Genetic Programming with Symbiotic Functions

Ian Glover

MSc in Artificial Intelligence
Division of Informatics
University of Edinburgh
2000

Abstract

This dissertation presents and investigates a new technique for using sub-routines in genetic programming. The technique differs from previous ideas by having separate populations for programs and functions and evolving the members of each concurrently. It was hoped that this technique will solve problems as successfully as existing methods with the additional advantage of having less structure determined in advance by the experimenter. It was also hoped that the technique would be superior to existing methods at finding generally useful sub-routines.

The initial algorithm was found to be better than genetic programming techniques that don't use sub-routines but inferior to Koza's automatically defined functions (ADFs). The reasons behind the performance are analysed and three refinements are investigated. One of these improved the performance but remained inferior to ADFs. The system was found to discover generally useful sub-routines on successful runs.

Acknowledgements

I would first of all like to thank my supervisor, Dr. Chris Mellish, for his help and advice in the research documented here. I'd also like to thank Julie Sadler for her patience as a proof reader and listener to my random ideas, and Harry Potter and friends for keeping me sane.

I also thank the EPSRC who provided financial support during my year of study.

Contents

1 Introduction **1**
- 1.1 Automatic Programming and Genetic Programming 1
- 1.2 Sub-routines in Programming 3
- 1.3 Aims 4

2 Literature Review **6**
- 2.1 Automatically Defined Functions 6
- 2.2 Encapsulation 9
- 2.3 Module Acquisition 10
- 2.4 Adaptive Representation 11
- 2.5 Automatically Defined Macros 13
- 2.6 Architecture Altering Operations 14
- 2.7 SANE 16

3 Method **18**
- 3.1 Project Aims 18
- 3.2 Basic Algorithm 20
- 3.3 Experiments 22
- 3.4 Even-4-Parity Problem 23
- 3.5 Expectations 24

4 Implementation **27**
- 4.1 Adaptation versus Innovation 27
- 4.2 Programming Language 29
- 4.3 Design 29
- 4.4 Evaluation of Evolved Programs 31
  - 4.4.1 Using the Lisp Evaluator 31
  - 4.4.2 Hand Executing 32
  - 4.4.3 Compromise Solution 32
- 4.5 Experiment Implementation 33
- 4.6 Optimization 34

4.7 Testing 36

5 Experimental Results **38**
5.1 Control Experiments 39
5.2 Basic Algorithm 40
5.3 Linkage Preserving Algorithms 43
5.4 Monotonic Algorithm 46
5.5 Mean Fitness 47

6 Discussion **49**
6.1 Performance 49
6.2 Structure Discovery 51
6.2.1 Number of Functions 51
6.2.2 Generally Useful Functions 52
6.3 Future Work 53

7 Conclusion **55**

Appendices **56**

A Glossary **56**

B The Interpreter **58**

Bibliography **59**

List of Figures

1.1 A simple program represented as a tree structure. This program can be written in lisp as `(and (or(not d1) (not d2)) (or (not d3) (not d4)))`. 3

2.1 The program of figure 1.1 represented as a genome with a single ADF. 7

3.1 The representation of the program from figure 1.1 represented by the technique being investigated in this problem. 19

3.2 The basic algorithm used for this thesis. 21

3.3 A solution to the even-4-parity problem using AND, OR and NOT with no sub-functions. 23

3.4 A solution to the even-4-parity problem using the sub-functions. 24

4.1 The main loop (in lisp-like pseudocode) of the GP with symbiotic functions. 30

4.2 The algorithm used for the minimal evaluation of programs and functions for SEFs. 36

5.1 Graph of the cumulative probability of finding a correct solution to the test problem against generation for ADFs and standard GP. 39

5.2 Graph of the cumulative probability of finding a correct solution to the test problem for ADFs, the first SEF algorithm and SGP. 40
5.3 Graph of the fitness of the best individual of each generation for a typical and a converging run for the original SEF algorithm plotted against generation. 42
5.4 Graph of cumulative probability of finding a solution against generation for ADFs, SGP, the original SEF algorithm and the two linkage preserving SEF algorithms. 45
5.5 Graph of the mean fitness of the best individual in the population (averaged over the trials carried out) at each generation for standard GP, ADFs and three of the SEF algorithms (original, cross mutation and monotone). 48

List of Tables

3.1 The settings for the even-4-parity problem. Adapted from Kinnear (1994). 25

4.1 Benchmark comparisons for runs of the system for standard GP. All these runs were done on a 450MHz Pentium III with 192 Mb using Clisp 1999-07-22, using the boolean even-4-parity problem. All runs of the same population size use the same seed, producing identical results. 35

5.1 The number of runs for each of the algorithms discussed. . . . 38

Chapter 1

Introduction

1.1 Automatic Programming and Genetic Programming

Giving computers the ability to program themselves was one of the earliest goals of computer science in general and artificial intelligence in particular. The term "machine learning" was originally used to mean precisely this. However the difficulty of the problem has caused the machine learning community to concentrate on more tractable problems in the meantime.

The idea of automatic programming is to have the computer generate the program code necessary to solve the problem it is faced with. The rationale is that computer programs are complicated (and are generally becoming more so) and that reducing the human effort needed to produce them by having the computer do parts will speed development and reduce the problems caused by errors.

All machine learning techniques involve a guided search through a hypothesis space. Many different search methods have been tried since the birth of the field. The idea of using a biological approach for searching in computer science has a long history. Turing (1950) gave the initial details of

how this might occur.

> We cannot expect to find a good child-machine at the first attempt. One must experiment with teaching one such machine and see how well it learns. One can then try another and see if it performs better or worse. There is an obvious connection between this process and evolution, by the identifications
>
> Structure of the child machine = Hereditary material
> Changes of the child machine = Mutations
> Natural Selection = Judgement of the experimenter

Since the publication of this paper the human guided manner which Turing imagined for the search has been largely overtaken by technological advances. Modern evolutionary techniques use populations that far exceed the size that humans can easily evaluate and improve by hand[1].

Modern genetic programming was brought to the fore by John R. Koza (1992, 1994). He adapted the technique of genetic algorithms developed by Holland (1992) to computer programs. Genetic algorithms work using a population of possible solutions expressed as genomes. These are then assessed with respect to their success in solving the problem. A new generation is produced using counterparts to biological genetic crossover and mutation. Koza moved on from the standard linear genome structure of genetic algorithms to use programs represented as tree structured genomes such as that shown in figure 1.1.

The mutation and crossover operators of genetic algorithms became manipulations of subtrees of the programs. To perform crossover on two individuals to produce two new individuals a random subtree from each is selected and these are swapped. Mutation can take several forms, the most standard being the replacement of a subtree with a new randomly created subtree.

[1] Although hardware abilities now clearly outstrip anything Turing could have imagined it is interesting to see the range of techniques than are foreseen in this paper.

Other mutation operators include the permutation of the branches at a node, and the replacement of a tree by one of its subtrees (hoist). The fitness of individuals is calculated by their performance on a number of test cases of the problem to be solved. For a good introduction to genetic programming see Banzhaf et al. (1998).

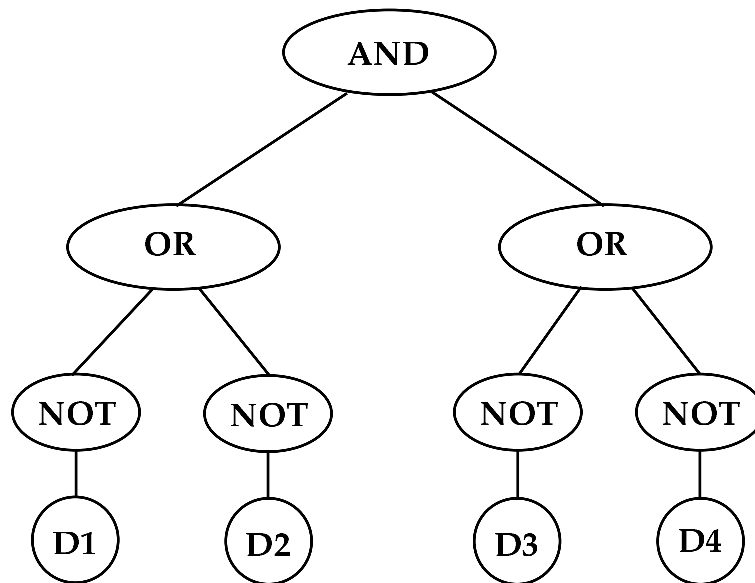

Figure 1.1: A simple program represented as a tree structure. This program can be written in lisp as `(and (or(not d1) (not d2)) (or (not d3) (not d4)))`.

1.2 Sub-routines in Programming

Any moderately complicated computer program uses sub-routines to simplify the structure. This is possible because most problems can be decomposed into smaller logically separate sub-problems. These sub-problems can be solved and then the pieces re-assembled to give a solution to the complete problem. Since most problems can be chunked into sub-routines, advantages of using those chunks are

- Repeated use of sub-routines can be made without error.
- Sub-routines from the solution to one problem can be used in another solution to another problem.
- Structural complexity of solutions is reduced.
- Parameterised sub-routines can use the same process in different situations allowing generalisation of the solution.

The breakdown into sub-routines thus speeds both the initial development stage and subsequent debugging of programs. A procedure written once can be checked and, once tested, is guaranteed to work whereas if the same piece of code were being continually rewritten typographical errors would evitably occur.

Given the advantages that subroutines bring to the human creation of computer programs it is natural to wonder whether they have similar benefits for the automatic generation of programs. There is also a precedent from the building block hypothesis of genetic algorithms. This states that solutions are found by the propagation of small above average segments of genome. If the spread of such segments can be encouraged then the performance of the system should be improved.

1.3 Aims

A number of techniques for using sub-routines in genetic programming have been investigated, and these will be examined in chapter 2. The aim of this dissertation is to examine a new method that attempts to avoid some of the shortcomings that are apparent in these methods.

The method explored in this dissertation differs from conventional techniques by having two interacting populations each of whose members are evolving. One of the populations contains the programs and the other functions that can be called by the programs. Both populations are subject to the

usual operators used in genetic programming. The fitness of the programs is calculated in the usual manner (ie. evaluation on a number of test cases). The fitness of each function depends on the fitnesses of the programs that call it.

Chapter 2

Literature Review

Since the publication of Koza (1992) and the emergence of genetic programming, a number of techniques have investigated the incorporation of sub-routines into genetic programming systems. Some of the major ideas will be examined here. In addition the neural network work of Moriarty and Miikkulainen which provided one of the inspirations for this thesis will be outlined.

2.1 Automatically Defined Functions

The most widespread method for incorporating sub-routines into genetic programming is Koza's automatically defined functions (ADFs). These were originally introduced by Koza (1992) and were the central theme in Koza (1994). ADFs use a syntactically constrained tree with at least two branches from the top node. One of these is the result producing branch that is evaluated to execute the program. The others contain function definitions that can be called by the result producing branch. The equivalent program to that given in figure 1.1 using a single ADF is shown in figure 2.1. In this diagram everything above the dotted line is fixed and the genetic operations only occur below the line.

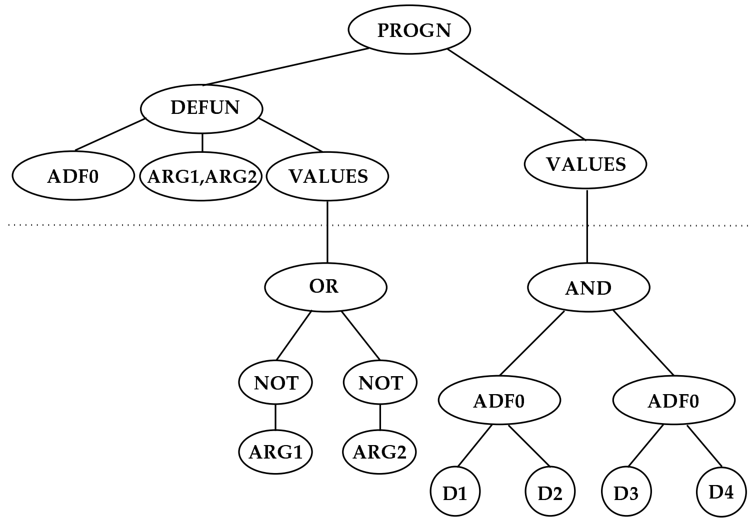

Figure 2.1: The program of figure 1.1 represented as a genome with a single ADF.

Each of the result producing branch and the function definition branches has a distinct set of functions and terminal constants that it can use. This restriction is primarily to stop problems caused by recursion. Since the result producing branch (RPB) has to be able to call at least one of the ADFs, if all branches had a common function set then at least one function would be able to call itself. A self calling function may not necessarily lead to an infinite recursion but over the course of an entire run it is probable that one would arise. Depending how the problem is set up it is, however, possible to have a hierarchy of functions that can call other functions.

Due to the distinct function and terminal sets the genetic operations on the individual have to be constrained. All the genetic operators are limited to act only within the body of the trees. Crossover is done by first selecting a branch of the same type from each of the parents and then swapping two subtrees from within these branches. Mutation also has to preserve the structure, so sub-tree mutation must use only the appropriate functions and terminals for the branch it is acting upon.

At the start of a run with a genetic programming system that uses ADFs the number and structure of the ADFs has to be decided. This involves deciding the function and terminal sets for each, including which ADFs can call other ADFs.

ADFs have been shown to be superior to standard genetic programming across a wide variety of problems. Koza (1994) analysed the performance of ADFs against standard genetic programming on a number of problems and generally found that the technique requires less computational power and produces structurally simpler solutions. Kinnear (1994) compared ADFs to both standard genetic programs and module acquisition (see section 2.3) using the even-4-parity problem (see section 3.4) and found that ADFs outperformed both other methods.

There are a variety of problems with this technique. One of these is that the number of ADFs used by a trial has to be decided before the start of the trial. This is undesirable as in many non-trivial problems it will not be clear a priori how many functions will be needed for a reasonable solution to the problem. In similar manner the structure of the ADFs is also decided in advance, in the form of the number of arguments each takes and the operators it is allowed to call. It is also not clear that the most obvious structure that a human would use to solve the problem is the most beneficial for a genetic programming system. Koza presents some rules of thumb for determining the structure and the number of functions but none are totally satisfactory[1]. A more promising strategy is to use architecture altering operations. This is examined in section 2.6.

A further possible flaw with this design is that the functions are inextricably linked to the programs. In human programming functions from one program are often taken and reused in other programs to solve other problems as they have been designed to be useful in a general context. It would

[1]The most systematic is to try several and see which works. This is clearly not ideal!

be advantageous if when solving problems the system could give us not only a solution but some information as to the structure of the problem (such as a natural hierarchy of sub-problems when one exists). This information could be used to inform the search for solutions to related problems. In Koza's scheme the close knit relationship between the program and function means that it is hard to tell what part of a function is generally useful and which is specific to the solution being evolved. There is also little pressure on the system to evolve generally useful functions.

There are advantages of ADFs over the other systems mentioned here. These include that they are the only system here that can use recursion. As mentioned above this is usually avoided but can be done with certain safeguards. These could include limiting the space or time the program is allow to run for, or keeping a track of the depth of the recursion and stopping it descending below a certain point. ADFs are also simpler to implement than several of the other systems here, such as adaptive representation. In addition and perhaps most importantly they are known to work on a wide variety of problems.

2.2 Encapsulation

Encapsulation was one of the earliest attempts to use sub-routines in genetic programming. It was introduced by Koza (1992) as a new elementary operator on the genome structure.

The encapsulation operator produces new terminal nodes from members of the existing programs. To produce a new terminal a random member of the population is selected. A non-terminal node from within this member is then chosen. The new terminal is formed from the subtree rooted at this node. The selected individual has the sub-tree replaced by the terminal. This replacement has no effect on performance of the individual but means that

the new terminal is available to spread through the population by crossover involving that individual. The new terminal is also made available to be introduced into the population via subtree mutation.

The transformation of the code into a terminal means that it cannot be affected by genetic operators (as the operators act on collections of nodes rather than the internal definition of the nodes). Thus the code is protected and the probability of it spreading is increased. If the code performs a useful function then this protection works to the benefit of the system.

The major problems with this approach are that only terminals are produced so the sub-routines that are being used are limited to having zero arity. Secondly the acquisition is random, so that while a useful piece of code may be selected it is more likely that junk code will be selected. It is therefore currently unclear whether encapsulation has any beneficial effects for genetic programming.

2.3 Module Acquisition

Module acquisition can be seen as a generalisation of the ideas of encapsulation and was introduced by Angeline and Pollack (1992, 1993). Whereas encapsulation produces new terminal nodes, module acquisition produces new functions of arbritrary arity.

In encapsulation a node was selected and the subtree rooted at that node became the new terminal. In module acquisition only the part of the subtree up to a certain depth is taken to form the module. The parts where the tree extends beyond this depth become the arguments of the module. As in encapsulation the module is protected from evolution once it has been formed. A second genetic operator of module expansion was also introduced. This replaces the call to a module in an individual with the module definition, allowing evolution to act on the code that was previously protected.

Kinnear (1994) compares module acquisition to ADFs and basic genetic programming on the even-4-parity problem (see section 3.4). In this study he found that there was no improvement over the standard genetic programming from using modules, and that both were substantially inferior to ADFs. He claims similar results for a second problem aimed at evolving a generalised sorting algorithm. It is therefore not clear that module acquisition has any benefits although some believe that more work is needed before the technique is discarded (Banzhaf et al., 1998).

2.4 Adaptive Representation

One of the major problems faced by both encapsulation and module acquisition is that they select the code to form new sub-routines at random. Adaptive representation (Rosca and Ballard, 1994) is an attempt to rectify this by choosing blocks of code in an informed manner. In doing so it also addresses one of the criticisms of ADFs, namely that the structure is predetermined.

Adaptive representation uses a three step method to create sub-routines.

- Useful blocks[2] of code are identified using an informed or heuristic technique.
- The blocks of code are generalised by the use of parameters.
- A number of random individuals using the new sub-routines are introduced to the population, replacing the least fit individuals.

The most difficult step of this procedure is the first. A number of methods have been tried including looking at the frequency of blocks or using a measure of block fitness. Rosca and Ballard (1996) examine a two-fold technique based on differential fitness and block activation.

[2]Blocks refer to pieces of code like the modules in Module Acquisition, they are subtrees that may be pruned at a certain depth.

Differential fitness is a heuristic that works on the principle that large changes in fitness are likely to occur due to the introduction (or removal) of useful blocks of code. The programs with the largest improvement in fitness in a generation were selected and their nodes are labelled by the frequency of their execution during the evaluation of all fitness cases. Small blocks of codes that are frequently used in the individuals with high differential fitness are highlighted to become the new sub-routines.

These sub-routines are then generalised by having their terminals replaced by parameters. The sub-routines are then placed in a separate population where they can be referenced by programs like members of the function set. This population evolves along with the main population. The evolution occurs by the replacement of low fitness individuals with newly created sub-routines[3]. To evaluate the fitness of the sub-routines they are assigned a utility value. This is calculated as the mean fitness of all the programs that have utilised the sub-routine over the past W generations (where W is some parameter).

In experiments on a Pac-Man based problem (Rosca and Ballard, 1996; Koza, 1992) it was shown that adaptive representation maintains a higher diversity in the population and is superior at discovering sub-routines to ADFs. Adaptive representation found superior solutions faster than both ADF and standard GP.

Adaptive representation is in some ways similar to the system that is developed in this dissertation. Both use a population of functions to try and address the restrictions caused by the pre-determination of the structure that occurs in techniques like ADFs. Adaptive representation has the advantage over the system put forward in this dissertation that the function and terminal sets for the evolved sub-routines and the number of arguments they

[3]This is not evolution in the normal sense of genetic programming, there is no crossover and mutation of members of the function population

take are not pre-determined. This approach is closer to the ideal method of incorporating sub-routines, but may not be advantageous in practice.

This method has several disadvantages compared to the system put forward in this dissertation. The most important is that the first step in the algorithm, that of selecting code to form sub-routines is difficult. Using a form of frequency analysis (such as the method put forward by Rosca and Ballard (1996) and several variants do) runs into the problem of introns in the evolved programs. Introns are pieces of code like (NOT (NOT ..) or an AND call with two identical arguments, these do not affect the result of the code but make functionally identical sections appear different to simple pattern matching algorithms. The second disadvantage is that adaptive representation is unable to use any form of recursion. A further difference is the form of evolution of the sub-routine populations; in adaptive representation the individuals do not evolve and the population does so only in the sense that sub-routines are added and unused ones are removed. In the technique that will be examined in this dissertation the population size is constant and the individuals in it evolve like in the same manner as programs.

2.5 Automatically Defined Macros

A variation on ADFs that has been proposed is automatically defined macros (ADMs) put forward by Spector (1996). Macros are operators that are implemented by transformation, and defined by saying how a call to it should be translated (for a good explanation of lisp macros see Graham (1996)). For example the following substitution macro would cause a robot to turn until the `sense-expression` returns true, and then return the value of the given `value-expression` in its final orientation.

```
(defmacro where-sensed (sense-expression value-expression)
   `(progn (while (not ,sense-expression)
             (turn))
```

```
,value-expression))
```

In the experiments by Spector (1996) only simple substitution macros were used, and the results were contrasted with those produced by ADFs. The semantics of ADFs and substitution ADMs are equivalent for domains where all operators are purely functional[4]. Thus there can only be an advantage from using ADMs in non-functional domains. Spector tested ADMs against ADFs using Koza's lawn-mower problem (Koza, 1994) without success and successfully using a variant of the obstacle avoiding robot (Koza, 1994) and Russell and Norvig's Wumpus World (Russell and Norvig, 1995).

These experiments contrast the abilities of ADFs and ADMs. There is clearly a possibility for the use of both ADFs and ADMs simultaneously to solve particularly difficult problems. However for this to be advantageous the use of macros other than substitution macros may be required.

2.6 Architecture Altering Operations

Architecture altering operations were introduced by Koza (1995) to attempt to reduce some of the shortcomings of ADFs. As mentioned previously one of the main problems of ADFs is that the structure has to be determined beforehand. This involves a number of decisions.

- The number of functions
- The number of arguments that each function takes.
- The terminals and functions available to each function.

Architecture altering operations address the first two of these. Six new genetic operators are introduced. These are

[4] A functional domain is one where two calls to the same operator with identical arguments will always have the same result. For example (+ 4 5) will always return 9 and so is functional but a robot control problem is not functional as the position of the robot and the external environment will affect sensor readings.

- Branch duplication: A new function defining branch is added to an individual. The initial definition of the new function is copied from one of the previous existing functions.

- Argument duplication: The number of arguments of a function is increased. The initial values of that argument in all calls is the same as for one of the other arguments.

- Branch deletion: A function defining branch is removed from the individual.

- Argument deletion: A function argument is removed from one of the functions of an individual.

- Branch Creation: A new function defining branch is added to an individual. The initial definition is a block chosen from another function branch or the result producing branch.

- Argument Creation: A new function argument is added to one of the functions of an individual by replacing a subtree in the function with the new parameter. The initial values of the argument in the calls to that function are the value of the replaced subtree.

These genetic operators are introduced much like the mutation operators of genetic programming. As a result it is clear that the population of individuals will rapidly become structurally diverse. This means that standard crossover techniques will not be guaranteed to produce valid offspring. To counter this Koza uses a technique called structure-preserving crossover (Koza, 1994).

In (Koza, 1995) a number of runs using this technique are carried out on the boolean even-3-parity problem. These successfully evolve individuals to solve the problem. It is not clear from these results how the performance compares to standard genetic programming techniques. However on problems that are not well understood it appears that the technique could be used to present insights into the optimal structure.

2.7 SANE

The SANE system (Symbiotic, Adaptive Neural Evolution) was proposed by David E. Moriarty and Risto Miikkulainen. This is not a genetic programming system, however it is reviewed here because the ideas in this project are based on the ideas that are used in SANE. The system is designed to evolve neural networks using a genetic algorithm in a two layer structure. At one level the individual neurons are evolved by representing their input weights and threshold. These neurons are then combined to solve the target problem. In the initial work on SANE the neurons were combined randomly into the networks (Moriaty and Miikkulainen, 1996).

Later work moved on to evolve the structure for the neural network simultaneously with the neurons (Moriarty and Miikkulainen, 1998). This work used a population of neurons as described above and then a population of network blueprints that specified which neurons were to be included in the network. The neurons were evaluated based on the average performance of the networks they participated in.

The reproduction process for the neurons is an aggressive strategy where for each neuron in the top 25% of the population a second member of the top 25% is chosen to mate with it. The mating operation produces two offspring: a copy of one of the parents and a child from one-point crossover (the second child is discarded). There is a small chance of mutation for the entire population at the end of the reproduction phase. The network blueprints are reproduced in an identical way except for mutation operators. Two mutation operators are used. The first changes a neuron to a randomly chosen neuron. The second is more selective taking a neuron that has bred and replacing it with one of the two offspring[5]. This second mutation operator was introduced to aid the system in exploring new neurons.

[5] This is why one of the offspring is kept identical to one of the parents - to stop this mutation operator causing too large an adverse affect.

Moriarty and Miikkulainen tested SANE against genetic algorithms evolving just the neurons in random combinations (similar to the original SANE) and evolving the entire network as a single entity. The problem was to position the hand of a robot arm within 10cm of a object, using inputs from a camera on the hand and knowledge of the three joint positions. The neuron based approach whilst being the fastest initially generally stalled converging to suboptimal solutions. Both the network based approach and SANE produced successful solutions with SANE being almost as fast as the neuron based approach in the early stages.

Chapter 3

Method

3.1 Project Aims

As outlined in chapter 2 there are a number of problems with most of the systems used to incorporate sub-routines into Genetic Programming. The purpose of this thesis is to put forward a new method that addresses two of these problems. The first is the predetermination of the number of functions that solutions are able to have. The second is to try to evolve generally useful functions.

To achieve these aims the relationship between functions and programs will be changed. Instead of a single population with the functions and programs sharing individuals there will be two populations (see figure 3.1). The first population will consist of the programs, the second of functions. Each function in the function population will have a unique identifier and can be called by any number of programs. Conversely the programs can call any number of functions. To evaluate the fitness of programs they will be run on a number of test cases in the usual manner. To evaluate the fitness of functions in this way is clearly not possible. Instead each function will be assigned the mean fitness of the programs that call it (programs that call

it twice being counted doubly). The evolution of the two populations will occur concurrently.

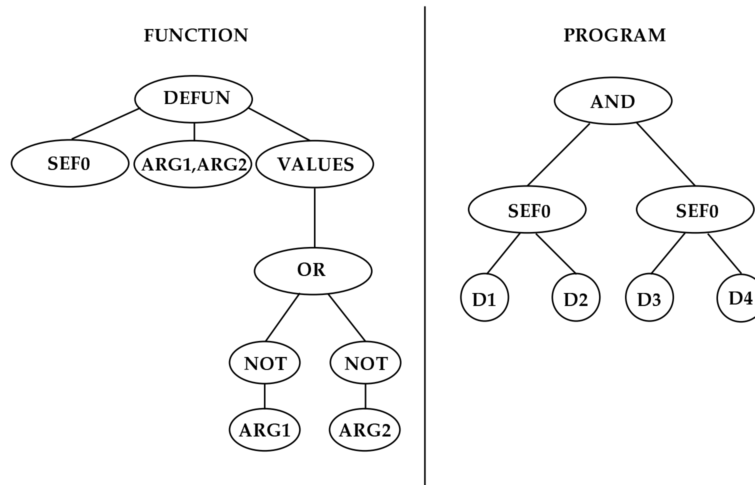

Figure 3.1: The representation of the program from figure 1.1 represented by the technique being investigated in this problem.

This approach is inspired by the work of David E. Moriarty and Risto Miikkulainen on their SANE neural network system (see section 2.7). In SANE the two populations consist of neurons (which correspond to the functions) and networks composed of those neurons(corresponding to programs). Initially it was felt that the aggressive breeding strategy of SANE was inappropriate to the more delicate forms of genetic programs, but this was tried later (see section 5.3). Each program can call as many functions as it wishes (subject to the size of the function population). The system is released from the grip of guess work as to the optimal number, allowing evolution to decide for itself. Since functions can be called by multiple programs there is also more pressure on them to become generally useful.

Although several of the techniques outlined in chapter 2 use two populations the behaviour of the function population differs in all of them. All of the techniques in chapter 2 have a function population that is initially

empty and then has members added during the run. In contrast the system proposed here has a constant number of members throughout the run. Also in the established techniques once functions are added to the function population their internal structure is not affected by genetic operators. In the system to be examined the population undergoes standard genetic evolution.

Since the members of the function population have to co-operate with each other they can, like the neurons of SANE, be regarded as symbiotic and so will be referred to throughout this project as symbiotically evolving functions or SEFs.

3.2 Basic Algorithm

The basic algorithm that was used for the new technique is given in figure 3.2. This algorithm is much the same as standard genetic programming except for the incorporation of the second population.

The most important difference to standard genetic programming that arises is that each program can reference the members of the function population to incorporate the sub-routines. To make this sensible each member of the function population has a unique name. When the new function population is being generated a new function is generated by crossover or mutation using tournament selection. A function for it to replace is chosen by picking the loser in a separate tournament. The new function is then given the name of the loser and takes its place in the population. Thus any program that previously called the replaced function will now call its successor. Program reproduction is exactly analogous to standard genetic programming.

One important factor to decide was the rate of change of the function population with respect to the program population. The extreme cases are a combination of being generational and being steady state; either with one of each or both the same type. The faster one population evolves with re-

```
Create initial program population P
Create initial function population F
Repeat until termination criteria satisfied
  For each program p ∈ P
    Evaluate program p
  For each function f ∈ F
    For each program p ∈ P
      For each time p calls f
        CallerFitnessSum(f) = CallerFitnessSum(f) + Fitness(p)
        Calls(f) = Calls(f) + 1
    Fitness(f) = CallerFitnessSum(f) / Calls(f)
  Generate a new function population
  Generate a new program population
```

Figure 3.2: The basic algorithm used for this thesis.

spect to the other the more emphasis there is on that population to solve the problem. But the more it would have to work around the constraints introduced by the second populations. To have a rapidly evolving program population and slow functions would lead to a situation rather similar to module acquisition (see 2.3), which has been seen to give no advantage over standard GP. It is also hard to see the advantage of evolving the functions quicker than the programs as then the inflexibility of the programs would stifle the functions. It was therefore decided to evolve the programs and the functions at approximately the same rate. To try and increase the chances of the functions evolving into useful forms it was decided to make the evolution fairly gentle by using a steady state GP algorithm with tournament selection (Kinnear, 1994).

During the experimental trials it be came apparent that refinements to this algorithm were needed. These are detailed in sections 5.3 and 5.4, where the motivation for them is more obvious.

3.3 Experiments

The experiments were carried out using the even-4-parity problem used by Kinnear (1994). This is described in more detail in section 3.4.

It is usual to run genetic programming trials with a population of the order of 1000 (Koza (1994) uses sizes upto 16000 on the even-3-parity problem). However the speed of a genetic programming system is heavily dependent upon the size of the population (this is discussed in section 4.6). Thus to increase the variety of experiments tried it was decided to reduce the size of program population in order to increase the speed of the system. However decreasing the population size has the effect of reducing the number of individuals examined, and thus the chance of finding a solution. To give an indication of the performance of the system on the larger populations it was decided to use a three way comparison. To do this the SEF experiments would be run using a program population of 250. The ADF and standard genetic programming experiments would be run using population 250 and be repeated with a population of 1000 (the size used by Kinnear (1994)). The scale of performance on the ADF and standard GP should give an indication of the scaling that would occur with SEFs[1].

This three way comparison was used because SEFs are necessarily slower and require more computational effort than the SGP and ADF techniques. This is because although the number of newly bred programs evaluated is the same, the actual number of programs evaluated is not. This is caused by the changing functions; when a function is changed the behaviour of the programs that alter it will be changed as a result. This means that more programs are evaluated over a run compared to conventional techniques and hence the time for the run is increased.

[1]This method was suggested in a discussion with John Hallam.

3.4 Even-4-Parity Problem

The even-4-parity problem is a boolean logic problem. Given four boolean (true/false) inputs the program should return true if an even number of them are on and false otherwise. Although this is a simple problem to describe, a solution is not entirely trivial.

```
(or
;; An even combination of d1/d2 are on and an even
;; combination of d3/d4
     (and (or (and d1 d2)
              (and (not d1)
                   (not d2)))
          (or (and d3 d4)
              (and (not d3)
                   (not d4))))
;; An odd combination of d1/d2 and an odd combination of
;; d3/d4
     (and (or (and d1
                   (not d2))
              (and (not d1)
                   d2))
          (or (and d3
                   (not d4))
              (and (not d3)
                   d4)))))
```

Figure 3.3: A solution to the even-4-parity problem using AND, OR and NOT with no sub-functions.

The problem was used by Koza (1994) to test ADFs against standard GPs and Kinnear (1994) to test ADFs and standard GP against each other and against module acquisition (see section 2.3).

It is a problem that has a natural use of functions. The structure `(or (and ..) (and ..))` can be seen to repeat in the solution and if we use this then we can simplify the solution to that seen in figure 3.4. The function defined is the (NOT (XOR .. ..)) in standard boolean algebra terms.

```
Program = (func (func d1 d2)
                (func d3 d4)
func(arg1 arg2) = (or (and arg1 arg2)
                      (and (not arg1)
                           (not arg2)))
```

Figure 3.4: A solution to the even-4-parity problem using the sub-functions.

The even-n-parity problems can make use of the XOR function in a recursive structure. If f_n is a solution to the even-n-parity problem then

$$(f_{n+1}\ d_1 \ldots d_{n+1}) = (Xor\ d_{n+1}\ (f_n\ d_1 \ldots d_n)))) \tag{3.4.1}$$

and (generalising figure 3.4)

$$(f_{2n}\ d_1 \ldots d_{2n}) = (Not\ (Xor\ (f_n\ d_1 \ldots d_n)\ (f_n\ d_{n+1} \ldots d_{2n}))) \tag{3.4.2}$$

and the even-2-parity function is exactly NOT XOR.

The settings used for the problem are given in table 3.1. The function set available included the boolean operators AND, OR and NOT for all three techniques with the terminals being the four inputs. For automatically defined functions one ADF was added which takes two arguments and uses AND, OR and NOT. This was chosen as it is the most natural form for the solution to evolve to (see above). For SEFs the function set includes the members of the SEF population which have the also take two arguments and use AND, OR and NOT.

3.5 Expectations

Should the proposed scheme be successful in producing solutions to problems it could have a number of advantages over standard sub-routine techniques. The first of these is that it could highlight generally useful sub-routines, for the test problem this is most likely to be (NOT XOR). Since each sub-routine

Objective	Evolve an expression which will return `t` when an even number of arguments are `t` and `nil` otherwise.
Terminal set	`d1`, `d2`, `d3`, `d4` generally. `arg1`, `arg2` when using ADF or SEF.
Function set	`AND`, `OR`, `NOT` throughout. May include `ADF0` or an SEF for those problems.
Fitness cases	All sixteen combinations of four boolean inputs
Raw Fitness	The number of fitness cases for which the expression fails to generate the correct value of the even-4-parity function. Equivalent to the Hamming distance to the correct solution.
Standardised Fitness	Same as raw fitness.
Hits	Not used.
Wrapper	Not used.
Parameters	Population size either 1000 or 250. Maximum generations = 50.
Success predicate	When one individual has a raw fitness of 0.

Table 3.1: The settings for the even-4-parity problem. Adapted from Kinnear (1994).

can be used by all programs those that are useful in a variety of situations should prosper over those that require very exact conditions to be useful. In a similar manner to the neurons in SANE the competition between the functions means that if more than one function is advantageous then the functions will have to evolve in a manner that allow co-operation between them. In this respect it should have some of the advantages of adaptive representation.

The system should be able to find the number of functions that are needed to solve the problem without extra experimentation. This is in contrast to ADFs where the optimal number of functions has to be discovered by multiple trials.

It would also be possible to extend the technique to multiple problems simultaneously. This would evolve several program populations all making use of the same function population. This would further increase the pressure to evolve general useful sub-routines because of the slightly varying nature of the problems. It would be most likely to work in situations where the problems have a reasonable relationship (eg different boolean n-parity problems Koza (1994) or the various ant trail problems Koza (1994)).

Reversing the previous idea to have multiple function populations called by a single program population would be more useful when nothing is known about the problem structure. It would be expected that populations that do not have a suitable structure could be evolved out of use. This could show some of the advantages of architecture altering techniques.

Chapter 4

Implementation

4.1 Adaptation versus Innovation

The first design decision that had to be made was whether to adapt an existing system or write a fresh one from scratch. There are arguments to be made both ways in such a discussion.

An existing system has already been tested, debugged and documented. This should make the experiments on standard genetic programming and ADFs for comparison should be very easy to set up and run. However the idea that is being investigated has large underlying differences from classic GP systems; primarily the interaction of the two populations to solve the problem. This could be arranged as a co-evolution with co-operating rather than the more usual competing populations. Thus using an existing system with minimal modification would require a system that already supported co-evolution. Moreover if specialist genetic operators were later desirable then these would have to be added to the code, which might not easily support them.

Another requirement is that the source code for the system is firstly available and secondly comprehensible. This is desirable for a number of reasons.

One is so that specialist operators and other code can be added if it is found necessary. Secondly if unexpected behaviour is seen then it should be possible to check if there is a fault and where it lies. If it is found to lie within the system it should be rectifiable (relying on writing to the system author and then hoping that he responds within the time allowed for this project would be a dangerous policy). The availability of the source code is not a particular difficulty. Most GP systems are non-commercial and are distributed in source code form. The requirement for comprehensibility is more of a challenge. Most systems are written in C/C++, Java or Lisp. The author knew none of these languages at the outset of this project[1].

Four systems were looked at with a view to adapting. GPData (written in C++), lilGP 1.1 (C), EC (Java) and the Lisp code from Koza (1994). All of these have support for ADFs. Given ignorance of all 4 programming languages this did not present a bias. It is a truth universally acknowledged that the reading of somebody else's source code is unpleasant. With this in mind the less code there is the better. The Systems varied wildly in size. EC is 30,000 lines, GPData and lilGP are both around 13,000, whereas Koza's code is a mere 1,500. Thus the decision was made to look at Koza's code to start with.

The examination of this suggested that it would not be an ideal starting point. The code has been written to apply only to ADFs. (Koza used a separate although substantially similar system to do standard genetic programming.) This means that certain structural features (such as the number of ADFs being two) are deeply embedded in the system. However the examination did suggest that the creation of a fresh system from scratch would not be particularly difficult. So this was the next possibility examined.

[1]But did have coding experience, mainly in Pascal and Python.

4.2 Programming Language

If the system is to be created from scratch the choice of programming language again rears its head. The choice was to be made from amongst C, C++, Java, Lisp, Python and Pascal. The first four on the grounds that it would be possible to examine previous systems for ideas when problems were encountered. The last two being those that the author was most familiar with at the commencement of this project.

Python is a interpreted language and although it has many nice features has nothing to particularly recommend it for this project. Java is also (essentially) interpreted and a brief look at the EC code suggested it was moderately incomprehensible to a beginner if ideas from the code were needed. C/C++ are probably the fastest languages of the group but speed was not absolutely critical. In addition C/C++ could not be described as rapid development languages. Lisp had several advantages over the others. Koza's code was short and comprehensible, and even though large parts would need to be rewritten some could be used. Being a high-level language it is fairly quick to develop in, but can be compiled later for further speed if necessary. It is eminently suited for genetic programming with its easy handling of tree and list structures. It is also very easy to learn. It was therefore decided to design and code the system from the ground up, using Lisp and code from Koza where possible.

4.3 Design

There were certain features that were considered desirable for the system structure. The most important of these was that the system should be able to work with standard genetic programs (no functions), Koza's automatically defined functions and the symbiotic functions with the minimum of alteration. This is an important feature to enable comparisons to be made

between the three methods being investigated. The second feature was that the system be modular in that the problem specifications were separate files to the main kernel of the system. This would allow easy alteration of the problem should more than one be investigated.

The main loop for the resulting system is shown in figure 4.1. The main loop for the classic GP system is essentially the same without the references to the function population. The algorithm is tail recursive as this is the most neatest way of writing it and it can be optimized to run in constant memory by all ANSI-compliant Lisp compilers (Graham, 1996). With the decision to use a steady state population and more specifically tournament selection most of the rest of the design followed naturally and will not be examined here.

```
(defun run-sef-gp-generations (generation
                               program-population
                               function-population)
  (reset-fitness program-population)
  (reset-fitness function-population)
  ;; Evaluate the fitness
  (evaluate-all-program-fitness program-population)
  (evaluate-all-function-fitness function-population program-population)
  ;; Return the final populations if we've finished otherwise recurse
  ;; down.
  (if (termination-predicate generation program-population)
    (values generation program-population function-population)
    (progn
     (report-on-generation)
     (run-sef-gp-generations (+ generation 1)
                             (breed-new-population program-population)
                             (breed-new-population function-population)))))
```

Figure 4.1: The main loop (in lisp-like pseudocode) of the GP with symbiotic functions.

4.4 Evaluation of Evolved Programs

Another important decision that had to be made was how to execute the produced programs, so as to build the functionality in. There are two methods of doing this, by using the Lisp `eval` macro or by hand building an interpreter.

4.4.1 Using the Lisp Evaluator

It is possible to set the system up so that all functions the evolved program can call are defined as Lisp functions (including ADFs and SEFs). The program is then executed by a single call to `eval`. This is faster than hand evaluation. However setting up the evolved functions to be executed is harder. In Koza's code (Koza, 1994) this method is used but the number of ADFs is hard coded into the kernel and so the appropriate number of functions (2) is also included. Since it had been decided to have no problem independent restrictions for the number of SEFs, this solution was not available.

When executing SEFs there could be several hundred and having this number of preset functions is wasteful (especially since the number could vary significantly during testing to find the optimal population size). It would be possible to get around this by using a syntactic constraint on the tree so that instead of the functions being called as `(SEFxx (..)  (..))` they are called as `(SEF xx (..)  (..))`. So instead of an unique name for each function there is a single wrapper procedure whose first argument acts as an identifier as to which function is being called. The problem with this approach is enforcing the constraint in the reproduction of trees; it would be an illegal operation to replace the identifier with one of the arguments of the function call.

4.4.2 Hand Executing

The second solution is to use a hand coded evaluator. A simple evaluation model of Lisp is to take the present list, evaluate each member of it and then apply the first member to the rest (Abelson et al., 1996). When the evaluator finds a list whose first element is an evolving function then it should find the function and apply it appropriately. The most fundamental problem with this method is that the simplistic evaluation model given above does not cope with special forms and macros (for example, of the then-clause and else-clause in an if statement, only one should be evaluated. This is particularly important if there are operators that rely on side-effects). A second problem is dealing with variables and scoping. A simple solution would be to declare the variables needed by the program as global, however there is a (small) chance that this could cause a clash with variables in the system kernel. This is unsatisfactory as the implementor of the problem file should not have to worry about the internal operation of the kernel beneath a certain level.

4.4.3 Compromise Solution

As far as the main kernel of the system goes the decision was to opt out. All the details that were needed to execute the code would be made available and the problem file would be left to deal with the details of how to use these. While this simplifies the kernel and leaves the onus on the implementation of the problem specification, it still left the decision to be made with regard to the actual problems being used.

Initially it was hoped that `eval` calls could be used to do the problems. While this worked for standard GPs and ADFs the execution of the SEFs as expected proved difficult. This is due to the fact that until runtime the number and hence the names of the SEFs are unknown. Thus the functions are created dynamically. Although this was tried it was not successfully

achieved.

It was therefore decided to take the slower hand-evaluation option for SEFs and the lisp interpreter for ADFs and standard GP. The code to do this is given in appendix B . The SEF definitions are placed in a global hash table `*sefs*` and inserted as required. To deal with the problems of scoping variables the `evaluate` functions are called with a `progv` wrapper to set all the required variables. For the even-4-parity problem studied an `if` operator was not required.

4.5 Experiment Implementation

One of the advantages of the even-4-parity problem is that it is very easy to implement once the interpreter has been written. The problem file was designed to pass the parameters for the run to the main program as global variables and then provide three functions:

- `termination-predicate` which given the program population and the current generation should return true if the run should be terminated (ie. if we have exceeded the maximum number of generations or have a correct solution).
- `fitness-better-p` which given two individuals should return true if the first has a better fitness than the second. For the even-4-parity problem better fitness has a lower value.
- `evaluate-individual-fitness` which given an individual returns its fitness.

The parameters that the system returns specify the population sizes, mutation rate, function and terminal sets for the different populations or branches of an individual etc.

To calculate the fitness of an individual it is evaluated with all 16 possible input combinations and the outputs are compared to the correct values that are stored in a look up table.

4.6 Optimization

The major problem that had to be overcome once the system was built was speed. Genetic programming is not a fast process and with the limited amount of time available for the project, results had to be produced reasonably quickly. In the initial system a run of 50 generations with a population of 1000 using standard genetic programming with the hand coded evaluator, would take in excess of 50 days[2].

A number of approaches were tried. The parameter settings that Koza uses for his runs were to be the base point for comparison. However he uses very large populations, typically of the order of 1000. Since a steady state GP is being used, this translates to 50000 iterations for 50 generations. If the fitness of each individual is recalculated with each iteration, then the order of the run time[3] is quadratic in population size (the number of iterations is proportional to the population size for a steady state GP and then this is also multiplied by the number of individuals to be evaluated each iteration which is the population size). However for standard and ADF GPs there is normally nothing that would cause the fitness of an unchanged individual to change from one generation to the next. Thus for these strategies the system was changed to evaluate new individuals only. This reduces the run time to being linear with population size. For a trial run on the boolean even-4-parity problem the time for a run was reduced by a factor of 70 (see table 4.1)

[2]A run with a population of 100 took over twelve hours (see table 4.1) and with the original system the time is proportional to the square of population.

[3]Assuming that the evaluation of fitness is the biggest bottleneck in the system.

A further increase in speed was made by compiling the Lisp source code. For the benchmark problem this gave a further 5-fold speed increase which makes Koza style runs far more feasible (see table 4.1).

	Population Size	Time / minutes
Uncompiled, Full Evaluation	100	> 720
Uncompiled, Minimal evaluation	100	11.1
Compiled, Minimal Evaluation	100	2.55
Compiled, Minimal Evaluation	1000	95.1

Table 4.1: Benchmark comparisons for runs of the system for standard GP. All these runs were done on a 450MHz Pentium III with 192 Mb using Clisp 1999-07-22, using the boolean even-4-parity problem. All runs of the same population size use the same seed, producing identical results.

While the strategy of only evaluating new individuals is possible for standard and ADF GPs, it is neither as straightforward to implement nor as time saving for SEFs. This is because a new program will change the fitness scores of all the functions that it calls. In addition a new function will change the scores of all the programs which call it, in turn changing the fitness of all the other functions that those programs call. Thus the speed up depends on the interconnection of the programs and functions which varies from run to run. However a lazy evaluation algorithm was implemented and gave a small speed increase. The algorithm used is given in figure 4.2.

The second main technique used to quicken the run-times was to reduce the population. For a steady state GP algorithm the (maximum) number of iterations, and hence run-time is proportional to the population size. Thus reducing the population from 1000 (as used by Kinnear (1994)) to 250 should bring approximately a fourfold speed increase. However the downside is that four times fewer individuals are examined and hence the probability of finding a solution is reduced. To try and offset this the three way comparison was carried out (see section 3.3).

```
Mark new functions as unevaluated.
Mark new programs as unevaluated.
For each Program P in the program population
  Find the functions F1, ..., Fn used by P.
  Ifany of F1, ..., Fn are unevaluated.
    Set P as unevaluated.
For each Program P in the program population
  Find the functions F1, ..., Fn used by P.
  IfP is unevaluated
    Set each of F1, ..., Fn as unevaluated.
Evaluate all unevaluated programs.
Evaluate all unevaluated functions.
```

Figure 4.2: The algorithm used for the minimal evaluation of programs and functions for SEFs.

Another method that was considered but not implemented was the hashing of programs and their fitnesses. For this each program is associated with a unique numeric identifier and the correspondence between this and the fitness of the program is stored in a database of some form. The theory is that the encoding of the trees and look up of the value can be carried out rapidly. However with the implementation of the non-evaluation of old individuals and the minimal probability of two identical individuals being reached it was not deemed to be advantageous enough to use.

4.7 Testing

Being stochastic processed genetic programming systems are not entirely trivial to test. However to ensure that the system was working it was possible to carry out some tests.

The program interpreter was tested by evaluating programs whose output was known. It was also checked by evaluating programs from actual test runs to ensure that the system was passing all the appropriate information

correctly.

The standard check for genetic algorithm and programming systems is to seed the population with a solution that is known to be good and check that it is propagated from generation to generation correctly. This was done for the systems. However it should be noted that this is not necessarily a correct test for the SEFs system. This is because if the solution uses a function (as would be desirable for checking the system) there is a chance that the function might not propagated. This would occur if the other programs that use it do not have a particularly good fitness. If the function is replaced at some point then the solution will be altered, probably for the worse. (The exception to this is the monotone SEF algorithm, see section 5.4.)

Chapter 5

Experimental Results

Since genetic programming is a stochastic process it is not possible to draw conclusions from a single run. Instead the properties being measured should be averaged over a number of trials. The number of trials for each of the methods discussed here varies. The number depended on the time that was available and the time that each took, as many trials as possible within those constraints being carried out. The number of runs for each is given in table 5.1.

Algorithm	Population Size	Number of Runs
ADF	1000	13
ADF	250	21
SGP	1000	14
SGP	250	36
SEF, original	250	10
SEF, cross-mutation	250	11
SEF, SANE	250	40
SEF, monotone cross-mutation	250	8

Table 5.1: The number of runs for each of the algorithms discussed.

5.1 Control Experiments

For the purpose of making comparisons a number of runs using standard genetic programming and automatic function definition were performed. As discussed in section 3.3, these were done at two different population sizes. Figure 5.1 shows the cumulative probability of finding a correct solution over the length of a 50 generation run. From this it can be seen that with a full size population of ADFs there is a high probability of finding a solution. As would be expected reducing the population from 1000 to 250 reduces the likelihood of finding a solution.

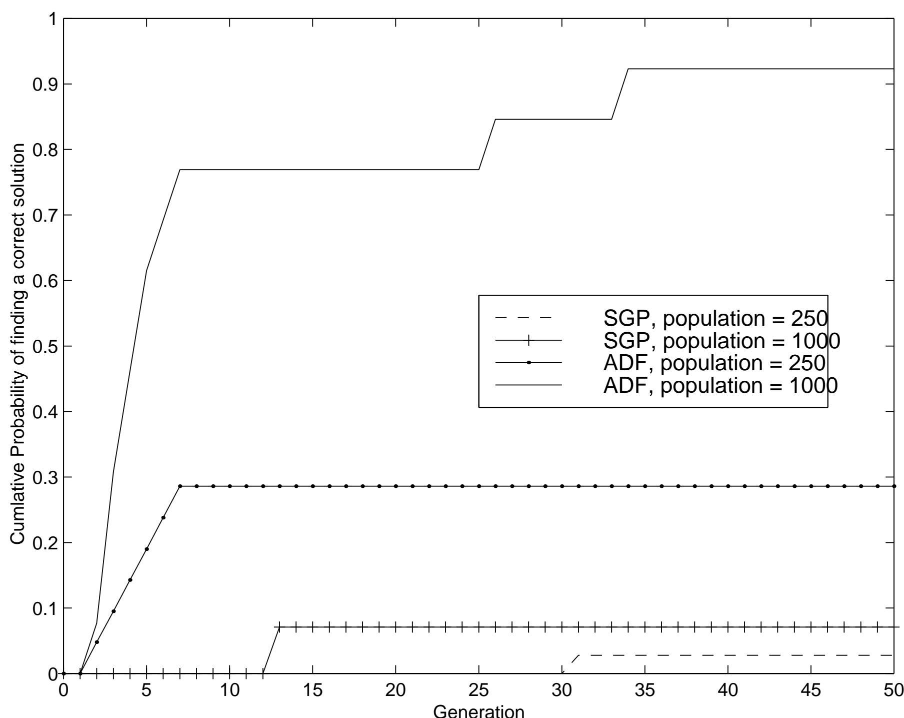

Figure 5.1: Graph of the cumulative probability of finding a correct solution to the test problem against generation for ADFs and standard GP.

In comparison to ADFs standard genetic programming struggles to find a solution. Again as would be expected the increased population improves the

probability of finding a solution, however both are distinctly inferior to even the small population ADF results. The comparison between the performance of the large population standard GP and ADFs is a similar result to those in Kinnear (1994).

5.2 Basic Algorithm

The cumulative probability of finding a correct solution to the problem with the algorithm described in section 3.2 in comparison to the SGP and ADF approaches is shown in figure 5.2. From these results it can be seen that while the technique performs better than standard GP, it is significantly worse than ADFs.

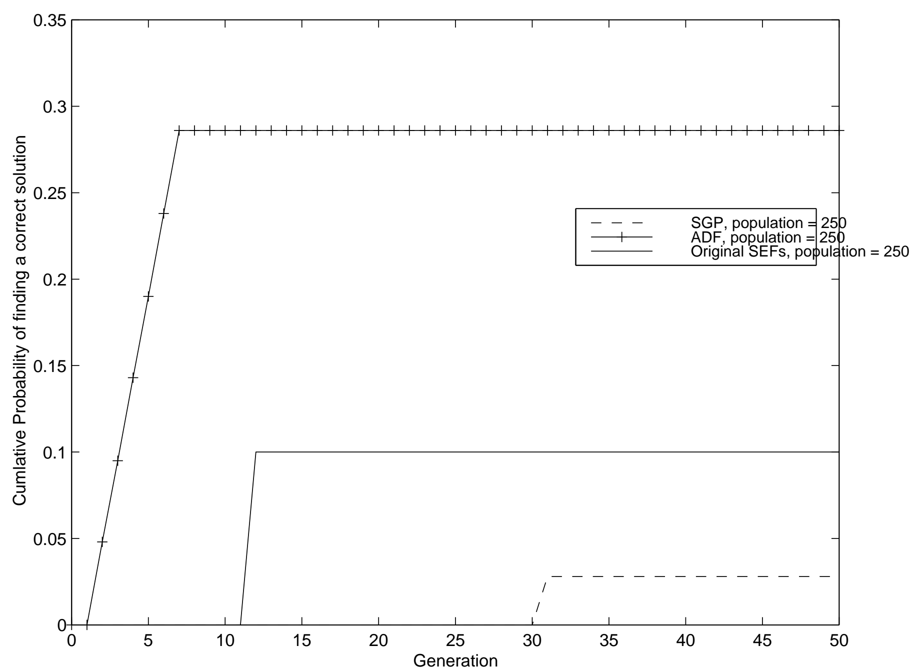

Figure 5.2: Graph of the cumulative probability of finding a correct solution to the test problem for ADFs, the first SEF algorithm and SGP.

Although the superior performance compared to standard GP is promis-

ing it would be preferable to have performance comparable to ADFs which is not the case. To explain why this is we can examine the details of the runs. The graph in figure 5.3 shows the fitness of the best individual at each generation, for a converged trial and a typical non-converged run. (Recall from table 3.1 that the fitness of a converged individual is 0.) The most notable feature of this graph is that the fitness in the typical run is not monotonically decreasing[1]. This is in contrast with the normal behaviour of steady state algorithms. In most steady state GPs the choice of a poor individual to be replaced means that the best individual cannot be lost and so the best fitness can not increase[2]. Although in SEFs the best individuals in each population are protected it is very possible for the best program to call several functions some of which may not have a good fitness in comparison to the rest of the function population. If one of the functions used by the current best program is replaced by a new function then the best fitness of the program population will probably increase (worsen).

The chance of a good candidate program being upset by a sudden change in one of the functions it calls is a two-edged sword. It is detrimental in that the program population will have adapted to use the function in its previous state (since good segments of code such as the call to that function will spread through the population). Thus when this function is changed there is likely to be a considerable worsening in fitness as a result. However this should introduce a evolutionary pressure to use only a few functions, since the fewer the number of functions used the less likely that one will be suddenly altered. This bias towards parsimony should be beneficial to producing generally useful functions, since it is preferred to have a few functions called repeatedly

[1] It should not be assumed from the graph that only a run where the fitness of the best individual at each generation decreases in a monotonic fashion can converge.

[2] The description of increasing and decreasing is with respect to the fitness method used in this problem. If a numerically greater fitness indicates an improvement then the descriptions of increasing or decreasing would reverse.

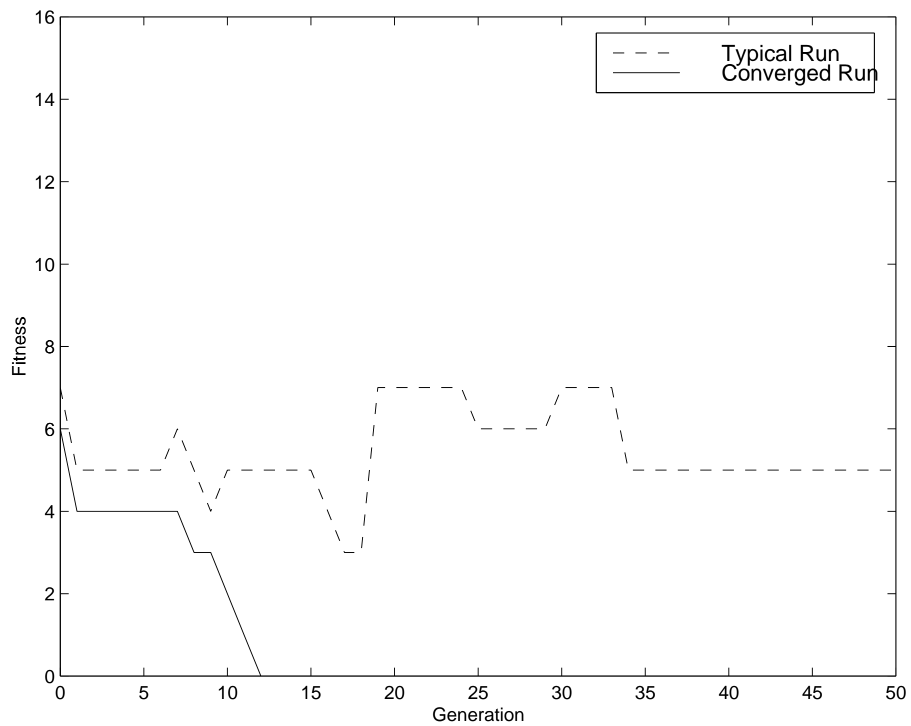

Figure 5.3: Graph of the fitness of the best individual of each generation for a typical and a converging run for the original SEF algorithm plotted against generation.

rather than lots of functions each only called rarely.

It is also likely that there is another factor hampering the evolution. When a function is reproduced its child is placed in the slot previously occupied by another poorer performing individual. The newly created function will therefore only be called by the programs that called the replaced function, in contrast to ADFs where if a new function is created it is only called by the same program as called one of its parents. These are firstly unlikely to be particularly good programs (as functions are regarded as performing poorly if the programs that call them perform poorly). Secondly they are unlikely to be using the new function in the sort of situation its parent was used in and performed well in. Thus the function can be regarded as losing the linkage to the calling programs and situations that made its parent useful. It was this linkage that caused Moriarty and Miikkulainen to introduce a specialist mutation into the SANE system. This operator worked in the following manner: if a neural network was being reproduced that used one of the reproduced neurons then with a 50% probability the reference to the reproduced neuron was switched to reference the child (see section 2.7).

5.3 Linkage Preserving Algorithms

Since the other cause of the non-monotonic behaviour could also offer potential benefits the first factor to be investigated was the linkage. To carry out this investigation two further algorithms were tried. One was simply the addition of a new mutation operator to the basic algorithm described in section 3.2. The second was to completely replace the population reproduction method with that used in SANE (see section 2.7), with functions corresponding to neurons and programs to networks. The algorithm is so similar to Moriarty and Miikkulainen's that it is not reproduced here.

The mutation operator is used after the new function and program for

the current iteration is created. If the newly created program uses the parent of the new function then the calls to the parent function in the new program are switched to call the child function. This operator will be referred to as cross-mutation. This mutation operator was applied whenever possible.

The cumulative probability of finding a solution for the two new methods can be compared to the original algorithm, ADFs and SGP in figure 5.4. As can be seen from this the SANE algorithm preforms poorly compared to the original algorithm, although still better than standard GP. It is likely that this is because as suspected the programs are rather more delicate than the neurons that the algorithm was successful on (see section 3.1). Thus the rapid changes to the population that the algorithm causes disrupt the evolution away from the good solutions as much as it moves towards them.

The cross-mutation technique seems more successful, performing better than the original method, although still not as well as ADFs. This suggests that linkage is one of the problems that is hampering the technique, although this technique could still be improved upon.

Structural complexity of programs can be measured in a number of ways, such as size, number of function calls, tree depth, et cetera. The freedom to choose from a large population of functions means that inclusive of the function definitions SEFs are almost always going to have a greater size than ADFs[3]. For a fairer comparison the number of function calls was examined instead. The mean number of function calls made by the best solution of each generation (over all trials and all generations) of the ADF system was 9.9. For SEFs using cross-mutation this figure was slightly lower at 8.6[4]. The difference between the two is probably not significant, but suggests that SEF

[3] Koza (1994) uses size of solution as the sole method of comparing structural complexity of ADFs against that of SGP.

[4] This is the number of function calls not the number of nominally distinct functions used which was 4.8. The number of functions that are distinct in terms of the results they give will probably be slightly lower and was not investigated.

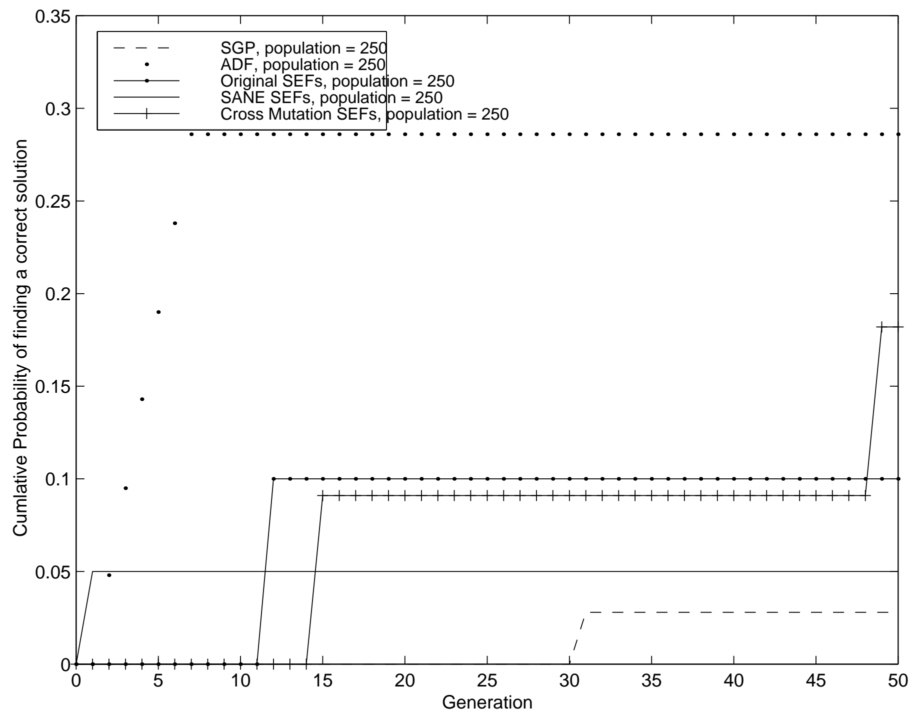

Figure 5.4: Graph of cumulative probability of finding a solution against generation for ADFs, SGP, the original SEF algorithm and the two linkage preserving SEF algorithms.

solutions are in some ways no more complex than the ADF equivalent.

5.4 Monotonic Algorithm

The last technique investigated was to make the fitness of the best individual in the cross-mutation algorithm runs become monotonic[5]. To do this the function reproduction was restricted. Only functions that were not used by the current best program were allowed to be replaced by new functions. This restriction cannot hold if the best program uses all the functions, to deal with this the restriction was waived if the current best program used more the 80% of the function population. 80% is an arbitrary figure chosen to be high enough that it should not interfere with most runs where a sensible number of functions are being called.

It was not possible due to time constraints to carry out as many trials for this system as for the others. Out of the eight trials that were completed only one converged to a successful solution. This suggests that the system is no better than the non-monotonic cross-mutation algorithm and may be slightly worse, but the lack of trials makes firm comparisons difficult.

The feared loss of parsimony in terms of the number of functions used occurred in at least some runs. In one run the number of functions used by the best program exceeded 80% of the function population (ie. 40 functions) causing the fitness to worsen. The average number of distinct functions called by the best individual in each generation almost doubled from 4.8 for the standard cross-mutation algorithm to 9.1 (the equivalent figure for the original SEF algorithm is 5.7), with the number of function calls also doubling from 8.6 to 16.7.

The general trend was for the fitness to decrease more rapidly than the

[5] The change was applied to the cross mutation algorithm as this was the most successful so far.

usual cross-mutation algorithm but then become stuck at around 2 or 3 indicating premature convergence. It is possible with a larger population size that premature convergence may be avoided.

5.5 Mean Fitness

The graph in figure 5.5 shows the mean fitness of the best individuals at each generation. A number of features stand out on this graph.

All five algorithms start roughly equal and remain so for the first few generations. This is as expected as at this stage the random generation of the initial populations will dominate. The first to break away is the standard GP, followed soon after by the original SEF algorithm which then fluctuates around a fitness level generally 0.5 higher than SGP. The cross mutation algorithm also fluctuates significantly, going down to the same level as ADFs at about twenty generations. Surprisingly despite being the most successful of the SEF algorithms tried it then finishes with the highest mean fitness at the end of the run.

The monotone SEF algorithm and ADFs show very similar results both having a mean fitness about 1.5 better than SGP from generation 10 onwards, with the monotone algorithm slightly better in general. This is unexpected as it did not perform as well at finding solutions. The monotone SEF initially descends the most rapidly reaching a mean fitness of two after about 10 generations. However it the flattens and does not improve further over the remainder of the trial length.

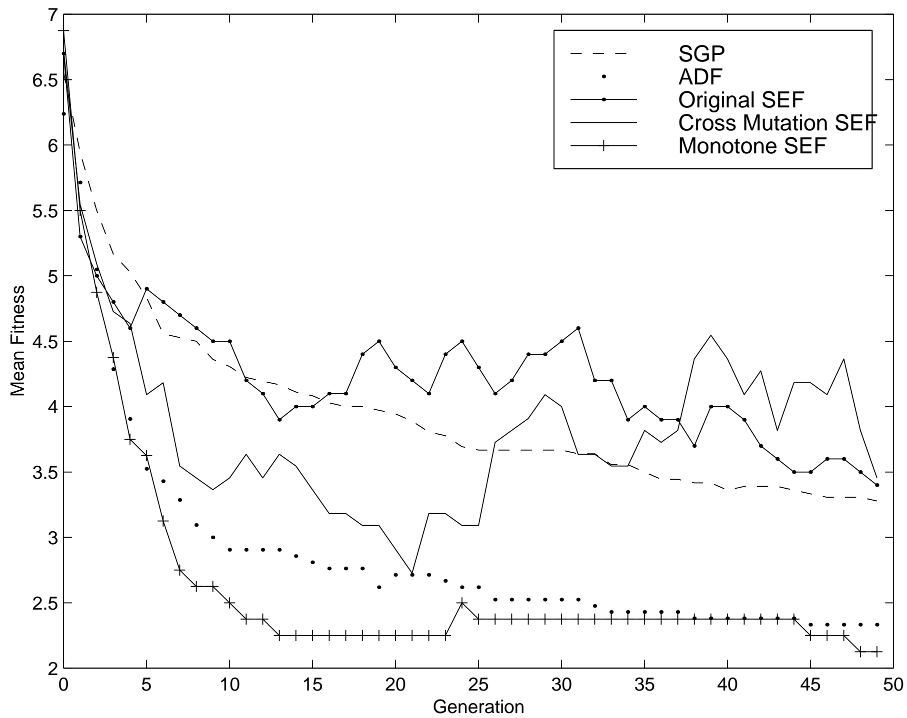

Figure 5.5: Graph of the mean fitness of the best individual in the population (averaged over the trials carried out) at each generation for standard GP, ADFs and three of the SEF algorithms (original, cross mutation and monotone).

Chapter 6

Discussion

6.1 Performance

Overall the results as given in chapter 5 are promising rather than successful. The original algorithm and the cross mutation both appear to do substantially better than standard GP but substantially worse than ADFs. Using the scaling that is indicated by the performance of the different sized populations for SGP and ADF suggests that with a population of 1000 the cross-mutation SEF technique would converge approximately 50% of the time on this problem.

However there is a caveat to this optimism. The SEF algorithms take more computational effort than the SGP and ADF methods. Although the same number of program reproductions occur the number of programs actually evaluated is substantially higher due to the changes to the functions they call. Thus there is an unfair advantage to SEFs on problems where the test is done on a fixed generational span and the problem is well understood as here. However in problems where the structure is not known this extra computational effort may not be so problematic, as the cost of running many unsuccessful SGP trials or a large number of ADF trials to find the correct

structure could well be higher.

The improvement of the performance when function to program linkage is increased by the cross-mutation operator suggests that this is one direction that could be explored further. It appears that close coupling is needed between a program and the functions it uses for them to work in harmony. This closeness is inherent in ADFs but has been lost to an extent in the new technique. The methods that improved this showed an improved performance and improving it may lead to performance rivalling ADFs with some additional benefits.

The results using the monotone algorithm suggest that protecting functions is not beneficial for the performance. The graph of mean fitness (figure 5.5) shows the typical pattern for premature convergence. This is that the fitness descends rapidly to a plateau on which it remains for the duration of the run. This occurs because not enough diversity is left in the population to find a better solution. However since increasing population size is one of the standard methods for countering premature convergence, it is likely that increasing the population to 1000 will yield bigger benefits for this algorithm than for the non-monotone variants.

The monotone algorithm clearly damages the parsimony (in terms of number of function calls). Parsimony is not a common concern in GP systems as it is hard to control structural simplicity due to the existence of introns, and sub-routine systems either have a hard limit on the number of functions available (eg ADFs) or do not worry about the number of calls (eg adaptive representation). However for a system such as the one investigated here it would be preferable to have some pressure to produce smaller solutions. The difficulty is incorporating this into the fitness in a way that does not damage the system's ability to solve the problem.

6.2 Structure Discovery

The discovery of a natural program structure to solve the problem occurs in two ways. These are the number of functions that are needed and the definitions of those functions (see section 3.5).

6.2.1 Number of Functions

In terms of the discovery of problem structure the results overall are mixed. Of the two most successful approaches (the original and cross-mutation) only one run out of all those attempted used an average of less than three functions in the best individual over the run and in terms of fitness this was one of the worst performing runs (the fitness was never less than 5, in most runs the lowest value over the run was about 2). The mean number of functions used by the best program at each generation with the cross mutation technique was 4.8 and with the original SEF algorithm was 5.7. For the monotone technique where the functions used by the best program were protected the mean rose to 9.1; this loss of parsimony was discussed in section 5.4.

However the use of multiple functions may not be as damaging to the structure discovery as it appears. Two distinct functions may actually be equivalent. (This is in fact guaranteed for the problem used here as we have a population of 50 functions and there are only $2^4 = 16$ distinct possible functions.) This is another example of the intron growth that occurs in genetic programming. There has been a large amount of research into introns (for references see Banzhaf et al. (1998, section 7.5)). These have shown that introns comprise around 50% of all code in the middle of GP runs and even more towards the end. These figures suggest that the mean number of functions used can be halved for a rough approximation of actual number of functions used.

The existence of introns is not surprising as they act as protection against

evolution for individuals as they propagate. However in terms of discovering what structure has been found introns confuse the picture. To clarify the structure the programs and the functions should be simplified at the end of the run and the introns removed before examining the resulting structure. However this can be a painful process. In addition judgement is sometimes required to determine what constitutes an intron and this could reassert the experimenters prejudices on the system.

The problem that remains from ADFs in this technique is the determination a priori of the function and terminal sets and the number of arguments for each population of evolving functions. This could to some extent be dealt with using multiple function populations but it is not clear how well this would work. The system does have the advantage over all the systems in chapter 2 other than ADFs that it can be set up for use of recursive evolved functions (subject to the constraints outlined in section 2.1).

6.2.2 Generally Useful Functions

Examining the definitions of the functions used on successful runs shows they do simplify to recognisable logic gates[1]. In one successfully converged run the six functions used by the successful solution simplified to the following two-input logic gates:- Arg1, (NOT XOR), NAND, Arg2, (NOT Arg1), (NOT XOR). Two of these functions (one of which was the most commonly called both in the solution and the entire population) are the NOT XOR function that was discussed in section 3.4 as being central to the problem (the other was the third most commonly called across the program population with the NAND function second). This suggests that the hope of finding generally useful functions is not in vain, and that the programs can evolve to use those

[1]However perhaps not too much should be read into the recognisability as with Arg1, Arg2, their complements, AND, OR, NAND, NOR, XOR and NOT XOR, T and Nil we have 12 of the 16 available functions.

functions.

For more conclusions to be drawn about the ability of the system to find generally useful sub-routines more work needs be done. This should include investigations on problems that use a richer function space. It could be interesting to take functions evolved in one run and try them as non-evolving functions in a separate run (probably on a slightly different problem). This would give a good indication of the utility of the evolved sub-routines for problems that cannot be easily analysed (eg non-functional problems).

6.3 Future Work

There are many options that can be experimented with for future work in the topic. The most urgent is perhaps to repeat the experiments detailed here, firstly with more trials, since 10 runs of a configuration is the bare minimum with which to start drawing inferences. Secondly the population size should be increased to 1000. Thirdly trials should be made on other problems including ones that are not so well understood and/or offer a greater wealth of functions. Should these continue to be promising then further refinements could be investigated, such as those suggested below.

- Investigating the effect of the parameters that the system offers. The first of these is size of function population, either absolute or relative to the program population.Other parameters include the fitness value for unused SEFs, and more subtly, the exact relative rates of evolution of the two populations.

- Solving multiple problems simultaneously with a single function population as outlined in section 3.5. Examples of problem families that could be susceptible to this approach include the various ant trail problems in Koza (1992, 1994) , or boolean parity problems of different sizes.

- Further improving the linkage between newly created functions and the programs that call their parents. This currently seems the approach that will have the greatest beneficial effect on the ability of the system to solve problems.

- Investigating different methods of protecting functions to give monotonic or near monotonic behaviour. This could include a fitness function that took the number of functions being used into account along with the correctness of the solution. Care is needed with such an approach to balance the requirements for a good solution and for a simple solution. The difficulty of such a balance will depend on the type of problem being tackled.

- Investigating the use of multiple function populations, A and B say, where the members of population A could call the members of population B (but not vice versa). Alternatively the functions in the different populations could have different structures (number of arguments, function and terminal sets).

Chapter 7

Conclusion

The SEF technique put forward in this dissertation has not worked as well as was hoped in terms of solving the test problem. A solution was found on about 10% of trials compared to around 30% for ADFs, however it outperformed standard GP which only succeeded on 3% of trials. The technique is also more computationally intensive than ADFs and SGP. The results of the mean fitness comparison suggest that the monotone algorithm may scale up to perform on a par with ADFs.

In terms of finding generally useful functions and problem structure the results are hopeful but not conclusive. The technique needs to be tried on problems with a greater richness of available subroutines for confirmation of the optimism felt.

The overall conclusion is therefore that the idea is promising but more work is needed to see if it can live up to its potential.

Appendix A

Glossary

ADF Automatically defined functions (see section 2.1).

ADM Automatically defined macros (see section 2.5).

Cross-mutation A mutation operator introduced to try and evaluate new functions more fairly. See section 5.3.

Generational Form of GP algorithm where the entire population is replaced at every iteration.

Introns Sections of code that if removed from a program would not alter the result produced. Examples include (NOT (NOT (..))) for boolean systems or $a = a + 0$ for arithmetic systems.

Monotonic Changing only in one direction. Monotonic decreasing is equivalent to not increasing (ie. it can includes periods of no change).

SANE Symbiotic Adaptive Neural Evolutions (see section 2.7).

SEF Symbiotic Evolving Functions, the system developed in this project.

SGP Standard genetic programming, as described in Koza (1992). No subroutines are used in the evolved programs.

Steady-state Form of GP algorithm where only the one individual is reproduced each iteration.

Subtree mutation The replacement of a subtree in an individual by a randomly generated subtree.

Terminal See *terminal node.*

Terminal node The constants and 0-arity functions in a program. (If the program is viewed as a tree then the terminal nodes form the leaves.)

Appendix B

The Interpreter

The interpreter that was used to execute the evolved programs.

```
(defun evaluate (tree)
  "Interpreter for the evolved programs."
  (if (eq *problem-type* 'sef)
    ;; Only need to do this the hard way if dealing with SEFs.
    (cond ((atom tree) (eval tree))
          (t
           (case (first tree)
             ;; Check for special forms
             (and (and (evaluate (second tree))
                       (evaluate (third tree))))
             (or (or (evaluate (second tree))
                     (evaluate (third tree))))
             (not (not (evaluate (second tree))))
             ;; Only other option for this problem is an SEF.
             (otherwise
              (let* ((sef-name (first tree))
                     (definition (gethash sef-name *sefs*)))
                (progv '(arg1 arg2)
                  (list (evaluate (second tree))
                        (evaluate (third tree)))
                  (eval definition)))))))
    ;; otherwise use eval.
    (eval tree)))
```

Bibliography

Abelson, H., Sussman, G. J., and Sussman, J. (1996). *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 2nd edition.

Angeline, P. and Kinnear, K., editors (1996). *Advances in Genetic Programming 2.* MIT Press, Cambridge, MA.

Angeline, P. J. and Pollack, J. B. (1992). The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Conference of the Cognitive Science Society*, Hillsdale, NJ. Lawrence Erlbaum Associates.

Angeline, P. J. and Pollack, J. B. (1993). Competitive environments evolve better solutions for complex tasks. In Forrest, S., editor, *Proceedings of the 5th International Conference on Genetic Algorithms, ICGA-93*, pages 264–270, University of Illinois at Urbana-Champaign. Morgan Kaufmann.

Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. (1998). *Genetic Programming An Introduction.* Morgan Kaufmann.

Graham, P. (1996). *ANSI Common Lisp.* Prentice Hall Series in Artificial Intelligence. Prentice Hall.

Holland, J. H. (1992). *Adaption in Natural and Artificial Systems.* MIT Press, 2nd edition.

Kinnear, Jr., K. E. (1994). Alternatives in automatic function definition: A comparison of performance. In Kinnear, Jr., K., editor, *Advances in*

Genetic Programming, chapter 6, pages 119–141. MIT Press, Cambridge, MA.

Koza, J. R. (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, MA, USA.

Koza, J. R. (1994). *Genetic Programming II, Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, MA, USA.

Koza, J. R. (1995). Evolving the architecture of a multipart program in genetic programming using architectue altering operations. In McDonnell, J. R., Reynolds, R. G., and Fogel, D. B., editors, *Evolutionary Programming IV Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 695–717, San Diego, CA, USA. MIT Press. http://www.genetic-programming.com/EP95.ps.

Moriarty, D. E. and Miikkulainen, R. (1998). Hierarchical evolution of neural networks. In *Proceedings of the 1998 IEEE Conference on Evolutionary Computation.* IEEE. http://www.cs.utexas.edu/users/nn/pages/publications/abstracts.html.

Moriaty, D. E. and Miikkulainen, R. (1996). Efficient reinforcement learning though symbiotic evolution. *Machine Learing*, (22):11–33. http://www.cs.utexas.edu/users/nn/pages/publications/abstracts.html.

Rosca, J. P. and Ballard, D. H. (1994). Learning by adapting representations in genetic programming. In *Proceeding of the 1994 IEEE World Congress on Computational Intelligence*, Orlando, FL. IEEE Press.

Rosca, J. P. and Ballard, D. H. (1996). Discovery of subroutines in genetic programming. In Angeline and Kinnear (1996), chapter 9.

Russell, S. J. and Norvig, P. (1995). *Artificial Intelligence: A Modern Approach.* Prentice Hall.

Spector, L. (1996). Simultaneous evolution of progams and their control structures. In Angeline and Kinnear (1996), chapter 7. http://hampshire.edu/ lasCCS/publications.html.

Turing, A. M. (1950). Computing machinery and intelligence. *Mind*, LIX(236):433–460. www.abelard.org/turpap/turpap.html.