effectiveness of our CC PLGP/BS PLGP caching algorithm. CC PLGP/BS PLGP systems demonstrated a significant decrease in program execution time when caching was applied. In addition our empirical results matched, or slightly exceeded

theoretical predictions. This strongly supports the validity of the equations in section 8.3.1.

Chapter 9

Conclusions

In this chapter we present the conclusions of the work covered in this thesis, with respect to the research questions posed in chapter 1. We then outline some possible future work directions.

9.1 Conclusions

The overall goal of this thesis was to improve the effectiveness and increase the efficiency of LGP. To achieve this we have focused on developing, extending, and optimizing a new LGP architecture which minimises the number of instruction dependencies. The overall goal has been successfully achieved through our research in this area. This research has led to the PLGP architecture; the CC PLGP, Hybrid PLGP and BS PLGP algorithms; and caching techniques for LPG, PLGP, CC PLGP and BS PLGP. These methods constitute a new approach to LGP with improved performance and increased efficiency. In the remainder of this section we summarise the conclusions stemming from this work.

9.1.1 PLGP

We developed a new LGP architecture with fewer instruction dependencies called PLGP. We showed PLGP significantly outperforms conventional LGP on a range of classification problems.

PLGP is a natural extension of the LGP architecture, where each program consists of n instruction sequences called factors. Each factor is independently executed to produce n results vectors, which are summed to produce the final program output. PLGP programs have fewer instruction dependencies than equally sized LGP programs, as there are no inter-factor dependencies.

We compared the effectiveness of PLGP to that of LGP on three classification tasks for a range of program sizes.

We found that PLGP and LGP have comparable performance when small programs are used. This is to be expected as small programs are not well suited to factorization. We also found that PLGP significantly outperformed LGP on all three problems when using large programs. Large LGP programs have many dependencies and are easily disrupted during evolution, whereas large PLGP programs are more robust due to their modular structure.

We also found that using the PLGP architecture leads to speciation. PLGP has implicit subpopulations because recombination can only occur between factors with the same position, and factor position cannot change. Individuals within each implicit subpopulation specialize to solve some aspect of the problem resulting in a homogenous subpopulation. The homogenous subpopulations resulting from speciation contain highly compatible individuals, increasing the likelihood of crossover resulting in viable offspring. Speciation is important, because it helps us to understand why the PLGP architecture is so successful.

9.1.2 CC PLGP

We developed a new LGP algorithm by applying the concept of Cooperative Coevolution to the PLGP architecture. We showed that CC PLGP is significantly more effective than PLGP during the first stage of evolution, particularly for large PLGP programs with many factors. We also showed that CC PLGP has significant performance disadvantages *during the later stages of evolution. Finally we combined PLGP and CC PLGP into a hybrid algorithm possessing the strengths of both algorithms.*

CC PLGP is an algorithm which applies the ideas of cooperative coevolution to the PLGP architecture. CC PLPG maintains n subpopulations of factors, together with a single population of fixed size factor combinations known as blueprints. Blueprints serve as a mechanism for remembering and evolving good factor combinations. The strength of CC PLGP is its ability to simultaneously optimize all program factors.

We compared the performance of CC PLGP with that of PLGP on three classification tasks for a range of program sizes. We found that PLGP significantly outperforms CC PLGP when small programs are used. This is to be expected as small programs have few factors, so there is little advantage in optimizing all program factors in parallel. We also found that CC PLGP significantly outperforms PLGP during the first stage of evolution when large programs are used. Large programs have many factors, and optimizing these factors in series is time consuming, leading to slow algorithm convergence. Using CC PLGP we can optimize all program factors in parallel allowing rapid convergence to a population of high quality solutions. PLGP still significantly outperforms CC PLGP during the later stages of evolution, as PLGP excels at fine tuning existing high quality solutions.

We combined PLGP and CC PLGP into a single hybrid algorithm possessing the strengths of both approaches. CC PLGP has the advantage of allowing rapid initial population convergence, while PLGP excels at fine tuning existing high fitness solutions. Therefore Hybrid PLGP uses

CC PLGP until population convergence occurs, after which a transition to PLGP is effected. We compared the performance of Hybrid PLGP with that of CC PLGP and PLGP on three classification tasks. We found that Hybrid PLGP is significantly more effective than both CC PLPG and PLGP in all cases.

9.1.3 BS PLGP

We developed an extension of the CC PLGP algorithm called BS PLGP which uses PSO to search a structured search space for high fitness blueprints. We showed that BS PLGP is significantly more effective than LGP, PLGP, and CC PLGP on a range of classification problems.

BS PLGP is a logical extension of the CC PLGP algorithm which uses the current factors to search for high fitness blueprints. The effectiveness of the CC PLGP algorithm is limited by the blueprint quality. Bad blueprints give rise to bad factor fitness estimates, which in turn lead to the wrong factors being selected for reproduction. Bad blueprints often occur because the blueprint population used in CC PLGP provides a dated guess at which blueprints are good. BS PLGP solves this problem by constructing a structured search space of all possible blueprints, and using PSO to effectively and efficiently search this space for good blueprints.

We compared the performance of BS PLGP to that of CC PLGP and PLGP on three classification tasks for a range of program sizes.

We found that BS PLGP significantly outperforms both CC PLGP and PLGP. This performance improvement was caused by an increase in blueprint quality. Higher quality blueprints lead to better factor fitness estimates which in turn lead to better factor selection. In addition, high quality blueprints are the final goal of the system, as they represent high quality solutions to the problem. CC PLGP struggles to find high quality solutions, as maintaining a blueprint population provides only out of date estimates. In contrast, BS PLGP actively searches for high quality solutions

using the current population of factors.

9.1.4 Caching

We developed caching techniques for LGP, PLGP, CC PLGP, and BS PLGP based on inter-generational caching. We established theoretical results which we used to predict the optimal cache parameter configuration together with an estimate of the execution time savings. Finally we obtained empirical results which confirmed our theoretical predictions.

Algorithms which use the PLGP architecture are naturally suited to caching. Conventional LGP programs consist of a single instruction sequence with many instruction dependencies which interfere with caching. In contrast PLGP programs consist of n independently executed factors perfectly suited to caching. We exploited the parallel structure of PLGP to develop caching algorithms for PLGP, CC PLGP and BS PLGP.

We also developed a caching algorithm for LGP called execution trace caching. Caching for LGP stores the state of the registers every n instructions. This cache is used to expedite the execution of modified program offspring by allowing execution to begin part way through the offspring programs. We undertook a theoretical analysis of the execution trace caching algorithm. This analysis showed that the optimal number of cache points is proportional to the square root of the number of instructions. Furthermore we showed that caching is cost effective when the number of instructions is at least 18 times larger than the number of registers. We also showed that at best caching can decrease program execution time by 50%. As a final step we obtained empirical results which support our theoretical conclusions.

We developed a caching algorithm for PLGP. Caching for PLGP stores the final program output together with a copy of the modified program factor. This cache is used to expedite the execution of modified program offspring by limiting execution to the single modified program factor. We

undertook a theoretical analysis of caching for PLGP. This analysis showed that our PLGP caching algorithm has the potential to decrease execution time by more than an order of magnitude. Furthermore this savings is proportional to the number of program factors. As a final step we obtained empirical results which support our theoretical conclusions.

We developed a caching algorithm for CC PLGP and BS PLGP. Caching for CC PLGP and BS PLGP caches the output of all factors prior to executing any blueprints. This cache is used to expedite the execution of all blueprints by avoiding factor execution during blueprint evaluation. We undertook a theoretical analysis of caching for CC PLGP and BS PLGP. This analysis showed that our caching algorithm has the potential to decrease execution time by more than an order of magnitude. Furthermore this saving is proportional to the number of blueprints, and the size of the program factors. As a final step we obtained empirical results which support our theoretical conclusions.

9.2 Future Work

In this section we outline possible future work directions inspired by the work presented in this thesis.

9.2.1 PLGP

- **Weighted PLGP**
  The output of each PLGP program is calculated by summing the output of all program factors. One obvious approach to further improve the performance of PLGP is to replace this sum with a *weighted sum*. Weighted PLGP would associate a single real valued number called the weight with each factor. These weights could be optimized using techniques such as least squares regression to deterministically improve performance. Furthermore, this optimization could occur

either during evolution, or as fine tuning for the final solution after the original algorithm has terminated.

- **Variable length factors**
  One avenue not yet explored involves PLGP programs consisting of factors with different and variable lengths. In the present approach all factors are of equal and fixed length. This simplifies many aspects of both the implementation, and the calculations, however it is unknown whether this approach gives the best performance. Many problems consist of a number of subproblems, with good solutions consisting of good solutions to each subproblem. PLGP programs exploit this by enforcing explicit subpopulations, and allowing solutions to subproblems to evolve within these subpopulations. It is natural to assume that the complexity of subproblems will vary, and that the solution length will also vary. Hence one possible future work direction is to relax the constraint on factor sizes, and either explicitly include factors with different lengths, or allow factor lengths to vary during evolution.

9.2.2 CC PLGP

- **Alternating Hybrid PLGP**
  A possible future work direction is to develop a hybrid algorithm which alternates between PLGP and CC PLGP. The current hybrid PLGP algorithm uses CC PLGP for the first n generations, and uses PLGP for all subsequent generations. Another possible hybrid algorithm is one which switches between PLGP and CC PLGP every m generations. Such an algorithm could exploit the strengths of both PLGP and CC PLGP throughout the entire duration of the algorithm.

9.2.3 BS PLGP

- **Alternative PSO Variants**
  There are PSO variants in the literature which differ in a variety of ways from the PSO variant used in this work. For example particles can be linked together in a graph for the purposes of defining local optima. Different graphs have been shown to have distinct advantages on certain problems, and it is likely that there exists a PSO particle graph we have not explored which gives performance advantages.

- **Alternative Search Algorithms**
  Another interesting question is whether or not PSO is the best approach for searching the space of possible blueprints. PSO can be theoretically motivated, and empirical testing has demonstrated it gives good results, however it may well be possible to achieve superior performance by using a different search technique. One technique which might be particularly well suited is genetic algorithms.

- **Alternative Distance Metrics**
  Is it possible to find a better distance metric than the Hamming distance. The success of this algorithm depends heavily on the assumption that the Hamming distance is a good choice for approximating the "difference" between two programs. The major problem with the Hamming distance is that it does not take into account the similarity between program components. For example, two programs which differ in the value of a positive constant are far more likely to have similar output than two programs which differ in an operator. Therefore one direction for future work is to explore alternative distance metrics for comparing programs.

- **Alternative Traveling Salesman Algorithm**
  The PSO algorithm used by BS PLGP relies on spacial locality in the

blueprint search space to find good solutions. If we can improve the search space by finding a better factor ordering, then we can improve algorithm performance. Finding a good factor ordering is equivalent to solving the traveling salesman problem. At present, we find an approximate solution using the nearest neighbor algorithm. While this algorithm is attractive in its simplicity, the approximation provided by this algorithm may differ substantially from what is optimal. There are a number of algorithms which exist in the literature which provide better approximate solutions than the nearest neighbor algorithm. Therefore one possible future work direction is to integrate these algorithms with the BS PLGP method.

9.2.4 Caching

- **Intra-Generational Caching**
  We have focused on inter-generational caching: caching between programs in different generations. Future work may focus on the alternative and complementary technique of caching between programs within the same generation. Many of the execution traces are very similar, indicating we are often computing the same function multiple times and hence performing unnecessary work. By caching parts of execution traces and retrieving them when necessary, we should be able to further decrease the evaluation time of LGP programs. This form of caching would be complementary to the existing method of execution trace caching, hence by combining the two techniques we would hope to arrive at a composite caching technique which outperforms either of its components.

- **Caching using Hashing**
  Many caching techniques exploit hashing to significantly reduce the overhead cost associated with caching. In this thesis we have developed a hashing technique for LGP and PLGP programs. One possi-

ble future work direction is to use a hashing algorithm to develop an improved caching algorithm.

- **Reducing the cost of Caching**
  Currently there is a significant cost, both in terms of memory and time, associated with caching execution traces. While the benefits of caching already outweigh the cost, decreasing the cost further will result in greater computational savings as well as making caching cost effective for a greater range of problems. To this end, we aim to investigate how the use of more innovative caching techniques can decrease the cost of caching without negatively impacting on the benefits already obtained.

9.2.5 General

- **More Data Sets** It is important that we evaluate the new algorithms presented in this thesis on additional data sets. The algorithms presented in this thesis are evaluated on three tasks from the problem domain of object classification. This an important and challenging problem domain, and one which serves us well for comparing algorithm performance. However in order to confirm that the results presented in this thesis generalize to a wide range of problem domains it is vital that we evaluate these new developments on a wider variety of data sets.

Bibliography

[1] ALBERT, J. H. Calculated bets: Computers, gambling, and mathematical modeling to win. *The American Statistician vol 56* (November 2002), 329–330.

[2] ALPAYDIN, E. *Introduction to Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2004.

[3] ANGELINE, P. J. Subtree crossover: Building block engine or macromutation? In *Proceedings of the Second Annual Conference on Genetic Programming* (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 Publishers, pp. 9–17.

[4] ARNAU, G. M., MANRIQUE, D., RÍOS, J., AND PATÓN, R. A. Initialization Method for Grammar-Guided Genetic Programming. *Knowledge-Based Systems 20* (2007), 127–133.

[5] ARYA, S., MOUNT, D., NETANYAHU, N., SILVERMAN, R., AND WU, A. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *Journal of the ACM (JACM) 45*, 6 (1998), 891–923.

[6] BÄCK, T. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, Oxford, UK, 1996.

[7] BANZHAF, W. Genetic programming for pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms* (San Francisco, CA, USA, 1993), Morgan Kaufmann Publishers, pp. 628–632.

[8] 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 Publishers, 1998.

[9] BEYER, H. G., AND SCHWEFEL, H. P. Evolution strategies: A comprehensive introduction. *Natural Computing: an International Journal 1*, 1 (2002), 3–52.

[10] BISHOP, C. M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Secaucus, NJ, USA, 2006.

[11] BRAMEIER, M., AND BANZHAF, W. Effective linear genetic programming. Tech. rep., Neural Networks in Medical Data Mining IEEE Transactions on Evolutionary Computation, 2001.

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

[13] BULL, L., AND KOVACS, T. *Foundations of Learning Classifier Systems*. Studies in fuzziness and soft computing. Springer-Verlag, 2005.

[14] BUSCH, J., ZIEGLER, J., BANZHAF, W., ROSS, A., SAWITZKI, D., AND AUE, C. Automatic generation of control programs for walking robots using genetic programming. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002* (Kinsale, Ireland, 3-5 April 2002), J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, Eds., vol. 2278 of *LNCS*, Springer-Verlag, pp. 258–267.

[15] CASTRO, L., AND TIMMIS, J. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer-Verlag, 2002.

[16] CHITTY, D. M. A data parallel approach to genetic programming using programmable graphics hardware. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation (GECCO)* (2007), ACM, pp. 1566–1573.

[17] DEZA, M. M., AND DEZA, E. *Encyclopedia of Distances*. Springer-Verlag, 2009.

[18] D'HAESELEER, P. Context preserving crossover in genetic programming. In *International Conference on Evolutionary Computation* (1994), pp. 256–261.

[19] DORIGO, M., AND STUTZLE, T. *Ant Colony Optimization*. Bradford Books. MIT Press, 2004.

[20] DOWNEY, C., AND ZHANG, M. Multiclass object classification for computer vision using linear genetic programming. In *Proceedings of the 24th International Conference on Image and Vision Computing New Zealand (IVCNZ)* (November 2009), pp. 73 –78.

[21] DOWNEY, C., ZHANG, M., AND BROWNE, W. N. New crossover operators in linear genetic programming for multiclass object classification. In *Proceedings of the 12th Annual Conference on Genetic and Evolutionary Computation (GECCO)* (2010), ACM, pp. 885–892.

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

[23] EKLUND, S. E. A massively parallel GP engine in VLSI. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC)* (2002), IEEE Press, pp. 629–633.

[24] ESPEJO, P. G., VENTURA, S., AND HERRERA, F. A survey on the application of genetic programming to classification. *IEEE Trans. Sys. Man Cyber Part C 40* (March 2010), 121–144.

[25] FOGEL, D. B. *Evolutionary Computation: The Fossil Record*, 1st ed. Wiley-IEEE Press, 1998.

[26] FOGELBERG, C., AND ZHANG, M. Linear genetic programming for multi-class object classification. In *Proceedings of the 18th Australian Joint Conference on Artificial Intelligence* (2005), vol. 3809 of LNCS, pp. 369–379.

[27] FRANK, A., AND ASUNCION, A. UCI machine learning repository, 2010.

[28] GALVAN LOPEZ, E., POLI, R., AND COELLO COELLO, C. A. Reusing code in genetic programming. In *Proceedings of the 7th European Conference on Genetic Programming (EuroGP)* (2004), vol. 3003 of *LNCS*, Springer-Verlag, pp. 359–368.

[29] GATHERCOLE, C., AND ROSS, P. Dynamic training subset selection for supervised learning in genetic programming. In *Parallel Problem Solving from Nature III* (1994), Springer-Verlag, pp. 312–321.

[30] GIACOBINI, M., TOMASSINI, M., AND VANNESCHI, L. Limiting the number of fitness cases in genetic programming using statistics. In *Proceedings of the 7th International Conference on Parallel Problem Solving from Nature* (London, UK, 2002), PPSN VII, Springer-Verlag, pp. 371–380.

[31] GOMEZ, F., AND MIIKKULAINEN, R. 2-d pole balancing with recurrent evolutionary networks. In *Proceeding of the International Conference on Artificial Neural Networks (ICANN)* (1998), Springer-Verlag, pp. 425–430.

[32] GOMEZ, F., AND MIKKULAINEN, R. Incremental evolution of complex general behavior. *Adapt. Behav. 5* (January 1997), 317–342.

[33] GOMEZ, F. J., AND MIIKKULAINEN, R. Solving non-markovian control tasks with neuroevolution. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI)* (1999).

[34] GREFENSTETTE, J. Rank-based selection. In *Handbook of Evolutionary Computation*, T. Bäck, D. B. Fogel, and Z. Michalewicz, Eds. Institute of Physics Publishing and Oxford University Press, Bristol, New York, 1997, pp. C2.4:1–6.

[35] GUTIN, G., YEO, A., AND ZVEROVICH, A. Traveling salesman should not be greedy: Domination analysis of greedy-type heuristics for the TSP. *Discrete Appl. Math 117* (2002), 81–86.

[36] HAMMING, R. W. Error detecting and error correcting codes. *Bell System Technical Journal 29*, 2 (1950), 147–160.

[37] HANDLEY, S. On the use of directed acyclic graph to represent a population of computer programs. In *International Conference on Evolutionary Computation (ICEC)* (1994), pp. 154–159.

[38] HEITKTTER, J., AND BEASLEY, D. The hitch-hiker's guide to evolutionary computation: A list of frequently asked questions (faq), 2000.

[39] HINTON, G., AND SEJNOWSKI, T. *Unsupervised Learning: Foundations of Neural Computation*. Computational neuroscience. MIT Press, 1999.

[40] HOLLAND, J. H. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor : University of Michigan Press, 1975.

[41] JOHNSON, D. S., AND MCGEOCH, L. A. *The Traveling Salesman Problem: A Case Study in Local Optimization*. John Wiley and Sons, London, 1997, pp. 215–310.

[42] KANAS, S., AND LECKO, A. Metric spaces. *Formalized Mathematics 1* (1990).

[43] KEIJZER, M. Alternatives in subtree caching for genetic programming. In *Procedings of Genetic Programming 7th European Conference (EuroGP)* (2004), Lecture Notes in Computer Science, pp. 328–337.

[44] KENNEDY, J., AND EBERHART, R. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN)* (Aug. 1995), vol. 4, pp. 1942–1948.

[45] KENNEDY, J., AND EBERHART, R. C. *Swarm Intelligence*. Morgan Kaufmann Publishers, 2001.

[46] KOZA, J. R. *On the Programming of Computers by Means of Natural Selection*. A Bradford book. MIT Press, 1996.

[47] KOZA, J. R., KEANE, M. A., STREETER, M. J., MYDLOWEC, M., YU, J., AND LANZA, G. Genetic programming IV: Routine human-competitive machine intelligence. *Genetic Programming and Evolvable Machines 6*, 2 (2005), 231–233.

[48] KRAUSE, E. *Taxicab Geometry: An Adventure in Non-Euclidean Geometry*. Dover Publications, 1986.

[49] KRAWIEC, K., AND BHANU, B. Visual learning by evolutionary and coevolutionary feature synthesis. *IEEE Transactions on Evolutionary Computation 11*, 5 (Oct. 2007), 635–650.

[50] LANGDON, W. B. Pareto, population partitioning, price and genetic programming. Research Note RN/95/29, University College London, Gower Street, London WC1E 6BT, UK, 1995.

[51] LANGDON, W. B., AND POLI, R. *Foundations of Genetic Programming*, 1 ed. Springer-Verlag, Mar. 2002.

[52] LOVEARD, T. *Genetic Programming for Classification Learning Problems*. PhD thesis, RMIT University, School of Computer Science and Information Technology, 2003.

[53] LOVEARD, T., AND CIESIELSKI, V. Representing classification problems in genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation* (May 2001), vol. 2.

[54] LU, H., AND YEN, G. Dynamic population size in multiobjective evolutionary algorithms. *Computational Intelligence, Proceedings of the World on Congress on Computational Intelligence 2* (2002), 1648–1653.

[55] MACCLELLAND, J., AND RUMELHART, D. *Parallel Distributed Processing: Explorations in The Microstructure of Cognition. Psychological and Biological Models*. No. v. 2 in Bradford Book. The MIT Press.

[56] MACKAY, D. J. C. *Information Theory, Inference and Learning Algorithms*. Cambridge University Press, 2003.

[57] MILLER, B. L., GOLDBERG, D. E., AND GOLDBERG, D. E. Genetic algorithms, tournament selection, and the effects of noise. *Complex Systems 9* (1995), 193–212.

[58] MITCHELL, M. *An Introduction to Genetic Algorithms*. Complex adaptive systems. MIT Press, 1998.

[59] MITCHELL, T. M. *Machine learning*. McGraw Hill, New York, 1997.

[60] MONTANA, D. J. Strongly typed genetic programming. *Evolutionary Computation 3* (1994), 199–230.

[61] MORIARTY, D. E. *Symbiotic Evolution of Neural Networks in Sequential Decision Tasks*. PhD thesis, Austin, TX, USA, 1998. UMI Order No. GAX98-02963.

[62] MORIARTY, D. E., AND MIIKKULAINEN, R. Forming neural networks through efficient and adaptive coevolution. *Evolutionary Computation 5*, 4 (1997), 373–399.

[63] MORIARTY, D. E., MIIKKULAINEN, R., AND KAELBLING, P. Efficient reinforcement learning through symbiotic evolution. In *Machine Learning* (1996), pp. 11–32.

[64] NEILSON, T. P., AND PERKIS, T. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence* (1994), IEEE Press, pp. 148–153.

[65] NESHATIAN, K., AND ZHANG, M. Genetic programming and class-wise orthogonal transformation for dimension reduction in classification problems. In *Proceedings of the 11th European conference on Genetic Programming (EuroGP)* (Berlin, Heidelberg, 2008), Springer-Verlag, pp. 242–253.

[66] NORDIN, P. A compiling genetic programming system that directly manipulates the machine code. In *Advances in genetic programming* (Cambridge, MA, USA, 1994), MIT Press, pp. 311–331.

[67] OLAGUE CABALLERO, G., ROMERO, E., TRUJILLO, L., AND BHANU, B. Multiclass object recognition based on texture linear genetic programming. In *Applications of Evolutionary Computing (EvoWorkshops)* (11-13 Apr. 2007), vol. 4448 of *LNCS*, pp. 291–300.

[68] OLAGUEA, G., CAGNONI, S., AND LUTTON, E. (eds.) special issue on evolutionary computer vision and image understanding, pattern recognition letters. **27**(11), 2006.

[69] POLI, R. Genetic programming for image analysis. In *Proceedings of the First Annual Conference on Genetic Programming (GECCO)* (Cambridge, MA, USA, 1996), MIT Press, pp. 363–368.

[70] POLI, R. Discovery of symbolic, neuro-symbolic and neural networks with parallel distributed genetic programming. In *Proceedings of the 3rd International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)* (1997).

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

[72] POTTER, M. A., AND JONG, K. A. D. A cooperative coevolutionary approach to function optimization. Springer-Verlag, pp. 249–257.

[73] POTTER, M. A., AND JONG, K. A. D. Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation 8* (2000), 1–29.

[74] QUINLAN, J. *C4.5: Programs for Machine Learning*. Morgan Kaufmann series in machine learning. Morgan Kaufmann Publishers, 1993.

[75] ROBERTS, M. E. The effectiveness of cost based subtree caching mechanisms in typed genetic programming for image segmentation. In *Proceedings of the 2003 International Conference on Applications of Evolutionary Computing* (Berlin, Heidelberg, 2003), EvoWorkshops, Springer-Verlag, pp. 444–454.

[76] ROBERTS, M. E. The effectiveness of cost based subtree caching mechanisms in typed genetic programming for image segmentation. In *Proceedings of the 2003 International Conference on Applications of Evolutionary Computation (EvoApplications)* (Berlin, Heidelberg, 2003), EvoWorkshops, Springer-Verlag, pp. 444–454.

[77] ROBILLIARD, D., MARION-POTY, V., AND FONLUPT, C. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines 10* (December 2009), 447–471.

[78] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. *Learning Internal Representations by Error Propagation*. MIT Press, Cambridge, MA, USA, 1986, pp. 318–362.

[79] SCHLIMMER, J. C., AND LANGLEY, P. *Paradigms for Machine Learning*. John Wiley and Sons, 1991.

[80] SMART, W., AND ZHANG, M. Probability based genetic programming for multiclass object classification. In *Proceedings of the 8th Pacific Rim International Conference on Artificial Intelligence (PRICAI)* (2004), vol. 3157 of *LNCS*, Springer-Verlag, pp. 251–261.

[81] SMART, W. R., AND ZHANG, M. Classification strategies for image classification in genetic programming. In *Proceedings of the International Conference on Image and Vision Computing NZ (IVCNZ)* (Nov. 2003), Massey University, pp. 402–407.

[82] SMITH, J. Modelling GAs with self adaptive mutation rates. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (2001), Morgan Kaufmann Publishers, pp. 599–606.

[83] SPEARS, W. M., JONG, K. A. D., BÄCK, T., FOGEL, D. B., AND GARIS, H. D. An overview of evolutionary computation. In *Proceedings of the European Conference on Machine Learning* (London, UK, 1993), Springer-Verlag, pp. 442–459.

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

[85] SUTTON, R. S., AND BARTO, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, Mar. 1998.

[86] TACKETT, W. A. Genetic programming for feature discovery and image discrimination. In *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA)* (1993), Morgan Kaufmann Publishers, pp. 303–309.

[87] TETKO, I. V., LIVINGSTONE, D. J., AND LUIK, A. I. Neural network studies, 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences 35*, 5 (1995), 826–833.

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

[89] WONG, M. L., AND LEUNG, K. S. Evolutionary program induction directed by logic grammars. *Evolutionary Computation 5*, 2 (1997), 143–180.

[90] WONG, P. Removing redundancy and reducing fitness evaluation costs in genetic programming. Master's thesis, School of Mathematics and Computer Science, Victoria University of Wellington, 2008.

[91] WONG, P., AND ZHANG, M. Scheme: Caching subtrees in genetic programming. In *IEEE Congress on Evolutionary Computation* (2008), IEEE, pp. 2678–2685.

[92] YANG, Z., TANG, K., AND YAO, X. Large scale evolutionary optimization using cooperative coevolution. *Inf. Sci. 178* (August 2008), 2985–2999.

[93] YAO, X. Evolving artificial neural networks. *Proceedings of the IEEE 87*, 9 (1999), 1423–1447.

[94] ZHANG, B.-T., AND CHO, D.-Y. Genetic programming with active data selection. In *Selected papers from the Second Asia-Pacific Conference*

on Simulated Evolution and Learning (SEAL) (1999), Springer-Verlag, pp. 146–153.

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

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

[97] ZHANG, M., AND FOGELBERG, C. G. Genetic programming for image recognition: An LGP approach. In *Proceedings of the 2007 EvoWorkshops on Applications of Evolutionary Computing* (Berlin, Heidelberg, 2007), Springer-Verlag, pp. 340–350.

[98] ZHANG, M., GAO, X., AND LOU, W. A new crossover operator in genetic programming for object classification. *IEEE Transactions on Systems, Man and Cybernetics, Part B 37*, 5 (Oct. 2007), 1332–1343.

[99] ZHANG, M., AND SMART, W. Multiclass object classification using genetic programming. In *Applications of Evolutionary Computing, EvoWorkshops* (2004), vol. 3005 of *LNCS*, Springer-Verlag, pp. 369–378.

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