View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by RMIT Research Repository

Utilising Restricted For-Loops in Genetic Programming

A thesis submitted for the degree of

Doctor of Philosophy

Xiang Li,

Master of Information Technology,

School of Computer Science and Information Technology,

Faculty of Applied Science,

Royal Melbourne Institute of Technology,

Melbourne, Victoria, Australia.

February 28, 2007

Declaration

I certify that:

1. Except where due acknowledgment has been made, the work is that of the candidate alone.
2. This work has not been submitted previously, in whole or in part, for any other academic award.
3. The content of the thesis is the result of work which has been carried out since official commencement date of the approved research program.

Xiang Li
School of Computer Science and Information Technology
Royal Melbourne Institute of Technology
February 28, 2007

Acknowledgments

I thank Vic Ciesielski, my supervisor, for his direction in this research. With his help, I successfully completed research experiments, conference paper writing and thesis writing. I found it was helpful to follow his style in writing, which is explicit questions followed by explicit steps in proofing. In addition, I feel grateful for his endless patience in helping me with English writing. I am not a native English speaker, and it is a painful and very time consuming task to understand my meaning in writing and get the thesis done. Vic helped to correct the thesis sentence by sentence and continuously encouraged me to complete the task even when I lost my temper. I really appreciate his effort and help.

I thank the people in the evolutionary computing and machine learning group, especially my second supervisor, XiaoDong, and my colleagues, Andrew, Andy and Anthony for their support in solving numerous technical issues and for sharing ideas and providing thoughtful arguments in my research period. Andy and XiaoDong also helped to provide teaching positions to me when my scholarship ran out. In this, I also thank Jenny Zhang who constantly supports me and provides paid work when she knows my situation.

I thank my current manager Martin Nightingale who gave me a job that I enjoy and makes efforts to encourage my study and use knowledge that I have gleaned from research skills in the workplace. I like working for this manager. I thank him and my previous manager John Gray for providing me time to complete this study.

I thank my colleagues, Evan, Stuart, Daniel, Michelle, Anthony, Jesse and Tooker in the SENSIS corporation for their happy lifestyle, for their endless jokes that made my daily life happy in this study period.

I thank a person I have never met, Michael Ciesielski, who helped me to proofread and make corrections in the writing of Chapters 2 and 6. I am curious about him.

I also thank Sara Ciesielski for the final proofreading of the thesis.

Finally, I thank my wife Ayeeda Xuan Liu whom I met and married during my study and who is my only relative in this country. She has limited knowledge in my research area, but tried hard to make my life easier by doing more housework, cheering me up when I was depressed and even helping me read a number of the thesis' chapters.

Contents

List of Figures

List of Tables

Abstract

Genetic programming is an approach that utilises the power of evolution to allow computers to evolve programs with little human involvement. It has demonstrated its usefulness in solving many experimental problems as well as many real world problems. However, it suffers from weaknesses in using repetitions effectively. While loops are natural components of most programming languages and appear in every reasonably-sized application, they are rarely used in genetic programming. Extending the power of genetic programming by encouraging more use of loops will bridge the gap between the current state-of-the-art in programs evolved with genetic programming and those written by humans, and improve this automatic programming method.

The goal of the work is to investigate a number of restricted looping constructs in which infinite loops are not possible and to determine whether any significant benefits can be obtained with these restricted loops. Possible benefits include: Solving problems which cannot be solved without loops, evolving smaller sized solutions which can be more easily understood by human programmers and solving existing problems quicker by using fewer evaluations.

In this thesis, a number of explicit restricted loop formats were formulated and tested on the Santa Fe ant problem, a modified ant problem, a sorting problem, a visit-every-square problem and a difficult object classification problem. A maximum number of iterations based on domain knowledge was used to avoid the infinite iteration problem. The experimental results showed that these explicit loops can be successfully used in genetic programming. The evolutionary process can decide when, where and how to use them. Runs with these loops tended to generate smaller sized solutions in fewer evaluations. Solutions with loops were found to some problems that could not be solved without loops.

From these experimental problems, the modified ant problem and the visit-every-square problem were selected to analyse differences between using and not using loops with respect to the search spaces, the patterns captured by genetic programming and the sensitivity to changes

in the maximum number of iterations on CPU time. The analysis of the search spaces found that there were more fitter programs within a limited tree depth for programs with loops. To solve the same problem without loops required a larger tree depth and this exponentially increases the number of possible programs and may decrease the chance of finding a good solution. The analysis of the patterns captured found that runs with loops captured repetitive patterns of the problem domain and repeated them to improve the fitness. The analysis of the effect of different values of maximum number of iterations showed that CPU time per evaluation increased as the maximum number of iterations increased. However, solutions were found in fewer evaluations. There was a large range of values for maximum number of iterations for which the overall CPU time was lower. Good choices for maximum number of iterations could be found from domain knowledge.

Overall, the results and analysis have established that there are significant benefits in using loops in genetic programming. Restricted loops can avoid the difficulties of evolving consistent programs and the infinite iterations problem. Researchers and practioners of genetic programming should not be afraid of loops.

Chapter 1

Introduction

1.1 Motivation

Loops are powerful constructs of programming languages. They provide a mechanism for repeated execution of a sequence of instructions. The task of most programs is usually to execute a code segment a number of times by a loop. Variables within the loop body may be updated sequentially during the execution.

There are many different ways of writing loop statements in programming languages, but there are basically only two different types of loops. In one type of loop, it is not known how many times the loop body will be executed when the loop body is entered, and a condition needs to be specified to exit the loop. A loop that reads data until end of a file is a loop of this kind. We call such loops unbounded loops. In these kinds of loops infinite loops are possible. In the other type of loop, the number of times the body will be executed is known at entry. A control variable is given start and end values and the loop will iterate from the start to the end, thus infinite loops are not possible. We call this type bounded loops. In many programming languages a *for* statement is used to code this type of loop, and in this thesis we refer to it as a *for-loop*.

Recursion is another method to implement iteration. However, this thesis is concerned primarily with bounded loops and recursion is not the focus.

There are various ways to specify bounded loops in different programming languages and some of these are shown in Figures 1.1 to 1.4. In Pascal, the statement inside a bounded loop is executed a number of times depending on the control condition (see Figure 1.1). The value of

the control variable, *control_variables*, cannot be changed inside the loop. In Fortran, a bounded loop is specified by a DO statement as shown in Figure 1.2. The loop body is executed once for each value of the control variable. In LISP, a bounded loop is specified by a *loop* statement which iterates the body from a start value to an end value (see Figure 1.3). As shown in example 2, this form can also be used to step through the items in a list.

A full specification of a bounded loop consists of four components. They are the initialization branch that is used to assign starting values to variables, the condition branch that specifies the termination criteria, the updating branch that provides the way to change the control variable and the loop body which contains the code that is repeatedly executed (see Figure 1.4).

Higher order iteration constructs are also available in LISP and Java. In LISP, loops can use mapping functions such as *mapcar* (see Figure 1.5). $Mapcar$ applies the function given in its first argument successively to the list elements in the other arguments and generates the results in a new list. In Java, a for statement can be used to iterate through each element in an array or a collection (see Figure 1.6).

Genetic programming (GP) is a newly developing field in artificial intelligence and it allows the computer to "learn how to solve problems without being explicitly programmed" [124, p35]. Darwinian principles of natural selection and recombination are used to evolve a population of programs towards an effective solution to a specific problem. The genetic programming technique has been successfully applied to a large number of tasks including robot control [34], financial trend prediction [213], electronic circuit design [230], image classification [220] and object detection [260].

Despite the success of genetic programming on different useful problems, loops have been rarely utilised and the inclusion of loops in evolved programs has not been systematically studied. There have been some attempts in using loops to solve a number of "toy" problems, but compared to the versatility and flexibility of loop usage in other programming languages, as described earlier, loops are hardly used in GP.

The reasons for this are:

- Firstly, loops are hard to evolve.

  It is necessary to evolve the start point, the end point and the body and to make sure they are consistent. For example, in some kinds of loops, an index variable must appear in the body of the loop. In conventional programming languages as described earlier, this task is

Format	Example 1	Example 2

for control_variable [to|downto] final_value
do
[begin
program_statements
end];
	for i := 10 downto 1
	do
	write(i+1);
		for j:=1 to 5 do
		begin
		s:=s+j;
		write(j)
		end;

Figure 1.1: A typical bounded loop in Pascal

Format	Example 1	Example 2
DO control_variable, final_value, [increment] instruction block END DO	DO i=1, 5 PRINT *,i END DO	DO i=2, 10, 2 PRINT *,i END DO

Figure 1.2: A typical bounded loop in Fortran

Format	Example 1	Example 2
loop for variable [type_spec] [expr] program_statements	(loop for n from 1 to 100 (sum n))	(loop for i in (a b c d) do (print i))

Figure 1.3: A typical bounded loop in LISP

Format	Example 1	Example 2

for(control_variables; condition; increment)
program-statements
	for(i=0; i<10; i++)
	printf("%d", i);
		for(i=3,y=0,total=0;
		i<y-3;i++,y+=2)
		total=y+i;

Figure 1.4: A typical bounded loop in C or Java

Format	Example 1	Example 2
mapcar function car-lists	(mapcar #'+ '(1 2 3 4 5) '(10 20 30 40 50))	(mapcar #'cons '(a b c) '(1 2 3))

Figure 1.5: A higher level bounded loop in LISP

Format	Example 1	Example 2
for(element : elements) element-handle-statements	for(int i : int[] a) result += i	for(Book i: Books) totalPrice += i.price()

Figure 1.6: A higher level bounded loop in Java

the responsibility of the programmer, while the evolutionary process in GP can only take a trial and error approach to generate consistent programs. This situation can be illustrated in example 1 of Figure 1.1. In Pascal, the control variables cannot be changed inside the loop but an evolutionary process could generate a statement like $i := 50$ in the loop body. A mechanism needs to be formulated to avoid this kind of problem.

- Secondly, programs with loops generally take longer to evaluate and some mechanism must be implemented for dealing with infinite loops.

  Genetic programming uses population-based search methods. Initially, a group of possible solutions are generated and they are evaluated and selected for mating or mutating based on their fitness. Evaluating a group of programs with loops for a number of generations is potentially high in computation cost. Furthermore, the evolution may generate some combinations which contain infinite loops. The design of the GP process must take this into consideration and restrict the number of iterations while still allowing the necessary number of repetitions to solve the problem.

- Thirdly, there is a large class of useful problems which can be solved without loops.

  At the web site by *http://www.genetic-programming.com* (visited on May 6th, 2006), there are 36 instances in which genetic programming has produced human-competitive results on a range of difficult problems. Only one of these uses loops. Many problems, for example symbolic regression, have been solved without loops.

- Fourthly, it is often possible to put the looping behaviour in the environment or into a terminal and execute the repetition implicitly.

  Currently, loops in genetic programming are mostly implemented in an implicit way. There are two ways of doing this. In the first way, loops are put into terminals. For example, in the robo-soccer problem [45], the dash-towards-the-ball function moves the robot forward until it reaches the ball. In the second way, loops are embedded in the environment. For example, in the Santa-Fe Ant problem [123], the evolved program is executed again and again until the task is finished or some maximum number of executions is reached.

Putting explicit loops into genetic programming could greatly enhance its power and could lead to the solution of some significant experimental as well as real world problems which cannot

be solved without loops. Moving looping behaviour from the environment or the terminals into the evolved program could have a number of benefits including: (1) It could enhance the understanding of the solution by explicitly showing the complete code. (2) It could improve the efficiency of the program by pruning out the elements where repeated execution cannot make a difference. (3) It could optimise the initialization, the condition and the updating branches in addition to the loop body by using fitness-driven crossover and mutation in the genetic search.

There are further reasons for using loops in genetic programming. There is a large class of problems with obvious looping characteristics that have not been tackled by GP. For example, problems involving large vectors or arrays, like a generalised sorting algorithm, need loops to access every element and reduce the length of the evolved code. Image detection and classification tasks also involve looping behaviours because pixels of images are usually stored in arrays. The classfiers evolved by GP are generally quick in decision making. However, evolving classifiers using GP is slow. Improving the GP search process to find an acceptable classifier is in strong demand. Furthermore, nearly every reasonable human written application has loops. Loops help in the design of concise and comprehensible algorithms. In order to increase the applicability of genetic programming, there is a need to narrow the gap between the wide use of loops in most other programming languages and the very limited use of loops in genetic programming.

This thesis is based on the premise that it is possible to construct restricted loops in a way that avoids the problems of infinite loops, but still leads to being able to solve problems that cannot be solved without loops. The work will show how loops can be utilised to take advantage of the repetitive patterns in problems without worrying about the infinite loop problem.

The recent decrease in the cost of computing cycles enables investigations of genetic programming which require more computation resources, such as programs with loops. Studying loops was not feasible in the past because generating large populations of programs with loops and evaluating them require very long elapsed times if the CPU speed is slow. Also, to obtain statistically significant results, experiments need to be repeated many times.

Today, with the increasing CPU speed, availability of large scale parallel computing as well as many more real world applications adopting genetic programming as the solution methodology, it is possible and necessary to systematically study the use of explicit loops. GP with explicit loops has the potential to utilise the allocated memory more efficiently by minimising duplication of code. For example, if the task is to direct a robot to enter into 10 locked rooms, the GP program

needs to evolve a solution either with 10 identical sequences of actions which are opening the door, entering the room and walking to another door, or to have a looping construct which captures one sequence of these actions and repeats them 10 times. Furthermore, if the required sequence of actions is difficult to discover, the probability of success will be higher with loops as this sequence needs to be discovered only once. Without loops, this discovery needs to be made ten times.

1.2 Terminology

The key terms used in this thesis are defined and described below:

- **Bounded loop:** refers to a type of loop in which the number of times the body will be executed is known at entry. A control variable is given start and end values and the loop will iterate from the start to the end, thus infinite loops are not possible.
- **Unbounded loop:** refers to a type of loop in which the number of times the body will be executed is not known at entry and infinite loops are possible.
- **Explicit loop:** refers to a bounded loop function which is explicitly denoted by *loop* or *for-loop* in the evolved program and the branches of the loop are evolved during the evolution.
- **Implicit loop:** refers to a loop facility which exists outside the evolved program, either the evolved program as a whole is executed a number of times or the looping is embedded in one or more terminals.
- **For-Loop:** is a generic term for bounded loops in a variety of formats.
- **For-Loop-?:** refers to a specific explicit loop format developed for a specific task and used in the experiments. The "?" is replaced by a number indicating different loop formats.
- **Restricted loop:** refers to an explicit loop that does not allow the branch which indicates number of iterations to have mathematical functions and nested loops, for example, (ForLoop numberOfIteration body).

- **Unrestricted loop:** refers to an explicit loop that allows the branch which indicates number of iterations to have mathematical functions and nested loops, for example, (ForLoop (5 + (ForLoop 4 body)) body).
- **Run:** refers to a completed evolutionary process which is either terminated by finding a solution or by reaching a maximum number of generations.
- **Successful run:** refers to a completed evolutionary process that generates a satisfactory solution.
- **Unsuccessful run:** refers to a completed evolutionary process that does not generate a satisfactory solution when the maximum number of generations is reached.
- **Mean best program fitness graph:** refers to a graph in which the *x-axis* indicates generations or evaluations and the *y-axis* indicates the fitness values. Multiple runs have been conducted for each experiment. The fitness of the best program for each run at each generation is recorded. The mean best fitness is the average fitness value of these best programs.

1.3 Goals of the Thesis

The broad aim of the thesis is to study how bounded loops, that is, a limited form of loops in which infinite loops are not possible, can be incorporated into tree-based genetic programming and to determine whether incorporating such loops delivers any major benefits.

1.3.1 Research Questions

1. How can we restrict the syntax and semantics of *for-loops* in a way that avoids problems of infinite loops and still provides useful benefits for genetic programming?

   The benefits could be higher success ratios, fewer evaluations, reduced overall CPU time for the evolutionary process and the evolution of smaller sized understandable solutions.

   The expectations are that the incorporation of the loops would decrease the number of evaluations for getting a solution for a range of problems and help GP to evolve smaller sized solutions. An evolved program with loops could require more CPU time, but the

chance of reusing repetitive patterns could accelerate the evolutionary process and the problem could be solved in fewer evaluations thus minimising the overall time.

2. Can GP with *for-loops* solve some problems that cannot be solved or are very difficult to solve without explicit loops?

   The expectation is that there will be such problems.

3. Can *for-loops* be used in a difficult object classification problem with similar performance gains to those achieved on relatively simple artificial problems?

   Initial experimental work showed that *for-loops* were beneficial for a number of "toy" problems. This question is a further exploration of a much harder problem in which repetition is not necessary for a solution. We expect for-loops could be still used, but are not sure whether they could deliver similar benefits to those in previous artificial problems.

4. How can the performance gains from using *for-loops* be explained?

   Use of *for-loops* has resulted in major benefits for a number of different problems.

   We expect that the performance gains can be explained through analysis of search spaces, of the patterns captured by the loops and of the settings for the maximum number of iterations.

1.4 Contributions

This is the first (PhD) thesis that presents a systematic analysis of the usage of explicit loops in genetic programming and provides a methodology that users can apply to different problems.

The thesis makes the following major contributions:

1. A methodology for incorporating explicit loops into tree-based genetic programming.

   The method shows how to formulate terminals and functions for looping constructs, how to restrict syntax and semantics by domain knowledge so that there will be no infinite loop problems and how to favour programs with fewer loops so that the evolved programs are small and comprehensible.

   This work was published in:

Vic Ciesielski and Xiang Li. Experiments with explicit for-loops in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 494-501. IEEE Press, July 2004.

2. A first demonstration that explicit looping constructs can be used to solve image classification problems.

   The study has demonstrated that GP with loops is not just for "toy" problems. GP with loops can be used to solve a more difficult image classification problem. On this more difficult problem, GP with loops had success rate around 25/100, while GP without loops had success rate around 5/100. Furthermore, the classifiers with loops were more robust and general, because they used sequences or areas of pixels while the classifiers without loops were more fragile because they used individual pixels scattered through the images.

   This work was primary published in:

   Xiang Li and Vic Ciesielski. Using loops in genetic programming for a two class binary image classification problem. In *Proceedings of the Australian Joint Conference on Artificial Intelligence (AI-2004)*, pages 898-909. Springer-Verlag, December 2004.

   Part of this work was published in:

   Vic Ciesielski and Xiang Li. Pyramid search: Finding solutions for deceptive problems quickly in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2003)*, pages 936-943. IEEE Press, December 2003.

3. A systematic analysis of the search spaces and evolved patterns which reveals why evolving programs with loops is beneficial.

   The analysis compares the search spaces for programs with and without loops. The results have shown that, with loops, there is a larger number of solutions and a larger number of small and fit programs.

   The analysis examines the patterns evolved during the evolution for programs with and without loops. The results have shown that patterns captured in the bodies of the loops are reflective of repetitive patterns in the problem domain, and repeated execution of the patterns is clearly associated with improvement in fitness.

   The analysis also looks into the relationship between the maximum number of iterations

and the overall CPU time. The results have shown that the increase in CPU time for each evalution for programs with loops can be offset by getting a solution in fewer evaluations and this decreases the overall evalution CPU time.

This work was primary published in:

Xiang Li and Vic Ciesielski. An analysis of explicit loops in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2005)*, pages 2522-2539. IEEE Press, September 2005.

Part of this work was published in:

Vic Ciesielski and Xiang Li. Analysis of genetic programming runs. In *Proceedings of the Asia-Pacific Workshop on Genetic Programming (ASPGP-2004)*, December 2004.

4. The results of the thesis encourage the use of loops in genetic programming.

   Overall, GP users tend to avoid the use of loops. The results of the thesis have shown that there is no need to be scared of loops. The kinds of loops described in this thesis are easily formulated, avoid the problem of infinite loops, use domain knowledge to constrain loop bounds and provide significant benefits in terms of success ratio and evolution times.

1.5 Structure of the Thesis

After the introduction, the thesis is organised as follows.

Chapter 2 presents a detailed review of the literature to date in the domain of evolutionary computation focusing on genetic programming, loops in genetic programming and image classification that have been successfully solved by genetic programming.

Chapter 3 establishes that programs with explicit loops can be evolved successfully for a number of GP problems.

Chapter 4 extends the work of chapter 3 and shows how loops can be utilised for solving two problems which are significantly difficult - a centered and a shifted binary image classification problem.

Chapter 5 presents an analysis of why explicit loops are helpful for evolution in terms of the search spaces, sensible patterns captured by the loop constructs and the computation time.

Finally, chapter 6 summarises the findings, draws conclusions and suggests future work.

Chapter 2

Literature Review

This chapter reviews the background of the investigation presented in the thesis and the relevant studies. Figure 2.1 shows the overall structure. The survey starts with an overview of evolutionary computation, which provides the basis for genetic programming (GP). After a brief overview, the review examines tree-based genetic programming — the experimental tool of this work. The examination covers GP representation, refinements of GP methods, key evolutionary processes, GP theory studies and current research trends. After this, the survey focuses on how implicit or explicit repetitions have been formulated in GP. In this section, the weaknesses of previous approaches are highlighted and the section presents the potential benefits of making use of explicit loops. This section also describes the test problems used in this work. Because the work has applied looping constructs to a difficult classification problem to show that explicit loops are useful not just for "toy" problems, the required features of classifiers and previous work on evolving GP classifiers are briefly reviewed.

2.1 Overview of Evolutionary Computation

Genetic programming is a sub-branch of evolutionary computation. This section reviews the general concepts, major steps and some of the major algorithms in evolutionary computation.

The origin of evolutionary computation can be traced from the late 1950's [15, 225]. Researchers in the US and Europe mimicked mechanisms in biological evolution in order to develop algorithms for problems of adaptation and optimization. At that time, most of the work was theoretical and there was little experimental work. The reasons for this include the fact that the

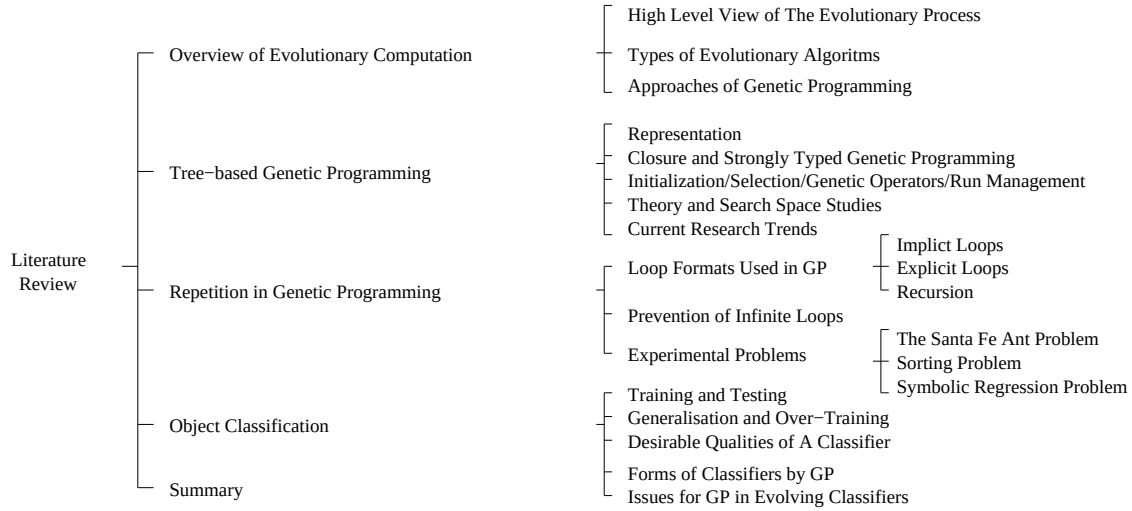


Figure 2.1: Overall structure of the literature review

computing power at that time was limited, there was incomplete knowledge on how to apply the theory to real world problems, and many optimisation problems at that time could be solved by tailored solutions developed within acceptable time.

Evolutionary computation is based on the biological principle of evolution ("survival of the fittest" [169]) and is generally used to solve complex optimisation problems. Such problems normally contain a huge search space, have complex parameter interactions and the potential solutions cannot be enumerated in a reasonable time. 'Smart' search algorithms need to be designed for these tasks. However evolutionary computation applications may not guarantee finding the best solution - the global optimum. For most practical problems, a good solution is acceptable without necessarily being the absolute optimum [172].

In this section, we present a high-level view of the evolutionary process and briefly describe five major evolutionary approaches: evolutionary programming (EP), evolutionary strategies (ES), genetic algorithms (GA), particle swarm optimisation (PSO), and genetic programming (GP).

2.1.1 High-Level View of The Evolutionary Process

The search in evolutionary computation is population based and involves four major steps: initialization, evaluation, selection and generation of descendants. The last three steps are iterated until the problem is solved or some other conditions are satisfied.

- Initialising the population

  The evolutionary process starts by creating a population of potential solutions to the problem. The initial population is normally randomly generated and the objective is to distribute individuals broadly in the search space (see Section 2.2.7, page 46). However, existing good solutions can also be used in the starting population.

- Evaluating individuals

  Each individual is evaluated against a fitness function according to the environment and a fitness value is assigned to describe how well an individual performs. The evaluated individuals can either have numeric values to reflect their fitness or just be ranked based on their performance.

- Selecting the fitter ones

  In the selection process, the fitter individuals are favoured for producing descendants. This is to simulate the biological evolution process in that the fitter individuals survive and have a greater chance to provide descendants that are fitter than their parents.

- Generating the descendants

  The selected individuals undergo a number of transformations, based on biological processes, to form the new population. In the new population, some individuals are generated by genetic recombination (crossover) in that parts of the parents are swapped and some by mutation in that some parts of the parents are varied.

The last three steps are iterated. After the descendants are generated, the new population is evaluated again and checked against the fitness function. Each iteration is called a *generation* [172]. In evaluation, if the solution is found, the whole process stops; if not, the process will continue until some maximum number of generations is reached or some other condition is satisfied. Figure 2.2 shows the whole process.

An evolutionary technique is defined by the data structure used to represent an individual and the genetic operators. For example, a binary string representation, binary crossover and mutation define the basic binary genetic algorithm, while a real-valued representation and mutation operators based on normal distributions define evolutionary strategies.

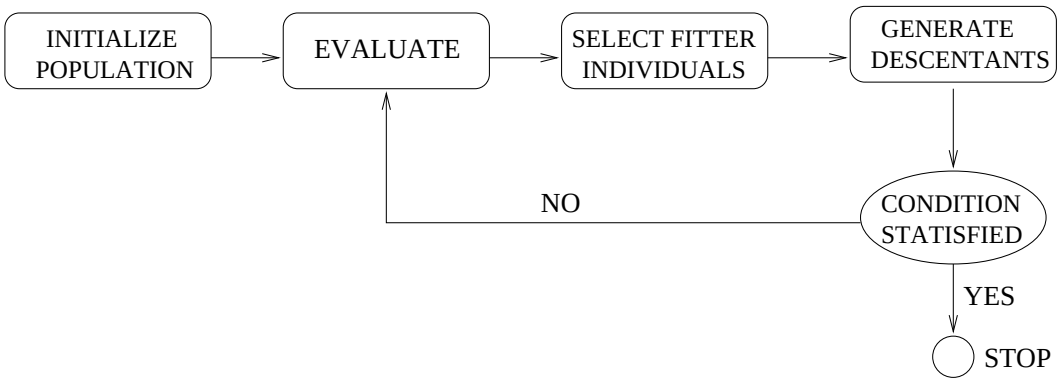


Figure 2.2: The overall process of evolutionary computation

2.1.2 Types of Evolutionary Algorithms

There are five main kinds of evolutionary algorithms. They are evolutionary strategies (ES), evolutionary programming (EP), particle swarm optimization (PSO), genetic algorithms (GA) and genetic programming (GP). There are also a number of other evolutionary algorithms, such as differential evolution [205] ant colony [60], etc. The first four are briefly described in this section and GP is described in depth in the following section.

Evolutionary Strategies and Evolutionary Programming

Rechenberg introduced evolutionary strategies (ES) in the 1960s [172]. In his formulation, a population consisted of two individuals - a parent and a child mutated from the parent. They were represented by a set of real-valued parameters. To optimise the parameters, the fitter of the two was selected to be the parent for the next generation. The fitter individual was mutated by incrementing or decrementing a real value according to a given distribution. The distribution of the parameters was also encoded as part of each individual and evolved with the parameters to be optimised. Evolutionary strategies were further developed by Schwefel [215] and the theory and application of evolutionary strategies have remained an active area of research [15].

Evolutionary programming (EP) was developed by Fogel [75] and extended by Burgin and others [15, 73]. In evolutionary programming, the potential solutions are represented by finite state machines. A finite state machine transforms a sequence of input symbols into a sequence of output symbols based on a finite set of states and transition rules. Mutation was the only source of variation in evolutionary programming. The fitness of a finite state machine is measured on the basis of the machine's prediction capability. A broader formulation of evolutionary programming

remains an area of research [72].

Genetic Algorithm

The genetic algorithm (GA) was first introduced and investigated by Holland [96] in the early 1960s. It is a population-based model that performs a multi-directional search by using recombination and mutation operators to exchange and update information encoded in an individual. The search is directed by fitness selection [253].

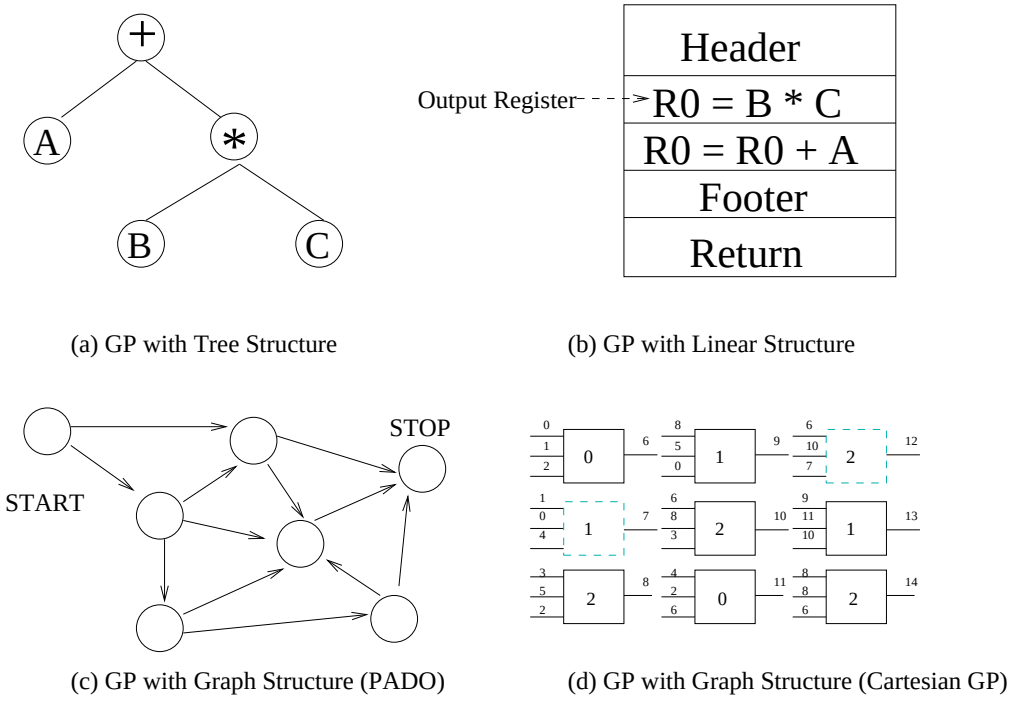
In the classic genetic algorithm, individuals are represented by binary strings. A binary string encodes all the genetic information needed to build an actual solution. However, this representation limits the precision required to encode a real number in a finite set of bits, thus it may not capture problem-specific knowledge. The real-valued genetic algorithm [106, 168] avoids this problem and incorporates natural, real valued data structures.

Genetic algorithms have been widely used in many real-world applications such as software design [233], pattern recognition and various engineering problems [168]. They remain an active area of research.

Particle Swarm Optimization

Particle Swarm Optimisation (PSO) was inspired by the flocking behaviour of birds and was proposed by Kennedy and Eberhart [114] in 1995. It is a stochastic optimization algorithm and is population-based. The population is called a *swarm*. The individuals are referred to as *particles*. The particles contain search space information and steps of the search. Each particle moves with an adaptable velocity within the search space and retains the best position it encountered in its memory. The whole population is able to respond to a good position found by individuals. Through the exploitation of the particles and the swarm's memory, particle swarm optimization is able to find a global best solution. It is claimed to have fast convergence and not be influenced by the number of peaks and dimensions. It delivers good results in static, noisy and continuously changing environments [115].

In recent years, particle swarm optimization has gained increasing popularity because of its effectiveness in performing difficult optimization tasks. It has been applied to multi-objective problems [70, 146], minimax problems [129, 144], integer programming problems [143] and numerous engineering applications [1, 62, 262]. It is an active area of research.



(a) GP with Tree Structure

(b) GP with Linear Structure

(c) GP with Graph Structure (PADO)

(d) GP with Graph Structure (Cartesian GP)

Figure 2.3: Examples of genetic programming structure

2.1.3 Approaches of Genetic Programming

The goal of genetic programming is to evolve computer programs automatically to solve problems. People have achieved this in a lot of different ways and used different methods, languages and formulations for representing programs. This section gives a brief description of these different genetic programming approaches.

Genetic programming (GP) is a type of genetic algorithm devised by Cramer [48] and popularised by Koza [123]. It provides hierarchical structure and dynamic variability. The programs evolved by GP can be executed directly and normally complete the whole task. In its original formulation, individuals are represented as LISP S-expression lists [123] and evaluated by a LISP interpreter. An S-expression can be visualised as a tree. In later work, different program representations have been investigated. They are linear structure, graph structure and two hybrid structures, linear-graph and linear-tree structure. Grammatical evolution is another way to generate programs by evolution and will be briefly discussed at the end of this section.

Tree-based GP

Tree-based GP is a commonly used form and there are a number of tree-based GP implementations publicly available [39, 131, 264].

In tree-based genetic programming, individuals are built recursively from a set of functions $\{f_1, f_2, f_3, ..., f_n\}$ and a set of terminals $\{t_1, t_2, t_3, ..., t_n\}$. Each function can have a number of arguments. Figure 2.3a shows an example of an individual in tree-based GP. It is composed of functions {+, *} and terminals {A, B, C}. Functions are located at inner nodes and terminals are at the leaves. "+" is the root node in this example. Terminals {A, B, C} will be assigned values from the environment in evaluation. This thesis uses tree-based genetic programming and this approach is described in detail in Section 2.2.

There is an approach in tree-based GP that uses a grammar as part of the process of evolution programs and is called grammar guided genetic programming.

Grammar Guided Genetic Programming

Grammar guided genetic programming is a form of tree-based genetic programming developed by Whigham in 1995 [248]. It is a way to guide the search process by utilising grammar rules.

In grammar guided genetic programming, a context-free grammar is used to define the structure of individual programs and to direct crossover and mutation operations. The derivation trees [248, 256] are used to determine production rules during the evolution. Crossover between two programs can only be carried out by swapping sub-derivation trees that start with inner nodes labeled by the same non-terminal symbol [94]. In mutation, a sub-derivation tree is replaced by a randomly generated sub-derivation tree from the same non-terminal symbol.

The main advantages of grammar guided genetic programming are that it allows a clear statement of inductive bias and controls over typing by the use of the grammar; it enables incremental learning [186]. Grammar guided genetic programming has demonstrated positive results on a 6-multiplexer problem [248] and a range of other problems [77, 94, 181].

The main disadvantages of grammar guided genetic programming are its bias and restrictions in the search space. A good grammar rule may improve the search while a bad one may significantly decrease the chance of finding good solutions.

Overall, the main advantages of tree-based genetic programming are that the generated individuals are easy to interpret and it is easy to perform crossover operations in a tree representation.

The individuals are interpreted as LISP programs and the crossover operations can be easily achieved by swapping parts of the trees.

The main disadvantage of tree-based genetic programming is that extra memory is needed to maintain the tree structure and a mechanism is needed to decide the points for crossover and mutation for individuals in a tree representation [200].

Linear GP

Linear GP [27] was first introduced by Cramer [48] and further developed by Banzhaf [16, 28]. In linear GP, programs are represented by bit strings which are lines of code for register machines. It can be stack-based [195] in which the program instruction takes its arguments from a stack, performs the calculation and returns the results back to the stack. There is a commercial package, AIMGP [184], for linear GP.

Figure 2.3b shows an example of a program with linear structure. It contains four parts: *header, body, footer* and *return* instruction. *Header* and *footer* do not participate in the evolution. *R0, A, B, C* are registers. *R0* is the output register. In evaluation, the instructions are executed sequentially and the value of *R0* is returned at the end of execution. The example in Figure 2.3b is performing the same calculation as the program represented by the tree structure shown in Figure 2.3a.

Brameier [27] gave a detailed description of linear genetic programming in his thesis. He claimed that programs with a linear representation are more suitable to be varied in small steps than in a tree structure, that programs in the linear structure are generally more compact due to multiple usage of register contents and an implicit parsimony pressure by the structurally non-effective code. He suggested that introducing and analysing new genetic operators in linear genetic programming environments, and testing programs that have a hybrid structure (like linear-with-graph) would be future research directions for linear genetic programming.

The advantages of linear GP are that the evolved programs can be binary machine code and are executed directly without interpretation during the fitness evaluation [183]. Also linear GP is uniform in node selection for mutation and crossover due to its representation [27]. It is claimed to be faster in evaluation than tree-based GP.

The disadvantages of linear GP are: the execution of linear GP programs is generally sequential, thus more work is needed in order to find a way to implement repetitions easily in this

linear structure; programs represented by binary machine code cannot be understood as easily as those in tree-based GP.

Graph-based GP

Graph-based GP was first used in the PADO system [237], a system developed by Teller and Veloso for object recognition. In graph-based GP, a program is constructed as an arbitrary directed graph of nodes.

In the PADO implementation, each node has two parts: an *action* and a *branch-decision*. The *action* does the calculation and the branch-decision directs the the path for execution. Each node possesses its own private stack-based memory and can also access a globally defined indexed memory. Figure 2.3c shows a program in this system. The program contains 7 nodes in a directed graph with arcs going out from each node. *Start* and *end* are special nodes in the program. The *start* node is always the first node to be executed when a program begins and the program halts when the *stop* node is executed. Inside an individual, some nodes may be subprogram calling nodes which can call private protected subprograms; some nodes may be library subprogram calling nodes which can call subprograms that are publicly accessible.

A later development of graph based GP is called Cartesian genetic programming (CGP) [171]. In Cartesian genetic programming, nodes are connected in a graph in a Cartesian coordinate system. A program is defined as a set of inputs, input connections, outputs, output connections and functions. Figure 2.3d shows a program in the Cartesian genetic programming system. In Miller and Thomson's implementation [171], the connectivity of the nodes is feed-forward but can be extended to allow more complicated structure like loops. Their work found that the Cartesian genetic programming representation has very large number of redundant nodes and this improves the balance between crossover and mutation during evolution, thus improving the search. Cartesian genetic programming is very effective for some boolean function learning [170] and has been used for image processing tasks [175].

The main advantage of the graph-based GP is that the graph representation allows a single individual to have multiple execution paths, and thus is easier to handle different situations.

The main disadvantage of graph-based GP is the extra complexity to maintain and manipulate the graph structure and to specify the possible arcs. It needs special crossover and mutation mechanisms. These make the implementation of graph-based GP difficult [17, p266] [235].

Hybrid Structures

In a recent GP structure development, Kantschik and Banzhaf proposed a linear graph structure for GP programs [109]. It was tested in two symbolic regression problems and gave good results with small population sizes of 10 and 100. Kantschik and Banzhaf proposed a linear-tree structure [108] which applied to an automatic quantum circuit design problem. They found it was suitable and achieved more "degrees of freedom". However, its scalability requires further studies.

The main advantages of these hybrid structures are that they extend GP representations and help to evolve solutions that are natural to the problems.

The disadvantages of these hybrid structures are that they are new GP representations and their effects for general problem solving and their scalability for harder problems require further research.

Grammatical Evolution

Grammatical evolution was developed by Ryan, Collins and O'Neill in 1998 [187, 211] and is a system that utilises Backus Naur Form (BNF) grammar definition as a mapping tool to evolve genetic programs. It can be used to automatically generate programs in any language.

Grammatical evolution can be viewed as a variation of genetic algorithms and linear genetic programming utilising Backus Naur Form in code mapping. Backus Naur Form is a meta-syntax to express context-free grammars and is widely used as a notation for computer programming languages. In Backus Naur Form, a grammar is represented by a tuple N, T, P, S, where N is the set of non-terminals, T the set of terminals, P a set of production rules that maps the elements of N to T, and S is a start symbol which is a member of N. This grammar is utilised to map the evolved linear code into programs. In the evolutionary process, a genetic algorithm is used to control the choice of production rules that are expressed by numbers which can be encoded as binary bits. A sample individual presented by Ryan et al. is 202|203| 17| 3|109|215|104| 30 [211] and it can be mapped to a concrete program based on the content of the mapping table described by Backus Naur Form.

Grammatical evolution has the ability to evolve any higher level language in addition to LISP, the widely used language in Koza's methods. It has been used to solve symbolic regression problems [212], generate classification rules [185] and evolve financial models [59].

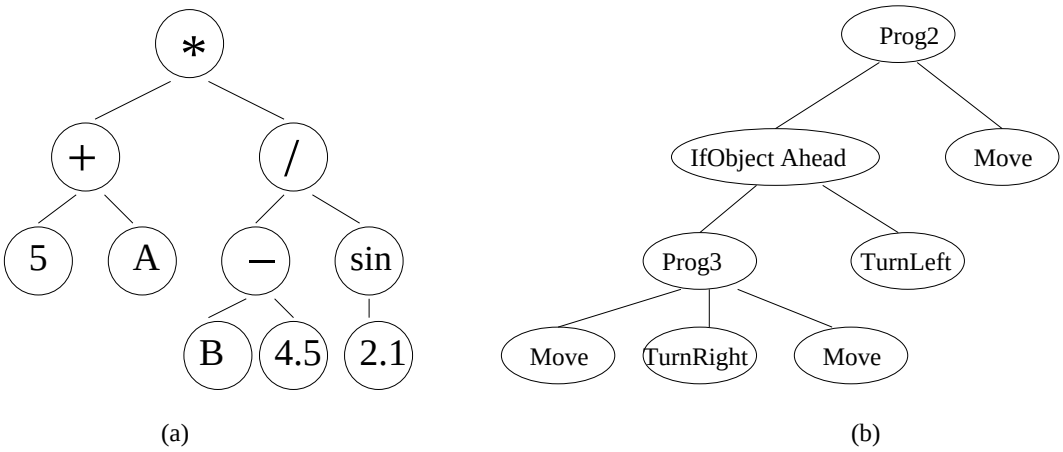


Figure 2.4: Samples of programs in tree-based genetic programming

The main advantage of grammatical evolution is that it utilises Backus Naur Form for the mapping, enabling the evolved individuals to be converted to a program of any language.

The disadvantages of grammatical evolution are: The evolved code is not understandable and requires a mapping to convert it to an understandable program.

2.2 Tree-based Genetic Programming

Tree-based GP is used in this thesis. This section presents a survey of the current research, with a focus on tree-based GP representation, handling closure, initialization, evaluation, selection, main genetic operators and run dynamics. Current GP theory studies are discussed at the end of the section, together with the research trends.

2.2.1 Representation

In general, an evolved program in tree-based GP is represented by a tree shaped program consisting of functions and terminals.

Terminals

As described in Section 2.1.3, the terminals are located at the leaves of the program trees. A single terminal can represent an individual.

Terminals can be input variables, such as A, B in Figure 2.4a. The value of the input variables comes from the environment during evaluation.

Terminals can be random constants, also called ephemeral random constants, such as 5, 4.5 and 2.1 in Figure 2.4a. The value of a random constant is generated during initialization or mutation. In GP implementations, terminals of random constants may be implemented by a *randomInt* or a *randomDouble* or a subroutine.

A terminal can also be any zero-argument node indicating some action to be performed on the environment. For example, in a problem where a robot needs to move in a two dimensional grid, the action can be *moveforward* or *turnright*, also shown in Figure 2.4b. As the program is evaluated, the corresponding action is performed on the environment.

Functions

As indicated in Section 2.1.3, functions are located inside the program trees. A single function itself cannot represent an individual.

Functions can be mathematical or arithmetic, such as *+, -, /, sin* in Figure 2.4a or boolean operators such as *and, or, not*. They accept the values returned by the terminals or functions, perform calculations accordingly and return the result.

Functions can be conditional operators, such as *IfObjectAhead* in Figure 2.4b. These conditional operators check the current condition of the environment and execute different tree branches based on the result.

Functions can be linking functions, such as *Prog2, Prog3* in Figure 2.4b and they "glue" the nodes together. *Prog2, Prog3* take two and three arguments respectively and execute them sequentially.

Functions can be domain specific, such as *TurnRight, TurnLeft* in Figure 2.4b. They are specially designed for this problem.

2.2.2 Closure and Strongly Typed Genetic Programming

The 'closure' property requires that all functions can accept, as their arguments, any value and data type that is returned by any function in the function set and any value and data type that is assumed by any terminal in the terminal set [123].

There are three basic ways of achieving closure.

- Users can allow programs to have functions and terminals of different types and allow invalid programs to be generated either in initialization or in the process of evolution.

Checking is performed in evaluation and invalid programs are discarded or assigned a bad fitness [123]. The advantage of this approach is that nothing needs to be done at set-up time. The disadvantage is that generating and discarding invalid programs during evaluation is very costly.

- Users can set all functions to accept arguments of a single type and return values of the same type [91]. The advantage of this is that any element can be a child node in a program tree for any other element without violating the data type constraints. The disadvantage is that it causes a serious limitation to GP because individuals can only have one data type.

- Users can implement a certain mechanism in functions so that they can accept any data type in an argument and deal with it appropriately, performing any necessary type conversion [123]. The advantage of this is that functions are free to accept any value returned by functions or by any types of terminals. The disadvantage is that it adds to programming complexity since all possible cases need to be anticipated inside every function.

None of the above methods are particularly satisfactory and there has been some work on more sophisticated methods of achieving closure. These include strongly typed genetic programming [174] and grammar guided genetic programming [248]. Grammar guided genetic programming was described earlier (see page 19).

Strongly Typed Genetic Programming

Strongly typed genetic programming (STGP) was developed by Montana [174] and initially it allowed only two levels of type hierarchy. Later on, it was updated by Haynes et al. [91] to allow more than two levels of type hierarchy.

In strongly typed genetic programming, the data type of each argument of each function is defined at set-up time, as is the data type returned by each function. In initialization, strongly typed genetic programming only constructs individuals that satisfy type constraints. In genetic operations like crossover and mutation, only functions and terminals of the same type can be swapped or mutated, thus only valid descendants are generated.

The main advantages of strongly typed genetic programming are that it saves time in checking for invalid programs at execution time by avoiding the generation of illegal programs and reduces

the search space by only allowing explorations of valid programs.

However, the type restriction may also have some side effects. The restriction may eliminate the possibility of some good solutions being evolved. For example, suppose numeric values have been used in a robot control program to move the robot in a simulated play field. If the values indicating *degree of direction* and *force of movement* have been set to be different data types, there will be no chance to reuse the values evolved in *degree of direction* as *force of movement*, and vice versa, while good solutions might be generated by swapping these values during evolution.

Strongly typed genetic programming has been used for solving many genetic programming problems [13, 120, 151, 158, 204]. It is used for all the experiments in this thesis.

2.2.3 Initialisation

It is considered desirable for GP systems to generate programs that are distributed widely in the search space in order to create diversity at initialization. This is because at the beginning of the evolution, the search does not know where the good solutions are. A detailed discussion of the search space will be presented in Section 2.2.7 (page 46).

A number of algorithms for initializing the population have been proposed. The main ones are *grow*, *full*, *ramped half-and-half* [123], *random-branch* [36], *uniform* [25, 98], *probabilistic tree-creation 1* (PTC1) and *probabilistic tree-creation 2* (PTC2) [155]. Among them, *grow*, *full* and *ramped half-and-half*, introduced by Koza [123], are the most commonly used. A brief description of the above tree generation algorithms follows:

Grow: Randomly selects a tree root from the full set of functions and terminals. If the root is a function, the arguments are filled with random functions or terminals, then their arguments with random functions or terminals, and so on, until all branches have ended with terminals, as long as the maximum allowed tree depth has not been reached. The objective of the *grow* method is to produce trees with a wide variety of shapes.

Full: Starts by randomly selecting a tree root from a full set of functions and keeps selecting functions until one level before the specified maximum depth is reached. Then, the full method randomly selects terminals to create the leaves. The objective of the *full* method is to generate full trees of a specified depth.

Ramped half-and-half: Takes a tree depth range and for each depth in the range, with 50%

probability uses the *grow* method and with 50% probability uses the *full* method to generate individuals. The objective of the ramped half-and-half method is to have a balanced distribution of individuals of irregular and full tree shapes.

RandomBranch: Takes a requested subtree size. The arguments of this subtree are filled by requesting a tree of size ($\frac{RequestedSize}{NumberofSelectedFunctionArguments}$). The process recursively continues until the requested size becomes zero and the leaves are filled with random selected terminals. The objective of the *RandomBranch* method is to generate a tree with a specified size.

Uniform: Computes a number of tree distribution tables with different functions and terminals, including a table of numbers of trees for all sizes up to some maximum size requested offline beforehand. During evolution an individual of a particular size is requested, the uniform method picks a function selected from a distribution derived from the precomputed tables and generates a tree accordingly. The objective of the *uniform* method is to create trees with uniform distribution of nodes and shapes.

Probabilistic tree-creation 1: Is a modification of the *grow* and the *uniform* method. It takes the probability of using each function and the requested tree size into consideration. The objective of the probabilistic tree-creation 1 algorithm is to provide user control over the generated tree size and generate trees around an expected size with given probabilities of appearance of functions.

Probabilistic tree-creation 2: Is a variation of probabilistic tree-creation 1 which allows the user to provide a probability distribution of requested tree sizes. It picks a random tree size from the user provided distribution table and builds a tree of that size or slightly greater. The objective of the algorithm is the same as the probabilistic tree-creation 1, that is, to produce trees around an expected size with given probabilities of appearance of functions. In addition, probabilistic tree-creation 2 allows control over the variance in tree sizes.

The overall objective of different tree creation algorithms is to give the population an advantage at the beginning of the evolution and to generate individuals which are evenly distributed in the search space. The research found that the ramped half-and-half method is unable to

generate a population with evenly distributed terminals and functions as the method is biased towards the full method. This is because half of the trees created are full trees and there are many more different tree shapes which are not full trees [32]. The uniform method proposed by Bohm and Geyer-Schulz [25] addresses the bias towards full trees problem in the ramped half-and-half method and eliminates it in tree creation by precomputing the distribution of tree shapes with different functions and terminals. However, the uniform method requires off-line calculation and this is an extra cost. In addition, some researchers found that uniformity in initialization appears to have little consequence in improving fitness [159].

Luke and Panait [159] compared all of the above tree generation algorithms on a symbolic regression problem, the artificial ant problem and an 11-boolean multiplexer problem. They concluded that various tree creation algorithms do not have a significant impact on fitness. They suggested that more attention be devoted to additional features, other than tree size, which could give evolution an 'extra push' [159] during the initialization phase.

The experiments presented in this thesis use the ramped half-and-half method for initialization, because it is a commonly used method and this choice is not expected to make a significant difference in evolution, based on Luke and Panait's research [159].

2.2.4 Fitness and Selection

Fitness describes the quality of the solution while selection directs the evolutionary search. In the GP process, every individual program is evaluated and assigned a fitness. The fitness of an individual is used for deciding the individual's participation in the selection process.

Fitness

Fitness describes how well each individual computer program in the population performs in its problem environment [124, p36]. For example, in a robot navigation task, the fitness might be the distance traveled by a robot before crashing, while in an image classification task the fitness might be the number of correctly classified images. There are a number of ways to measure the fitness [17, 123]. The main ones are *raw fitness*, *standardized fitness*, *adjusted fitness* and *normalised fitness*. A brief description of these follows:

Raw fitness: is the measurement of fitness stated in the natural terminology of the problem itself. For example, the number of correctly classified images in an image classification

task is a raw fitness value.

Standardized fitness: is a transformed fitness function in which zero is the value assigned to the fittest individual. For example, the number of incorrectly classified images in an image classification task is a standardized fitness value.

Adjusted fitness: is given by $\frac{1}{1+standardfitnessvalue}$ and has a value between 0 and 1. The fittest individual is represented by 1, that is, when the standardized fitness equals 0.

Normalised fitness: is computed from adjusted fitness and ranges between 0 and 1. It is the adjusted fitness value of an individual divided by the sum of adjusted fitnesses of the whole population.

The objective of assigning fitness to individuals is to describe how good an individual is in order to select the better ones for later genetic operations and to decide whether a solution has been found.

There are a number of ways to assign fitness to an individual.

- The fitness can be assigned in a static environment, that is, fitness is computed by an algorithm or formula base on a static fitness function. For example, in the Santa Fe ant problem (see Page 63), the Santa Fe trail is unchanged (static) for every generation and for every run. The fitness is computed by counting how many pieces of food are left in the trail. This method is the most frequently used [17, 123, 124].

- The fitness can be assigned in a dynamic environment. For example in a robot soccer game, two teams of robots are competing against each other to score more goals. The fitness of a team is how good a team is compared with its rival. This fitness assignment method is also called 'co-evolution' [52, 128] and GP needs evolve an algorithm that is able to keep track of the changes of the opposing individuals and beat them in a changing environment.

- Fitness can also be assigned in a multi-objective way [21]. For example, small program size can be considered to be an objective in addition to how good the evolved program is at solving the problem [20, 21]. The fitness value used during the evolution needs to reflect both objectives.

- The fitness can be assigned interactively during the evolution by the user. For example, in [199] the user decides which individual is better for an image enhancement tasks. This fitness assignment method is frequently used for applications in design [19], arts [163] and other subjective optimisation problems [47].

The experiments in this thesis use standardized fitness, in that, the smaller the fitness value, the better the programs and the fitness is measured in a static environment.

Selection

Fitness-based selection is used to determine the parents for generating descendants. Selection is intended to find good individuals in order to pass on good genetic material to the descendants. In the ideal case, selection is not purely based on how well an individual solves the problem, but also on its evolvability [7, 61], that is, how well it can help to generate fitter offspring.

"Selection pressure" or "selection intensity" is often used to describe a key property of a selection method. This property is related to the difference between the average population fitness after and before selection [87]. The larger the difference, the higher the selection pressure or intensity.

Major selection methods for generating offspring are *fitness proportional* selection [197], *tournament* selection [35, 198], *rank* selection [252] and *truncation* selection [24].

Fitness proportional selection: is also known as "roulette-wheel" selection. In fitness proportional selection, the probability of an individual being selected is in agreement with its fitness value. The objective of fitness proportional selection is to choose fitter individuals based on a fitness comparison with the whole population. The better the fitness of an individual in the population, the higher the chance that it will be selected.

There are a number of variations of fitness proportional selection which involve scaling of selection probability. This is because when the best and the worst individuals have similar fitness values, the similar fitness values may cause selection probabilities to be the same for either the best or the worst. Major scaling methods are linear static scaling and exponential scaling [87].

Tournament selection: is achieved by randomly choosing a number of individuals from the population to participate in a tournament. The winner is the fittest individual and is

selected for subsequent genetic operations. In some cases, several winning individuals are selected. This process is repeated until a new population is completely formed. The objective of tournament selection is to control the "selection pressure" in order to maintain some not-so-fit individuals with the expectation these not-so-fit individuals can help to generate fitter descendants.

In tournament selection, if the tournament size is the whole population, then it is the same as fitness proportion selection; if the tournament size is 1, it is equivalent to random selection.

Linear ranking selection: is achieved by sorting individuals according to their fitness. Rank n is assigned to the best individual and the rank 1 to the worst individual. The selection probability is linearly assigned to the individuals according to their ranks in which the best individual gets $\frac{n}{\Sigma n}$ selection probability and the worst individual gets $\frac{1}{\Sigma n}$ selection probability. The objective of linear ranking selection is to prevent bias due to large differences in fitness values.

Truncation selection: is achieved by ordering candidate solutions by fitness, then selecting a fraction of the best individuals. These best individuals have the same selection probability to produce descendants. Truncation selection is mainly used in genetic algorithms. The objective is to bias the selection towards the fittest individuals.

The choice of selection algorithm is problem specific. There is no systematic comparison of these selection schemes for genetic programming, but there has been work in genetic algorithms [23] and evolutionary algorithms [24]. In [23, 24], it was shown that different selection mechanisms could be used to set different selection intensity. A high selection intensity generally leads to the loss of diversity and rapid convergence to a local optimum, while a low selection intensity helps to maintain diversity, but may slow down the speed of getting a solution. As stated in [23, 24], whether to have a high or low selection intensity depends on the problems.

Depending on when selection is performed, there are generation-based GP and steady state GP [88, 117]. In generation-based GP, selection is made after a generation has been processed. In steady state GP, there are no fixed generation intervals, because better offspring continuously replace the less fit existing individuals.

The experiments in this thesis use *generation-based fitness proportional* selection.

2.2.5 Main Genetic Operators

Genetic operators perform actions that simulate natural genetic operations and manipulate structures of individuals during evolution. There are three principal genetic operators and they are *reproduction, crossover* and *mutation*.

Reproduction

Reproduction is the process that selects individuals, usually the fittest, and copies them into the next generation. When the reproduction process only selects the fittest individuals and copies them to the next generation, it is called *elitism*. The objective of reproduction is to keep good individuals and have good genetic material in the population. If elitism is used, the best fitness of the whole population will not drop during evolution.

Crossover

The crossover operator exchanges genetic material between two individuals by swapping part of one individual with part of the other. The objective of crossover is to exploit the existing genetic material in a population.

There are a number of crossover methods for tree-based GP. The main ones are *standard* crossover, *one-point* crossover, *size fair* crossover and *homologous* crossover [36, 57, 84, 133, 134, 161, 188, 234].

Standard crossover: is a frequently used crossover method. Two individuals are picked, one crossover point in each of them is randomly chosen and the subtrees are swapped to obtain two offspring. The objective of standard crossover is to generate two offspring by randomly exchanging genetic material of two parent individuals.

One-point crossover: has three phases [200]. Firstly, two parent trees are traversed, starting from the root nodes, to identify the parts with the same shape. Recursion is stopped when an arity mismatch in trees is detected. All traversed nodes and links are stored. Secondly, a crossover point is randomly chosen with a uniform probability from the stored links. Thirdly, the two subtrees below the common crossover point are swapped in exactly the same way as in standard crossover. The objective of one-point crossover is to encourage global search. It has been found that standard crossover only performs local search and

does not explore the search space effectively while one-point crossover can perform global search early in a run [200]. This is easy to understand with an example. Nearly half of the nodes of an individual represented by a binary tree are terminals, while the rest are functions. Changing a terminal in an individual is likely to have less effect on the performance than changing a function. One-point crossover gives a slight preference for crossover at function nodes as traversal is stopped when an arity mismatch in the trees is detected.

Size fair crossover: takes the size of sub-trees to be swapped into consideration. The method picks two individuals from a population as parents and then selects the crossover point in the first parent. The size of subtree to be deleted from the first parent is calculated. The crossover point in the second individual is randomly chosen from subtrees of the same size. If there are no such trees, a second crossover point is chosen from the first parent and the process repeats until subtree of the same size is found. The objective of size fair crossover is to control the size growth of descendants, so that the offspring will be no more than the first parent in size.

Homologous crossover: is an extension of size fair crossover. Instead of randomly choosing between all available subtrees of desired size in the second parent, distance is computed between the first subtree and all possible crossover points in the second parent tree. Crossover points for subtrees which have similiar position and size are chosen. Research conducted by Hansen [89] found that homologous crossover reduces the tendency of evolved programs to grow larger without correlated fitness improvements, is consistently better in performance in terms of accuracy, and finds solutions earlier than standard crossover. The objective of homologous crossover is to exchange only subtrees that have similar size and location.

Crossover methods need to be modified for strongly typed genetic programming (see Section 2.2.2, page 25) because randomly choosing crossover points can lead to invalid individuals. Only points of the same data type can be swapped during crossover in strongly typed genetic programming.

The experiments in this thesis use standard crossover, because standard crossover is a frequently used crossover method and the focus of our work is not on crossover.

Mutation

The mutation operator randomly changes some functions or terminals in an individual. The objective of mutation is to explore, that is, to introduce new genetic material to an existing individual.

There are a number of mutation methods for tree-based GP. The main ones are *standard* mutation [123, p106], *point-based* mutation [193] and *uniform subtree* mutation [242].

Standard mutation: is the most frequently used mutation method. A node is selected at random and whatever is below is replaced by a randomly generated subtree. The objective of standard mutation is to allow a random exploration of subtrees or terminals.

Point-based mutation: changes a single node in an individual. A node is selected at random. If the node is a function, it is replaced by another function of equal arity. If the node is a terminal, it is replaced by another terminal. The objective of point-based mutation is to explore small changes in an individual.

Uniform subtree mutation: performs a number of standard mutations in one operation. The nodes for mutation are chosen with uniform probability from the program tree. The objective of uniform subtree mutation is to explore massive changes of an individual.

The experiments in this thesis use *standard* mutation, because this is a frequently used method and the focus of our work is not on mutation.

Balance Between Crossover and Mutation

The traditional view of crossover and mutation is that crossover is primarily responsible for improvements in fitness and is the main driving force of evolution, while mutation is a relatively unimportant operation that helps to reintroduce random genetic material[123, p105] [223]. Mutation is generally used in a small proportion, however, some work has found that mutation can be more helpful than expected [93].

Luke and Spector [161] compared various combinations of crossover and mutation rates, from 90% crossover (c) to 90% mutation (m), with a fixed 10% reproduction (r) rate ($c\%+m\%+r\% = 100\%$) for a number of classic genetic programming problems. They found that a higher crossover rate was more successful in a majority of the tests. In their refinement work [162], they found

that the difference in fitness between different settings of mutation and crossover rates was small in most of the tested problems. They claimed that in general mutation was more successful for small populations, while crossover performed better for large populations.

Setting Reproduction, Crossover and Mutation Rates

Two ways to set reproduction, crossover and mutation rates are in use.

In method one, the sum of the assigned percentages of reproduction (r), crossover (c) and mutation (m) is 100% ($r\% + c\% + m\% = 100\%$). The new population is created in three steps. Firstly, $r\%$ of good individuals from the old population are copied to the new population. Secondly, $c\%$ of the individuals in the new population are created by crossover. Thirdly, $m\%$ of individuals in the new population are created by mutation. Individuals generated through these three operations form the new population. This is the most commonly used method [123, 161].

In method two, the sum of assigned percentage of reproduction (r), crossover (c) and mutation (m) is higher than 100% ($r\% + c\% = 100\%$, $r\% + c\% + m\% > 100\%$). The new population is also created in three steps. Firstly, $r\%$ of the individuals in the new population are created by reproduction. Secondly, $c\%$ of the individuals in the new population are created by crossover. In the created new population, $m\%$ of the individuals are mutated. For example, in [117], 10% of the new population was generated by reproduction and 90% was generated by crossover. After these two operations, 10% of the resulting new population was altered by the mutation.

The experiments in this thesis use the first method to set reproduction, crossover and mutation rates. In this thesis, the default elitism rate is 2%, the default crossover rate is 70% and the default mutation rate is 28%.

2.2.6 Run Management

Run management reviews all other parameters that control the genetic environment during evolution but have not so far been addressed. Run management includes population size management, number of populations, program size/depth control, diversity control, premature convergence control and self-adaptive parameters.

Population Size Management

In the GP evolution process, evaluation generally consumes a large amount of computation and comprises most of the evolution time. In a run, the overall computation cost (number of evaluations) equals (population size x number of generations). The objective of population size management is to find a good population size so that a run can find a good solution with the least number of evaluations.

In order to decide population size, there are two main methods. One is to use a fixed population and the other is to change population size during runs. Methods of changing population size are further divided into three streams - self-adaptive population size based on the run performance (reviewed in page 41), decreasing the population size and increasing the population size.

In fixed size population methods, the early practice was to use a large population [79]. Later, researchers found that small populations (e.g. 50 to 400) over longer generations were better for some problems [82] and multiple shorter runs tend to find successful solutions earlier thus lowering the overall computation cost [156]. Recent research by Ashlock [12] found that GP with very small (4-7) population sizes gave more solutions faster for a 4-parity and a Tartarus problem.

In methods of changing population size, decreasing the population size during a run can have good effects. The main method in this approach is *decimation*. Decimation is used primarily at the initialization. A large population (usually thousands of individuals) is generated and evaluated. All individuals with bad fitness after one or two generations are removed. After decimation, a fixed population size (usually only hundreds of individuals) is maintained. Decimation has been found to be effective in reducing the number of evaluations to get a successful solution [80]. Luke et al. [157] found that decimation never performed worse than various fixed-sized population strategies. Nanduri [179] reported similiar findings.

There is also work showing that increasing the population size can help. An investigation of three GP test problems showed that in order to ensure getting solutions within an error tolerance, the population size had to be increased when evolutionary runs did not converge [214].

There is no theoretical guide to decide which population size is appropriate for a given problem [214]. The experiments in this thesis use fixed population sizes. Where appropriate,

we have used the same population size as in previously published work to facilitate comparison.

Number of Populations

Number of populations relates to how individuals are organised - a single population or a number of small sub-populations which can be evolved simultaneously. There are two main objectives for deciding the number of populations. One is to maintain population diversity (see page 40), the other is to fully utilise the power of parallel processing of networked computers.

There are two basic models: *coarse-grained* [105] and *fine-grained* [67, 240]. In the coarse-grained model, populations of individuals are evolved separately and interact periodically to exchange good individuals. This model is also frequently referred as the multi-population model. In the fine-grained model, a global population is laid out on a two dimensional array and individuals are only allowed to interact with their neighbors. The overlap of the interactions between neighboring individuals enables implicit communication through crossover and the global population is gradually improved.

There is also a varied model. Ciesielski, Loveard and Li implemented a variation of the coarse-grained model together with population-based decimation [41, 150]. They evolved multiple populations simultaneously at the beginning and removed the slow or non-performing populations during evolution. This was called the *pyramid strategy*. The model was tested for a number of classification problems and some commonly used genetic programming problems like symbolic regression and the Santa Fe ant problem. The results demonstrated great saving in overall computation cost (see page 36).

It is hard to decide whether several populations are better than a single population. A number of investigations found that the coarse-grained model is helpful to maintain diversity, thus leads to better performance in finding solutions earlier [105, 240]. However, a published paper on the royal tree problem [206] showed the opposite. In that work, a single large population always outperformed a group of smaller populations using the *coarse-grained* model.

The experiments in this thesis use a single population because this is a frequently used method and the focus of our work is not on deciding whether a single population or several populations are better.

Program Size Control and Reducing Bloat

Program size control has a close relationship with reducing 'bloat'. Bloat refers to the phenomenon in which the size of an individual grows, but performance does not improve [138]. The objective of program size control is to minimise non-effective code. In some work [154], the non-effective code is also called inviable code. The non-effective code appears and grows during evolution in tree-based GP [123, 124] and slows down evolution in achieving an optimal solution. An example of an individual that contains non-effective code is $(A + C + 0 * (X))$. X in this program can be a single terminal or a subtree, but whatever it is, it has no effects on the overall fitness because it multiples 0. Any mutation or crossover point within X is regarded as neutral.

There are a number of common explanations of the cause of bloat [22, 154, 222]. Firstly, bloat may be caused by protection from destructive effects [22]. Non-effective code makes it difficult for the evolutionary process to effectively change an individual by increasing numbers of neutral crossover or mutation points [22]. In our example $(A + C + 0 * (X))$, all crossover or mutation points in X are neutral points. However, there are controversial findings. Sean Luke in his work [154] investigated three genetic programming problems - a symbolic regression, an 11-bit and a 6-bit multiplexer problem and found that even when crossover at neutral points was not allowed, the bloat continued. He suggested more investigations in this area. Secondly, bloat may be caused by removal of bias [221]. This assumes that non-effective code is more densely concentrated near the leaves of program trees. The bias refers to the evolution in favor of offspring created by removing a small subtree and against offspring created by removing a large subtree. There is no bias for adding subtrees. So, in order to preserve an individual, there is a penalty for removing large subtrees, but no penalty for inserting large subtrees. Thirdly, bloat may be caused by genetic drift [29]. There are more larger-sized solutions and this is obvious because adding non-effective code to smaller sized solutions will not change the fitness of these solutions. The genetic drift effect moves a small sized population to a population of large size.

In order to find small sized solutions, it is desirable to restrict the size or depth of the evolved programs. A theory study has found that the number of different unlabeled binary tree shapes is doubly exponential in depth [5]. After a depth of 5 the number of possible tree shapes becomes massive (see Table 2.1). If a tree shape is not binary but ternary, quaternary etc., the number of possible tree shapes will be much larger, and so will the search space. It is hard to find solutions

Table 2.1: Number of Binary Trees to Height of 8

Height	Number of Binary Trees
0	1
1	1
2	3
3	21
4	651
5	457653
6	210065930571
7	44127887745696109598901
8	1947270476915296449559659317606103024276803403

in a very large search space.

Researchers have proposed a number of size control methods in order to reduce the effect of bloat, to search the space efficiently and to evolve small solutions. Firstly, program size can be constrained by setting a maximum number of nodes or a fixed maximum depth. This is commonly used. An empirical study [49] showed that methods that restrict number of nodes and methods that restrict tree depth were similar, except that methods that restrict number of nodes provide finer control. Secondly, a penalty may be be added to bias the search towards small programs [22]. For example, the fitness of a program can equal to the size of the program multiplied by a user specified rate ($fitness = standardizedFitness + programSize * userDefinedRate$). Thirdly, modifications to crossover methods can help to reduce the non-effective code growth during evolution. For example, Langdon proposed the *size-fair* crossover method (see Section 2.2.5, page 33) that helped to reduce bloat for four GP benchmark problems [135]. Terrio and Heywood biased the crossover selection mechanism. They evaluated the fitness of each individual at each node and favored crossover between subtrees or nodes with better fitness. Their work demonstrated that this biased crossover method could reduce non-effective code and improved the speed of evaluation for a number of problems [238]. Fourthly, program size control may be achieved in a multi-objective way. Some researchers have used multi-objective techniques with small size as one of the objectives [20, 21, 54]. The results demonstrated clear reduction in size while improving fitness. Fifthly, researchers have proposed a number of techniques to identify non-effective code and remove it [219]. However, non-effective code detection is non-trivial.

In the experiments in this thesis, the size of the evolved program is constrained by setting a maximum tree depth because it helps to restrict maximum program size and also gives the

flexibility to compare the size differences for programs with and without loops. We expect that the programs with loops will be smaller than those without loops.

Diversity Management and Avoiding Premature Convergence

Diversity management and avoiding premature convergence are related. They both deal with how to avoid local optima and how to maintain differences between individuals during evolution. Diversity is a *variety* measurement and indicates the number of different individuals in a population [123]. The differences between individuals could be in structure (representation of a program) or in behaviour (fitness). Premature convergence occurs when all programs in a population become similar (loss of diversity) but are not the optimal solution to the problem at hand [44].

There are a number of ways to measure diversity. Firstly, diversity can be measured by differences in subtrees. Keijzer used the ratio of unique subtrees over total subtrees as the diversity indicator [111]. However, to obtain this indicator is computationally expensive as each individual needs to be traversed to count number of unique subtrees and total subtrees. Tackett measured structural diversity using the frequencies of subtrees or schemata [231]. Secondly, diversity can be measured by the degree of graph isomorphism. Rosca found [209] that the properties of functions used in genetic programming would require a special, complex implementation of isomorphism and an approximate measure of the number of isomorphic trees could be found by using a {terminals, functions, depth} combination. Thirdly, diversity can be measured by the differences between two individuals based on string edit distance. Edit distance indicates the number of changes needed to convert a string to another string. To perform an edit distance calculation in tree-based GP, two trees are overlaid at the root node. Two different overlaid nodes score a distance of 1 while identical nodes score a distance of 0. The sum of all different nodes is the edit distance of two programs. O'Reilly used edit distance to understand the underlying dynamics for the effects of crossover and mutation [190]. De Jong et al. used edit distance to measure the diversity for one of the objectives in a multi-objective approach [55]. Fourthly, entropy, calculated by fitness value together with the program size, can be used to as a measure of diversity. Rosca found the population appeared to be stuck in local optima when entropy did not change or decreased in successive generations [209]. Fifthly, frequency of the terminals, functions and tree shapes in a population can be used as a measure of diversity

[50].

There are a number of methods to promote diversity and to avoid premature convergence. In the classic view, a diverse population will have more chances in evolution. Firstly, diversity can be promoted by geographical distribution of individuals to limit their interactions. This is described in population organisation management (see Section 2.2.6, page 37). Secondly, diversity can be promoted by fitness sharing, which considers behaviour and structure similarities and content management [6, 56, 166]. Thirdly, diversity can be promoted in a multi-objective way. De Jong et al. promoted diversity by setting diversity as an objective in their multi-objective optimisation to solve an n-Parity problem [55]. Fourthly, diversity can be promoted by changing population size at run time. Fernandes and Rosa used varying population sizes and non-random mating to maintain diversity for a Royal Road problem [69]. Fifthly, diversity can be promoted by some replacement strategies. Ciesielski and Mawhinney inserted different programs during evolution for a robot soccer problem in order to improve diversity and avoid early convergence [44].

The experiments in this thesis do not use any mechanism to control diversity, because diversity is not the focus of the work.

Self-adaptive Genetic Parameters

Self-adaptive genetic parameter management addresses the issue of adjusting generic operations in the run time. The self-adaptive approach eliminates the task of GP practitioners to pre-set the different genetic parameters like population size, crossover and mutation rates or even types of crossover/mutation methods. These parameter settings tend to be problematic and it is hard for new algorithm developers to decide whether it is the setting of these parameters or the newly developed algorithm which makes a difference in results.

There are few papers on self-adaptation issues. Firstly, the number of individuals in a population can be adjusted at the run time. Some approaches have already been described in Section 2.2.6 (see page 36). Rochat, Tomassini and Vanneschi have experimented with changing the size of the population in genetic programming on three classic genetic problems in order to improve the search efficiency and decrease the amount of computing resources required [208]. In their composition, they deleted individuals from a population while the best individual found so far kept improving, and added individuals when there was no improvement. The results showed

that they got higher success rates with less computing effort. Secondly, crossover and mutation rates can be self-adaptive. The previous work related to this is mainly for genetic algorithms [14, 83, 218, 227], but can apply to genetic programming. Thirdly, choosing crossover methods can be decided dynamically at run time [90, 224]. Spears composed a method to choose between two different forms of crossover at run time for an n-Peak problem in GA [224]. The composed adaptive mechanism solved that n-Peak problem. However, Spears did not claim his work was better than the non-adaptive method.

The experiments in this thesis do not use any self-adaptive mechanisms as we want to compare the loops approach with non-loops in a relatively simple environment.

2.2.7 Theory and Search Space Studies

GP has been successfully applied to many real world problems, but there are people who are still reluctant to accept GP as a workable approach and use it for their problems. These people believe that any workable algorithm needs to be explained by mathematics and an algorithm without mathematical proof is weak. Using mathematics to model GP algorithms is beneficial for GP practitioners because it can improve the understanding of the evolutionary process and help to refine the existing techniques.

Currently, there are only a few theories developed in the GP domain and most of them are extensions from other evolutionary algorithms. They are the schema theorem, Price's covariance theorem, the no-free-lunch theorem and studies in fitness landscapes.

The Schema Theorem

The schema theorem was proposed by Holland [96] and has been heavily studied in GA [85, 228, 229].

In binary GA, a schema is determined by the defining bits (0 and 1) and by their position [201]. For example, the schema *1*10 can match any bit strings like 11010 and 01010, etc. The number of defined bits in a schema is referred as the order of a schema, while the number of bits between the first and last defined bits is referred to as the length of the schema.

The schema theorem states that schemata with above-average fitness increase their frequency in the population each generation at an exponential rate when only a few individuals in the population have these schemata [8]. The schema theorem helps to understand why evolutionary

algorithms work because it is believed that evolutionary algorithms solve problems by hierarchically composing relatively fit, short schemas to form complete solutions [201].

Researchers have extended the schema theorem to genetic programming in order to explain why genetic programming works [123, 192, 203, 249]. There are a number of issues relating to these extensions.

Firstly, definitions of a schema in GA and GP are different. There are a number of definitions of a schema in GP.

Koza defined a schema as the subspace of all trees which contain a pre-defined set of subtrees and a schema H is represented as a set of S-expressions [123]. For example, {(+ A 5)} represents all programs that have at least one occurrence of the expression (+ A 5).

O'Reilly refined the schema definition by Koza [192]. Her schema definition takes GP variable length representation into consideration. In her definition, a schema is an unordered collection of subtrees and tree fragments. The fragments are trees with at least one leaf that is a "don't care" symbol "#" which can be matched by any subtree including subtrees with only one node. For example, {(+ A #)} represents all programs that have at least one occurrence of any tree fragments like (+ A B) or (+ A 5), because "#" can be replaced by any node.

Whigham proposed a definition of a schema based on the concept of his context-free grammar GP [249]. In his definition, a schema is a partial derivation tree rooted in some non-terminal node. For example, a schema (NODE => FUNCTION NODE NODE) can represent a derivation tree (NODE (FUNCTION -) (NODE (TERMINAL 2)) (NODE (TERMINAL x))) for the program tree (- 2 x).

Secondly, the extended schema theorem in GP is similar to the schema theorem in GA. It takes the number of occurrences of a schema, the mean fitness of programs which have the corresponding schemata and the consecutive generations into consideration and works out the weighted sum of the fitness [192]. The schema theorem in GP describes the way in which components of the representation of a schema propagate from one generation to the next, which is the same as the schema theorem in GA.

Thirdly, the schema theory has been extended to analyse crossover operations. Poli, McPhee and Langdon have composed two sub-schema theorems for crossover operations [196, 202, 203]. One is a microscopic schema theorem and is applicable to crossover operators which replace a subtree in one parent with a subtree from the other parent to produce offspring. This theorem

relates the total transmission probabiliy for a fixed-size-and-shape GP schema to the selection probability of the schema, the crossover probability and the probability of selection of the crossover points. This result is a subtree-swapping crossover operator without mutation. The other is a macroscopic schema theorem and is valid for a large class of crossover operators in which the probability of selecting any two crossover points in the parents depends only on their size and shape. This schema theorem is similiar to the microscopic schema theorem and relates the total transmission probability for a fixed-size-and-shape GP schema to the selection probability of the schema, the crossover probability and the probability of selection of the crossover points. The result of this theorem is for a node-invariant subtree-swapping crossover operator without mutation.

Fourthly, variations of the schema theories should not be regarded as competitors. Poli and Langdon reviewed the schema theorems and pointed out that all these schema theorems should be regarded as mathematical tools to describe the search space and are different views of the same subspace of the space of possible solutions [201].

Fifthly, there are doubts on the usefulness of the schema theorem for GA [38, 74]. This work expressed concerns over the schema theorem ability to predict progression of multiple generations rather than the just adjacent next generation. Compared with GA, the evolutionary process in GP is similiar but the representation is more complicated. This may make prediction in GP more difficult even for the adjacent generation as well as multiple generations in the future.

The schema theorem and its variations provide a nice idea that the process of the evolution can be modeled and understood by some sort of mathematical formulas. However, the problems investigated in these theory studies are artificial and furthermore, genetic operations for these artificial problems are restricted to be crossover. It is hard for a GP practitioner to apply these theories to understanding why the evolution process solves their real problems. The usual practice is that GP practitioners have already solved their problems or solved their problems to a certain degree and then try to use some techniques to find some traces of the schema or repeated patterns by looking into log files after many runs are finished. They do this in order to understand what is going on with their evolution systems with a hope to evolve better programs faster. However, it is hard to say whether they really find schemata that help the evolution or just repeated patterns that occur because of the settings or because of the natural of genetic operations. Despite the impression of the powerful mathematical formulation, at the current

stage, the schema theory can not really reveal what is going on for the whole evolutionary process and there are no results that show that the schema theory has been really helpful in solving hard real-world problems.

In our work, we applied the some techniques developed in the schema theorem research. In Chapter 5, we used the methods similar to O'Reilly's schema work and found repeated fragments with some matching nodes and some "don't-care #" nodes for runs with and without loops. The difference between O'Reilly's approach and ours (see Section 5.5.1, page 153) is that we count all the code fragments larger than a certain length and repeated in individuals, while O'Reilly's work only checks complete statements in evolved individuals. We found these repeated patterns, compared them for the runs with and without loops and interpreted them to decide whether the repeated patterns that emerged in the runs with loops are especially helpful to solve the problem.

Price's Theorem and The No Free Lunch Theorem

Price's covariance and selection theorem and the no-free-lunch theorem are other theorems that help to understand the evolutionary process and analyse the strengths/weaknesses of evolutionary algorithms.

$$w \triangle z = cov(w_i, z_i) + E(w_i \triangle z_i) \tag{2.1}$$

Price's theorem is an equation which describes evolution and natural selection. Originally it was used in the biological area. In this equation, w is the average fitness and $\triangle z$ is the change of the population in a genetic character (property). $Cov(w_i, z_i)$ is the covariance of the characteristic with respect to the fitness (w) for the population or group (i). A group has certain characteristic z_i. $E(w_i \triangle z_i)$ is the expectation of the fitness times the change in the characteristic. Overall, the theorem illustrates the effects of selection on a population in terms of covariances between fitness and the property of effects due to transmission [142]. It shows that the covariance between parental fitness and offspring traits is the means by which selection directs the evolution of the population. Altenberg found that Price's theorem helped to reveal how the fitness function, program representations, and genetic operators interact to produce evolvability [8]. In his definition, evolvability means the ability of a population to produce descendants that are fitter than the existing population. With the help of Price's theorem,

he conducted an investigation to analyse the change in population fitness and representations and then proposed a number of new selection operators and other techniques [7]. These new operators and techniques delivered better control over the evolution and improved evolutionary performance.

To GP practitioners, Price's theorem provides very limited help. In GP, it is understandable that the change of the property (the representation of the program) may lead to the change of fitness, but it is doubtful that this fact can really help the understanding of the evolution and thus lead to better algorithms. Currently, there is no research dealing with hard practical problems using Price's theorem as an explanation of the results.

The no-free-lunch theorem for optimization was formulated by Wolpert and Macready [255]. The theorem shows that for both static and time dependent optimization problems, the average performance of any pair of algorithms across all possible problems is identical. This means if some algorithm performance is superior to that of another algorithm over some set of optimisation problems, then the other algorithm performance must be superior to that algorithm over the set of all other optimisation problems. The no free lunch theorem is relevant to genetic programming because the evolutionary process can be considered as one of finding the best solution to a problem.

For GP practitioners, the no-free-lunch theorem cannot really help to improve the evolution, but by understanding this theorem, we know that there are limitations for different genetic programming techniques and there does not exist a super optimisation algorithm which can cater for all problems [26, 46]. For GP practitioners, it is worthwhile to analyse the problem at hand and decide for which classes of problems the proposed setting or new algorithm works and the classes of problems for which it does not. Generally speaking, we are not interested in optimising all possible problems [142] and there are certain ways in which we can classify problems and determine the corresponding optimisation methods.

Search Space and Fitness Landscape Studies

Search space and fitness landscape studies do not involve formulating mathematical formulas as in theory studies. But as with theory studies, understanding the search space or the fitness landscape facilitates the analysis of problem difficulty and helps to understand why some settings work for some problems and some not.

Search spaces and fitness landscapes are related but different. A search space indicates all possible evolved programs. A fitness landscape is the idea that the processes of evolution can be studied by visualizing the distribution of fitness values across the search space as a landscape.

The idea of a fitness landscape was first proposed by Wright in 1932 [258]. The concept of the fitness landscape was first used to understand the difficulties of optimisation problems in GA before genetic programming by Kinnear [119].

Conceptually, a *fitness landscape* is a mapping of individual programs to an *x-y* plane and plotting their fitness on the z-axis (see the examples in Figure 2.5). Ideally, programs that are close together on the *x-y* plane are those that are most likely to be created from one genetic operation. When the fitness is measured by *standardized* fitness, this creates a surface where the peaks are the locations of program with poor fitness and the basins show the locations of the programs with good fitness [119]. Finding the best solution to the problem becomes searching for the deepest basin.

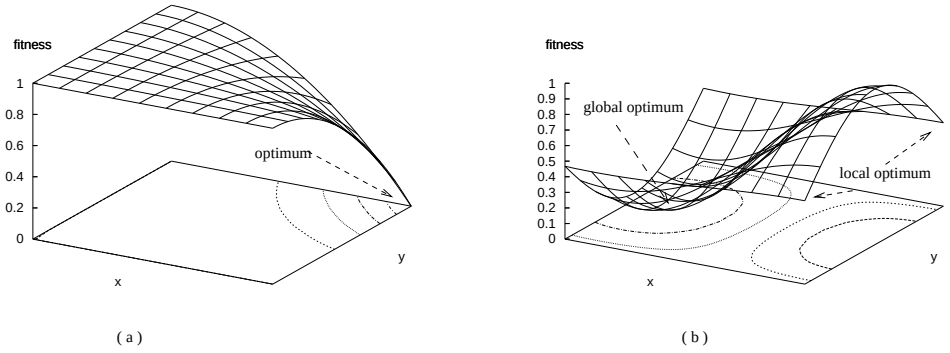


Figure 2.5: Examples of fitness landscapes for two numeric parameters optimisation problems

Fitness landscapes deliver an idea of how hard the problem is as well as how the search alogorithm can handle the problems at hand. For example, Figure 2.5 (a) shows a fitness landscape for two numeric parameters (x and y) optimisation problem. Individuals with different x and y values are organised and put into the *x-y* plane. Standardized fitness is assigned to the individuals and is showed in z-axis. There is only one optimum in the fitness landscape. On the other hand Figure 2.5 (b) shows a different fitness landscape for a two numeric parameter optimisation problem. In this fitness landscape, there are three optima and two of them are local and one is global. The problem shown in (b) is harder than the problem illustrated in (a), because a gradient search performed close to the local optima in (b) is easily attracted by the

basin of the local optima. Once the search has fully moved to the local optima, it will be hard to escape because all neighbourhood individuals of the local optima are relatively less fit.

However, mapping the individuals to an x-y plane is a hard problem. Problems tend to have more parameters (besides the simple x and y), and various operations. A "neighborhood" definition is essential because it defines how individuals are arranged as immediate neighbors. In his original diagram [258], Wright used a contour map with the contour lines representing levels of adaptiveness of biological genes. But the original diagram included no labels for the axes and no indication was given as to how the various genes should be arranged on the landscape.

Much work involving fitness landscapes avoids a rigorous definition of the landscape and neighbourhood [103] and usually presents landscapes by arranging the individuals with single bit or node mutation as neighbors. On such landscapes, crossover operations are assumed to take multiple steps of mutation and this is a deficiency of the work.

For evolutionary algorithms in general, there have been a number of attempts to overcome the deficiency of avoiding rigorous definition of the neighborhood and the deficiency of creating a fitness landscape with only mutation operations. Kauffman presented an NK fitness landscape model [110]. In this model, a fitness landscape is defined by a fitness function and the function is defined by two parameters: the number of genes (N) and the number of links between genes (K). The NK landscape model is widely used to generate landscapes for test functions for search and optimisation techniques [92, 104, 216]. Weinberger [247], Jones [102] and Hordijk [97] proposed a number of similar fitness landscape models in which the landscape is represented as a graph on which the vertices correspond to individuals and have associated fitness values. Traversing the edge of the graph corresponds to the genetic actions like mutation and crossover. In Jones's "one operator, one landscape" model [102, 103], genetic operations like selection, mutation and crossover are mapped into three separate landscapes. The evolutionary algorithm takes steps from mutation landscapes to crossover landscapes and then to selection landscapes. The neighborhood in this model is defined by the genetic operation in each landscape. This approach provided a totally new definition of the fitness landscape in 1995 and the work was not accepted for publication at that time. With more and more research on genetic algorithms and genetic programming, people have started to accept his idea and this work has been frequently referenced in recent fitness landscape study papers.

For genetic programming, the variable sized program representations, various terminals and

functions and type constraints make it harder to define *neighborhood* [119], thus it is harder to map the actual programs into a fitness landscape. Researchers in the GP area have proposed a number of ways to overcome the problem of defining *neighborhood* and handling different genetic operations in addition to mutation. In parallel GP, Slavov and Nikolaev regarded subpopulations as neighbors and plotted the fitness landscape accordingly [217]. They analysed the performance of inductive learning algorithms on a set of artificial problems for constructing decision trees and found this landscape model delivered hints for possible improvements of system components and adjustments to their parameters. In linear GP, Langdon analysed programs with point mutation [136]. This work describes fitness landscapes similar to genetic algorithm landscapes because of the linear program representation. Recently, Moraglio and Poli [176] presented a new topological framework and redefined the mutation and crossover operators to be more tightly linked to the fitness landscape. In their model, the genetic operators were defined by the fitness landscape upon which they operated and the genetic operators were a natural consequence of the neighborhood and distance metric of the fitness landscape. Later, they also extended their work to analyse the landscape of homologous crossover (see Section 2.2.5, page 33) together with mutation operations [177].

In essence, the fitness landscape is a metaphor - a metaphor in which individuals are organised and the difficulties of the problem can be viewed as a 3-dimensional terrain, where peaks and valleys represents high and low fitness respectively when using standardized fitness as stated earlier. So in a broader sense, any work relating to problem difficulty or fitness correlations between different programs can be viewed as analysing the fitness landscape or a property of the fitness landscape for the problem. For GP, Kinnear tried to use a fitness correlation between parents and children to interpret the ruggedness of the fitness landscape [119]. In 2003, Vanneschi et al. [244] used structural distance to calculate a fitness distance correlation coefficient to express difficulty levels for a set of problems, i.e. unimodal trap functions, royal trees and the MAX problem. Their structural distance is calculated by weighted sum differences between two trees overlapped at the root node. The results showed this method was useful to measure the difficulty levels. Later, Vanneschi et al. [243] extended their work and described the idea of a fitness cloud. This is a way to show correlations between individuals and their neighbourhoods and was used for binary landscapes [246] to predict the difficulty of tree shaped GP problems. In the fitness cloud formulation, individuals and their neighborhoods were randomly sampled

and they were all put into a scatter-plot where the x-axis represented fitness and the y-axis represented the fitness of the neighborhood. The advantage of this approach is that it allows GP practitioners to understand the distribution of fitness in a narrowed region and give direction on where the search should be performed.

The approach used in this thesis for analysing the differences between solving a problem with loops and without loops is similar to the method developed by Langdon and Poli [141] for analysing difficulty of the Santa Fe ant problem (see Section 2.3.3, page 63). Langdon and Poli enumerated all possible programs up to a length of 14 and then randomly sampled programs to length of 500. All enumerated and sampled programs were evaluated, fitness was assigned and number of programs with the same fitness and same length was counted. The result was plotted in a 3 dimensional graph in which x indicated the length of the program, y the fitness and z-axis showed the proportion of the individuals with the given fitness. They called this a fitness distribution graph. In a way, this fitness distribution graph provides a simple but useful way to show the search space. As we have described previously in Section 2.2.6 (page 38), the number of possible programs exponentially increases with increase in tree depth or size. It is intuitively understandable that it is better to have more good solutions in smaller sized search spaces, and thus quicker to find a solution. In Section 5.4 (page 145), we use a similiar way to measure the distribution of fitness of individuals in order to establish that for GP with loops, there are more good solutions than for GP without loops with the same program depth setting.

Overall, in the area of fitness landscape studies, there is no generally accepted definition of what constitutes a fitness landscape. There is no agreement on what a fitness landscape is and what is the neighborhood of an individual. Due the representations used in GP, the fitness landscape is difficult to define even for simple problems with basic genetic operations. Researchers have proposed various ways to demonstrate the difficulty of their problems and the fitness relationship between different individuals. These approaches can all be regarded as studying features of fitness landscape or can be regarded as some sort of the fitness landscape. For GP practitioners, understanding the difficulty of the problem is important. In addition, GP practitioners should be aware that different genetic settings, different functions, terminals and operations vary the difficulty of the problem, thus varying the fitness landscape. It is desirable to find better operators, functions and settings in order to form a relatively simple fitness landscape for the problem to facilitate the search for a solution.

2.2.8 Current Research Trends

This section presents a survey of the current main research trends. The rapid development and use of GP in many real applications has already proven that GP is a human-competitive technique to solve some hard problems [127]. However, the GP techniques suffer from a number of problems like bloat and diversity control, and there is a need to develop practical theories to understand and utilise GP tools to their full advantage.

Weakness in Theory Studies

A review of theory studies has been presented in Section 2.2.7. All major theories developed have limitations. The current results on the schema theorem studies are unable to predict the path of evolution after several generations. Extending the power of the schema theorem or finding new theorems to predict a longer process or the whole evolution are needed. Also, only artificial problems with constrained restrictions were used for developing theories. Can these methods to be applied to real world problems? How to define the 'neighbourhood' in GP is a problem in the fitness landscape studies and needs more work. Overall, further research on theory is needed to promote GP to be a theoretically sound, explainable, predictable and powerful optimisation tool.

Improving the Speed of Evaluation

In GP evolution, evaluation tends to be the most costly process. To minimize the number of evaluations is a key research area. Work in this area includes : 1. Converting evolved programs to linear machine code to improve the overall execution speed (see Section 2.1.3, page 20); 2. Using grammar guided evolution to restrict the search domain (see Section 2.1.3, page 19); 3. Caching the already evaluated subtrees or programs to avoid reevaluating the evaluated parts [42, 100, 112]. Also, minimising the number of evaluations thus improving speed of evolution is an active research area.

Maintaining Diversity and Preventing Premature Convergence

When an evolutionary run does not produce desirable results, loss of diversity in a population is often the cause. Without diversity a population of individuals cannot generate variation and it is hard to move out from the local optima. This situation is often called premature

convergence. There are two research areas relating to diversity. One is to measure diversity and the other is to maintain and promote diversity. For diversity measurement, researchers in [33, 55, 123, 140, 142, 167, 190] are investigating edit distances, entropy and other measures to define the differences between individuals. For promoting or maintaining diversity, methods include geographical distribution of individuals [67, 105, 240], pair-wise mating [68], a multi-objective approach [55] in which improving diversity is one objective and using replacement strategies to remove similar programs [44]. The research trend is to maintain diversity but not at the cost of too much extra computation to find a solution.

Minimising Bloat

Bloat refers to the exponential growth of the code during evolution without fitness improvement. The research on this has been discussed in Section 2.2.6 (page 37). Generally, bloat slows down the search or decreases the efficiency. However, in some cases, bloat protects programs and leads to areas of the search space which has fitter programs. Different methods have been proposed for minimising bloat, such as size and depth limits [49, 173], code editing [173], hill climbing [189], double and proportional tournament [160] and breeding influence by spatial structure [251]. There is still room for improvement in this area.

Adaptive Parameter Setting

Automatically setting parameters can reduce reliance on the user's experience and knowledge as well as removing bias in performance due to the selection of the parameters. Research on self-adaptive parameter setting has been reviewed in Section 2.2.6 (page 41). However, there is not much work in this area for GP [10, 11, 182] but quite a lot of work in genetic algorithms and this is a research direction for GP.

Multi-objective GP

It is natural to have different requirements for a problem and this brings the multi-objective concept and implementation into evolution. Multi-objective GP has already helped to minimise bloat [20, 21], to evolve group behaviour for agents [164] and to solve some classification tasks [116]. There is more potential in this direction.

Evolving Understandable Programs

Genetic programming has been used in a variety of fields to solve difficult problems and produce human-competitive results, but the solutions evolved are often hard to understand. Evolving understandable programs helps to promote the use of genetic programming by improving trust in the evolved algorithms. There is some recent work addressing this issue, such as the work to evolve understandable mathematical formulas [130] by picking the right function sets, correct data structures, adding a parsimony pressure and utilising explicit loops to evolve solutions for a number of classic or modified classic genetic programming problems [43]. These are the beginning of exploration in this direction.

Applying GP to More Real World Applications

Genetic programming has demonstrated applicability to numerous real world problems in different areas, like financial data prediction [99], robot control [63], designing electronic circuits [245] antennae for NASA's space mission [149], image classification [220], object detection [40, 260]. This has been and will continue to be an important area for future research.

2.3 Repetition in Genetic Programming

This section reviews how repetition has been achieved previously in GP history and what the advantages and disadvantages are, as the task of the thesis is to promote the use of loops with restricted formats in genetic programming.

Repetition is useful for many problems, and the *for-loop*, as mentioned in Chapter 1, is a widely used structure in many programming languages [2, 18, 58, 113]. In contrast to considerable research on loops in other programming languages, there are few studies on loops in genetic programming.

The review of repetition is organised into three sections. Previously used *for-loop* formats in GP are reviewed in Section 2.3.1, how *for-loops* were restricted to avoid infinite iteration is reviewed in Section 2.3.2 and previous experimental problems with loops or potential problems with loops are described in Section 2.3.3.

2.3.1 Loop Formats Used in GP

Broadly speaking, loops in GP have been achieved implicitly and explicitly.

Implicit Loops

Implicit loops in genetic programming have been briefly mentioned in Chapter 1 (Section 1.1, 6). Details of their implementations are described here.

Ciesielski, Mawhinney & Wilson 2002, Robot Soccer Problem

Looping can be achieved implicitly by putting it within a node. For example, in order to evolve successful robot soccer players, Ciesielski et al. [45] provided *moveTo* as a function in the evolution. The *moveTo* takes a *position* as an argument and moves the robot forward. Internally, this function utilises a *loop*. (*moveTo TheirGoalPos*) uses a loop to move the robot repetitively in steps from the current position to *TheirGoalPos*.

Koza 1992, Santa Fe Ant Problem

Looping can be achieved externally by the environment. In the Santa Fe ant problem [141], the evolved program is executed repeatedly to direct the ant to eat all the food. Besides solving the problem, the objective of the evolution is actually evolving the body of a loop instead of a single program that is only executed once. The details of the Santa Fe Ant problem are described in Section 2.3.3 (page 63).

Strengths and Weaknesses

The strengths of the implicit approach are: The implicit approach is easy to set up and there is no need to explicitly specify initial values, updating processes, how to pass values, and how to restrict the number of repetitions to avoid infinite iteration. The evolved programs are small and easier to understand because there are no complex looping structures.

The weaknesses of the implicit approach are: The explicit steps in execution of the evolved program may not be easily understood, because repetitions are hidden in the nodes or by the environment. Formulating these nodes with loops and setting up the environments needs human knowledge. The user needs to formulate them by their understanding of the problem. The embedded loops are fixed in structure and this limits their flexibility.

Explicit Loops

Explicit loops have been used in genetic programming but there have been only a few investigations which used them. The objective of most of this prior work was to solve a problem at hand and not the study of the formulation and effects of loops.

Koza 1992, 9-Block Stack Problem

An iterative function *DU* (do until) was first used by Koza to solve a 9-block stack problem [123, p461]. The goal of this 9-block stack problem is to produce a single stack of blocks that spells "UNIVERSAL" from a variety of initial configurations. Every block is always either part of the stack or on the table.

The terminal set consists of the following: CS, which dynamically specifies the top block of the stack; TB, which specifies the highest block on the stack and whether all blocks below it are in the correct order; and NN, which specifies the block that should be on top of TB in the final stack.

The functions are: MS, which takes a block as its argument and, if it is on the table, moves it to the stack and returns T (otherwise it returns NIL); MT, which takes a block as its argument and, if it is anywhere in the stack, moves the top block of the stack to the table and returns T (otherwise it returns NIL); NOT, which is the normal LISP boolean negation function; and EQ, which is the normal LISP equality predicate.

As well as the above functions, there is a DU function and it takes two bodies of code, both of which are evaluated repeatedly until the second returns non-NIL. The format is

(DU WORK PREDICATE)

Spector [226] extended the research done by Koza for the block stacking problem and pointed out the limitations of Koza's approach: 1. It can only solve a single stack problem. 2. It uses an unusually powerful set of functions and terminals. Spector used a different macro that implements a limited iteration control structure and the format is

(DO-ON-GOALS BODY)

in which the body is limited to accept certain values. However, this is also a higher level function so that a major part of the implementation is buried in the function. Both loops are a kind of unbounded loop in our definition (see page 8) in which the number of times the body will be executed is not known at entry.

Both works were successful in terms of evolving programs to solve the problem, but neither

of them provided any empirical results. Also, neither of them provided a comparison of results and analysis of solutions with loops and without loops, because examining looping was not the main objective and DU or DO-ON-GOALS are just functions that help to get the solution.

The strengths of the iteration formats are: They are simple in structure and are customised for this problem. The weaknesses are: The DU function is carefully tailored to the specialized nature of the domain; the refinement by Spector has not significantly advanced the use of iteration by slightly modifying the format.

Koza 1999, Computing Average in A Vector Problem, Automatically Defined Functions and Automatically Defined Loops

Automatically Defined Functions have been suggested by Koza [124] to allow genetic programming to form more complex structures - sub-routines that can be reused at different points in the program. In automatically defined functions, a number of function-defining branches and a result-producing branch are specified for each individual and the result-producing branch can call the function-defining branches multiple times. Automatically defined functions have been tested in a number of problems [124, 241] and have been extended to address issues of iteration, looping, recursion and storage [125]. These extensions of automatically defined functions are called automatically defined iterations (ADIs), automatically defined loops (ADLs) and automatically defined recursion (ADRs). They are ways to utilise iteration explicitly [125, p122-p154].

In [125], Koza et al. used a compute-the-average problem as an example to demonstrate the use of automatically defined loops. The problem is to compute the numerical average of all elements in a vector. If such computation is performed by a C program with a for-loop, it will involve the following steps. Firstly, the program initializes a total value holder to zero. Secondly, the program utilises a for-loop and traverses every element in the vector and adds it to the total value holder. Thirdly, the program divides the total value by the length of the vector and gets the result.

For convenience, an individual with ADLs is used to explain how automatically defined loops work (see Figure 2.6) The *progn* function has two branches, a loop definition branch and a result branch. The loop definition branch has a set of defined sub branches which take different values and performs the loop initialization, updating and condition checking tasks and specifies the loop body. In the loop definition branch (defloop), SETV1 sets variable V1 to a value of 0.

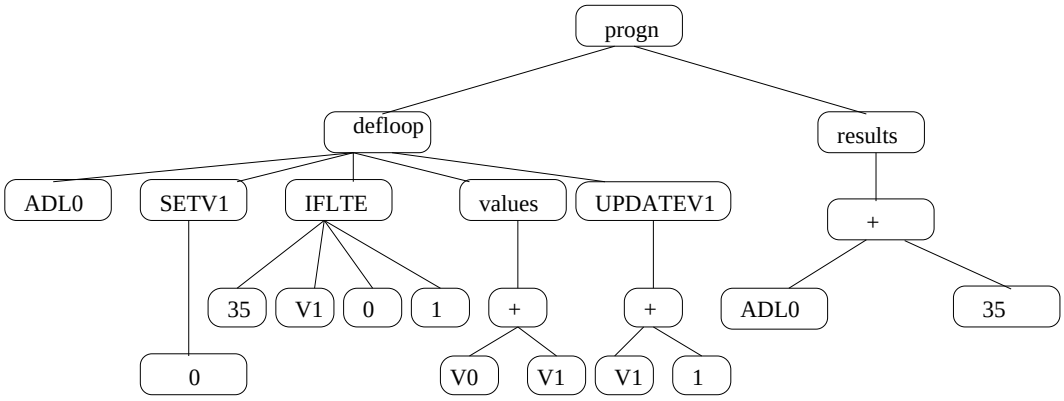


Figure 2.6: An example of automatically defined loops

IFLTE is an if-less-than function and compares the value 35 and the value in variable V1. If the comparison value is true, it equals 0 and the loop exits, otherwise, it returns 1 and the loop will continue. The loop body is the *values* branch and variable V0 is added to variable V1. The value of V0 is taken from the environment while V1 is a local variable and is initialised in the SETV1 branch. UPDATEV1 is the updating branch and increments the value of V1 at each iteration. The result branch adds the value returned from the loop definition branch referenced by ADL0 to 35 (a value returned by random value terminal).

The work was successful in terms of evolving programs to solve the compute-the-average problem and a sorting network problem and in terms of clearly demonstrating that ADLs helped to decrease the number of evaluations.

The strengths of the ADF and ADL approach are: They allow a degree of freedom in capturing useful patterns, and they create loops automatically by the power of evolution and reuse them at any point in the result branch, thus saving the effort of evolving them again if other parts of the program or their descendents need to use them. They clearly define the branches of the program separating the main program with the sub-functions. However, there are a number of weaknesses. Firstly, setting up environments for ADFs or ADLs needs major adjustment of types for functions and terminals and adjustments to genetic operations. Apart from specifying the result and loop definition branches and restricting crossover and mutation from those branches, the user needs to specify the number of the ADLs allowed and a mechanism to reference them in the result branch. Secondly, it is hard to control the program size. The sub-branches and the arity of the possible functions used by the loop definition branch tend to generate a very large bushy tree and it is hard to control the overall program size. Thirdly, [125]

states that the ADLs allow nesting, but it is hard to find concrete examples of nested ADLs. In addition, nesting significantly increases the evaluation cost for programs with ADLs. These considerations explain why Koza et al. restricted the evolved programs to allow only a few ADLs [125, p322] and Köppen and Nickolay [122] discarded the use of ADFs for their image processing algorithms. There are very few publications on learning of automatically defined functions and automatically defined loops. A recent report by Nanduri [179] gave an empirical comparison of automatically defined functions with a population management method - decimation (see Section 2.2.6, page 36). The report shows that programs with automatically defined functions used more time to evolve and evaluate than decimation for a number of classic genetic programming problems. The results of the report also showed that there was no improvement in the success rate for programs with ADFs.

Kinnear 1993, Sorting Algorithm Problem

The task of sorting is to reorder an array of elements so that elements will be in order by the key. Details of the sorting problem will be explained in Section 2.3.3 (page 66).

Kinnear [117, 118] used an iterative operator with an index variable to evolve a sorting algorithm. The format of the loop described in the work is

(dobl start end work)

Two special terminals, *index* and *len* are used to hold the sequence index and length of the sequence. Each occurrence of *dobl* will not iterate more than 200 times and the sum of all *dobl* iterations within a single test will not exceed 2000.

The experiment was successful in evolving sorting programs that can sort arrays correctly up to a length of 40. In his experiments, 20 out of 60 runs achieved success.

The strengths of his approach are: The format is generic. The weaknesses of this approach are: It needs hard coded limits to restrict each iteration as well as the overall number of iterations. The focus of the paper is on evolving sorting algorithms, not generalising or experimenting with iteration formats, so the format and specification of the loops used are domain specific and only cater for that sorting problem. He did not comment on or provide an analysis of issues relating to loop formats and comparisons without loops.

O'Reilly and Oppacher 1992, The Sorting Vector Problem

O'Reilly and Oppacher used GP for evolving a program to sort vectors [191]. They used two specialised looping formats derived from the following functional notation:

(do-until loopVar startValue endValue varDelta loop-test true-form)

loopVar is set to *startValue* and each time through the loop is changed using *varDelta*. *StartValue* is compared to *endValue* using the relation *loop-test*. If the result is true, *true-form* is executed. *True-form* can be a program.

The two derived *loop* functions for iteration are:

(do-until-Up-with-*j* startValue endValue true-form)

(do-until-Down-with-*i* startValue endValue true-form)

Variables $*j*$ and $*i*$ work as implicit *loopVars*. $*j*$ is always incremented by 1 using an implicit *varDelta*, while $*i*$ is always decremented by 1. At each iteration, *true-form* is executed if $*j* <= endValue$ or $*i* >= endValue$. The details of the sorting problem will be explained in Section 2.3.3 (page 66).

O'Reilly and Oppacher claimed that they got 3 correct programs that could sort arrays ranging in size from 2 to 6, but they did not provide the evolved solution and did not provide empirical results of the runs. Thus, we cannot determine how successful the experiment was.

The strengths of these formats are the same as Kinnear's approach and they are suitable for this vector sorting problem. The weaknesses are the same as before: lack of analysis on how these functions affect the evolutionary process and lack of comparison with programs without loops. The objective of the work is to demonstrate that hierarchically forming complex programs from general functions in GP is feasible and worthwhile and the work does not specifically address or promote the use of loops.

Langdon 1996, The List Structure Problem

Langdon used an explicit loop structure

(for-while s e l)

for evolving a list structure [132].

The *s* is the initial condition. The index of the loop *i0* is assigned the value of *s* and *l* is the list. When the for-while function returns zero, the loop exits. According to Langdon's specification, loops can only be contained in an ADF and an ADF must contain at least one loop. Nested loops are not allowed. As the focus of his work was on evolving a list data structure from basic primitives, the utilisation of loops was put to second place.

Langdon was successful in evolving list structures. Two out of 56 runs produced solutions which passed all the tests.

The strengths of this approach are: It allows the evolutions to develop different loop sub-branches and it is optimised for this list problem. The weaknesses of this approach are similar to those described in ADLs and the Kinnear's loop format. It needs a hard coded limit to restrict the overall number of iterations and nested loops are not allowed. The focus of the paper is on evolving a list structure, not a generalised iteration format. The format used is domain specific and caters only for this list structure problem.

Finkel 2003, The Integer Factoring Problem

The solution to the integer factoring problem *factor(n)* returns a list of the prime factors of n [71]. For example, factor(24) returns {2, 2, 2, 3}.

Finkel [71] utilised a *do-while* structure to evolve an algorithm to correctly factor positive integers. The format is

(do-while arg_cond arg_body)

The *arg_cond* returns a value less than one or greater or equal to one to indicate whether the *arg_body* will be executed or not. In his setting, nested loops were allowed. His do-while loop is a type of unbounded loop in our definition (see page 8). He restricted each *do-while* to a maximum of 100 iterations and all *do-while* loops were collectively allowed a maximum of 200 iterations.

Finkel was successful in his final run to get a solution for integers up to 100. But he did not state how many runs in total were used to find this solution and did not state the success rate.

The strengths of his approach are: The format is less restrictive than some of the others and simplified for evolving the factoring problem. The weaknesses of this approach are: the *arg_cond* is hard to automatically evolve, so constraints are always needed to avoid infinite loops. The objective of the paper is to show how to evolve a correct algorithm, so no analysis of the performance of each function, including loops, is given. He stated that one of his future goals would be to optimise the number of iterations to evolve faster algorithms. The format used in the work is domain specific.

Chen and Zhang 2005, The Factorial and The Modified Ant Problem

Chen and Zhang defined two explicit while loop structures

(WhileLoop1 start end body)

(WhileLoop2 condition body)

for evolving solutions for the factorial problem and a modified ant problem [37].

Their work was motivated by our early work [43] and extends our results to another problem and a loop structure variation. Their *WhileLoop1* is the same as our loop format 2 (see page 75) and their *WhileLoop2* is an unbounded loop (see page 8). They simplified the ant problem and applied *WhileLoop2* to the modified problem.

Their experiments with *WhileLoop2* were successful and 42 out of 50 runs found solutions. They also noted that when they used the for-loops described in our work [43], they got the same number of solutions in 50 runs. However, GP with their WhileLoop2 was more effective in improving the fitness and used fewer evaluations to solve the problem.

The weakness of this work is that their results were empirical only and lack a serious analysis of why GP with WhileLoop2 was better. Is it because of changing the problem or because of using a highly customised while-loop condition? Our work explores the reasons why loops were beneficial by analysing the search space and the evolved patterns for programs with loops (see Chapter 5, page 144).

Overall, apart from the ADL approach, which takes the 'general' into consideration, the other work by Kinnear [117, 118], Langdon [132] and Finkel [71] is limited. They focus on solving specific problems and have not addressed the issues of whether the loop formats can be used for other problems, whether efficiency could be improved by loops and whether more sensible solutions could be evolved. In their reports, the benefits of loops have not been pointed out and there are no comparisons for runs without loops. Although the ADLs take the 'general' into consideration, they suffer from high computation costs and high complexity. In a way, our approach is not fully 'general'. However, we attempt to make the loop formats 'general' while keeping them simple and easy to use. We have proposed several formats and applied each of them to a number of problems and provided analysis of each. By using our formats, we have demonstrated that even with these simple loop structures, evolution can still find useful patterns to repeat for success.

Recursion

Recursion can be regarded as another form of repetition. Brave [30], Wong and Leung [257], Whigham [250], Yu and Clack [259], Agapitos and Lucas [3, 4] designed different methods to evolve recursive functions or behaviours for genetic programming. In [3], they have successfully used recursion together with higher-order functions to evolve a sorting algorithm with complexity

$O(n \times log(n))$. In this work, they also have designed a number of fitness functions to measure the sequence order. However, their work did not relate to loops directly.

2.3.2 Prevention of Infinite Loops

Prevention of infinite loops is a major issue in evolving loops in GP. Numerous combinations of terminals and functions are allowed in the evolutionary process and infinite loops are highly likely. These infinite loops will keep executing without generating a result as they keep iterating the same pieces of code. However, it is hard to design a method to avoid infinite loops while not interfering with the evolutionary process because it is hard to know in advance how many times it is necessary for the evolved pieces of the code to be repeated to get a solution.

For implicit loops, the environment settings by the user restrict the allowable number of iterations. In the robo-soccer problem [45], the stop condition is that when the location is reached by *moveTo*, the iteration will stop. Infinite loops are not possible.

Five methods have been proposed for explicit loops.

A maximum number of execution steps is set in advance to avoid infinite loops. In the Santa Fe ant problem [123, p461], the evolved body will be repeatedly executed, but only a maximum of 600 steps of movement are allowed. Once the maximum number is reached, the evaluation will be stopped and a fitness will be assigned to the program based on the pieces of the food left. This setting needs a counter for the evaluation process and the fitness of the problem needs to have *cumulative* characteristics, that is it improves gradually and partial success is possible.

A maximum number of iterations together with a maximum number of total iterations are set to avoid infinite loops. For evolving the sorting algorithms [117], the function *dobl* was set to allow a maximum of 200 iterations and the total number of iterations for a single program was not to exceed 2000. Langdon [132] set a small number of iterations (32) for the search for the list algorithm. Finkel [71] used the same strategies and allowed 100 maximum iterations for *do-while* and 200 iterations in total for an individual.

A maximum number of iterations can be set dynamically based on the tree-depth of the individual. Brave [30] used GP to evolve programs with recursive ADFs to perform a tree search. A recursive ADF may incur infinite repetition. He specified the depth of the tree being searched as the limit of the number of recursive calls to avoid this problem. This is

somewhat similiar to setting a hard limit on the number of iterations in a loop.

A CPU time limit can be set for each evaluation to avoid infinite loops. Wong and Leung [257] used an execution time limit to halt the program in their grammar guided evolution to generate a recursive generalised solution for the even-parity problem. This approach suffers from the same problem as setting a maximum number of iterations for loops and may discard some potentially good solutions because they cannot be evaluated within the specified time frame.

A coroutine execution model can be set to avoid infinite loops. The model proposed by Maxwell [165] tolerates individuals with infinite loops or recursion while still allowing evolutionary progress and can be regarded as an extension and refinement of the first two approaches - setting a maximum number of execution steps or a maximum CPU time limit. In this model, each individual is allowed to execute for some amount of time in a pseudo-parallel manner and a fitness is given for the partially evaluated programs. Newly formed offspring are executed until they reach the same age as the rest of population and then synchronised with the population. For example, in the Santa Fe ant problem, every individual can be evaluated for a fixed time, then fitness is assigned based on the current situation and the evolutionary process will use this fitness for the selection and generation of descendents. The weakness of this model is also the same as for setting a maximum steps, that is it needs the programs have the *cumulative* characteristic so that a partial fitness can be accurately given to the partially evaluated individuals.

Currently, the most frequent practice is to set a maximum number of iterations for each loop or to set a maximum number of iterations for each individual or a combination of these two methods, because the methods are generic, independent of problem domain and easy to configure.

2.3.3 Experimental Problems

A number of problems which have been used in earlier work and have utilised loops implicitly or explicitly, or have the potential to utilise loops, are used in this thesis. They are described in this section. They are the Santa Fe Ant problem [141], a sorting problem [118] and a symbolic regression problem [95].

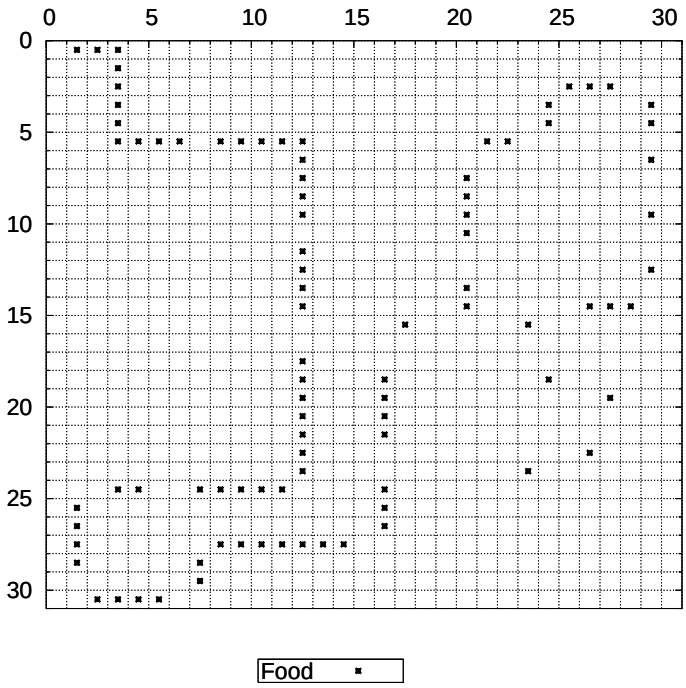


Figure 2.7: The Santa Fe trail for the artificial ant problem

The Santa Fe Ant Problem

The Santa Fe ant problem was introduced by Koza [123] and has been frequently used as a benchmark problem in GP [101, 123, 138, 139, 171, 186].

The objective of the problem is to evolve a program that controls a robot ant to follow the Santa Fe trail. The irregular Santa Fe trail is located in a 32x32 grid and consists of 89 pieces of food. The trail is not straight and contains 21 turns. The food scattered is not continuous and it has single gaps, double gaps and triple gaps. Some of these gaps are at corner positions which increases the difficulty of the traversal. A map of the trail is presented in Figure 2.7. Food is represented by black squares.

The goal is to evolve a program that can successfully direct a robot ant, which starts from the top left square, to eat all the food along the trail within a limited number of steps. The frequently allowed number is 600 [43, 141]. Some work has different counting strategies and requires the ant to complete the task in 400 moves [101, 156]. The fitness is based primarily on the number of pieces of food eaten.

There are three terminals in the ant problem which represent three primitive actions: {*turnRight, turnLeft, move*}. *TurnRight* and *turnLeft* change the facing direction of the ant to the right or

left by 90 degrees without moving the ant forward. *Move* pushes the ant forward in the direction it is facing. If it moves into a square that has food, the ant consumes it. Every primitive action costs one step.

There are three functions in the ant problem, which are {*ifFoodAhead, Prog2, Prog3*}. *IfFoodAhead* is a sensor function for the ant with an arity of 2. It inspects the square in front of the ant. If there is food in that square, the first argument is executed and if not, the second argument is executed. *Prog2* and *Prog3* are linkage functions and they take 2 or 3 arguments and execute them sequentially. These three functions do not count towards the steps used by the ant.

The commonly used genetic parameters for this problem [123, p114] are population size 500 to 1000, crossover rate 90% to 70%, mutation rate 0% to 30%, reproduction rate 1% to 10%. The maximum allowed tree size is usually small, ranging from 5 to 7. Researchers have developed different strategies for finding solutions or finding better solutions in smaller size and increased understandability [101, 139, 161, 162, 171, 242].

We consider that explicit loops can be used for this problem despite the usual practice that iterations are conducted implicitly, that is, the evolved program is executed again and again until the ant uses up the allowed number of steps or eats all the food on the trail. In this approach the evolved programs are generally small. Executing the evolved code once cannot complete the task. The restriction by maximum number of steps avoids infinite iterations. An example solution and the traversal pattern can be viewed in Figure 2.9 and Figure 2.8.

Overall, the special characteristics of this problem make it an ideal problem to test the use of loops and to benchmark situations where loops are not used. It can be used to understand why explicit loops are useful and why solutions with loops can be evolved quicker than those without.

The Sorting Problem

Sorting is one of the fundamental problems of computer science. The task of the sorting is to reorder an array of elements so that elements will be in order by the key. In many current applications, large arrays need to be sorted. Designing good algorithms is in high demand in many areas. However, applying genetic programming to evolve a sorting algorithm is not an easy task.

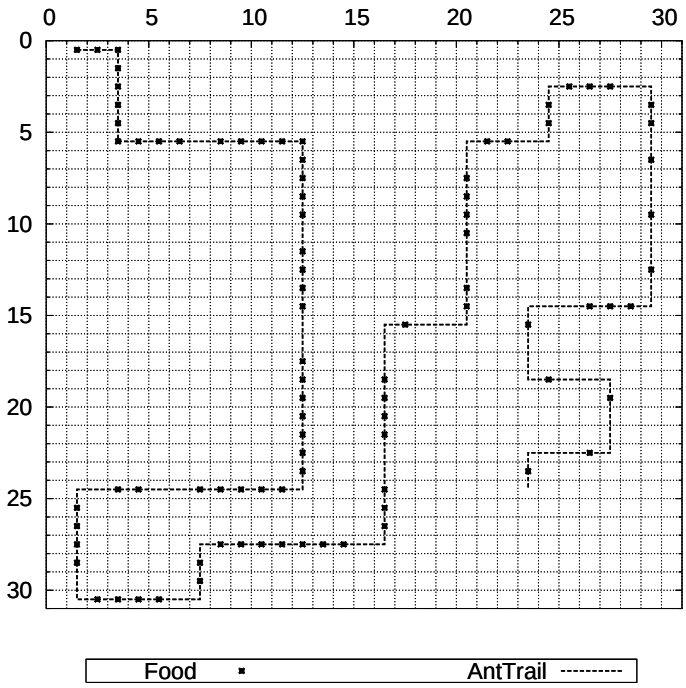


Figure 2.8: The traversal path for the artificial ant problem achieved by a perfect solution, see Figure 2.9

(Prog3 move (Prog3 (IfFoodAhead (Prog2 (IfFoodAhead move turnLeft) turnRight) turnRight) (IfFoodAhead (IfFoodAhead (Prog2 (Prog2 move move) (IfFoodAhead turnRight move)) (Prog2 (Prog2 turnRight turnLeft) move)) turnLeft) turnLeft) (IfFoodAhead (Prog2 move move) turnRight))

Figure 2.9: A perfect solution for the Santa Fe ant problem

There are two kinds of sorting tasks for GP: evolve an algorithm that can sort a limited number of elements efficiently [126, 153] or evolve a generalised sorting algorithm that can sort arrays of any size [118, 191]. The second task is considerably harder for GP, because no amount of testing can ever establish generality for an algorithm that operates on an infinite domain of data [117]. The objective of the thesis focuses on promoting the use of the loops. So only experiments that have used iteration are discussed.

The sorting process requires the two primitive functions, {*swap, compare*}. *Swap* is a function of arity 2 and exchanges the array elements it takes. *Compare* is a function of arity 4. The first two arguments are elements and the second two arguments are actions. It compares the first two elements. Depending on the result, it does the action indicated by the third and fourth arguments. *Compare* can be decomposed into two primitive functions: {*wisBigger, wisSmaller*},

which-is-bigger and which-is-smaller respectively. They specify the return result and decide the follow-on actions [117]. *Swap* and *compare* can be combined into a *compare-exchange* function.

There are two methods to evaluate the correctness of the sorting. One is by enumerating all the possible orderings of the array. The other is by using the "zero-one principle" [125, p337], that is, if a program correctly sorts n binary bits into non-decreasing order for all 2^n possible combinations of n bits, it will correctly sort any set of n distinct numbers into non-decreasing order. The second method significantly decreases the number of the testing cases. For example, to ensure an evolved algorithm correctly sorts an array of length seven, 5040 test cases are needed to ensure every possible combination has been tried, but only $2^7 = 128$ test cases are needed utilising the "zero-one principle".

Incorporating loops is natural for the sorting problem. Sorting uses positions in an array as terminals and allows the above functions to manipulate the values in those positions. The loops can take advantage of repetitive compare and swap actions and reuse them to solve the problem, thus saving evolutionary effort to form these patterns again and again in a consecutive order.

The detail of three different loops formats by Kinnear, O'Reilly and Koza for the sorting problem has been reviewed in Section 2.3.1. Kinnear's work had significant success in evolving generalised algorithms [117, 118]. He found that there is a possible connection between program size and generality in evolution. The generality is inversely proportional to size. It is preferable to include the program size in the fitness calculation and it helps to evolve a 'general' sort. O'Reilly and Oppacher's work pointed out that GP needs to construct its solutions in an explicitly hierarchical manner in addition to their hierarchical representation [191]. Both investigations found that incremental learning is important for finding generalised algorithms. ADLs have been utilised by Koza et al. to solve a minimal sorting network problem. This work addresses the problem of efficiently sorting a fixed length array. They have successfully found a 16-step seven sorter and the algorithm employs the minimum number of comparison-exchange operations [125, p335].

The work of this thesis does not focus on evolving generic or efficient sorting programs with minimum comparison-exchange, but on how to use generalised formats for sorting and how the loop format can make a difference compared to the non-loop approach in terms of the number of evaluations, size and understandability of the evolved solutions. The sort experiment in this work uses a similiar loop format to Kinnear's, that is, the loop function takes three arguments

start position, *end* position and *body*.

The Symbolic Regression Problem

The task of symbolic regression is to find a mathematical expression that can provide a perfect fit between a given sampling of values and the associated values of the dependent variables [123]. Rarely have researchers used loops for symbolic regression problems. However, loops could still be helpful for some symbolic problems like the quintic polynomial [124, page 118] in Equation 2.2 or the sextic polynomial [124, page 110] in Equation 2.3.

$$f(x) = x^5 - 2x^3 + x \tag{2.2}$$

$$f(x) = x^6 - 2x^4 + x^2 \tag{2.3}$$

In [124], Koza used automatically defined functions to capture the regular patterns in these two problems and found ADFs saved evaluation effort. This leads us to think of the use of loops to take advantage of these repetitive calculations.

The set of problems proposed by Hoai et al. [95] provides a way to gradually increase the potential repetitive patterns. The target functions used are:

$$F1(x) = x^2 + x$$
$$F2(x) = x^3 + x^2 + x$$
$$F3(x) = x^4 + x^3 + x^2 + x$$
$$F4(x) = x^5 + x^4 + x^3 + x^2 + x$$

There are four binary functions: {+, -, *, /} and four unary operators: {sin, cos, exp, log}. x is the only terminal for these problems. The fitness is the aggregate differences between the evolved function with the actual function at 20 random points in the interval [-1, 1]. The system used a population of size 500 with a tournament size of 3 and the maximum allowed number of generations was 30.

In their approach, they used tree adjunct grammar guided GP, a variation of Whigham's grammar guided GP system [248]. The work demonstrated a far better performance than the original GP approach [123].

This thesis utilises these functions and extends them to higher powers of x to demonstrate how, with incremental repetition potential, loops can be incrementally helpful.

2.4 Classification by Genetic Programming

The work conducted in this thesis involves a difficult object classification task and this section briefly reviews the classification tasks, how the tasks can be achieved in GP and what the required features of classifiers are.

A classification task involves building models that are able to identify new instances as one of a set of defined classes. Classification has been used in a wide range of applications, such as face recognition [236] and diagnosing medical conditions from the output of medical tasks [207, 254].

The following sections explain the process of training and testing, discuss generalisation and over-training issues and list some of the accuracy measures in GP in evolving the classifiers.

2.4.1 Training and Testing

The process of evolving the classifier through evaluating a set of training cases is called *training*. To evaluate whether the evolved classifiers are capable of correctly classifying a set of cases which have not been involved in training is called *testing*.

The classification accuracy is measured by the percentage of correctly classified cases. Training accuracy and test accuracy refer to the number of correctly classified cases divided by the total number of cases in the training data and test data respectively.

Deciding what the training and test data will be, and how much training data is needed for a classification task is a hard problem. Methods such as cross-validation and bootstrap selection address this issue [121].

2.4.2 Generalisation and Over-Training

Normally, a successful classifier is expected to correctly classify data which has not been used in training. However, there can be a significant gap between the performance of a classifier on training and testing data. This is because the learning becomes too specific to the training data and causes poor performance for the testing set. This phenomenon is called *over-training* or lack of generalisation.

Over-training occurs mainly in three circumstances: the size of data set is too small, the data set is biased to one class, or the data set is not representative and classes are overlapping

in characteristics [9, 239]. If the training data set is too small, then it may not contain enough information or not be representative. A biased data set indicates that the training data set is dominated by one or two classes and does not contain enough information about others, therefore, cannot be successful in identifying some objects of unknown classes. If two classes of data have too many common features, it will also be hard to differentiate them in the testing even if the classifier is successful in the training.

2.4.3 Desirable Qualities of a Classifier

There are four desirable factors for a classifier. They are:

- *High Accuracy.* It is natural to demand a classifier with very few errors unless there are some other considerations.
- *Understandable.* To have an understandable classifier is essential for some domains. For example, in medical diagnosis, it is hard to convince a doctor to believe a classification system unless the doctor can understand the reason. Also, evolving an understandable classifier can help to contribute new knowledge to a domain.
- *Fast in Execution.* In many real world applications, there are time constraints. It is preferable to have a classification task done in a short period of time. For example, to automatically classify a human action to be dangerous in an image taken by airport video, classification speed is crucial so that there is time to prevent a disaster.
- *Fast in Training.* In most situations, training time is less important than execution time. A classifier can be trained off-line for several days or months and then applied to a system. However, in some other extreme cases, the environments are continually changing and the classifiers need to be updated to adapt to the new situations.

2.4.4 Forms of Classifiers by GP

GP has mainly been used to develop three types of classifiers. They are *numeric expression classifiers* [81, 152], *decision tree classifiers* [107], and *classification rules* [107].

Numeric Expression Classifiers

A numeric expression classifier is represented by a mathematical expression which returns numeric value. The numeric output value will then be interpreted to a class value. For a two-class problem, zero is generally selected as the boundary point to distinguish the two classes [147]. Any positive value returned by the mathematical expression is classified as one class and a negative value will be the other. For multi-class classification, the problem can be broken up into multiple two-class problems or the range of real values can be divided into a number of chunks to indicate different classes [261].

There are a number of investigations that successfully used the numeric expressions to handle classification problems [66, 151, 220]. In [66], Eggermont et al. compared a numerical classification method with a method using Boolean functions and claimed that the method using Boolean functions was transparent but lacked flexibility. In [151], Loveard and Ciesielski tested five different numerical methods to classify a set of data chosen from the UCI machine learning repository [180] and found that the dynamic range selection method, in which a subset of training examples are used to determine the class boundaries, are well suited to the task of multi-class classification. In [220], Song et al. used dynamic range selection and static range representations to evolve classifiers for a set of texture images. They found that the dynamic range representation approach have good performance over a variety of texture data.

In this work, the classification task performed in Chapter 4 uses numeric expression classifiers.

Decision Tree Classifiers

A decision tree is a tree structure in which non-terminal nodes represent branches on one or more attributes and terminals indicate the class. GP has successfully been used in training decision tree classifiers for a number of problems, ranging from artificial [123] and UCI sample data classification [64, 178] to real world medical data classification [76].

Classification Rules

Classification rules are rules represented by "if-then" structures. Several investigations in GP have attempted to generate classification rules that are comprehensible to human interpretation [53, 232, 263].

2.4.5 Issues for GP in Evolving Classifiers

Genetic programming is a good method to evolve classifiers. It can perform classification by numeric expressions; it allows search in a large search space and a different variety of solutions can be found which may provide deeper information on the problem; the evolved classifiers are normally quick in execution time and can perform very fast classification; and it allows domain constraints to be built into the fitness function which is difficult to achieve with other classification methodologies.

However, certain disadvantages accompany these good aspects in evolving classifiers in GP and further research is needed.

- **Long training times.** GP is a population-based search algorithm. Normally it takes many evaluations and a much longer training time to get a successful classifier than other algorithms, such as C4.5 [107]. The performance of each run varies and normally many runs are needed to get a successful solution, thus, it is not efficient. Cost sensitive classification methods have been implemented with GP [145], but much more improvement is needed.

- **Hard to understand.** The classifiers found by GP are generally large in size and formed by a random ordering of terminals and functions, thus are hard to interpret and will not be accepted for some cases as explained in Section 2.4.3 (page 69).

GP has been applied for various classification tasks for some real world problems [51, 86, 137, 194, 210]. It is preferable to design some new techniques to overcome these disadvantages and improves the techniques to make GP a more useful and more desirable choice in classification tasks.

Surprisingly genetic programming has also been used for a knowledge discovery task involving millions of records [65]. In [65], Eggermont found that genetic programming was able to perform a global search for a model, in contrast to the local greedy search of most traditional machine learning algorithms [78] and a user can easily choose, change or extend a representation which is convenient and useful for the data classification task.

The thesis will utilise loops to solve a difficult classification problem and explore the advantages of using loops for this kind of problem.

2.5 Summary

This chapter has presented a literature survey of the major work to date in genetic programming, loops in the genetic programming and a brief review of classification by GP. We have identified the key areas of research that relate to the work in this thesis and described those problems or techniques which will be used in the following chapters.

Chapter 3

Two Explicit For-Loop Formats

3.1 Introduction

In this chapter, we propose two explicit *for-loop* formats, test whether they can be used in genetic programming and determine whether they can provide some benefits to the evolutionary process. The reasons for the very low use of loops have been discussed in Section 1.1 (page 3). This chapter establishes that there are some easily formulated *for-loop* formats that can be used to advantage in GP.

Two formats of explicit *for-loops* with restricted syntax and semantics are presented to solve five problems, which are (1) a modified Santa Fe ant problem [43] (2) a sorting problem for an array of limited length (see Section 2.3.3, page 66), (3) the Santa Fe ant problem (see Section 2.3.3, page 63), (4) a symbolic regression problem (see Section 2.3.3, page 67) and (5) a visit-every-square problem [148]. The reasons for selecting these problems and using the proposed *for-loop* formats are: (1) we want to explore the use of the explicit *for-loops* for a range of problems of different characteristics and complexities in order to establish that there are significant problems which can be solved with loops; and (2) we want to try different *for-loop* formats with different restrictions on these problems in order to demonstrate that loops can be easily incorporated into genetic programs and are not hard to use (see Section 1.1, page 4). These problems have natural looping constructs in their solutions. A solution without loops is not possible unless the tree depth is large enough, while a large tree depth setting may dramatically decrease the chance of success. Cumulative probability of success and mean best fitness graphs are used to show that explicit loops are useful. With these two constrained explicit loop formats, GP finds

solutions with fewer generations and evolved solutions are generally smaller in size, thus, more easily understood than those without loops.

3.2 Chapter Goals

The main goal of this chapter is to answer the first two research questions of the thesis (see Section 1.3.1, page 9), that are:
1. How can we restrict the syntax and semantics of for-loops in a way that avoids problems of infinite loops and still provides useful benefits for genetic programming?
2. Can GP with for-loops solve some problems that cannot be solved or are very difficult to solve without explicit loops?
These questions have been divided into the following sub-questions:

- How can explicit *for-loops* be used in GP? What modifications are needed to utilise loops in a standard GP system?
- Will the use of loops improve the evolution, so that a solution can be found in fewer generations? Is the average size of the solutions smaller?
- Are simple loops with semantic restrictions easier to evolve than more complex loops with less restrictions?
- Will loops be more beneficial when the potential for loops in a problem is increased?
- What is the sensitivity of the evolution to different genetic parameter settings?
- Are there any problems that cannot be solved without loops?

Following the exploration of a number of problems, a sensitivity analysis is conducted to decide what the key factors are for the evolutionary process improvement.

3.3 Syntax and Semantics of the For-Loops

Two variations of *for-loop* formats have been composed by investigating the characteristics of problems and reviewing the loop structures used previously in GP and other programming languages (see Section 1.1, page 3 and Section 2.3, page 53).

3.3.1 Loop Format 1

The syntax of the first for-loop format is:

(*for-loop1 num-iterations body*)

and the semantics of *for-loop1* are straightforward. *Body* is executed *num-iterations* times. During evolution, both *num-iterations* and *body* undergo crossover and mutation.

The reason for composing this loop structure is that we want to examine whether the simplest kind of loop can be evolved. The evolution just needs to find *num-iterations* and the components of *body*.

Simple Loops

In the case of the simple loops, the value of *num-iterations* is restricted to a special integer type. The value is set to a random number between 1 and a programmer supplied value of *max-iterations*. During crossover and mutation, typing is preserved so *num-iterations* can only be changed to another integer of the same type.

Unrestricted Loops

In the case of the unrestricted loops, the value of *num-iterations* can be set by any functions. This could involve the mathematic functions {+, -} and nested *loops* are allowed.

3.3.2 Loop Format 2

The syntax of the second *for-loop* format is:

(*for-loop2 start end body*)

The semantics are also straight forward. *Body* is executed once for each value of a counter between *start* and *end*. If *start* is greater than *end*, *body* will not be executed.

The reason for composing this loop structure is that for a considerable number of looping problems, an index is needed in *body*, so that *body* can utilise the updated variable for traversal of arrays or vectors. The increment value of the index is set to 1.

Simple Loops

As for *for-loop2*, *start* and *end* are restricted to an integer type. Mathematical computations are not permitted.

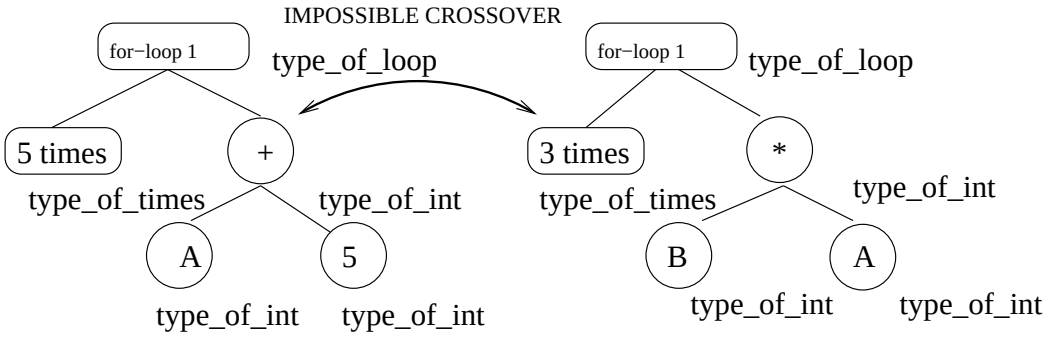


Figure 3.1: An example of impossible crossover in strongly typed genetic programming

Unrestricted Loops

As for *for-loop2*, *start* and *end* can be the result of any possible computation. Arithmetic functions {+, -, *, /} can be applied and embedded loops are permitted.

The reason for composing the simple and the unrestricted forms is that we want to explore loops with different levels of restrictions, from a very constrained approach in which nesting and arithmetic calculation of *num-iterations* or *start* and *end* are not allowed, to a totally unconstrained approach in which any computation is permitted.

In the implementation of looping in loop formats 1 and 2, the maximum number of iterations and the values of *start* and *end* are constrained by domain information, thus infinite *loops* are not possible. Unlike some previous research (see Section 2.3.2, page 61), no special actions are necessary in fitness evaluation in these formulations.

3.4 Evolution of the For-Loops

Strongly typed genetic programming (STGP) is used in the experiments (see Section 2.2.2, page 25). STGP simultaneously allows multiple data types and enforces closure by only generating parse trees which satisfy the type constraints. In genetic operations like crossover and mutation, only functions and terminals of the same type can be swapped or mutated.

For crossover operations, Figure 3.1 shows an impossible crossover operation in strongly typed genetic programming. The figure contains two individuals and both have a looping function at the root node. The looping function takes two arguments, the first argument is of type

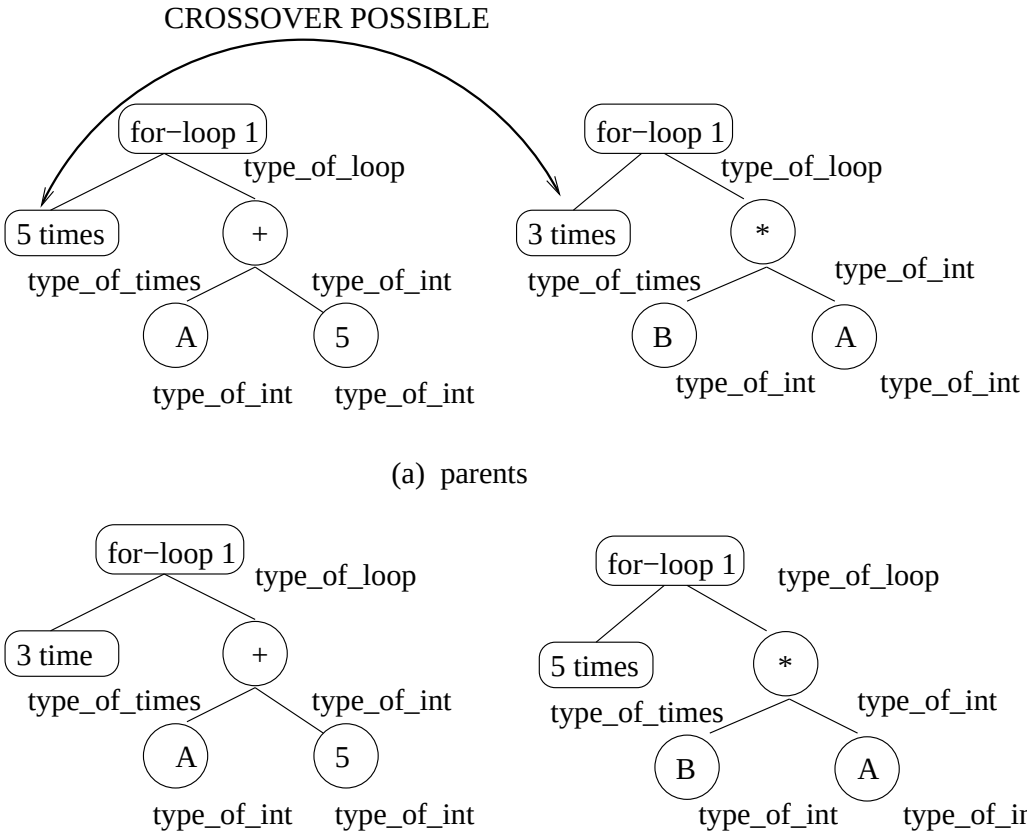


Figure 3.2: An example of successful standard crossover in strongly typed genetic programming

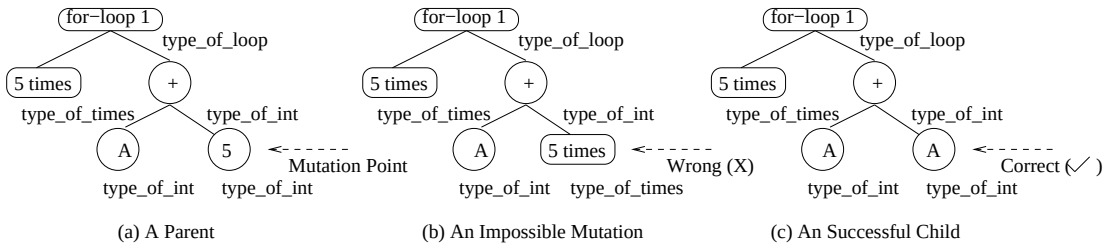


Figure 3.3: An example of the standard mutation in strongly typed genetic programming

times and the second argument is of type *integer*. The reason for using the different type settings is to constrain the times branch so that the number of iterations can be set easily and separately. When the crossover operator randomly picks the two points, terminal "3 times", which is of type *times* and function "+" which is of type of *integer* and tries to swap them, the operation is stopped because of the type mismatch.

Figure 3.2 demonstrates an example of a successful crossover in strongly typed genetic programming. This time the crossover operator randomly selects terminals "5 times" and "3 times" from two parents (see Figure 3.2a) and these two points are of the same type and can be swapped. The operation swaps the two nodes and successfully updates the two individuals to two new children (see Figure 3.2b).

Figure 3.3 shows an example of a mutation conducted in strongly typed genetic programming. The program has a *for-loop1* function as the root node. The left branch accepts type *times* and the right branch accepts type *integer*. The node "5" is selected as the random mutation point. In the figure, we can see that updating the node "5" to a node "5 times" is impossible because of the type mismatch as the node "5 times" is of type *times*. Updating the node "5" to a variable terminal "A" is correct, because they are of the same type.

During evolution, STGP takes care of type matching and ensures only that correct operations can be done.

3.5 Problem One — The Modified Ant Problem

The reason for formulating this modified ant problem is to have a problem which has obvious repetitive patterns that should be easily captured by the simple loops, since at this stage, we are not sure whether loops can be easily evolved and whether they can provide any benefits.

In previous work on the Santa Fe ant problem (see Section 2.3.3, page 63), there has been no explicit iteration in the evolved programs. Iteration is accomplished implicitly in the environment by invoking the program as many times as necessary to eat all the food or until some maximum number of steps (usually 600) has been expended.

In this work, the intention is to evolve programs in which there is no implicit looping. A program will be invoked only once, any looping behaviour must be explicitly in the program. The fitness of the program is the number of pieces of food remaining after 600 steps. In this modified problem, the size of the grid is 20×20 and 108 pieces of food are placed on the grid in

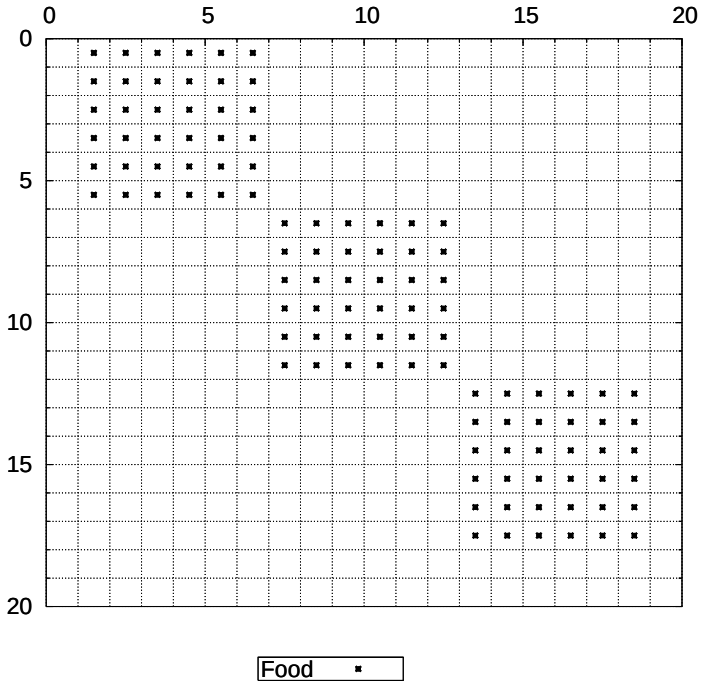


Figure 3.4: Food layout, the modified ant problem

3 blocks of 6×6 as shown in Figure 3.4. This regular placement of food is intended to encourage the evolution of loops within the evolved programs.

It is important to note that a solution to this problem by a program that is invoked only once and has no explicit loop constructs, will require a large tree. The optimal solution will require around 160 steps if the ant starts at position [0,0]. A brute force solution without loops which visits every square will need at least 400 moves and 80 turns. A binary tree of depth 9 has this capacity and this is the maximum tree depth setting for this experiment.

3.5.1 Genetic Environment Settings

Function Set and Terminal Set

The experiments for this problem replicate all functions and terminals from the Santa Fe ant problem (see Table 3.1). In addition, the experiments use the format (*for-loop1 num-iterations body*) (see Section 3.3). The terminal *RandTimes* and the function *for-loop1* are added in the loop approach (see Table 3.2).

Table 3.1: Definition of terminals and functions, standard approach, the modified ant problem

Nodes	Description
Move::Terminal	The robot moves one square forward and it costs one step.
TurnLeft::Terminal	The robot turns to its left and it costs one step.
TurnRight::Terminal	The robot turns to its right and it costs one step.
IfFoodAhead::Function	Takes 2 arguments and executes the first argument if there is a piece of food ahead, else executes the second.
Prog2::Function	Takes 2 arguments and executes them sequentially.
Prog3::Function	Takes 3 arguments and executes them sequentially.

Table 3.2: Definition of extra terminals and functions, loop approach, the modified ant problem

Nodes	Description
PlusInt::Function	Takes two integers and returns the sum.
MinusInt::Function	Takes two integers and returns the difference.
RandTimes::Terminal	Generates a random integer between 0-6 or 0-20 or 0-50.
For-Loop1::Function	Takes 2 arguments. The first argument indicates the number of times the second argument is executed. It returns the number of pieces of food left after the execution of the loop body.

Table 3.3: Variable settings, the modified ant problem

Variable Name	Value
Population Size	100
Mutation / Crossover / Elitism Rate	0.28 / 0.70 / 0.02
Maximum / Minimum Depth	9 / 1
Termination Criteria	2000 generations or all food (108 pieces) is eaten or 600 steps are reached.

Other Genetic Environment Settings

The rules to decide other genetic variable settings are:

1. If the problem is the same or similar to previously published problems, the published settings are used.

2. If the problem is new or there is no previous information, the default values are used (see Section 2.2.5, page 35) unless specified.

For this problem, the other genetic variable settings are listed in Table 3.3.

3.5.2 Experiments and Experimental Results — The Modified Ant Problem

All experiments have been run 100 times each with the functions, terminals and the environment settings shown in Tables 3.1, 3.2 and 3.3.

Experimental Results

Figure 3.5 shows the fitness of the best individual, averaged over 100 runs, for 2000 generations of evolution with *max-iterations* set to 6. These results were somewhat surprising. Since a large tree is necessary to solve the problem without loops, as described above, it was expected that programs with loops might perform better, which is the case. However, it was expected that simple loops would be easier to evolve than unrestricted ones. As Figure 3.5 reveals, the opposite was the case. This could be because *max-iterations* was too small and programs had to use more loops to capture the repeated behaviour, thus more nodes were required and this decreased the chance of finding a successful solution in the simple loops approach.

Figure 3.6 shows the cumulative probability of getting a successful solution[1], that is, the evolved ant eats all of the food. None of the runs without loops gave a successful solution. This is because it is hard to evolve the nodes without loops in an order that can solve this problem at this depth. At 2000 generations, 12 of the 100 simple loops runs and 23 of the 100 unrestricted loops runs gave a successful solution.

Figure 3.7 shows a comparison of the fitness of the best individual for different choices of *max-iterations* for simple loops. The figure shows that higher values of *max-iterations* lead to

[1]Figures relating to cumulative probability of success use number of evaluations rather than number of generation since this helps to compare runs with different population size settings.

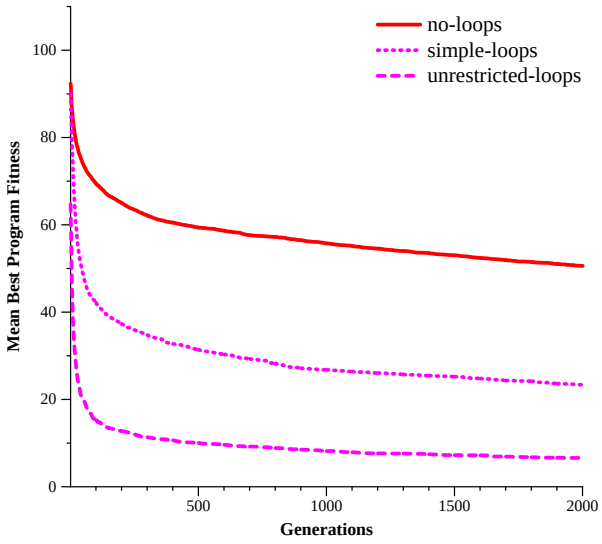


Figure 3.5: Mean best program fitness, *max-iterations*=6, average of 100 runs, the modified ant problem

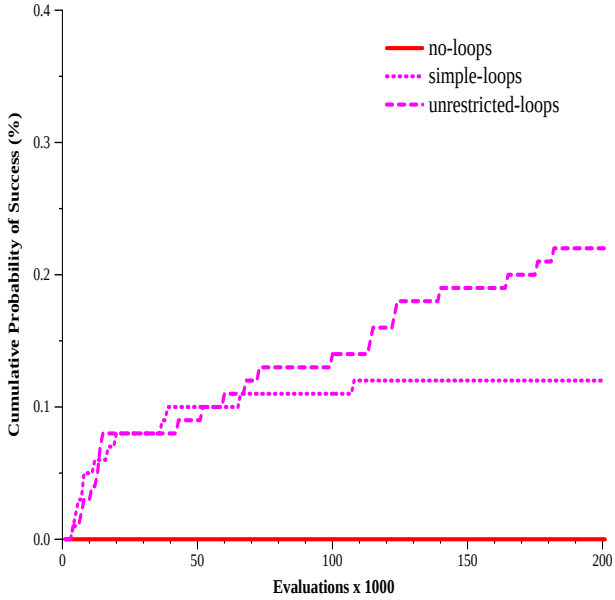


Figure 3.6: Cumulative probability of success, *max-iterations*=6, average of 100 runs, (the no loops line is on the x axis), the modified ant problem

better programs. A similar analysis for the unrestricted loops showed no difference for the same values of *max-iterations* (see Figure 3.8, page 83). There is, however, an unfortunate side effect that is not evident from the figure – the execution time rises dramatically. The 100 runs for *max-iterations* of 6, 20 and 50 took 1 hour, 3 hours and 1 day, respectively on our hardware. A detailed analysis of CPU time on the evaluation with variations of the setting of *max-iterations* will be presented later (see Section 5.6, page 159).

Since programs with a small number of loops are usually more understandable, a number of runs were performed in which the fitness function was modified to favour programs with fewer occurrences of *for-loop1*. This was done by counting the number of occurrences of *for-loop1* in the text of the program and adding it to the number of pieces of food left after program execution. Thus, if two programs consume the same amount of food, the one with fewer loops will be fitter. Figure 3.9 shows a comparison of best fitness for simple loops over the generations while Figure 3.11 shows a comparison of program size. Minimising the number of loops used in an evolved program has a dramatic effect on fitness for the simple loops but has no effect on the unrestricted loops (see Figure 3.10). Figure 3.11 reveals quite a difference in program size if fewer loops are favoured. All but one of the curves show an initial drop in program size. The

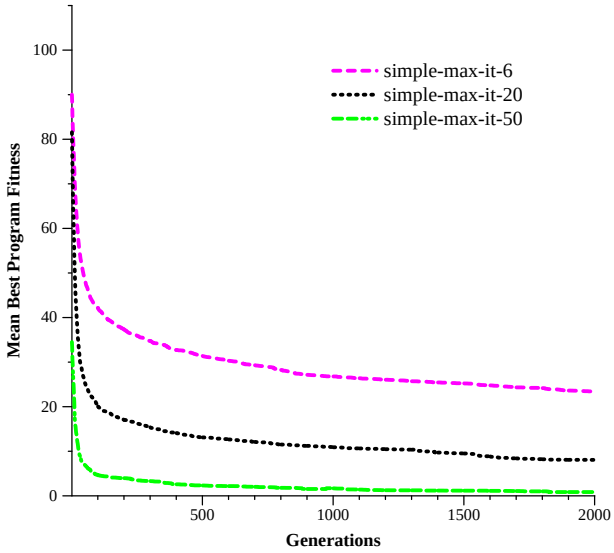


Figure 3.7: Mean best program fitness, simple loops, different values of maximum iterations, average of 100 runs, the modified ant problem

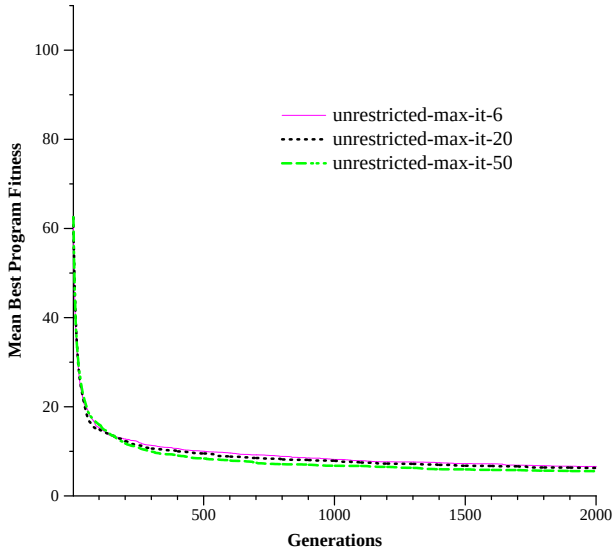


Figure 3.8: Mean best program fitness, unrestricted loops, different values of maximum iterations, average of 100 runs, the modified ant problem

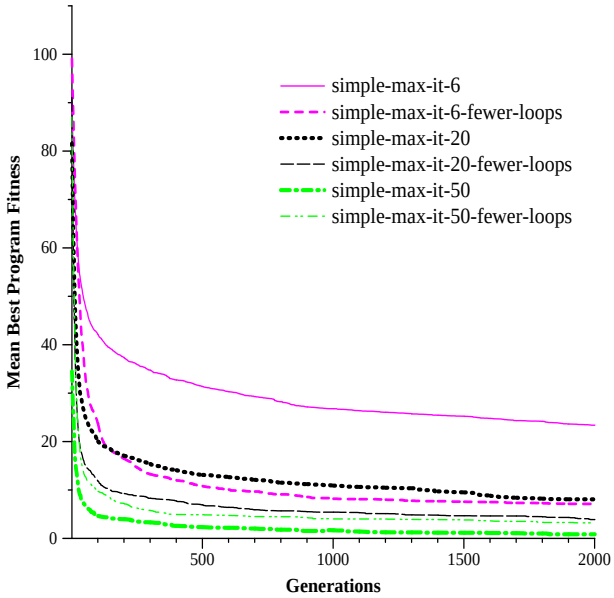


Figure 3.9: Favouring programs with fewer loops, simple loops, mean best fitness, averages of 100 runs, the modified ant problem

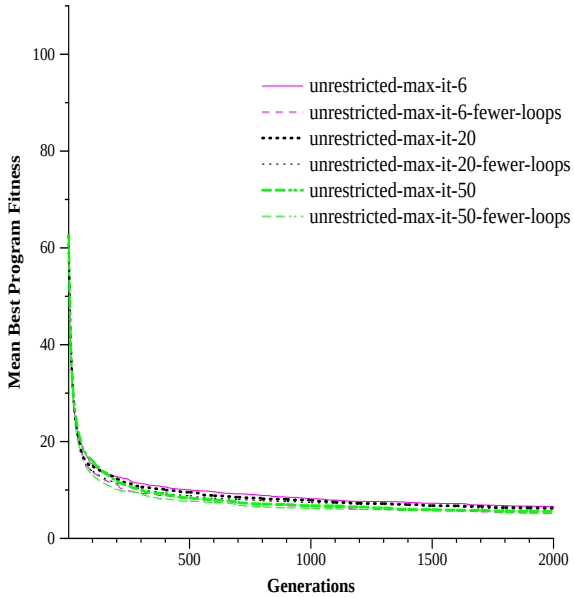


Figure 3.10: Favouring programs with fewer loops, unrestricted loops, mean best fitness, average of 100 runs, the modified ant problem

explanation for this is the following: the programs in the initial population are generated by the ramped half-and-half method. Larger programs are highly likely to have more occurrences of loops. In fitness evaluation, programs are terminated after executing 600 steps. Large programs will use up their allocation of steps before consuming much of the food and hence will not be as fit as the smaller programs. These unfit programs are not selected for mating and hence are removed from the next generation. Eventually these smaller programs increase in size as their fitness improves.

3.5.3 Analysis of Solutions for The Modified Ant Problem

The benefits of using explicit loops are shown clearly in the evolved solutions.

When *max-iterations* was large (20,50) the evolved solutions traversed every square in the grid. This is an intelligent solution to utilise the available resources to find answers quickly. A typical pattern of this is shown in Figure 3.12. The ant moves down and up, then one square to the right, then down and up again. This pattern is repeated until the ant reaches the right hand edge of the grid.

Solutions favouring a smaller number of *loops* tended to have larger *loop* bodies, smaller depth and size, and to be more understandable. An example of such a solution is shown in Figure 3.13. This solution, whose traversal pattern is shown in Figure 3.14, was found at generation 294 using the strategy of favouring programs with fewer loops. It uses 168 steps to eat the food and is close to optimal. The ant moves in a zigzag manner, moving its head left or right to detect food. If there is food ahead, it moves ahead and turns back by executing two $TurnRight$ actions. If not, it turns left. Depending on the result of sensing, the ant either does 2 forward moves or just one move and then senses again. In our setting, if an ant is on the grid border and continues moving forward, it will appear on the other side of the grid. This explains why there are three short lines at row 20. In essence, this solution shows that the evolution generated a piece of code starting from the first $IfFoodAhead$ as the loop body and repeats this 5×5×5 times to solve the problem.

In contrast, Figure 3.15 shows the smallest successful solution evolved when there was no favouring of programs containing fewer loops. This was generated with a value of 6 for *max-iterations*. The program has more nodes and fragments and it is harder to understand what the program is doing by analysing the code. Figure 3.16 shows the corresponding traversal pattern.

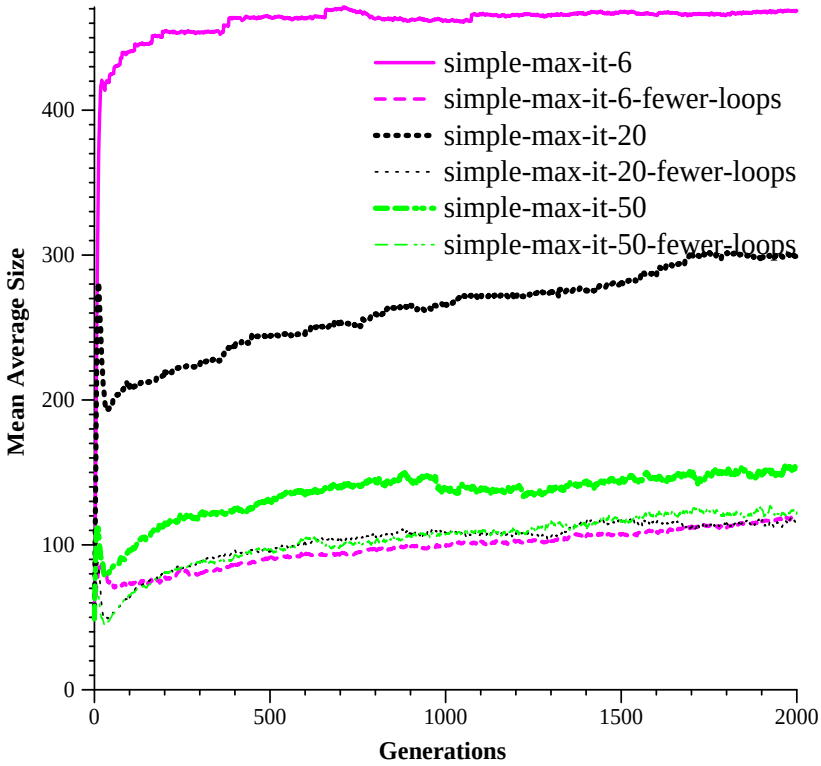


Figure 3.11: Favouring programs with fewer loops, simple loops, program size, averages of 100 runs, the modified ant problem

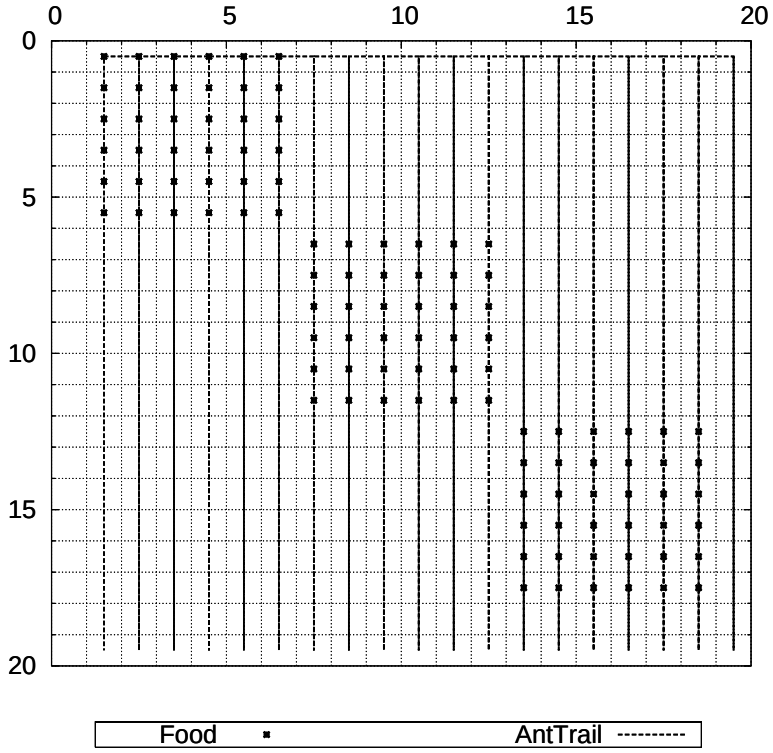


Figure 3.12: Traversal pattern for a solution evolved with *max-iterations*=20, the modified ant problem

```
SIMPLIFIED :
(ForLoop1 times5
  (ForLoop1 times5
    (ForLoop1 times5
      (IfFoodAhead
        ((Prog2 move turnRight) turnRight)
        (Prog2 turnLeft (IfFoodAhead (Prog2 move move) move))))))
```

Figure 3.13: A solution evolved by favoring programs with fewer loops, the modified ant problem

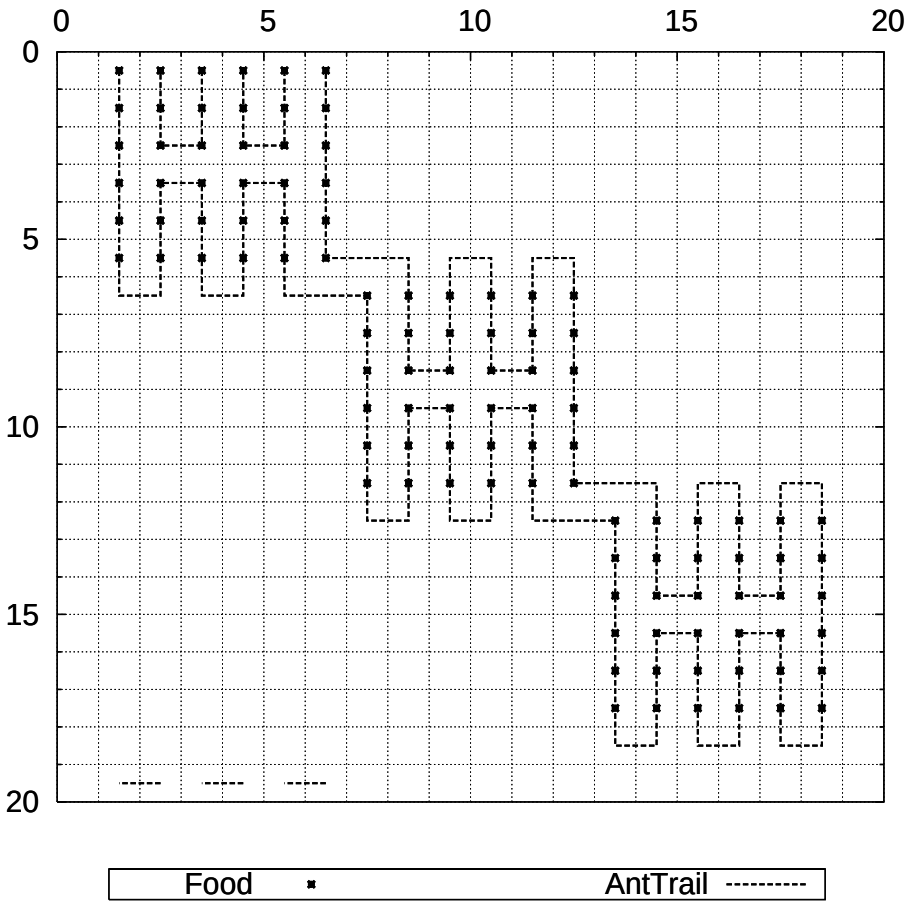


Figure 3.14: Traversal pattern of the program shown in Figure 3.13, the modified ant problem

(ForLoop1 times5 (ForLoop1 times4 (Prog3 (ForLoop1 times4 move) (Prog3 (IfFoodAhead (Prog3 move move move) move) (Prog3 (Prog2 turnRight move) turnRight (IfFoodAhead (ForLoop1 times4 move) (Prog2 turnRight turnLeft))) (IfFoodAhead (Prog3 move move move) move)) (Prog3 (IfFoodAhead (IfFoodAhead (Prog3 turnLeft turnRight move) move) turnLeft) turnLeft (Prog2 move move)))))

Figure 3.15: A solution evolved without favouring programs with fewer loops, the modified ant problem

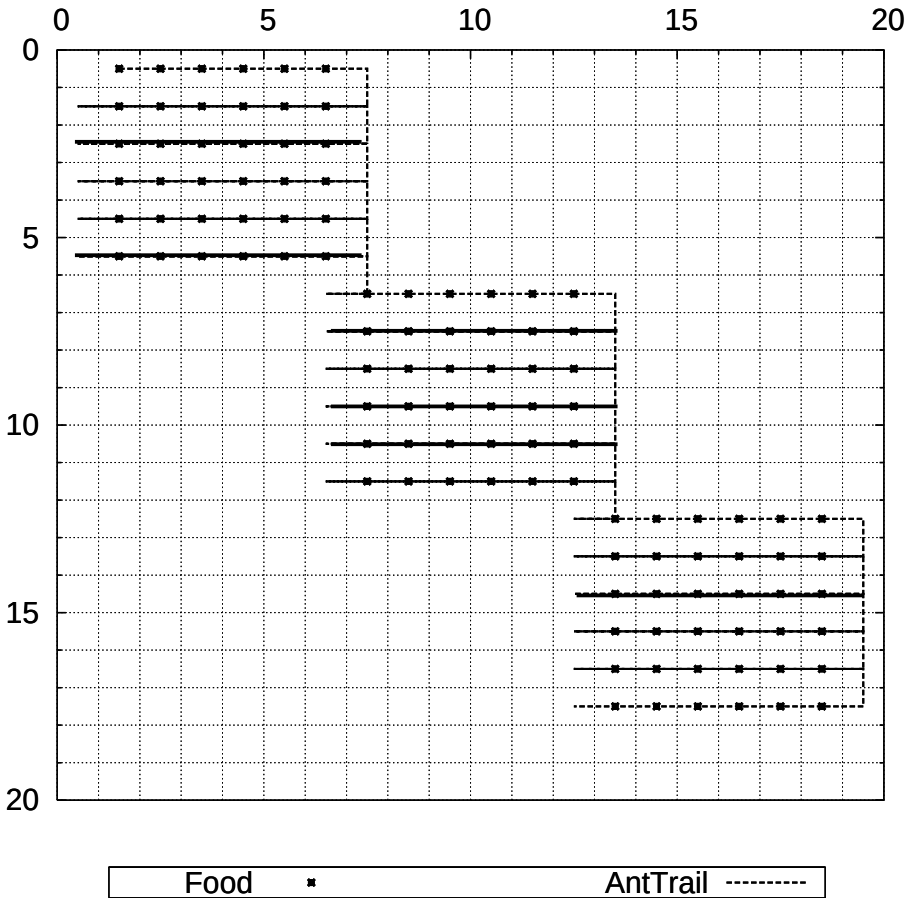


Figure 3.16: Traversal pattern of the program shown in Figure 3.15, the modified ant problem

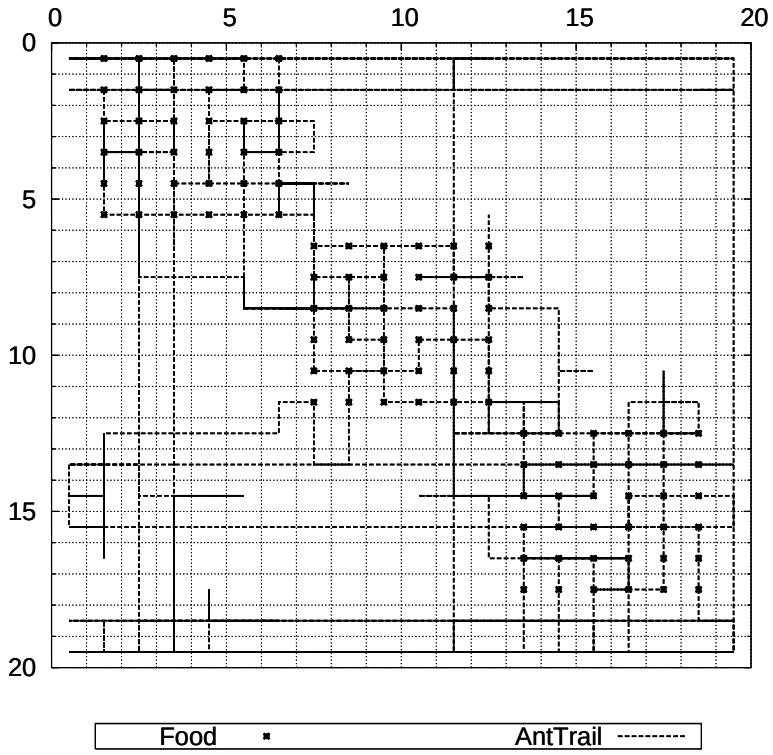


Figure 3.17: Traversal pattern of the only solution evolved by the no loops method, maximum depth=10, the modified ant problem

It is very hard to get a solution with the no loops approach and no solution was found at a maximum depth of 9. As we wanted to demonstrate a solution without loops, we increased the maximum depth to 10 and finally we succeeded. Out of three hundred runs with the maximum depth of 10, one run found an answer. The solution is enormous. It contains more than 5000 nodes and is impossible to understand. It takes four full A4 pages to print out. Figure 3.17 shows the traversal path of the solution. The ant uses 1704 steps to complete the task.

3.6 Problem Two — The Sorting Problem

The reasons for choosing sorting as the second problem are that the sorting task normally involves an array or a vector and loops are used in most human designed algorithms. In a sorting program, an index variable is used in the body of the loop. The variable used can index any element in the array or the vector. The sorting problem is suitable for experimenting with the second loop format, where there is a *start* and an *end*, and the body can use the index

Table 3.4: Definition of terminals and functions, standard approach, the sorting problem

Nodes Name	Description
Pos::Terminal	Random number in range 0..6
IfLessThanSwap::Function	Takes two arguments. If arg1 is less than arg2 the positions are swapped and the position of the larger value is returned
Prog2::Function	Takes 2 arguments and executes them sequentially.

Table 3.5: Definition of extra terminals and functions, loop approach, the sorting problem

Nodes Name	Description
ForLoop2::Function	Takes 3 arguments, start position, end position and body.
$+, -, \times, /$	Arithmetic functions with the usual meanings

variable.

There are two basic operations in sorting – comparing and swapping. Evolution of sorting programs is not well suited to genetic programming because of difficulties with fitness evaluation. It is very difficult to develop a tractable fitness function that guarantees any array of arbitrary length will be sorted after the evolved program has been executed.

Previous research on how to evolve a sorting program by GP has been described in Section 2.3.3 (page 66). The objectives of previous work were to evolve generalised sorting algorithms or to minimise the number of comparisons. Our focus is on the evolution of loops of different complexities and on the comparison of the loops and no loops solutions. The issue of a generalised sorting program is not addressed in this work.

3.6.1 Genetic Environment Settings

Function Set and Terminal Set

To minimise the number of functions and terminals used, this research follows the approach described in [124, p335]. The definitions of the functions and terminals without loops are given in Table 3.4. The extra functions for the loop approach are described in Table 3.5.

Fitness Function

The experiment only concerns arrays of length 7. Fitness is evaluated by applying an evolved program to all $7! = 5040$ permutations of the array elements, counting how far out of place each

Table 3.6: Algorithm for fitness calculation, the sorting problem

```
int calculateFitness(int _length, int *_array)
{
   int i, result = 0;
   for( i=1; i <= _length; i++)
      result += abs( _array[i-1] - i );
   return result;
}
```

Table 3.7: Variable settings, the sorting problem

Variable Name	Value
Population Size	100
Mutation / Crossover / Elitism Rate	0.28 / 0.70 / 0.02
Maximum / Minimum Depth	7 / 1
Termination Criteria	100 generations elapsed or the array is sorted.

element is and summing the values. The actual fitness calculation is shown in Table 3.6. Seven was chosen as the upper limit of array size so that the runs could be done in reasonable time.

Other Genetic Environment settings

For this problem, the other genetic environment settings are listed in Table 3.7.

3.6.2 Experiments and Experimental Results — The Sorting Problem

As before, runs were carried out with no loops, with simple loops, where START and END are restricted to an integer type, and with unrestricted loops, where START and END can be set by any mathematical calculation including loops. All experiments were conducted using the functions, terminals and other environment settings shown in Section 3.6.1.

Experimental Results

The fitness of the best individual for each method is shown in Figure 3.18. The corresponding cumulative probability of success is shown in Figure 3.19. Programs with loops are clearly fitter. In fact, for both kinds of loops nearly all runs found a solution within 40 generations, that is, a program with zero fitness. (One run of simple-fewer-loops found a solution in 87 generations). In contrast, at 100 generations, only 34 of the 50 runs without loops had found a solution.

There seems to be an inconsistency between Figures 3.18 and 3.19. The runs of the simple loops have the best possible mean fitness from the first generation and the unrestricted loops take several generations to evolve solutions, yet the cumulative probability of success rises faster for the unrestricted ones. This is because the y axis scale for the mean best program fitness is huge. The best programs with simple loops result in superior mean best fitness, but none of them actually reaches a solution in the first several generations. Because of the scale, the fitness diagram looks like the simple-fewer-loops approach gets to the solutions quicker.

The size of the best individuals is shown in Figure 3.20. Surprisingly the programs with loops are bigger than those without loops. This is because at an array size of 7, the programs without loops are still relatively small and the benefits of loops are not yet apparent. As the size of the array grows larger the no loops solution must also grow. Some preliminary work that we have done on an array size of 11 has led to similar results to the ant problem. The results show that the programs without loops were huge and the cumulative probability of success was very small. In contrast, the programs with loops were smaller and the cumulative probability of success was considerably higher.

The number of comparisons made by the best individual is shown in Figure 3.21. The programs with loops are making more comparisons. This is because programs with loops can easily have more comparisons than programs without loops at the similiar program size.

In Figure 3.20 and 3.21, the lines for the simple-fewer-loops and the unrestricted-fewer-loops stop earlier than the no loops. This is because all runs of these two methods (Figure 3.19) have achieved success before the maximum generation limit was attained.

Figure 3.22 shows one of the best evolved individuals without loops in terms of the number of comparisons and the number of swaps. Figure 3.23 shows one of the best programs evolved with the simple loops. Analysis of this program reveals a general strategy of moving large elements to one end, while the no loops program is very difficult to understand.

The results on the sorting problem are not as good as those on the modified ant problem. The main reason for this is that the sorting problem is considerably harder. It is known that sorting can be done with two nested loops, however, the limits of the inner loop need to be co-ordinated with the loop index of the outer loop. In our formulation of the problem this could only happen by random chance. This did not occur in any of the evolved programs that were analysed. The programs contained large numbers of uncoordinated loops.

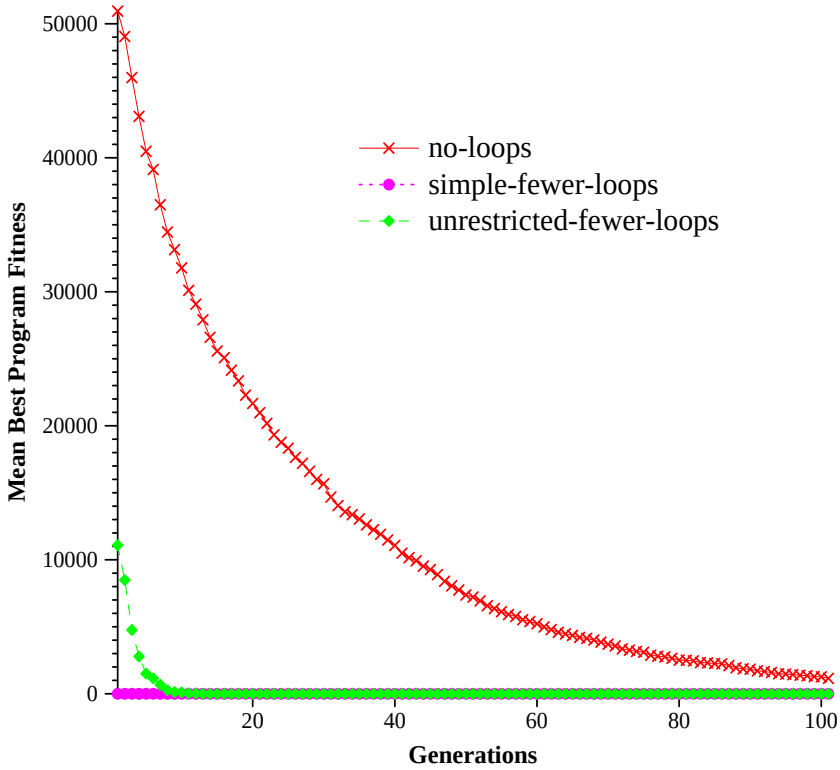


Figure 3.18: Mean best program fitness comparison, averages of 50 runs, the sorting problem

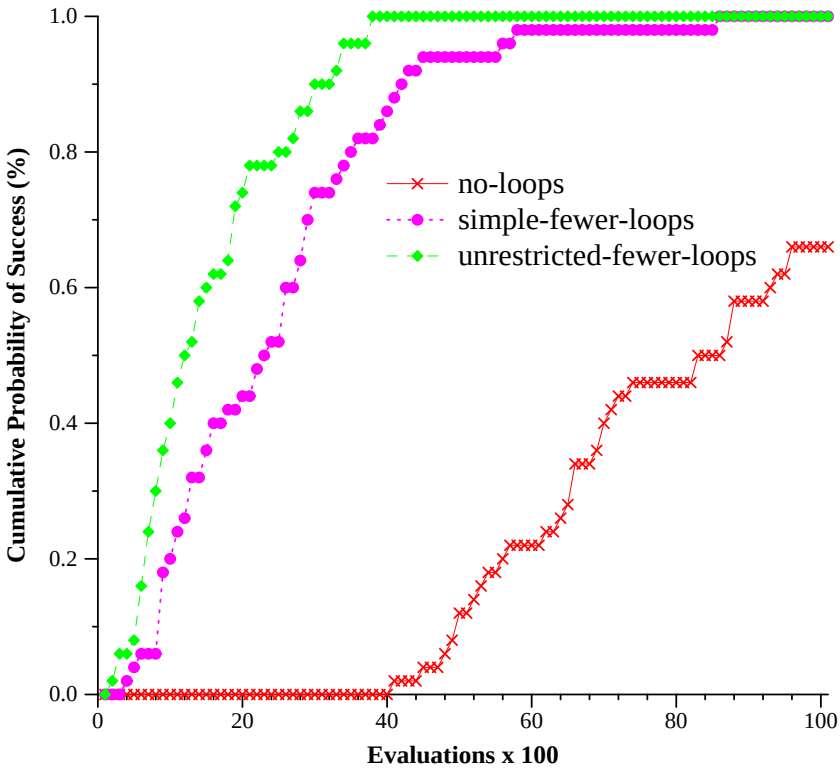


Figure 3.19: Cumulative probability of success, the sorting problem

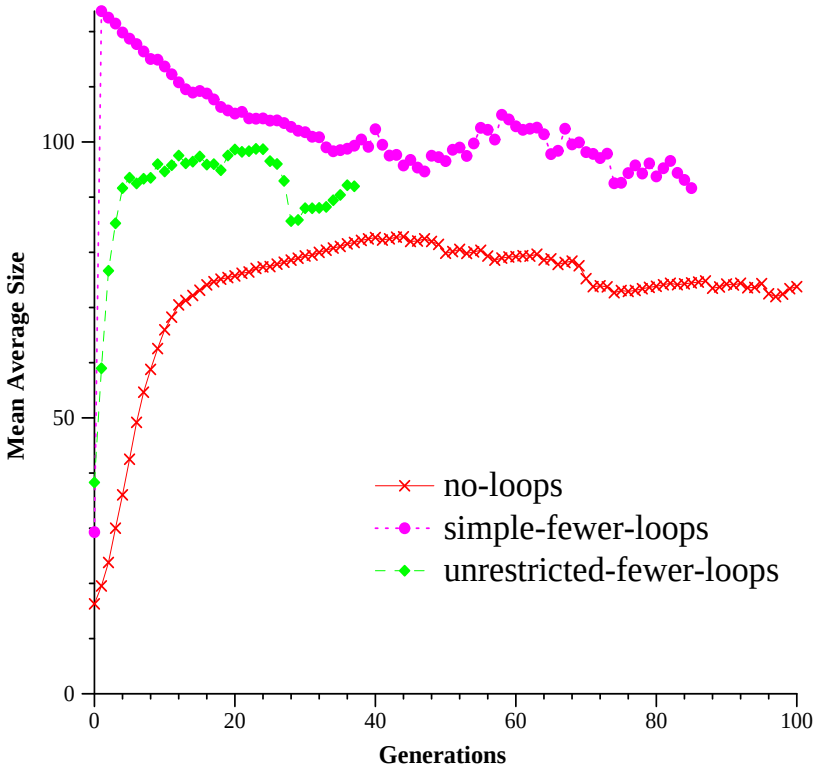


Figure 3.20: Size of the best individuals, averages of 50 runs, the sorting problem

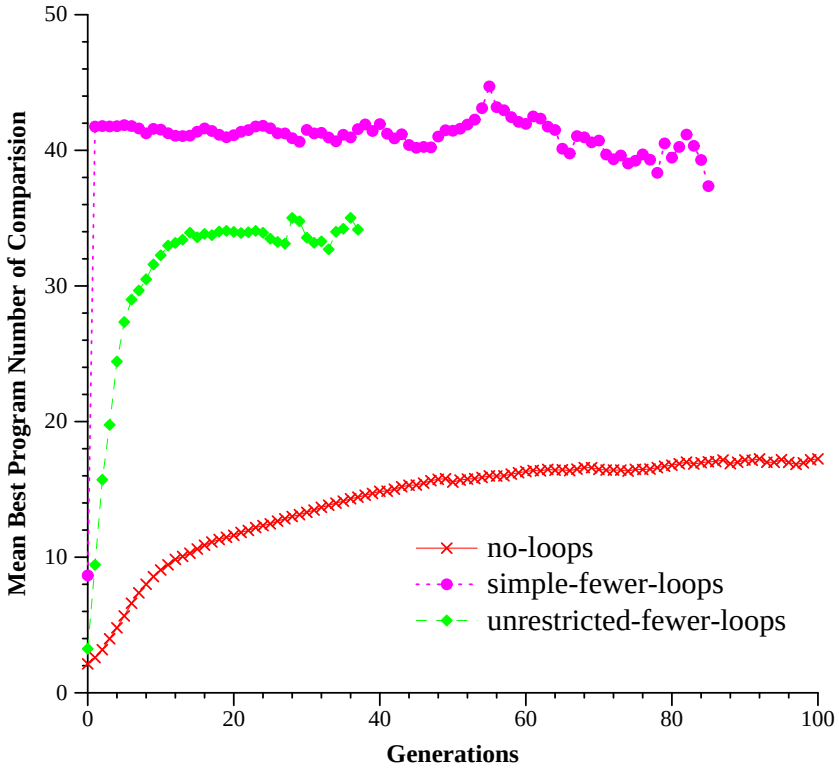


Figure 3.21: Number of comparisons made by the best individuals, averages of 50 runs, the sorting problem

```
(Prog2 (Prog2 (Prog2 (Prog2 (ILETs pos-0 pos-2) (ILETs pos-5 pos-1)) (ILETs pos-4 pos-6)) (Prog2 (Prog2 (ILETs pos-1 pos-6) (ILETs pos-0 pos-4)) (Prog2 (Prog2 (ILETs pos-4 pos-5) (ILETs pos-0 pos-4)) (ILETs pos-3 pos-4)))) (Prog2 (Prog2 (Prog2 (ILETs pos-1 pos-3) (ILETs pos-2 pos-4)) (Prog2 (Prog2 (ILETs pos-4 pos-6) (ILETs pos-5 pos-4)) (Prog2 (ILETs pos-1 pos-2) (ILETs pos-2 pos-5)))) (Prog2 (Prog2 (ILETs pos-0 pos-1) (ILETs pos-3 pos-5)) (Prog2 (Prog2 (Prog2 Dummy Dummy) (ILETs pos-2 pos-3)) (ILETs pos-4 pos-5)))))
```

Figure 3.22: One of the best programs evolved without loops, 18 comparisons and 8 swaps (ILETs = IfLessThanSwap), the sorting problem

```
(Prog2 (Prog2 (Prog2 (ForLoop2 pos-3 pos-4 (ILETs i (i+1))) (ForLoop2 pos-2 pos-6 (ILETs i (i+1)))) (Prog2 (ForLoop2 pos-3 pos-4 (ILETs i (i+1))) (ForLoop2 pos-4 pos-5 (ILETs i (i+1))))) (Prog2 (ForLoop2 pos-1 pos-6 (ILETs i (i+1))) (Prog2 (Prog2 (ForLoop2 pos-3 pos-2 (ILETs i (i+1))) (ForLoop2 pos-1 pos-3 (ILETs i (i+1)))) (Prog2 (ForLoop2 pos-1 pos-3 (ILETs i (i+1))) (ForLoop2 pos-0 pos-6 (ILETs i (i+1))))))))
```

Figure 3.23: A good program with simple loops, 22 comparisons and 10 swaps, the sorting problem

Table 3.8: Different sorting methods for 7 element arrays, 5040 test cases, the sorting problem

Methods	Comparisons	Swaps
Bubble Sort	21.00	10.50
Shell Sort	16.50	16.50
Insertion Sort	17.09	15.50
Selection Sort	21.00	6.00
Quick Sort	44.42	10.19
No loops	17.00	7.33
Simple loops	22.00	10.00
Unrestricted loops	21.00	9.00

Comparisons of the efficiency of the different approaches, as well as comparisons with standard sorting algorithms are shown in Table 3.8. For each row of the table the given algorithm was applied to all of the 5040 test cases and the number of comparisons and the number of swaps were counted. Quick sort has the highest number of comparisons for this problem and this is because quick sort does not perform well for small sized arrays. The numbers in Table 3.8 refer only to the average number of swaps and comparisons per test case. The evolved programs are competitive with conventional algorithms.

3.7 Problem Three — The Santa Fe Ant Problem

The reason for experimenting with the Santa Fe ant problem is to explore whether explicit loops can work for this classic GP benchmark problem, since loops have been found to be beneficial for the modified one.

The Santa Fe Ant problem has been described in Section 2.3.3 (page 63). In the original approach, iterations are buried in the environment and the evolved program is repeatedly evaluated until all the food is eaten or the maximum number of execution steps is reached.

The experiments conducted in this section compare the original approach (external loops) with the explicit loops approach (simple loops) and the no loops approach. In the no loops approach, the evolved program is invoked only once and no explicit loops and no implicit iterations are allowed.

3.7.1 Genetic Environment Settings

Function Set and Terminal Set

The experiments use the format (*for-loop1 num-iterations body*) which has been described in Section 3.3 (page 74). Only simple loops are examined, because previous modified ant problem experiments show that simple loops with reasonable *max-iterations* can deliver best results similar to the unrestricted loops.

The functions and terminals are the same as for the modified ant problem and can be viewed in Table 3.1 and Table 3.2.

Other Genetic Environment Settings

The tree depth setting is 10. The objective is to allow enough space for the no loops approach to find a solution as 2^{10} leaf nodes allow more than 600 steps. However, the chance of finding a solution without loops is still unknown. The other parameters are the same as for the modified ant problem. They are listed in Table 3.3.

3.7.2 Experiments and Experimental Results — The Santa Fe Ant Problem

The experiments were run 100 times each for the original approach (external loops), the simple loops approach and the no loops approach. In the simple loops approach, programs with a smaller

number of loops were favored with *max-iterations* set to 20 (*simple-loops-max-it-20-fewer-loops*).

Experimental Results

Figure 3.24 shows that the simple loops approach performed much better than the no loops approach. The best performing method is the one where the iteration control is external. The figure also reveals that without loops, it is hard to get fitness improvement for this problem.

Figure 3.25 shows the cumulative probability of success. It demonstrates the same pattern shown in Figure 3.24, that is external loops performed best with 14 solutions in 100 runs, while simple loops gave 2 solutions out of 100 runs. There were no successes for the no loops approach.

Figure 3.26 shows the average size of of the programs. The programs with simple loops or external loops are much smaller in size than the programs with no loops.

These three figures demonstrate the same patterns for this classic GP problem as in the previous two problems, that is, without loops, programs tend to be larger in size and the fitness does not improve significantly during the evolution.

3.7.3 Analysis of Solutions for The Santa Fe Ant Problem

One perfect solution obtained with explicit loops and one best program obtained without loops have been selected to demonstrate the differences.

Figure 3.27 shows the perfect solution evolved by favoring programs with fewer loops and Figure 3.28 shows the traversal pattern of this program. The results show that when favoring programs with fewer loops, the complete solutions obtained are similar to those obtained by the external loop approach (see Figure 2.8) but are not quite as good. This is because in the external loop approach, only the loop body needs to be evolved, while in the simple loops approach, the task is considerably more complex as the evolution needs to determine whether loops will be useful as well as the loop body and the number of iterations. These operations cost time and add variability to the evolution. The program in Figure 3.27 shows that most of the *for-loop* functions appear at the beginning of the program. Another finding which is worth pointing out, is that for the complete program with explicit loops, the ant continues to move after all the food has been eaten. This is because when all the food is eaten before a loop process is finished and if there are still steps left, the ant will continue to move and stop only when the loop finishes and the condition is then rechecked.

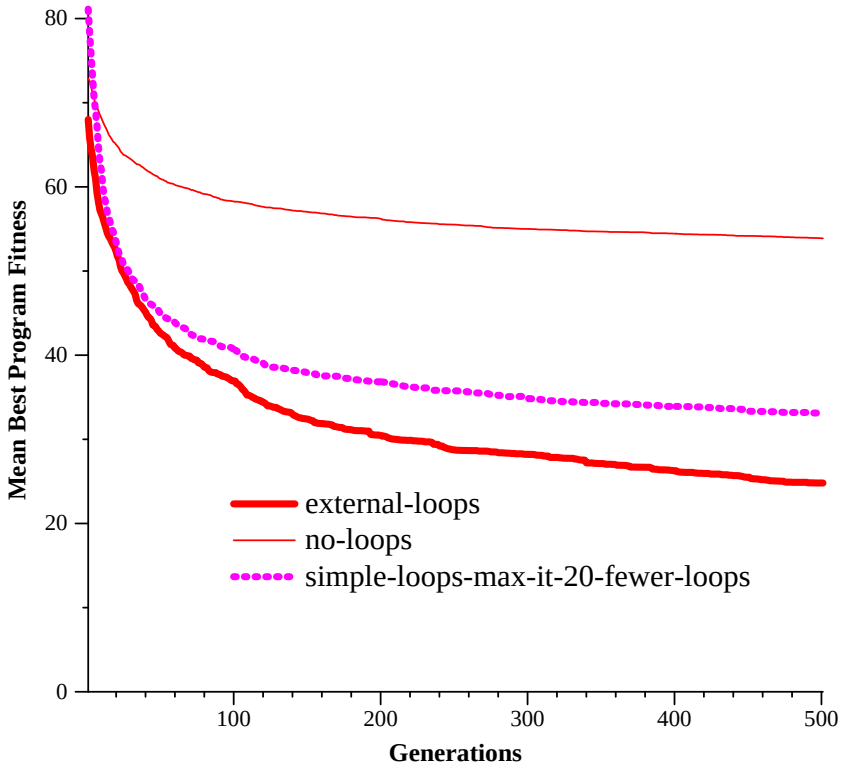


Figure 3.24: Mean best program fitness, averages of 100 runs, the Santa Fe ant problem

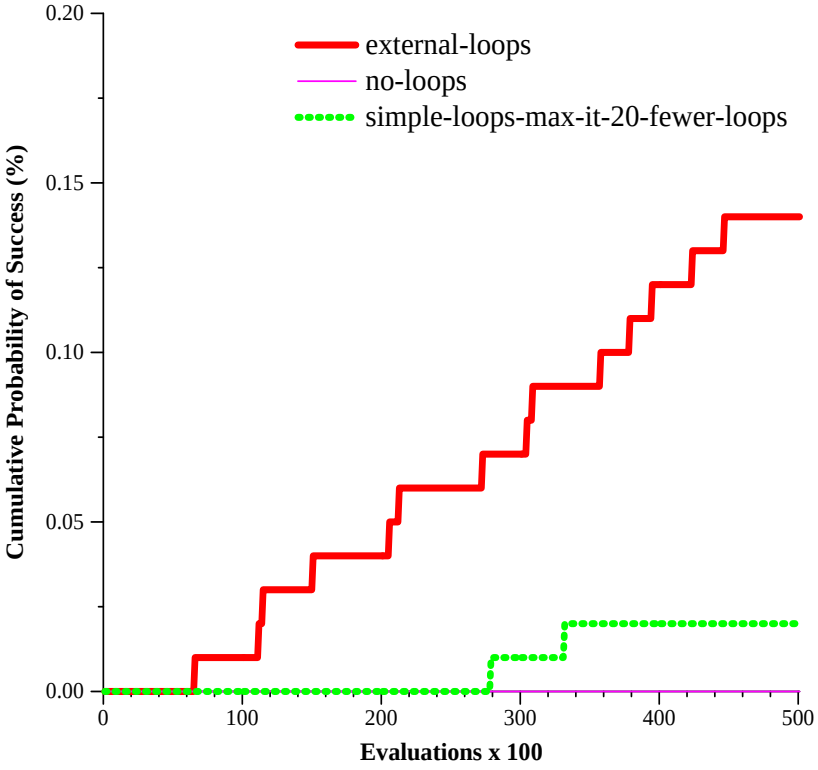


Figure 3.25: Cumulative probability of success, average of 100 runs, (the no loops line is on the x axis), the Santa Fe ant problem

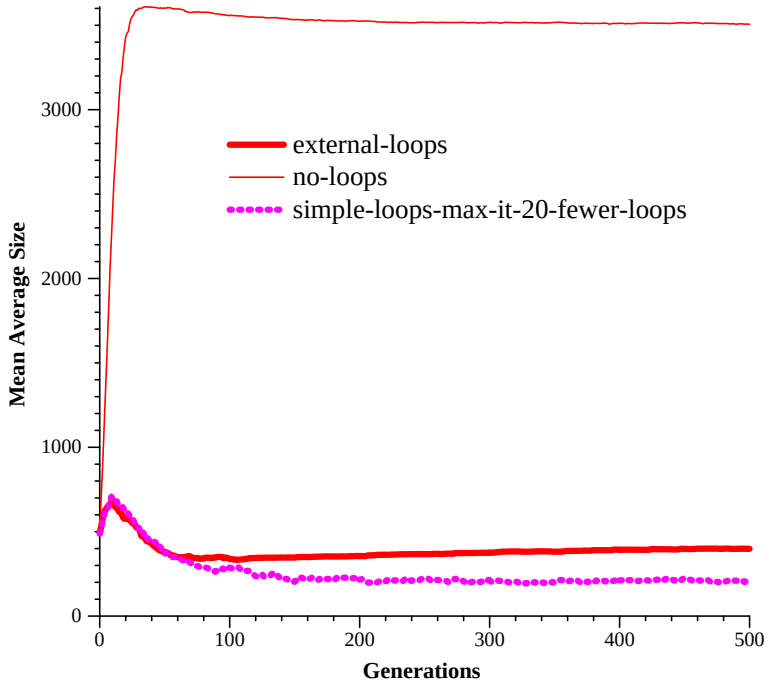


Figure 3.26: Program size, average of 100 runs, the Santa Fe ant problem

Figure 3.29 shows the traversal pattern of the best program found in the no loops approach. The ant ate 50 pieces of food on the trail and there were still 39 pieces left. The size of this program is more than 5000 nodes. It is very hard for GP to achieve a complete solution of such a large size.

The average size shown in Figure 3.26 together with the cumulative probability of success shown in Figure 3.25 suggests that the depth limit of 10 for the no loops approach may be too small. Our reason for setting a depth limit of 10 is that the original approach restricts a solution to 600 steps and maximum depth of 10 allows $2^{10} = 1024$ steps, which seems enough for a successful solution. However, in the experiments the programs quickly grew to the size limit. When programs became a full tree shape with maximum depth, they were hard to improve in this problem. A larger maximum depth may help to alleviate this, but it is clear that it will still be hard to evolve a solution without loops.

3.8 Problem Four — The Visit Every Square Problem

In the visit-every-square problem, a robot is required to navigate through every square in a $n \times n$ grid. The robot has four available actions – *left, right, up, down.* At the beginning, the robot is

(ForLoop times18 (ForLoop times16 (Prog2 (IfFoodAhead (IfFoodAhead (ForLoop 2 move) (Prog3 turnLeft turnRight (IfFoodAhead move (IfFoodAhead turnLeft move)))) turnRight) (Prog2 move (IfFoodAhead turnLeft (IfFoodAhead (Prog3 (Prog3 (Prog3 turnRight move move) turnLeft turnRight) (IfFoodAhead move turnLeft) move) (Prog2 (IfFoodAhead (IfFoodAhead move turnRight) (IfFoodAhead turnRight turnRight)) (IfFoodAhead (Prog2 turnRight turnLeft) (Prog2 turnLeft turnLeft)))))))))

Figure 3.27: A perfect solution evolved with favouring programs with fewer loops, *max-iterations*=20, simple-loops-max-it-20-fewer-loops, the Santa Fe ant problem

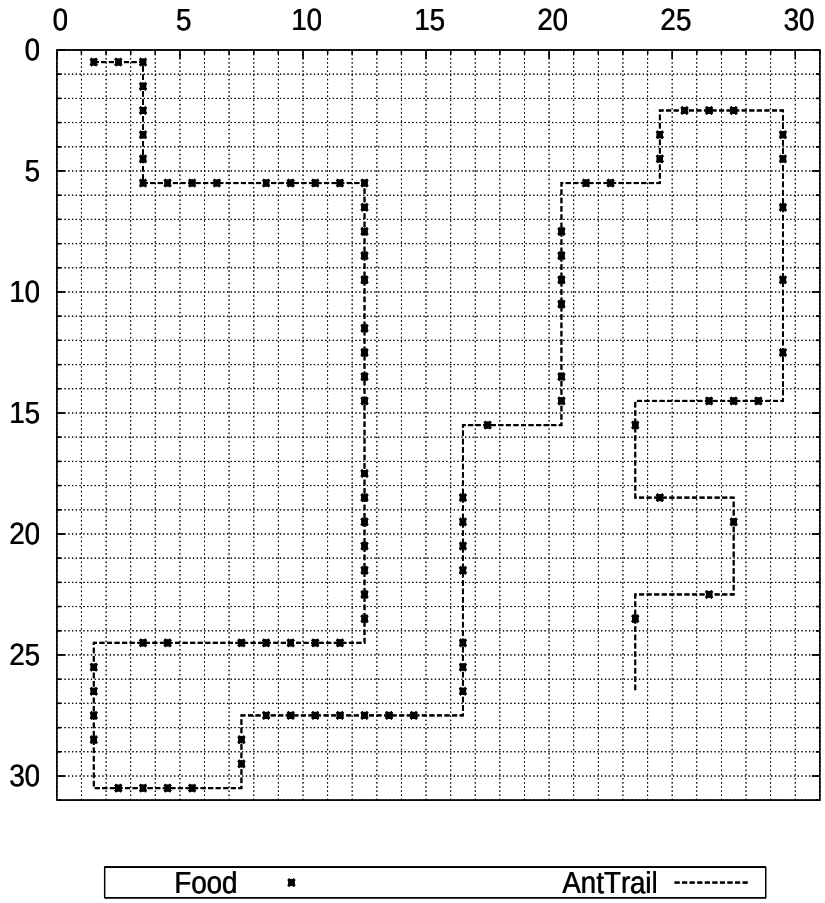


Figure 3.28: Traversal pattern for the perfect solution evolved with explicit loops shown in Figure 3.27, simple-loops-max-it-20-fewer-loops, the Santa Fe ant problem

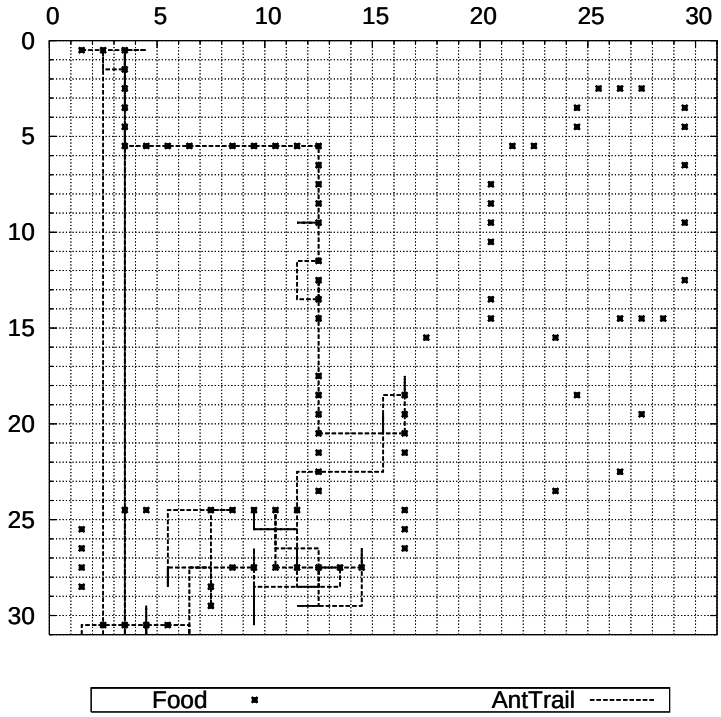
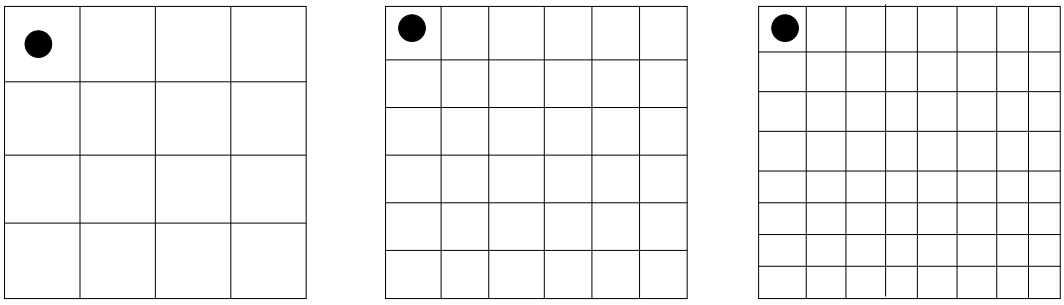


Figure 3.29: Traversal pattern for the best program found in the no loops approach in 100 runs, the Santa Fe ant problem



(a) 4×4 Problem (b) 6×6 Problem (c) 8×8 Problem

Figure 3.30: The visit-every-square problem

placed at the top left square of the grid. Each square can be visited more than once, but each square must be visited at least once in a solution. There is no penalty for visiting a square more than once.

The construction of this problem is stimulated by the idea of the Santa Fe ant problem and the modified ant problem where the search is performed in a two dimensional grid. The reasons for proposing this problem are that we want to analyse why GP with loops performs well (see Chapter 5) and to have a problem which cannot be solved without loops. It is difficult to adjust the complexity of the problems we have used so far. The difficulty of the visit-every-square problem can be adjusted with the size of the grid. The task can be made impossible for GP without loops by limiting the tree depth.

The specific task in this section is to direct a robot to navigate through a 4×4 or 6×6 or 8×8 grid (see Figure 3.30).

3.8.1 Genetic Environment Settings

Functions and Terminals

The terminals in the visit-every-square problem are simple. There are four actions represented by four terminals $Left, Right, Up, Down$. If a move through a border is required, for example, Up or $Left$ from the positions shown in Figure 3.30, the robot takes no action.

There is only one function for the no loops approach $Prog2$, which takes two arguments and executes them sequentially. The loop approach uses the simple *for-loop1* (*for-loop1 num-iterations body*) as an extra function (see Section 3.3.1).

Fitness Function

The fitness is the number of the non-visited squares, see Equation 3.1.

$$fitness = Total\ Number\ of\ Squares - Squares\ Visited \tag{3.1}$$

Table 3.9: Parameter settings, the visit-every-square problem

PARAMETERS	**VALUES**
Population Size	100
Max. Generation	50
Mutation / Crossover / Elitism Rate	28% / 70% / 2%
Termination Criteria	Successfully visited every square or 50 generations reached

Other Genetic Environment Settings

The maximum allowed depth for a program is 6, which allows 2^6=64 steps. 64 steps permit at least one solution for the no loops approach for the largest grid of 8×8.

The maximum allowed value for *num-iterations* is 50, which is large enough for any reasonable looping. The rest of genetic variable settings can be viewed in Table 3.9

3.8.2 Experiments and Experimental Results

The runs of the 4×4, 6×6 and 8×8 visit-every-square problem were conducted 100 times for GP with loops and without.

Experimental Results

Figure 3.31 shows the mean best program fitness for the no loops and the simple loops approaches. The dotted lines indicate the simple loops and the solid lines are runs without loops. The thicker the line the larger the grid size. It is clear from this figure that the larger the grid size the more difficult the program. Also, the larger the difference between the best fitness for programs with loops and without, the larger the benefit from using loops.

Figure 3.32 shows the cumulative probability of success for the six experiments. A solution means that the robot has visited every square of the grid. The plot shows the same pattern as the best fitness in Figure 3.31. For the 4×4 visit-every-square problem experiments, simple loops and no loops are similar in performance with simple loops just slightly better than the no loops. For the 4×4 problem, both approaches gave 100 solutions in 100 runs. The lines for the 6×6 and 8×8 problem for the simple loops and the no loops are contrasting. For both problems, the no loops approach did not get a single success in 100 runs, even though a tree depth 6 gives enough capacity for a solution. The simple loops approach performed well on both

problems, giving about 42 solutions for the 6×6 problem and 21 for the 8×8 problem. Figure 3.32 illustrates that with increase in grid size, the visit-every-square problem becomes harder and the no loops methods cannot adapt to the increase in the grid size.

3.8.3 Analysis of Solutions for The Visit Every Square Problem

We analysed a number of random solutions evolved for the 6×6 visit-every-square problem.

Figure 3.33 shows a solution evolved by the simple loops approach and Figure 3.35a is the traversal pattern of this program. The robot traverses in a zigzag fashion with some unnecessary steps to re-visit some squares, but achieves the objective. The size of the program is 21.

Figure 3.34 shows the best solution evolved by the no loops approach and Figure 3.35b is the traversal pattern of this program. The figure shows that the best solution for the no loops approach does not finish the task and 5 squares are unvisited. The program size is 63.

Overall, the experiments for this visit-every-square problem demonstrate that loops are beneficial. Loops help GP to get more successful solutions quicker and solutions tend to be smaller in size. Also, we have established that there are some problems that can only be solved with loops.

3.9 Problem Five — Symbolic Regression

The reason for experimenting with these three symbolic problems is to determine whether GP with loops can take advantage of increasing potential for loops and perform increasingly better than the no loops approach.

As described in Section 2.3.3, the task of symbolic regression is to find a function in symbolic form that fits a given finite sample of data. In the current GP literature, we cannot find any previous work which solves this problem with loops. In general, it is likely that the chance of repetitive patterns appearing in a symbolic function is small.

The kinds of symbolic problems used in this section are described in Section 2.3.3. They have been modified to a set of three problems with incremental potential for loops. The target functions are $F1 = x^3+x^2+x$, $F2 = x^5+x^4+x^3+x^2+x$, $F3 = x^8+x^7+x^6+x^5+x^4+x^3+x^2+x$. Functions (*F1, F2*) are exactly the same as those described earlier (see Section 2.3.3, page 67). *F3* is an extension of the problem. It increases the highest power of x to 8. We expect that the

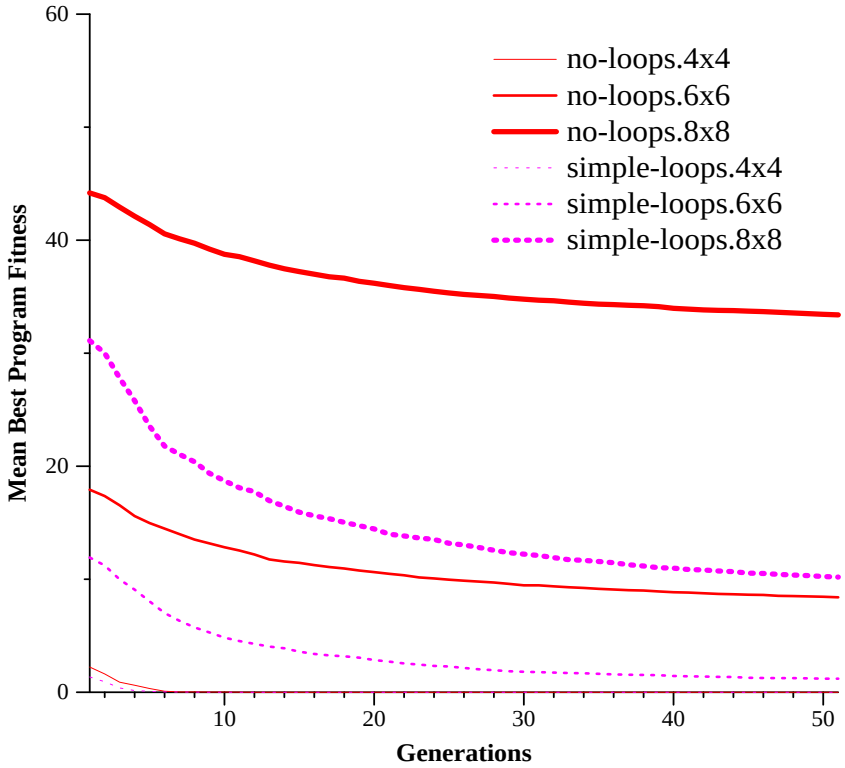


Figure 3.31: Mean best program fitness, averages of 100 runs, the visit-every-square problem

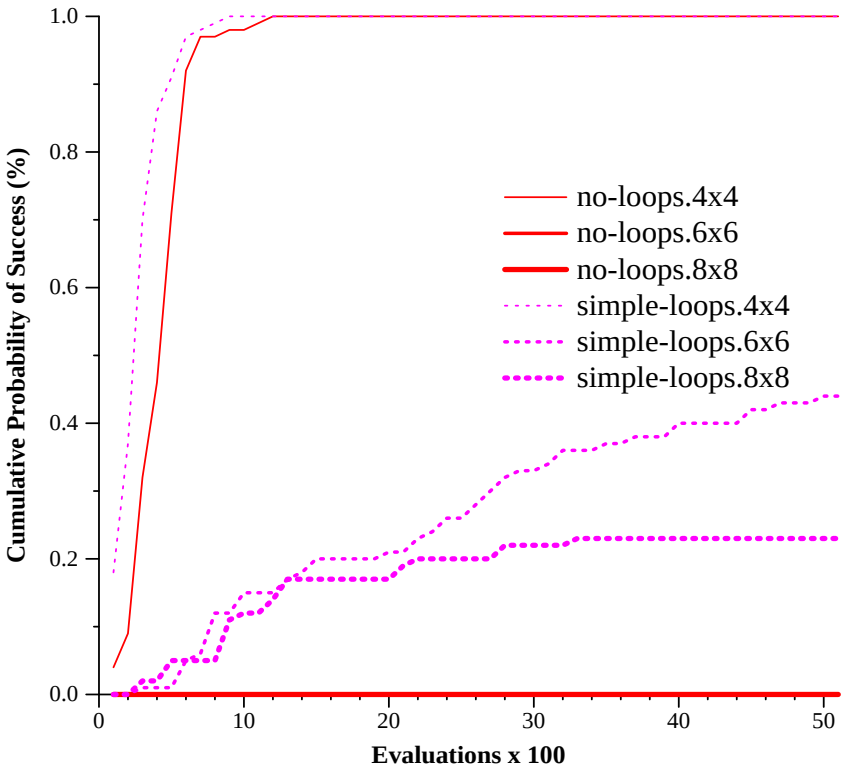


Figure 3.32: Cumulative probability of success, average of 100 runs, (the no loops 6×6 and 8×8 lines are on the x axis), the visit-every-square problem

(ForLoop t42 (Prog2 (Prog2 (Prog2 (ForLoop t6 right) down) (ForLoop t49 left)) (Prog2 (Prog2 right (ForLoop t48 left)) (ForLoop t50 right))))

Figure 3.33: A solution evolved by the simple loops method, *max-iterations*=50, size=21, the visit-every-square problem

program (prog2 (prog2 (prog2 (prog2 (prog2 down right) (prog2 right up)) (prog2 (prog2 left down) (prog2 down right))) (prog2 (prog2 (prog2 down left) (prog2 left down)) (prog2 (prog2 right right) (prog2 down down)))) (prog2 (prog2 (prog2 (prog2 right right) (prog2 up right)) (prog2 (prog2 up left) (prog2 up right))) (prog2 (prog2 (prog2 up up) (prog2 left down)) (prog2 (prog2 left down) (prog2 down down)))))

Figure 3.34: The best solution evolved by the no loops method, size=63, the visit-every-square problem

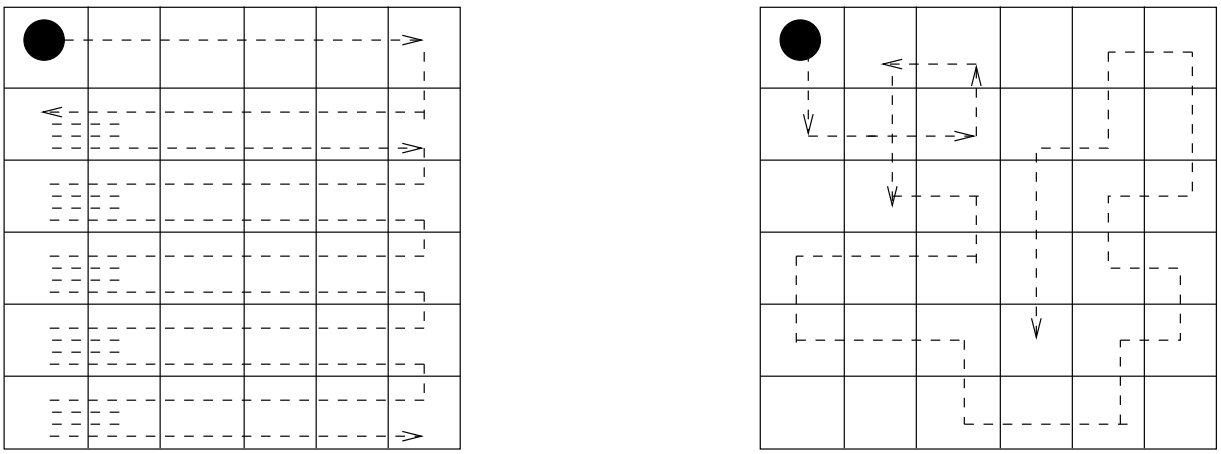


Figure 3.35: (a) shows the traversal pattern for Figure 3.33, (b) shows the traversal pattern for Figure 3.34, the 6×6 visit-every-square problem

Table 3.10: Variable settings, symbolic regression

Variable Name	Value
Population Size	100
Mutation / Crossover / Elitism Rate	0.28 / 0.70 / 0.02
Maximum / Minimum Depth	9 / 1
Termination Criteria	100 generations or the values of evolved program correctly match the target function in 50 points.

benefits of loops will increase with increasing value of the largest power.

3.9.1 Genetic Environment Settings

Function Set and Terminal Set

The experiments use x as the independent terminal. In the no loops approach, binary operators {+, *} are used as functions.

In the loops approach, the loop function (*for-loop1 num-iterations body*) is used. The syntax has been described in Section 3.3. Only simple loops are used and the semantics of *for-loop1* are revised, *body* will be multiplied *num-iterations* times. The maximum number of iterations is set to 12.

Fitness

The GP samples 50 points in the interval [0, 200] as the testing cases. The fitness is the sum of the absolute differences between the values returned by the evolved program at the different sampling points and the value from the known formula.

Other Genetic Environment Settings

Other genetic variable settings are shown in Table 3.10. These variable values are similar to values in previous experiments and the maximum depth is set to 9 to allow perfect solutions in the no loops approach.

3.9.2 Experiments and Experimental Results — Symbolic Regression

Experiments without loops and with loops have been run 100 times with the functions, terminals and other environment settings shown in Section 3.9.1.

Experimental Results

The increasing largest power value of x is regarded as an indicator of increasing potential for loops. *F3* with x^8, which has the highest power value of x among the three functions, has the highest potential for loops.

Figure 3.36 shows a comparison of total number of solutions for these three target symbolic functions out of 100 runs. The total number of evolved solutions for each problem is decreasing, indicating that *F1* to *F3* are increasingly difficult. The results show the same pattern as found in previous experiments with other problems, that is GP with loops achieved more successes than GP without.

Figure 3.37 shows the same data as figure 3.36, but showing successes with loops as a ratio of successes without loops. In the diagram, the success ratio for *F1* with loops is around 1.06, the success ratio for *F2* is increased to 1.82. *F3* has the highest success ratio of 3.5. This ratio is increasing with increasing power value of x and this indicates that, for this problem, GP with loops is more beneficial as the potential for loops increases.

3.10 Parameter Sensitivity Analysis

In this section, we analyse the effect of the settings of genetic parameters for the runs with and without loops in order to demonstrate that it is the explicit looping constructs that are the dominant factor in the evolution process, not the choice of the genetic parameters of crossover or mutation rates and population size.

The modified ant problem (see Section 3.5, page 78) and one of the symbolic regression problems *F2* (see Section 3.9, page 103) have been selected for analysis.

Two sets of experiments have been conducted in this sensitivity analysis. The first set varies the crossover and mutation rates. The elitism rate (2%) and the population size (100) are constant in these runs. The crossover rates are 10%, 20%, 55%, 70%, 95%. Because in our setting, the sum of the crossover, mutation and elitism rates is 100%, the mutation rates are 88%, 78%, 43%, 28%, 3% respectively. The second set of experiments varies the population size. The population size is set to 10, 50, 100, 200, 500 and the crossover, mutation and elitism rates are 70%/28%/2%.

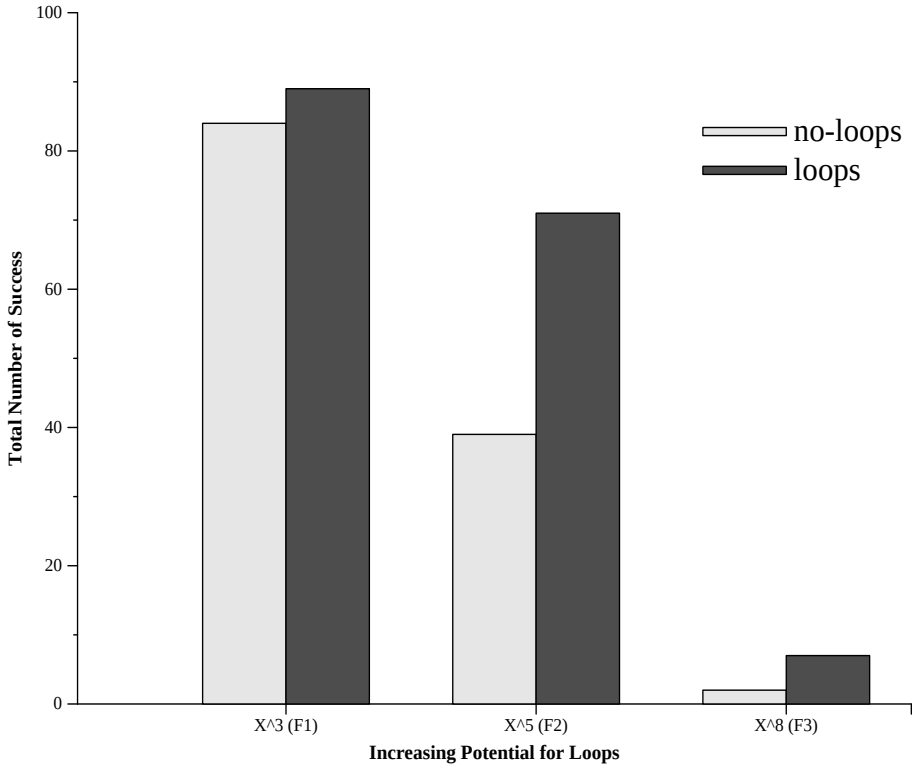


Figure 3.36: Total number of success for different target functions, 100 runs, symbolic regression

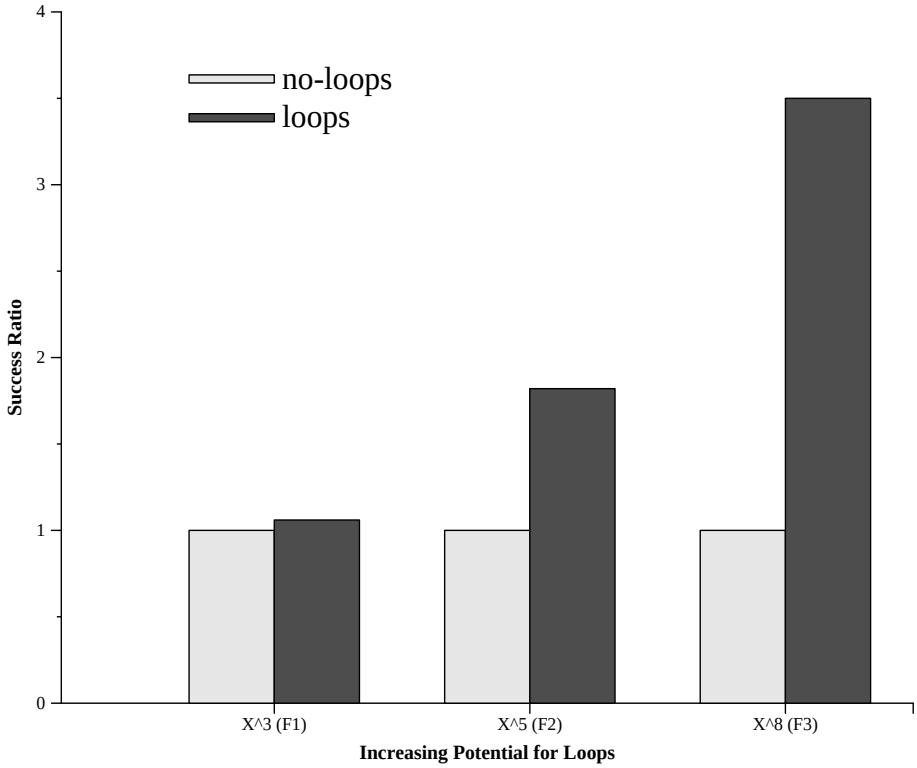


Figure 3.37: Success ratio for different target functions, 100 runs, symbolic regression

3.10.1 Experimental Results — The Parameter Sensitivity Analysis

One hundred runs with no loops and simple loops with the above parameter values have been carried out.

Figure 3.38 shows the mean best fitness for the experiments with loops and without at different crossover and mutation rates. The solid lines represent the runs without loops and the dotted lines represent the runs with simple loops. The thicker the line, the higher the crossover rate, thus the lower the mutation rate. The graph shows that all solid lines have worse fitness than the dotted lines and there is a distinct gap between them. The simple loops approach performed much better than no loops for all comparisons.

Figure 3.39 demonstrates the cumulative probability of success for the experiments on varying the crossover and mutation rates. All solid lines are on the x axis and the no loops approach does not get a single perfect solution in 500 runs.

Figure 3.40 shows the mean best fitness for the runs at different population sizes for the no loops and simple loops approaches. The solid lines represent the no loops runs and the dotted lines represent the simple loops runs. The thicker the line, the larger the population. Except for the *no-loops-pop500* method outperforming the *simple-loops-pop10* method, all other loop methods beat the no loops methods.

The cumulative probability of success shown in Figure 3.41 demonstrates the same pattern as in Figure 3.39. The no loops method does not get a single success while simple loops method has many successes. The graph also shows that there is an increasing number of successes.

The experiments conducted with increasing population size for the *F2* symbolic regression problem showed the similar patterns as in the modified ant problem (Figures not shown).

Overall, the sensitivity analysis experiments show that the use of looping constructs is the key factor in the improvement of the fitness and in getting more solutions. Although there were variations in the performance of the simple loops and the no loops methods with different settings, simple loops always outperformed no loops in cumulative probability of success.

3.11 Summary and Discussion

This chapter of the thesis describes and discusses two formats of explicit loops and gives empirical results for solving five artificial problems by GP with and without loops. By restricting the

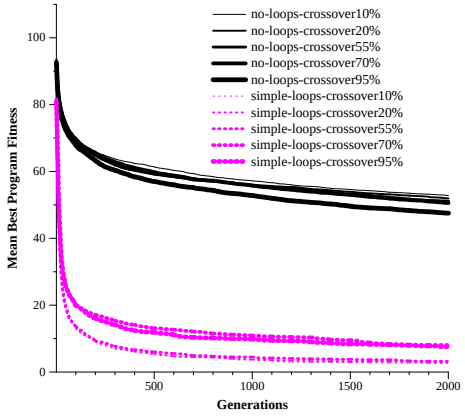


Figure 3.38: Mean best fitness at different crossover/mutation rates, population size = 100, average of 100 runs, the modified ant problem

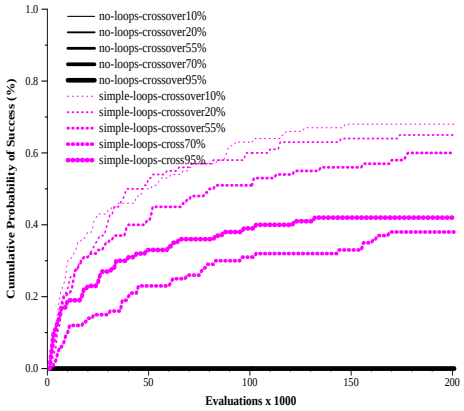


Figure 3.39: Cumulative probability of success at different crossover/mutation rates, population size = 100, average of 100 runs, (the no loops lines are on the x axis), the modified ant problem

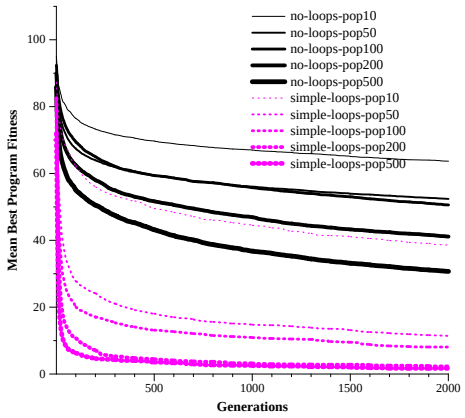


Figure 3.40: Mean best fitness for different population size, fixed crossover/mutation rates, average of 100 runs, the modified ant problem

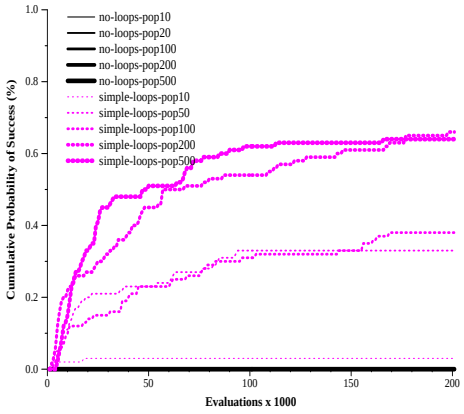


Figure 3.41: Cumulative probability of success for different population size, fixed crossover/mutation rates, average of 100 runs, (the no loops lines are on the x axis), the modified ant problem

semantic complexity of the for-loops, the experiments were successful in evolving small, efficient and reasonably understandable solutions to these problems. In relation to the research questions asked by this chapter the outcomes are as follows:

1. *How can we restrict the syntax and semantics of for-loops in a way that avoids problems of infinite loops and still provides useful benefits for genetic programming?*

   Explicit *for-loops* can be used in GP by introducing looping nodes and domain dependent limits on the maximum number of iterations. In our experiments, we have composed two formats of explicit *for-loops* to solve five artificial problems. The function node *for-loop* and the terminal nodes *number-of-iterations* or *start* and *end* need to be introduced. In addition, a maximum number of iterations needs to be specified or the value of the *start* and the *end* variables needs to be constrained by the domain information.

   Explicit *for-loops* provide useful benefits for genetic programming. The empirical results demonstrate that with loops, GP gets to a solution quicker in fewer generations thus fewer evaluations are needed. The major findings are:

   - Using a fitness function which favours programs with fewer loops was very beneficial. The programs evolved in this way were smaller and more understandable and generally fitter than programs evolved without this bias.
   - It is not clear whether simple loops with semantic restrictions are easier to evolve than more complex loops with less restrictions. Our initial experiments showed that programs tend to get better fitness when calculation in the branch of *num-of-iterations* is allowed. Further analysis found that this is because the value of *max-iterations* was too small. Set a big value for *max-iterations* improved evolution for the simple loops, but not for the unrestricted loops.
   - Loops are more helpful when there is more potential for loops. In the symbolic regression experiments, the loops approach performed better by getting a higher ratio of success than the no loops approach as the potential for loops increased.

2. *Can GP with for-loops solve some problems that cannot be solved or are very difficult to solve without explicit loops?*

Yes. GP with *for-loops* can solve some problems that cannot be solved or are very difficult to solve without explicit loops.

In the visit-every-square problem, we found that with the increasing grid size, the number of solutions for the no loops approach decreased dramatically, that is, the problem becomes more difficult. We did not get any solutions without loops for the 6×6 and 8×8 visit-every-square problem in 100 runs, while with simple loops, there were 20 to 40 solutions.

The modified ant problem is very difficult to solve without loops (in GP). The one solution from 300 runs contained over 5,000 nodes. In contrast, there were 10 to 20 solutions evolved by GP with loops and many of the solutions had fewer than 30 nodes.

The sensitivity analysis conducted at the end of these experiments shows that there are minor differences in performance for different genetic parameter settings. However, the looping constructs give major differences in performance and are the key factor for the good performance. In most runs, irrespective of the different parameter settings, the runs with loops performed much better than those without.

The results in this chapter suggest that looping constructs are worth considering when the problem domain has some repetitive characteristics. While evolution of generalised loops is currently not possible, looping constructs with carefully designed syntax and semantics can be used to great advantage.

Chapter 4

Solving A Binary Image Classification Problem

4.1 Introduction

In this chapter, the use of the *for-loops* with restricted semantics will be investigated for a problem in which there are natural repetitive elements, that of distinguishing two classes of images. The outcome of this experiment will answer the thesis's main research question 3, that is, whether restricted loops can be used in solving a difficult object classification problem and provide benefits as on other artificial problems (see Section 3, page 10). While we do not claim that this is a difficult computer vision problem, the problem presented in this chapter is more difficult than the previous artificial problems because the repetitive patterns are not obvious and capturing the repetitive patterns is not necessary for solving small instances of the problem. In this object classification problem, using our proposed formulations, classifiers with loops have been successfully evolved. They can capture the repetitions in pictures and perform much better than those without loops. The results suggest that loops with problem dependent formats can be successfully used in GP in the situations where domain knowledge is available to provide some restrictions on loop semantics.

The task of object classification has been described in Section 2.4. As stated, an image contains a large number of pixels and these pixels or features extracted from them need be input into the classifier for decision making. The evolution of a classifier takes time and the process of evolving a classifier may suffer from over-training (see Section 2.4.2, page 69).

This chapter presents three explicit *for-loop* formats. They are formulated through analysing the domain information of the problem and taking simplicity, generality and efficiency into consideration.

4.2 Chapter Goals

The aim of this chapter is to answer the main research question 3 in Chapter 1 (see Section 3, page 10), that is:
3. Can for-loops be used in a difficult object classification problem with similar performance gains to those achieved on relatively simple artificial problems?
This question explores a difficult image classification problem - an artificially constructed two class binary image classification problem in which the repetitive patterns are not obvious. The question has been divided into the following sub-questions:

1. How can *for-loops* be incorporated into evolved programs for image classification?
2. Does GP with *for-loops* perform better, that is, do classifiers with *for-loops* need fewer generations to evolve and are smaller, more accurate and more understandable than those without loops?
3. What variations of *for-loops* can be used?
4. What are the differences between decision strategies in the evolved loop and non-loop programs?

The expectations are that loops can be applied to the classification problem and classifiers with *for-loops* will be smaller in size and easier to analyse and thus more understandable.

4.3 The Binary Image Classification Problem

This image classification problem involves distinguishing two objects of interest, circles and squares. The objects are of similar sizes.

In the first classification task, the objects are centered in a 16×16 grid. The pictures were generated by firstly constructing full squares and circles and then manually removing groups of pixels or individual pixels to leave objects that a human would recognize as circles or squares.

This makes the classification task non-trivial. Examples of these images are shown in Figure 4.1 and Figure 4.2.

The second classification task involves shifted images. The centered objects have been randomly moved in the horizontal or vertical direction. This increases the difficulty of the task. Examples of the shifted images are shown in Figure 4.3 and Figure 4.4.

The task of the experiments is to let GP evolve a classifier by learning from training images and then use it on the test images to determine whether they are squares or circles. A successful classifier should correctly classify the training (see Figure 4.1 and Figure 4.3) and testing images (see Figure 4.2 and Figure 4.4). In our formulation, classifiers indicate a square when they return a value greater than or equal to 0; classifiers indicate a circle when they return a value less than 0. Small classifiers evolved with a small computation cost are desirable.

For simplicity, in the first set of experiments, the 16×16 grid is represented by a one-dimensional array of length 256. Pixel values are either 1 or 0. In the second set of experiments, a 16×16 array is used to represent an image in order to better use the spatial information in images. For each problem, we will evolve classifiers with and without loops and compare accuracy, size, computation cost and convergence behaviour.

4.4 Syntax and Semantics of the For-loops

Three loop formats have been composed for this problem. The logic behind these compositions is: For loop format 1, we wanted to solve the problem by utilising loop formats that we had successfully used earlier (see Chapter 3) and this requires the image to be represented as a one-dimensional array. Since a two-dimensional array is a more natural representation of an image, we then investigate loop formats for a two-dimensional representation.

4.4.1 Loop Format 1 — Traversing Lines in a One-Dimensional Representation

In loop format 1, the syntax is the same as *for-loop2* in Chapter 3 for solving the sorting problem (see Section 3.3.2, page 74), but the semantics are different. The syntax is:

(*for-loop-1d start end method*)

In this formulation, *start* or *end* are of type position. Terminals of this type are denoted by

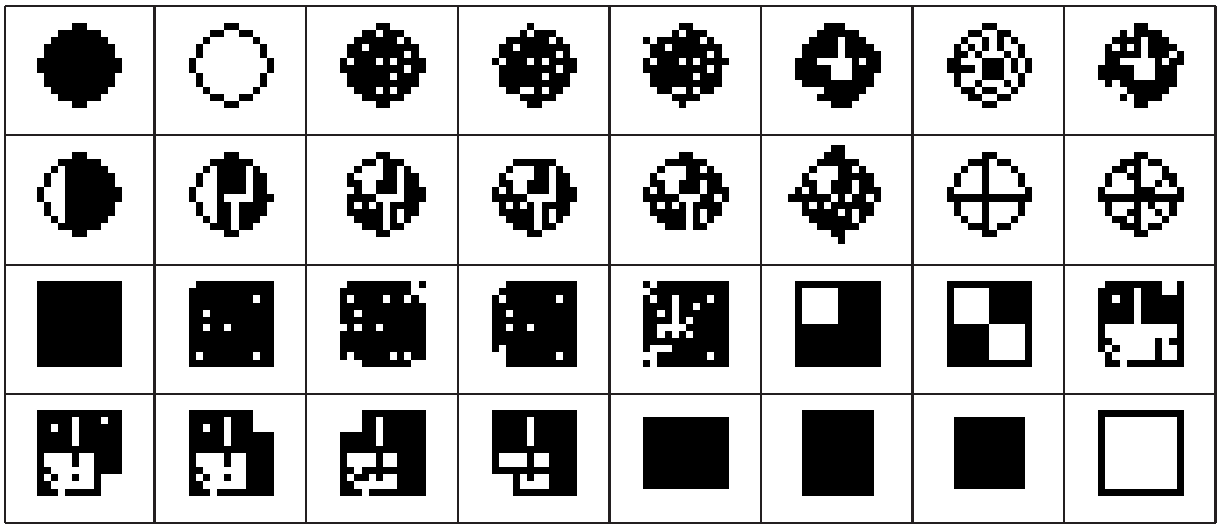


Figure 4.1: Centered binary images for training

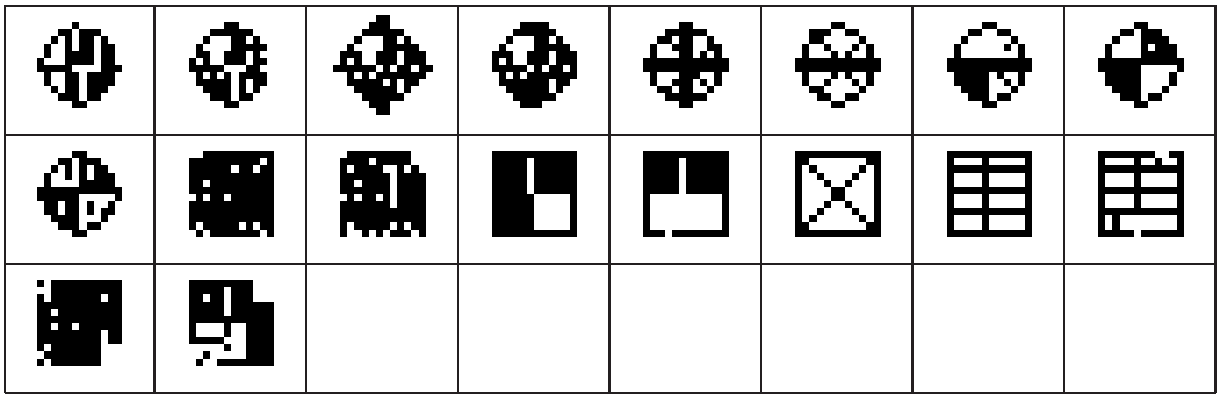


Figure 4.2: Centered binary images for testing

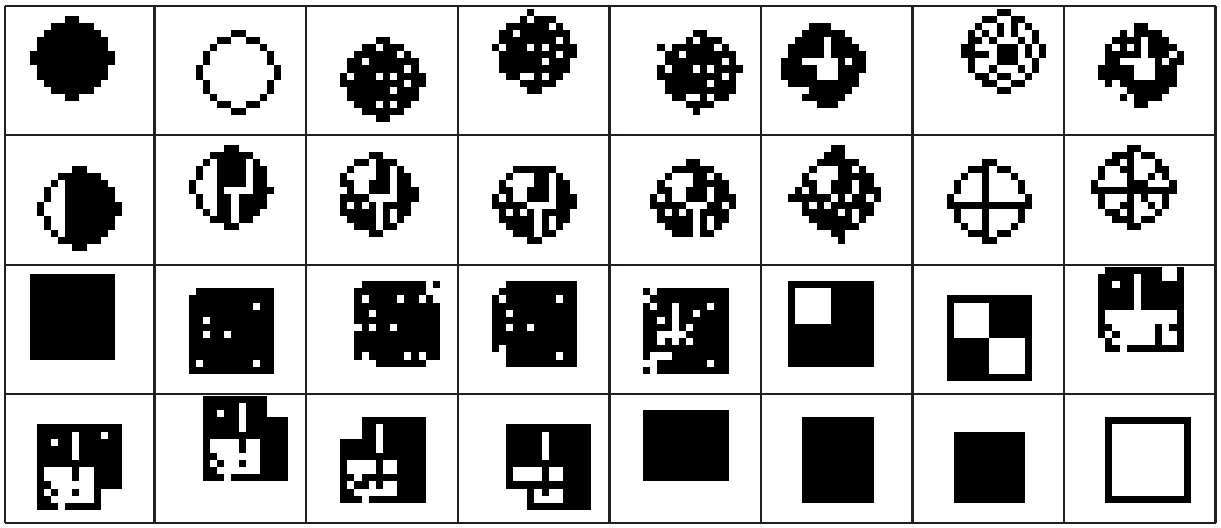


Figure 4.3: Shifted binary images for training

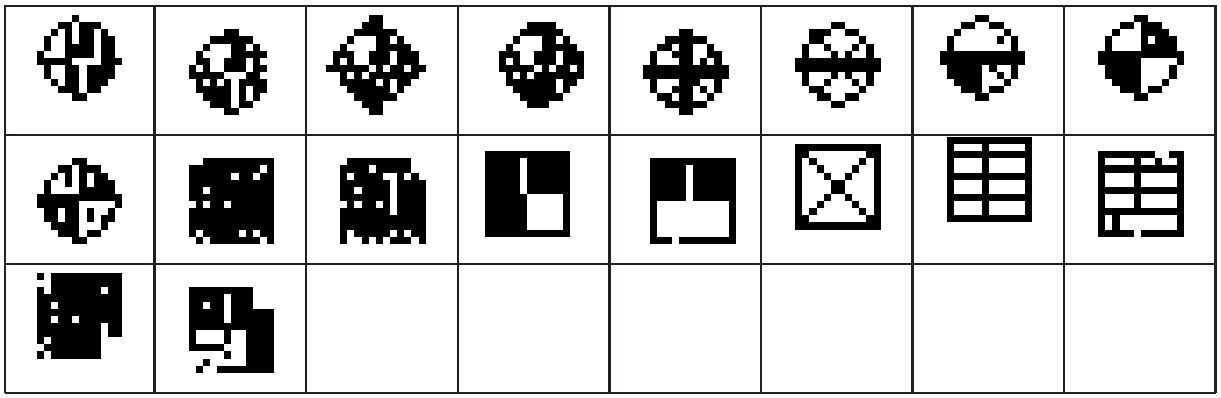


Figure 4.4: Shifted binary images for testing

pos-i and i is a value between 0 and 255. This allows loops to utilise positions to traverse the array and get pixel values in those positions. *Method* is a function selected from $\{plus, minus\}$. If *end* is greater than *start*, *for-loop-1d* will traverse the image from the *start* position to the *end* position, get the pixel values in these positions and perform the calculation indicated by *method*.

In evolved programs, terminals that return pixel values directly are also available. A terminal denoted by *pixel-value-i* returns the value of the pixel at position [i] and is of type double. We set different types for *pixel-value-i* and *pos-i*. This avoids potential crossover between these two terminals, so that *start* and *end* can only be valid positions and not pixel values. A summary of the terminals and functions used in this loop approach can be viewed in Tables 4.1. and 4.2.

An example of this loop format in an evolved program is:

(*for-loop-1d pos-3 pos-26 plus*) .

The program will add up pixel values, which are either 0 or 1, from array position 3 to 26 and return the sum.

If *start* is greater than *end*, *for-loop-1d* will calculate the result in the reverse order. The program *(for-loop-1d pos-26 pos-3 plus)* is equivalent to *(for-loop-1d pos-3 pos-26 plus)* while *(for-loop-1d pos-26 pos-3 minus)* may return a different value to *(for-loop-1d pos-3 pos-26 minus)* because of the order in computing.

In this implementation, infinite loops are not possible and no special actions are necessary in fitness calculation. In this chapter, methods using this loop format will be identified by *1d-?-loops* in graphs of results with *?* replaced by *centered* or *shifted* to indicate whether objects in the images are centered or shifted.

4.4.2 Loop Format 2 — Traversing Rectangles in a Two-Dimensional Representation

Loop format 2 has the same basic format as loop format 1 but been proposed for images in a two-dimensional representation. Instead of lines, loop format 2 traverses rectangles. The syntax is:

(*for-loop-2d-rect start end method*)

In this format, *start* and *end* are of type position. Terminals of this type for two-dimensional representations are denoted by *pos-i-j* where *i,j* are values between 0 and 15. These two positions

are regarded as the opposite corners of a rectangle. The program uses these two positions to traverse the rectangle. *Method* has the same meaning as loop format 1 and is a function indicator selected from $\{plus, minus\}$.

As before, terminals which return pixel values directly are also available. *Pixel-value-i-j* returns the value of the pixel at position [i,j]. As before, we set different types for *pixel-value-i-j* and *pos-i-j*. This avoids potential crossover between these two terminals, so that *start* and *end* can only be valid positions and not pixel values. A summary of the terminals and functions can be viewed in Table 4.3.

An example of this loop format in an evolved program is:

(*for-loop-2d-rect pos-3-0 pos-4-2 plus*) .

In this example, the *method* is *plus*. During evaluation, the loop will traverse a rectangle starting from point [3,0] with opposite corner at [4,2] and return the sum of pixel values in [3,0], [3,1], [3,2], [4,0], [4,1], [4,2].

Runs allowing this format of loops will be referenced by *2d-?-loops-rectangles* with *?* replaced by *centered* or *shifted*.

4.4.3 Loop Format 3 - Traversing Lines in a Two-Dimensional Representation

In loop format 3, the spatial information in images in a two-dimensional representation is utilised to allow loops to traverse straight lines at different angles. The proposed loop syntax is:

(*for-loop-2d-line start direction length method*) .

In this formulation, *start* is of type position and is denoted by *pos-i-j* as in loop format 2. *Pos-i-j* indicates the first pixel position [i,j] of the line. *Direction* represents the angle. There are 8 possible directions for a straight line. They are {*up, down, left, right, upLeft, upRight, downLeft, downRight*}. *Length* indicates the number of squares that a loop traverses. Depending on the location of the first pixel, the maximum length of a line is validated during the evolution, so that in any genetic operations, *length* will not be a value that makes a line across a border. *Method* is the same as in *for-loop-2d-rect* and is selected from {*plus*, *minus*}. A summary explanation of the terminals and functions can be viewed in Table 4.4 (page 122).

An example of this loop format in an evolved program is:

(*for-loop-2d-line pos-5-0 upRight length-3 plus*) .

In this example, the *method* is *plus*. During evaluation, the loop will traverse a line of length

4 (3 is the line length excluding the starting point), starting from position [5,0] in the *upRight* direction and return the sum of the pixel values in positions [5,0], [4,1], [3,2], [2,3]. The different types for position, pixel value and length ensure that only correct programs can be generated in initialization and only valid genetic operations can be performed during the evolution.

Runs with this format are referenced by *2d-?-loops-lines* with *?* replaced by *centered* or *shifted*.

4.5 Programs Without Loops

Runs without loops have been conducted for images in one and two dimensional representations and the results were compared with loop approaches.

4.5.1 No Loops in a One-Dimensional Representation

In evolved programs, a pixel is referenced by a terminal denoted by *pixel-value-i*. *Pixel-value-i* is of type double and *i* is a value between 0 and 255. *Pixel-value-i* returns the pixel value stored in the array position *i*. In addition, a random value terminal denoted by *drand-x* is used and generates random double values between 0.0 and 100.0. Plus and minus represented by {*d+*, *d-*} are the only functions used and both take two double value arguments.

A summary of terminals and functions can be viewed in Table 4.1.

4.5.2 No Loops in a Two-Dimensional Representation

In evolved programs, a pixel is referenced by *pixel-value-i-j*. *Pixel-value-i-j* is a terminal of type double and *i, j* are values between 0 and 15. *Pixel-value-x-y* returns the pixel value in the array position [i,j]. The other terminals and functions are the same as in the one dimensional representation.

4.6 Loops for Images in a One-Dimensional Representation

The images in a one-dimensional representation are examined first, because we want to see if we can be successful with the simpler situation first before going to more complex situations.

Table 4.1: Definition of terminals and functions, the no loop approach, images in a one-dimensional representation

Nodes	Type	Description
Drand::Terminal	double	Generates a double value between 0.0-100.0 denoted by '*drand-x*'.
Pixel-Value::Terminal	double	Generates a random position between 0-255, denoted by '*pixel-value-i*'. It returns the pixel value at position [i] and the pixel value is cast to a double.
d+::Function *(d+ double double)*	double	Takes two double values and returns the sum.
d-::Function *(d- double double)*	double	Takes two double values, subtracts the second from the first and returns the result.

Table 4.2: Definition of extra terminals and functions, the loop approach, images in a one-dimensional representation

Nodes	Type	Description
Pos::Terminal	position	Generates a random position between 0-255 denoted by '*pos-i*'.
Minus::Terminal	method	An indicator of the minus operation, denoted by '*minus*'.
Plus::Terminal	method	An indicator of the plus operation denoted by '*plus*'.
For-Loop-1d::Function *(for-loop-1d position position method)*	double	Takes 3 arguments. The first two are positions and the third argument is a method. The loop will traverse the array segment between the positions and perform the calculation indicated by the method.

Table 4.3: Definition of extra terminals and functions, 2d-loops-rectangles, images in a two-dimensional representation

Nodes	Type	Description
Pos::Terminal	position	Generates a random position in a two-dimensional array denoted by *pos-i-j*. The values of *i, j* are constrained between 0-15.
Minus::Terminal	method	An indicator of the minus operation and is denoted by '*minus*'.
Plus::Terminal	method	An indicator of the plus operation and is denoted by '*plus*'.
For-Loop-2d-Rect::Function *(for-loop-2d-rect position position method)*	double	Takes 3 arguments. The first two are of type position. They indicate two corner points of the rectangle. The third is of type method. The loop traverses the pixels in the rectangle and performs the calculation indicated by the third argument.

Table 4.4: Definition of extra terminals and functions, 2d-loops-lines, images in a two-dimensional representation

Nodes	Type	Description
Pos::Terminal	position	Generates a random position denoted by '*pos-i-j*'. The values of *i, j* are constrained between 0-15.
Direction::Terminal	direction	Generates a value selected from *up, down, left, right, upLeft, upRight, downRight, downLeft.*
Length::Terminal	length	Generates a value denoted by '*length-x*' to indicate the line length and the length will not exceed the border of the 16×16 grid
Minus::Terminal	method	An indicator of the minus operation and is denoted by '*minus*'.
Plus::Terminal	method	An indicator of the plus operation and is denoted by '*plus*'.
For-Loop-2d-Line::Function *(for-loop-2d-line position direction length method)*	double	Takes 4 arguments. The first three arguments indicate a straight line at a certain angle. The loop traverses the line and performs calculation indicated by the fourth argument.

Figure 4.5: Parameter settings

PARAMETER NAME	VALUES
Population Size	100
Generation Number	2000
Mutation/Crossover/Elitism Rate	28 % / 70 % / 2 %
Tree Depth	min : 1 max : 7
Initialisation Method	Ramped half-and-half, where grow and full methods each deliver half of the initial population
Selection Method	Proportional fitness
Termination Criteria	100 % accuracy on training set or 2000 generations reached
Number of Runs	100 runs each
The Training Set	32 pictures (16 squares/16 circles)
The Testing Set	18 pictures (9 squares/9 circles)

4.6.1 Genetic Environment Settings

Function Set and Terminal Set

In the *normal* (ie. no loops) approach, functions and terminals are similar to previous work in GP for object classification (see Section 2.4) and can be viewed in Table 4.1.

In the loops approach, GP will have all the functions and terminals in the *normal* approach and the extra terminals and functions for loops listed in Table 4.2 which have been explained in Section 4.4.1.

Fitness Function

In evaluation, each individual will return a value either less than, equal to or greater than 0. In our formulation, a value less than 0 indicates a circle, while a value equal to or greater than 0 indicates a square. The number of wrongly classified training images divided by the total number of training images (error rate) is used as the fitness, see Equation 4.1.

$$fitness = \frac{Number\ of\ Errors}{Total} \tag{4.1}$$

Other Genetic Variable Settings

The other genetic environment settings are illustrated in Table 4.5.

4.6.2 Experiments and Experimental Results

Experiments with the normal and loop methods have been run 100 times each.

Experimental Results

Figures 4.6 - 4.12 show data gathered during the experiments. In the figures, *centered-normal* indicates the experiments were done on centered images without loops. *Centered-loops* indicates the experiments used the extra loop functions and terminals on centered images (see Table 4.2). *Shifted* indicates the experiments were on shifted images.

Figures 4.6 and 4.7 show the overall convergence behaviour of the population. Figure 4.6 shows the cumulative probability of success for getting a perfect classifier. A perfect classifier means that the evolved program classifies all the training and test images correctly. If a classifier passes the training set, but fails to identify all the testing images, it is considered a failure. If a classifier correctly classifies only a portion of the training and/or test images, it is also considered a failure. The graph shows that for the centered images the loop method is much more likely to generate a successful classifier. At 600 generations (60×1000 evaluations) 82 of the 100 loop runs had succeeded while only 52 of the normal runs without loops were successful. The difference is even more pronounced on the more difficult shifted image problem. After 2,000 generations, 36 of the 100 loop runs had succeeded while only 2 of runs without loops were successful. Figure 4.7 shows the mean average training fitness. These curves are consistent with the success rates shown in Figure 4.6.

Figures 4.8 and 4.9 show the mean average fitness with one standard deviation on centered images for the no loops and loops approaches. The reason for showing these two graphs is that during the experiments a large variability was found in the loops approach. This was not so obvious in the previous experiments (see Chapter 3).

Figure 4.10 shows the mean best program fitness. The best program refers to the best evolved classifier in the training process. This may not be a successful classifier. There is not much difference in the mean best program fitness for the centered images between both approaches, even though Figure 4.6 shows that there are more successes by the loop method. This is because classifiers using loops have a larger variation in fitness (see Figures 4.8 and 4.9). The fitness in bad runs offsets the fitness in the good runs. For shifted images, there is a significant difference. Classifiers evolved without loops do not perform well. This trend is

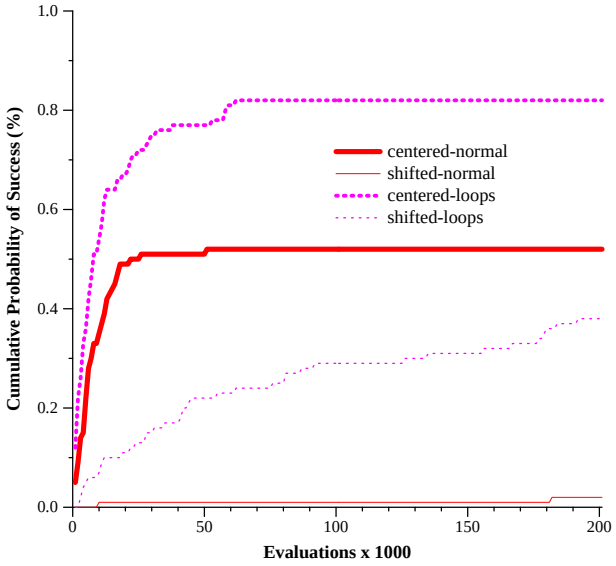


Figure 4.6: Cumulative probability of success, average of 100 runs, images in a one-dimensional representation

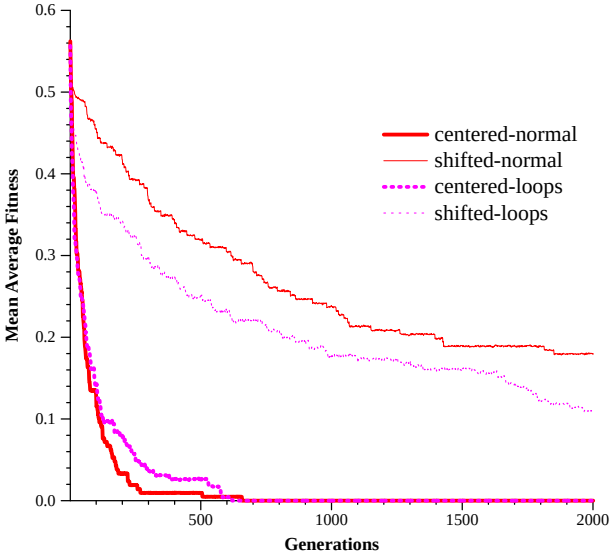


Figure 4.7: Mean average training program fitness, average of 100 runs, images in a one-dimensional representation

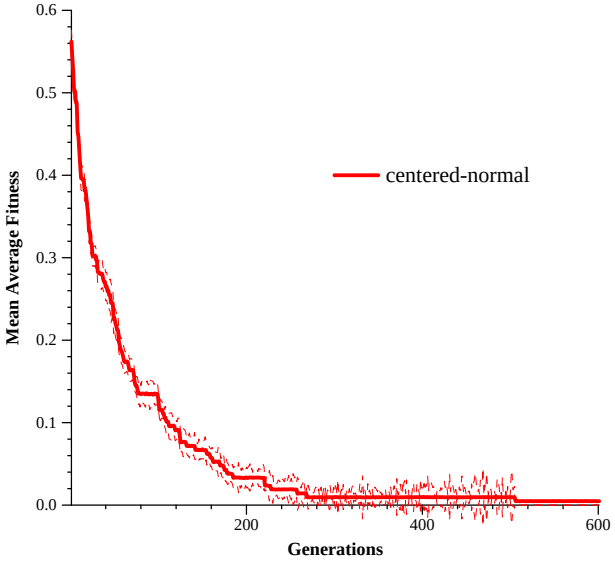


Figure 4.8: Mean average fitness with one standard deviation, centered objects, no loops approach, images in a one-dimensional representation

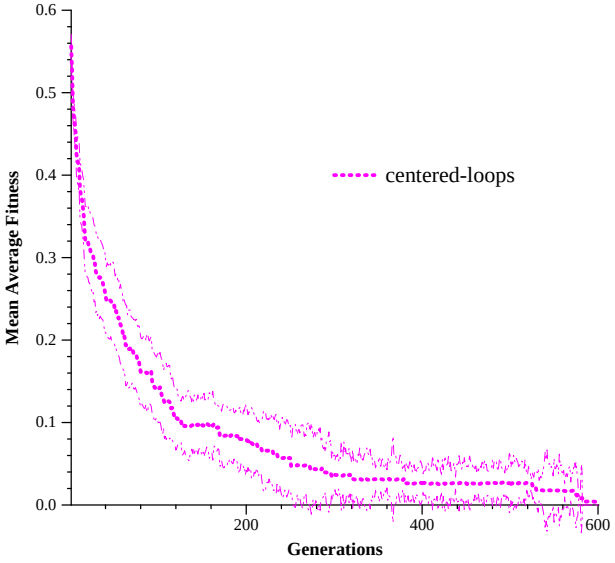


Figure 4.9: Mean average fitness with one standard deviation, centered objects, loops approach, images in a one-dimensional representation

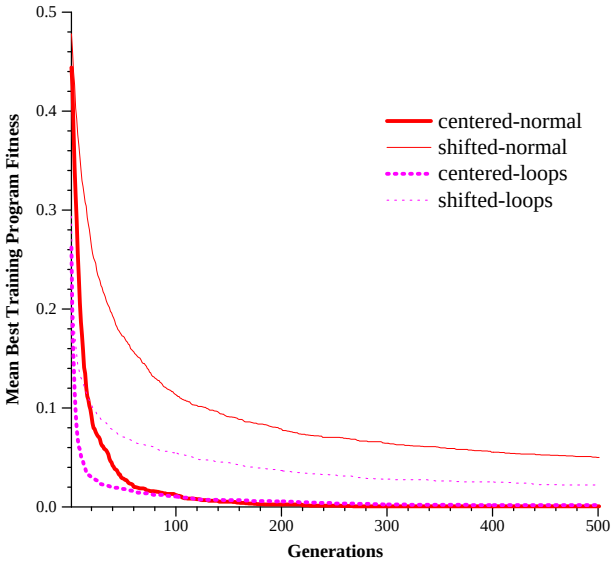


Figure 4.10: Mean best training program fitness, average of 100 runs, images in a one-dimensional representation

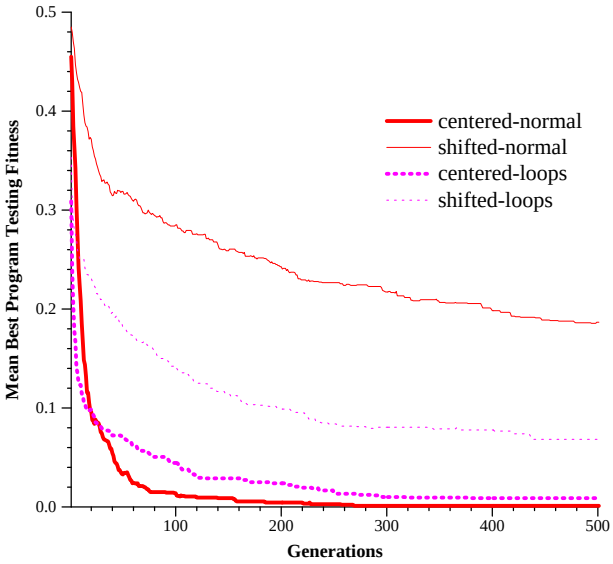


Figure 4.11: Mean best program testing fitness, average of 100 runs, images in a one-dimensional representation

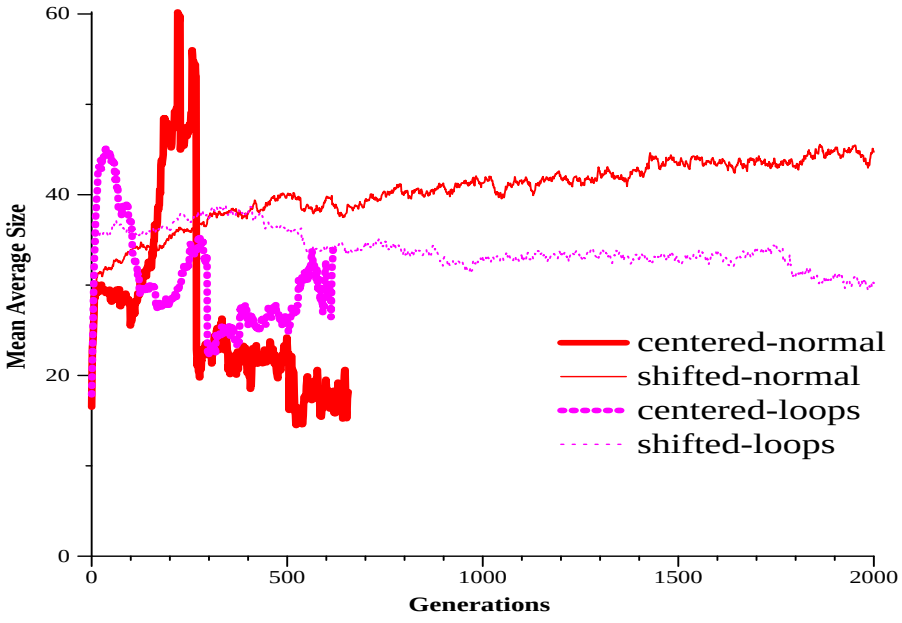


Figure 4.12: Mean average program size, average of 100 runs, images in a one-dimensional representation

further shown by the fitness of the best runs on the testing set as shown in Figure 4.11.

Figure 4.11 follows the same pattern as Figure 4.10. The loop method performs much better for shifted images and programs with loops have a wider variation in fitness. For the centered problem in Figure 4.11, none of the approaches actually get perfect solutions in all runs, but, because of the scale of the Y axis, it appears that zero fitness is reached.

Figure 4.12 shows the average size of the programs. Initially, we expected that programs with loops would be much smaller in size, but the results revealed that this was not the case. There are no wide differences for classifiers on the centered images or on the shifted images. The reason for this is that the training data is not hard enough and GP quickly found smaller sized solutions in both approaches and evolution stopped. However, many successful training classifiers evolved by the no loops method do not perform well on the test set. This suggests that the training set may be too small, but the classifiers with loops are more robust, that is they generalize better to unseen images. Figure 4.12 also shows that, for the centered images, both approaches resulted in perfect classification of the training data after about 800 generations and training stopped. Shifted image classification is a harder problem and the programs took longer to evolve. We observed that as fitness improved (see Figure 4.10), there was a decrease in size for the loop method and a slight bloating (see Section 2.2.6) in the *normal* method.

4.6.3 Analysis of Solutions

In this section, the solutions found by both methods are analysed and the decision strategies are compared.

Figure 4.13 shows one of the smallest classifiers evolved by the *normal* method and Figure 4.14 shows the points examined to distinguish the objects. The solution is small and elegant. It uses only two positions and took 4,797 evaluations to find. However, this solution has found an idiosyncrasy in the data and is clearly not general.

Typical solutions evolved by the *normal* approach are not so neat. Figure 4.15 lists a typical program evolved by the no loops approach and Figure 4.16 shows the points examined by the classifier. The program is large and the points examined are scattered all over the image. It took 13,030 evaluations to find this solution. This is much higher than the average number of evaluations (approx. 6,000) for finding a solution with loops.

Figure 4.17 shows one of the smallest classifiers evolved by the loop method and Figure 4.18

```
(d+ (d- pixel-value-37 drand-
0.441534) pixel-value-203)
```

Figure 4.13: One of the smallest classifiers evolved by the normal method, centered objects, images in a one-dimensional representation

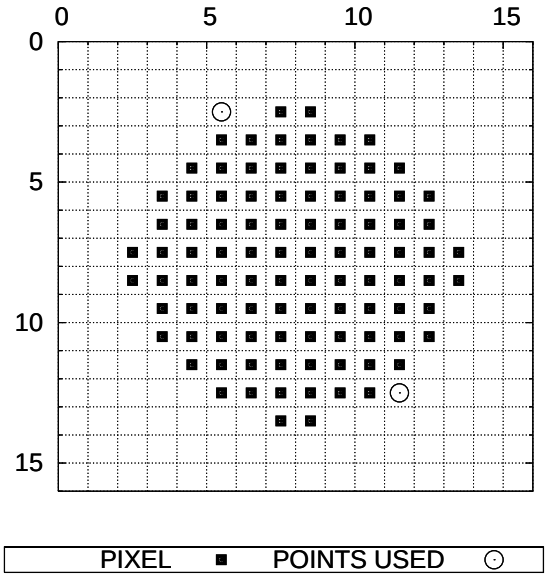


Figure 4.14: Points examined for the program shown in Figure 4.13, centered objects, images in a one dimensional representation

shows the points examined. The line goes from position 188 to position 288 and the program adds up all of the pixel values. By traversing this line, the program obtains enough information to distinguish the objects. This is in contrast to the random positions used by the no loop approach.

Figure 4.19 shows a typical solution evolved by the loop approach and Figure 4.20 shows the points examined. One of the main differences between the solutions with loops and those without is that a run using *1d-loops* examines more pixels in a linear manner, therefore, covers more areas of the image than really necessary for correct decision making. Apart from experimenting with more complicated formats of loops, this is another reason for composing the 2nd and 3rd loop formats of which are trying to minimise this effect.

Figure 4.21 displays one of the two solutions evolved by the normal (ie. no loop) method for shifted images and Figure 4.22 shows the points examined. They are scattered at the top and bottom to catch the information from the shifted objects. In contrast, the loop method (Figure 4.24) uses two lines to distinguish all of the shifted images.

In summary, the classifiers using loops examine a sequence of points to distinguish the objects. The no loop classifiers examine a seemingly random set of points in the image.

(d+ (d- (d+ (d+ (d+ (d+ drand-70.929252 pixel-value-188) drand-70.929252) (d- (d- drand-22.060454 drand-70.917456) (d+ drand-29.415353 drand-89.236116))) (d+(d- (d- pixel-value-2 pixel-value-155) (d- pixel-value-11 pixel-value-26)) (d+ (d+ pixel-value-150 pixel-value-37) (d- drand-52.450194 drand-38.299516)))) (d+ (d- (d+ (d+ pixel-value-133 drand-72.779942) (d+ pixel-value-139 pixel-value-130)) (d- (d+ drand-72.943129 drand-86.640064) pixel-value-114)) (d+ (d- (d- pixel-value-170 pixel-value-83) (d- pixel-value-194 pixel-value-133)) (d+ (d- pixel-value-225 pixel-value-172) (d- drand-29.415353 pixel-value-205))))) (d+ (d+ (d+ (d- (d- pixel-value-18 pixel-value-194) (d- drand-85.580583 pixel-value-209)) (d+ (d+ drand-61.098601 pixel-value-60) (d+ pixel-value-93 drand-59.032376))) (d+ (d+ (d+ pixel-value-224 drand-2.089882) (d- pixel-value-229 drand-82.981664)) (d- (d- pixel-value-135 pixel-value-209) (d- pixel-value-187 pixel-value-14)))) pixel-value-56))

Figure 4.15: A typical classifier evolved by the normal method, centered objects, images in a one-dimensional representation

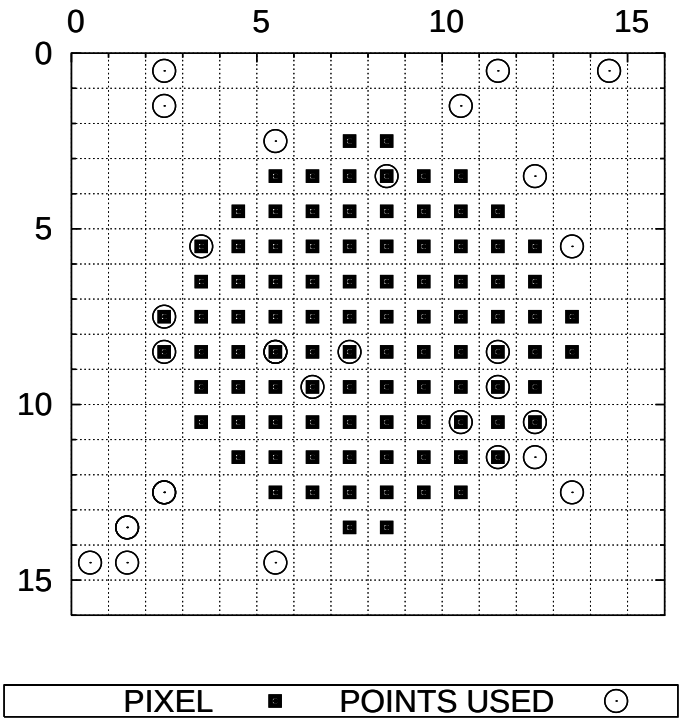


Figure 4.16: Points examined for the program shown in Figure 4.15, centered objects, images in a one-dimensional representation

```
(d- (for-loop-1d pos-228 pos-188
plus) drand-9.260122)
```

Figure 4.17: One of the smallest classifiers evolved by the loop method, centered objects, images in a one-dimensional representation

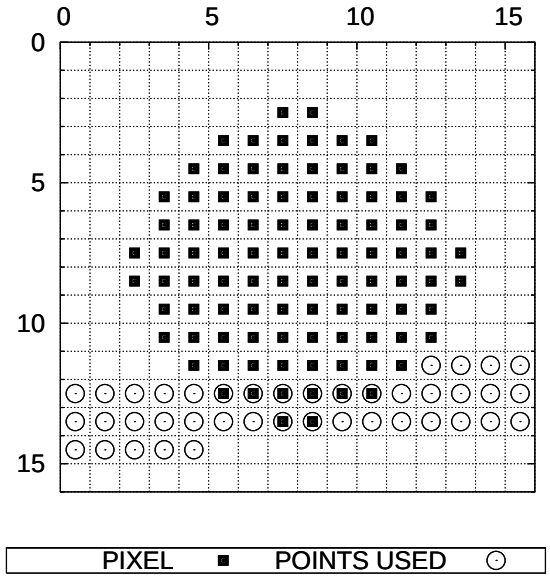


Figure 4.18: Points examined for the program shown in Figure 4.17, centered objects, images in a one-dimensional representation

```
(d- (for-loop-1d pos-161 pos-
228 plus) (d- (d+ (d+ drand-
76.701336 (for-loop-1d pos-144 pos-172
plus)) (d- (d- pixel-value-152 drand-
54.382222) pixel-value-157))
drand-14.021874))
```

Figure 4.19: A typical classifier evolved by the loop method, centered objects, images in a one-dimensional representation

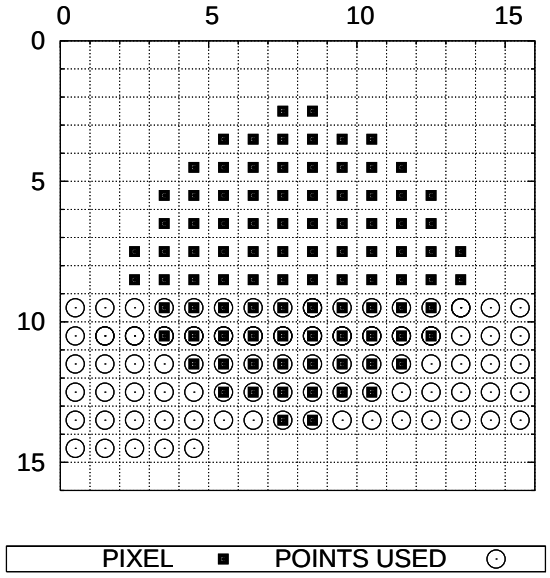


Figure 4.20: Points examined for the program shown in Figure 4.19, centered objects, images in a one-dimensional representation

```
(d+ (d+ (d- (d- (d+ drand-
34.087990 pixel-value-236) (d- pixel-
value-225 pixel-value-221)) (d- drand-
34.087990 (d- drand-96.220403 (d-
drand-72.832995 pixel-value-34)))) (d-
(d- (d+ (d+ drand-38.457827 drand-
2.639772)
drand-2.639772) (d- drand-96.220403
(d+ pixel-value-5 pixel-value-210))) (d-
drand-72.832995 drand-93.951264)))
drand-7.458120)
```

Figure 4.21: One of the two successful classifiers evolved by the normal method, shifted objects, images in a one-dimensional representation

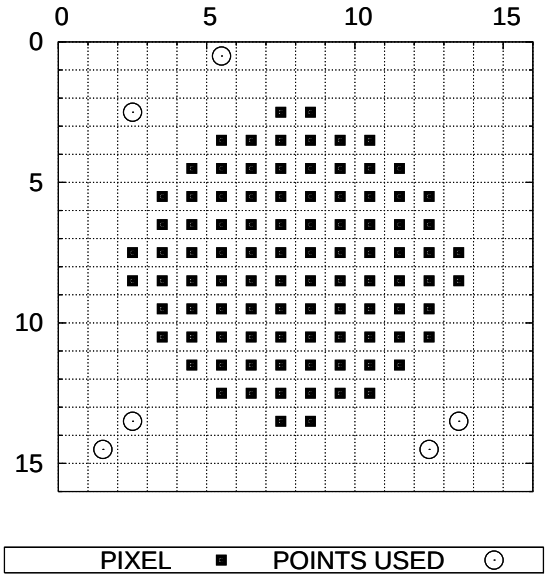


Figure 4.22: Points examined for the program shown in Figure 4.21, shifted objects, images in a one-dimensional representation

```
(d- (d- drand-87.318493 (d- (d+ (d-
(for-loop-1d pos-87 pos-45 minus) pixel-
value-165) drand-40.885102) (d- (for-
loop-1d pos-247 pos-199 plus) drand-
87.318493))) (d- pixel-value-198 drand-
17.579794))
```

Figure 4.23: One of the smallest classifiers evolved by the loops method, shifted objects, images in a one-dimensional representation

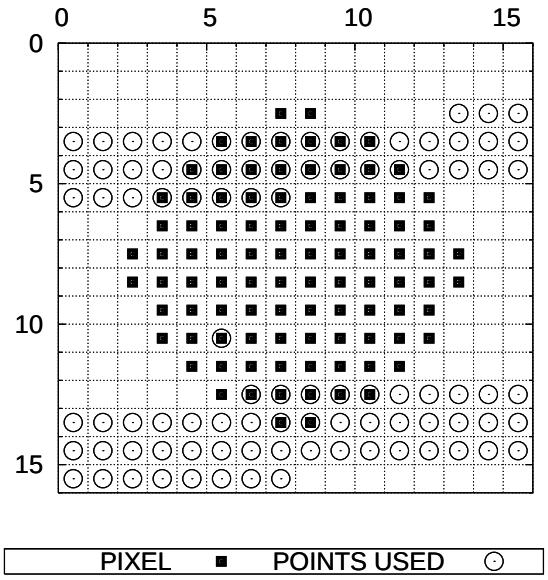


Figure 4.24: Points examined for the program shown in Figure 4.23, shifted objects, images in a one-dimensional representation

4.7 Loops for Images in a Two-Dimensional Representation

It is simpler in implementation to represent the images in one-dimensional arrays and use loops to evolve classifiers. However, it is more general and understandable to represent images in two dimensional arrays. In addition, runs with loops in the one-dimensional representation gave many solutions in previous experiments, but they used more pixels than the no loops approach. In some applications the number of pixels used can be an issue. If an image is small, using many pixels is not a problem. If an image contains millions of pixels, using too many pixels for classification costs a lot of computation power in evaluating each solution. A solution using a few pixels as in the no loop approach may just examine some ideal points in the training images and is not likely to generalize well to new images. We expect that the spatial information in images can be better utilised in the two dimensional representation to find a balance between the number of pixels used and the generality of the solution.

The experiments in this section utilise loop formats *for-loop-2d-rect* and *for-loop-2d-line* described in Section 4.4.3 for images in a two-dimensional representation. In these experiments, the same images are represented by two-dimensional arrays (16×16). The task is the same and is to distinguish centered or shifted squares and circles in the pictures (see Section 4.3).

4.7.1 Genetic Environment Settings

Function Set and Terminal Set

The normal approach uses the functions and terminals listed in Table 4.1 except that a pixel is referenced by *pixel-value-i-j*. The functions and terminals have been described in Section 4.5.

The *2d-loops-rectangles* approach utilises loop format *for-loop-2d-rect* which is described in detail in Section 4.4.2. The extra functions and terminals are listed in Table 4.3.

The *2d-loops-lines* approach utilises loop format *for-loop-2d-line* which is described in detail in Section 4.4.3. The extra functions and terminals are listed in Table 4.4.

Other Genetic Variable Settings and The Fitness Function

Other genetic variable settings and the fitness function are the same as those in the one-dimensional images. They can be viewed in Table 4.5. The fitness function is the same as before (see Section 4.6.1).

4.7.2 Experiments

Experiments with the *normal* approach were conducted again, but on this two-dimensional representation. One hundred runs each of the *2d-centered-loops-lines*, *2d-shifted-loops-lines*, *2d-centered-loops-rectangles*, *2d-shifted-loops-rectangles* methods were performed.

4.7.3 Experimental Results

Figure 4.25 shows the cumulative probability of success for the new runs for centered objects in a two-dimensional representation. Runs using loops (*1d-centered-loops*) and without (*1d-centered-normal*) in a one-dimensional representation were kept in the graph as the benchmark. The graph shows that runs without loops for centered objects represented either by a one-dimensional representation (*1d-centered-normal*) or by a two-dimensional representation (*2d-centered-normal*) are similar in the cumulative probability of success. The changes in representation do not affect the performance for the runs without loops. Runs with loops on a two-dimensional representation, *2d-centered-loops-lines* and *2d-centered-loops-rectangles*, have similar rates for cumulative probability of success. Their performance in the cumulative success is not as good as the runs with loops on a one-dimensional representation (*1d-centered-loops*). However, they still demonstrate an obvious superior cumulative success rate after 10,000 evaluations than those without *loops*.

Figure 4.26 compares the average number of the pixels used for centered objects on a one-dimensional and a two-dimensional representation using the different methods. In solutions with loops in a one-dimensional representation, a large number of pixels are used for decision making (see Figures 4.18, Figure 4.20). Figure 4.26 shows that the *1d-centered-loops* method consumed the highest number of pixels for decision making. The *2d-centered-loops-rectangles* method is the second highest, but is significantly less than the *1d-centered-loops* method. The *2d-centered-normal* method used smallest number of pixels among these five methods. Apart from *1d-centered-loops*, *2d-centered-loops-rectangles*, the other methods, *2d-centered-loops-lines, 2d-centered-normal, 1d-centered-normal* use approximately the same number of pixels for decision making. The plot-lines for *1d-centered-normal*, *1d-centered-loops* and *2d-centered-loops-rectangles* terminate before the end because all runs for these configurations had terminated successfully by then.

Figure 4.27 shows the cumulative probability of success for shifted objects. *1d-shifted-normal*

and *1d-shifted-loops* are kept in the graph as benchmarks. The graph shows the same trend as the runs on centered images (see Figure 4.25), that is, the loop approaches perform better than the *normal* ones in either one-dimensional or two-dimensional representations. Neither of the runs with loops on two-dimensional images (*2d-shifted-loops-lines, 2d-shifted-loops-rectangles*) performed as well as the loop approach on one-dimensional representations (*1d-shifted-loops*). This is because they used fewer pixels than the loop approach in a one-dimensional representation and solutions became less general and failed in the testing.

Figure 4.28 shows the average number of pixels used for decision making for shifted objects. A clear boundary of number of pixels used can be seen in this figure. The *1d-shifted-loops* method used the highest number of pixels, the same as the trend for loops for centered objects (see Figure 4.26). The *1d-shifted-normal* and *2d-shifted-normal* methods used the least number of pixels.

4.7.4 Analysis of Solutions

Analysis of solutions helps to understand how these classifiers with loops make decisions in two-dimensional representations.

Figure 4.29 shows the smallest classifier which uses loops to traverse rectangles in a two-dimensional representation for centered objects (*2d-centered-loops-rectangles*) and Figure 4.30 shows its traversal pattern. The program with loops uses 20 pixels to traverse a 4×5 square in the bottom right corner to decide whether it is a circle or a square. This classifier is similar to the 2-pixel classifier evolved in the no loop approach (see Figure 4.13) and finds an ideal rectangle in a location where it can just differentiate circles and squares for the set of images. Thus, it is not general since it only focuses on an narrow area in the grid.

Figure 4.31 shows the smallest classifier for shifted object classification by using loops to draw rectangles and Figure 4.32 shows the traversal pattern. Two rectangles are positioned in the upper and lower halves of the image to sense the shifted objects. This classifier is more robust compared to the previous smallest classifier. More pixels are used and different locations of the rectangles help to detect the shifted objects.

Figure 4.33 shows one of the smallest successful classifiers which uses loops to traverse lines in a two-dimensional representation for centered object detection (*2d-centered-loops-lines*) and Figure 4.34 shows its traversal pattern. It uses 6 pixels along a $downLeft$ direction line to make

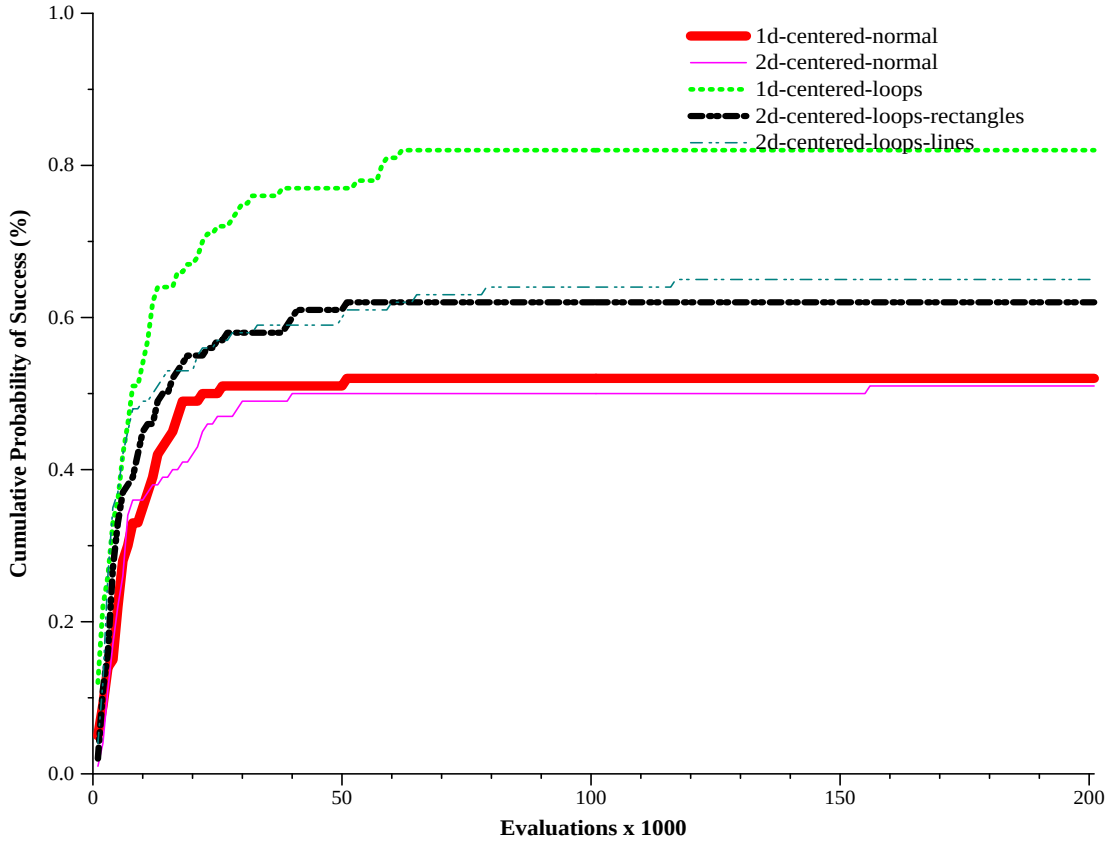


Figure 4.25: Cumulative probability of success, centered objects, average of 100 runs

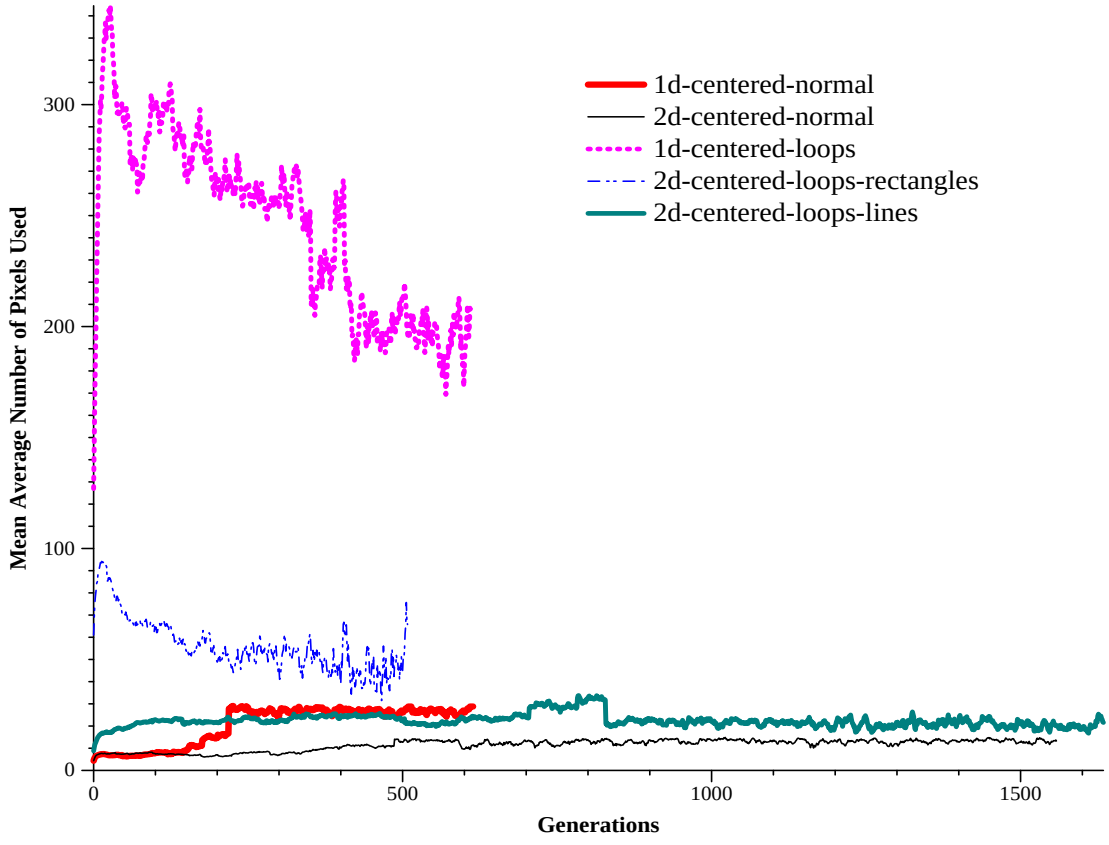


Figure 4.26: Mean average number of pixels used, centered objects, average of 100 runs

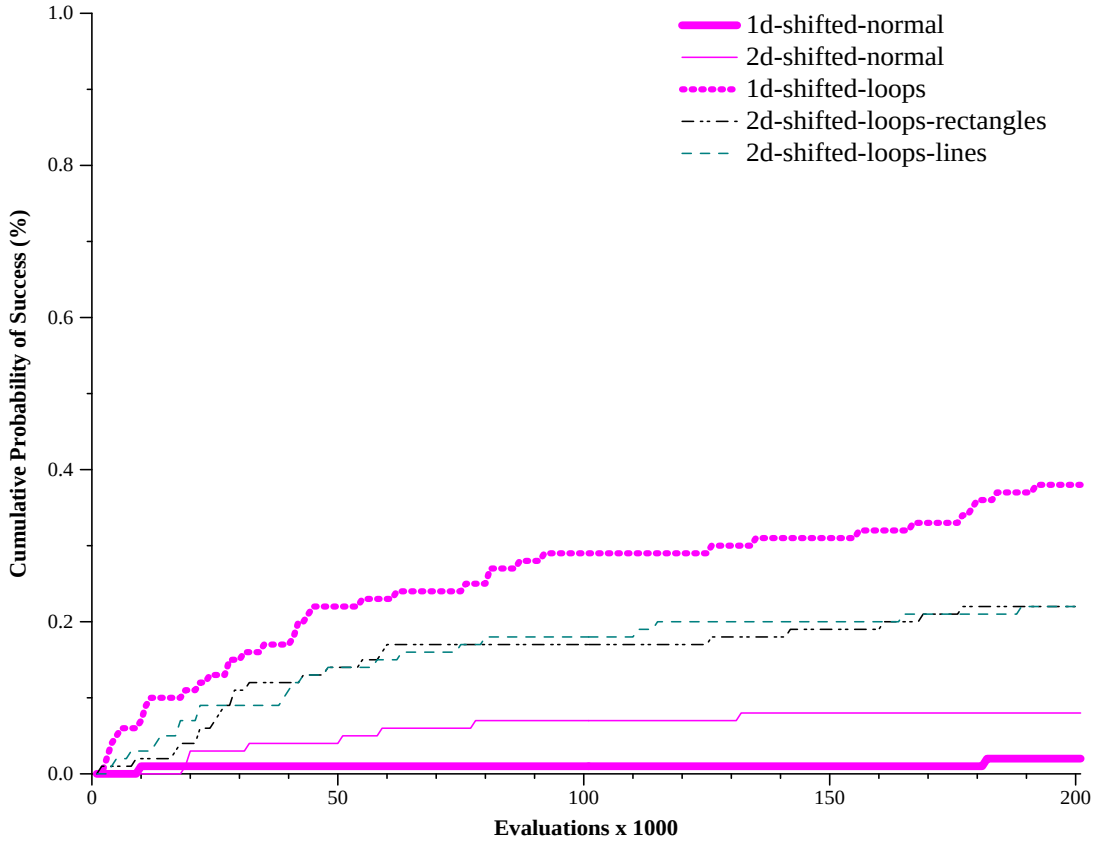


Figure 4.27: Cumulative probability of success, shifted objects, average of 100 runs

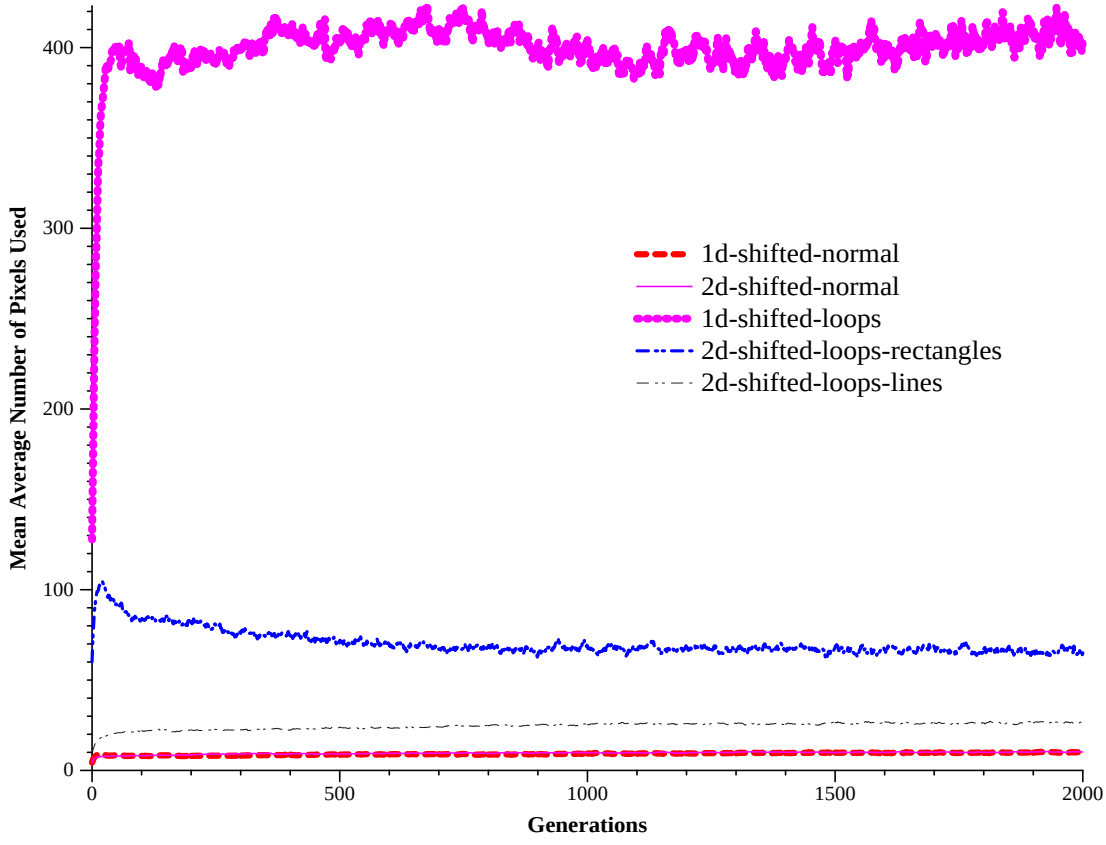


Figure 4.28: Mean average number of pixels used, shifted objects, average of 100 runs

```
(d- (d- (d- drand-75.060721 drand-
76.515305) (for-loop-2d-rect pos-14-15
pos-10-12 minus)) pixel-value-12-13)
```

Figure 4.29: One of the smallest classifiers evolved by 2d-loops-rectangles method, centered objects, images in a two-dimensional representation

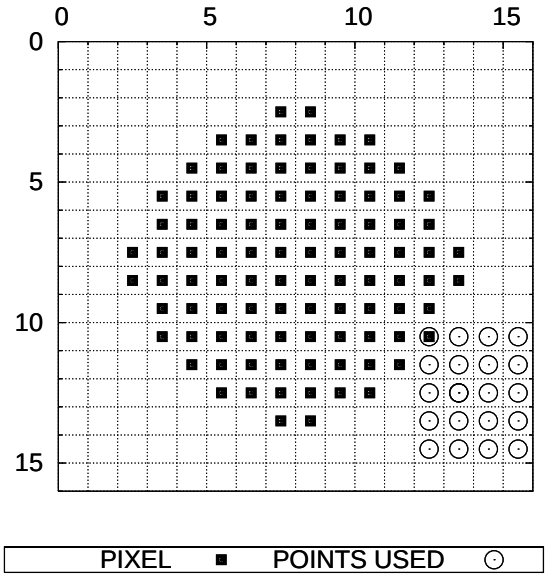


Figure 4.30: Points examined for the program shown in Figure 4.29, 2d-loops-rectangles method centered objects, images in a two-dimensional representation

the correct decision. By checking other solutions, we found most solutions had the same flavour and used pixel values along one line to make the decision.

Figure 4.35 shows the smallest classifier for shifted object detection using loops to traverse lines in two-dimensional representations and Figure 4.36 shows the traversal pattern. Four lines in four corner positions are used to detect the shifted objects.

The experiments showed that there is a trade-off in using a small number of pixels and using a large number of pixels. In our experiments, runs without loops gave few successes but the solutions generally used a small number of pixels. Runs with loops in one-dimensional representations gave the highest number of successes but the solutions generally used a large number of pixels. Runs with loops in two-dimensional representations were in the middle in terms of number of successes and number of pixels used.

While the last two image representations were two-dimensional, the loops are essentially '*one-dimensional*' in that there is one index per loop. The kinds of nested loops that a programmer would write in C or java for this problem would use two indices. Future research in this problem will focus on allowing two indices to freely participate in the evolution to form '*two-dimensional*' loops.

```
(d- drand-61.523841 (d- (d- (d- (d+
drand-85.844287 drand-60.960710) (for-
loop-2d-rect pos-6-15 pos-2-8 plus)) drand-
61.523841) (for-loop-2d-rect pos-13-14 pos-
14-2 plus)))
```

Figure 4.31: One of the smallest classifiers evolved by 2d-loops-rectangles method, shifted objects, images in a two-dimensional representation

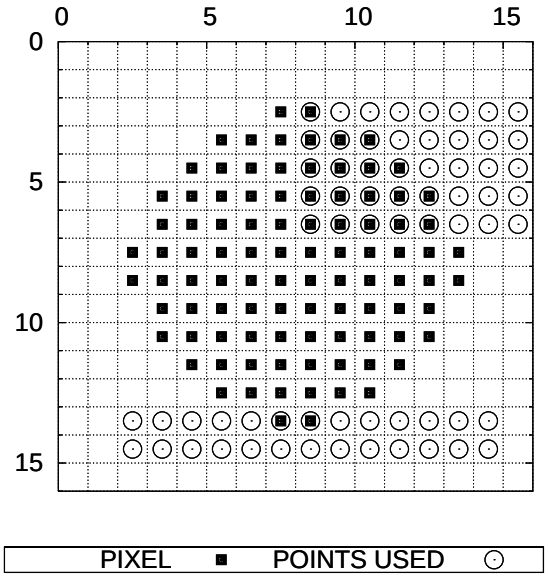


Figure 4.32: Points examined for the program shown in Figure 4.31, 2d-loops-line method, shifted objects, images in a two-dimensional representation

4.7.5 Computation Time

We have conducted an investigation on CPU times for experiments with and without loops, because we expect that programs with the loop functions may significantly consume more computation in evaluation.

However, there are issues in comparing algorithms based on CPU time. Computers do not simply execute one program at a time and they use priority-based scheduling schemes to continually switch from one process to another [31, p631]. These scheduling schemes are influenced by the network traffic and the timing of disk operations. The access patterns to the caches also depend on those concurrent processes. In addition, programming skills, different compilers and compiler settings affect execution times. The same algorithm written by different programmers with different levels of skill may be different in CPU time. Compilers with different optimisation settings may affect performance of the resulting binary code. Evan identical programs executing on the same computer may result in different execution times [31, p631].

Because we want to have more precise information on execution times for programs with and without loops, we have taken the following steps to minimise the problems in CPU time comparision. Runs have been conducted sequentially on a stand-alone linux computer. During

(d- (for-loop-2d-line pos-10-13 down-Left length-5 plus) drand-0.176959)

Figure 4.33: The smallest classifiers evolved by 2d-loops-lines method, centered objects, images in a two-dimensional representation

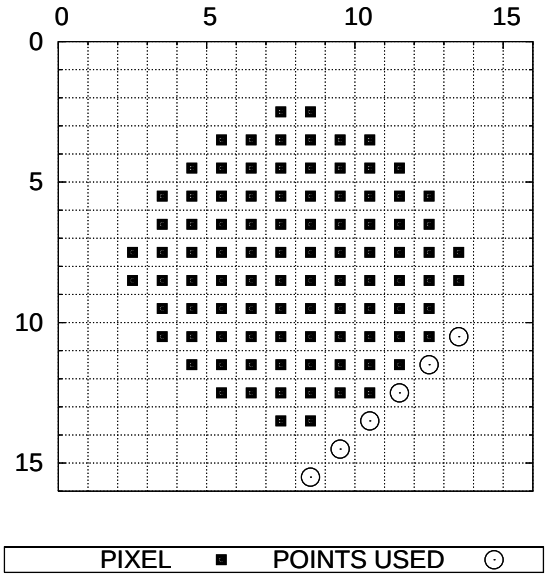


Figure 4.34: Points examined for the program shown in Figure 4.33, 2d-loops-line method, centered objects, images in a two-dimensional representation

a run, except for the operating system, there is no other user processes. The only difference between these runs is that some runs include the *for-loop* functions and others do not.

We have used three measurements. One is the average time for evaluating an individual, one is the average time for getting a successful classifier and the last is the ratio of average times for getting a successful classifier for programs with and without loops.

Table 4.5 shows a comparison of CPU times for the different methods. For easy visualization, problems are grouped by centered and shifted problems and comparisons are conducted for programs with and without loops. The last column shows the average time ratio. A value of 1.00 in ratio means there are no CPU time differences in getting a successful solution between the runs with loops and without loops; a value less than 1.00 means the no loops method is better; a value higher than 1.00 means the loops method is better. For centered image classification tasks, there are small differences in the average evaluation time for an individual with and without loops and differences in average CPU time in getting a successful solution. This is reflected by ratio values of 0.79, 2.74 and 1.16 separately. There are distinct CPU time differences in shifted image classification. The ratios are 16.44, 3.37 and 2.51 and they are much higher than 1 which means the loop methods are much better. Overall, the trend in this experiment is that

(d- (d+ (d- (d+ (d- pixel-value-3-8 (for-loop-2d-line pos-14-15 left length-3 minus)) (for-loop-2d-line pos-5-0 upRight length-5 plus)) (for-loop-2d-line pos-14-14 right length-1 minus)) pixel-value-1-12) (d+ (d- drand-5.458352 (for-loop-2d-line pos-13-4 left length-3 plus)) (for-loop-2d-line pos-9-9 upRight length-6 minus)))

Figure 4.35: One of the smallest classifiers evolved by 2d-loops-lines method, shifted objects, images in a two-dimensional representation

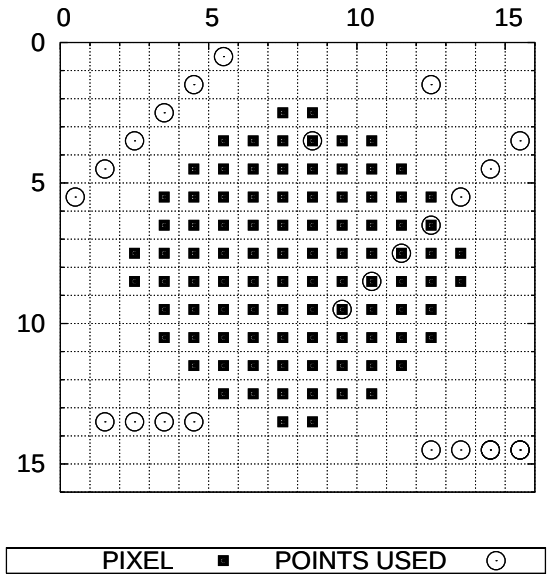


Figure 4.36: Points examined for the program shown in Figure 4.35, 2d-loops-line method, shifted objects, images in a two-dimensional representation

the loop methods are generally faster in CPU time in getting a solution than the runs without loops especially when the problems become harder and the no loop methods do not get many successes.

Table 4.5: Computation time in seconds, 100 runs each, the binary image classification problem

Problems	No loops			Loops				Average per sol. in no loops/ Average per sol. in loops
	Average per eval.	Number of successes	Average per sol.	Loop format	Average per eval.	Number of successes	Average per sol.	
1d-centered	0.000359	52	5.72	for-loop-1d	0.000547	82	7.26	0.79
2d-centered	0.000551	51	11.50	for-loop-2d-rect	0.000306	62	4.20	2.74
				for-loop-2d-line	0.000591	65	9.95	1.16
1d-shifted	0.000486	2	2707.19	for-loop-1d	0.000647	38	164.70	16.44
2d-shifted	0.000391	7	581.20	for-loop-2d-rect	0.000341	22	172.46	3.37
				for-loop-2d-line	0.000460	22	231.65	2.51

4.8 Summary and Conclusion

The goal of this chapter was to investigate the evolution of programs with loops for an image classification problem with implicit repeating patterns, that of distinguishing noisy circles and squares, and to check whether there are performance gains in utilising these loops. The experiments were successful. We have developed a set of loop formats, which lead to successful evolution of programs for the non-trivial image classification task and these loop formats have demonstrated their superiority in helping evolution in getting solutions more quickly and in evolving robust classifiers.

In relation to the research questions, the outcomes are as follows:

1. How can *for-loops* be incorporated into evolved programs for image classification?

   For-loops can be incorporated into evolved program for image classification by representing the images as one- or two-dimensional arrays, so that, the positions of the pixels can be referenced through the arrays. GP with *for-loops* can find the relevant parts of the image and utilise loops to traverse them to make the decision.

2. Does GP with *for-loops* perform better, that is, the classifiers with *for-loops* need fewer generations to evolve and are smaller, more accurate and more understandable than those without loops?

   GP with loops performs better than without. The classifiers with loops need fewer generations to evolve thus a saving in number of the evaluations. The difference was particularly evident in the more difficult shifted problem where GP without loops only gave 2 and 7 successes for one or two-dimension representation images, while GP with loops gave 36 and 18 successes respectively. The classifiers with loops were generally better than those without in that they were more accurate and easier to understand. However, in our experiments, there was little difference in size.

3. What variations of *for-loops* can be used?

   Loops in a simple format can be easily used for this task when the images are in a one-dimensional representation. Specified domain dependant formats of loops can be constructed for GP when these images are represented by two-dimensional arrays. In the

experiments, the proposed loop constructs draw lines or rectangles and use the visited pixels for classification.

There is a trade-off between the number of successes and the number of pixels used with these loop variations. Loops in one-dimensional representations used many pixels and gave many successes. Loops in a two-dimensional representation used fewer numbers of pixels but also gave fewer numbers of successes.

4. What are the differences between decision strategies in the evolved loop and no loop programs?

   The classifiers with loops were more robust in that they examined a sequence of pixels covering the areas in an image in which the circles and squares are different. In contrast, the classifiers without loops examined points randomly scattered throughout the images.

In our work, we have also demonstrated that evolution with loops is faster in CPU time in getting a solution than without loops. It may be noted that none of the loops here actually have an explicit loop index in the body. We have identified that other works utilise loops implicitly in a terminal. It may appear that we have done something similar. We have put the loop in a function in which *start*, *end* and methods of calculation in the loop body are randomly chosen. However, our approach is an advance in the evolution of programs with loops. Despite the loop index which is still implicit, our approach allows all parts of loops to be evolved easily instead of requiring a large amount of human intelligence to define a terminal that captures all components of loops with little flexibility.

In future work, more complex grey level object classification problems can be investigated and different arbitrary shapes can be traversed by the use of loops. Also, loops with explicit indices and nested loops in which an inner variable depends on an outer variable have not been explored and they will be in future work.

Chapter 5

An Analysis of Restricted Explicit For-Loops

5.1 Introduction

The objective of this chapter is to answer the last main research question, that is, whether the performance gain from using explicit *for-loops* can be explained (see Section 4, page 10). In the previous two chapters, several formats of explicit loops have been composed and successfully utilised by GP for solving five simple artificial problems and a more difficult object classification problem. The performance gain in getting faster, smaller and/or understandable solutions in all these problems leads us to investigate why this happens, what the roles of loops are in the evolution and whether these results are purely luck. GP theory studies are still weak (see Section 2.2.7, page 42). With complex terminals, functions and variable structures in our GP representations, it is not possible to use current GP theory results. However, the search space analysis for the Santa Fe ant problem by Langdon and Poli (see Section 2.2.7, page 46) stimulates us to use the same approach for some of our problems. The techniques of the pattern analysis in the schemata theorem studies (see Section 2.2.7, page 42) can also be utilised to check what has been captured by the loops during the evolution. Also, it is known that the setting of the maximum number of iterations affects the performance of evaluation, but in the literature (see Section 2.3, page 53) there are no experiments on how this number influences the evolutionary runs. Experiments on different settings of the maximum number of iterations will be conducted at the end of the chapter.

We have selected two problems for analysis - the visit-every-square problem (see Section 3.8) and the modified Santa Fe ant problem (see Section 3.5). For the visit every square problem, we analyse the search space. For both problems, we analyse the captured repetitive patterns. The reasons for selecting these two problems are: They are representative of a certain class of grid-based problems and their search spaces can be easily adjusted by the size of the grid. Also, the terminals and functions are straightforward, thus it is easy to interpret the patterns captured.

The visit-every-square problem is used again in the end to analyse the evaluation time of different maximum iteration settings.

The results show that there is a larger number of solutions in smaller tree sizes when explicit loops are introduced and the patterns captured by the bodies of the loops are helpful in solving the problems. Also the potential increased computational cost of evaluating an individual can be minimised by utilising domain knowledge to restrict the maximum number of iterations. These results are consistent with the other experiments presented in Chapter 3 and Chapter 4.

5.2 Chapter Goals

The main goal of this chapter is to find out why explicit *for-loops* are good for some GP problems and able to find smaller solutions with fewer evaluations. This main research question has been divided into the following sub-questions:

1. Why is it easier to evolve good small-sized solutions with explicit *for-loops*? Can a comparison of the search spaces with and without loops reveal the reasons?

2. Do looping constructs in programs capture useful patterns that help to improve the fitness and does repetition of these patterns lead to success?

3. Since the evolved programs contain looping constructs, parts of the tree could be evaluated many times, thus increasing the evaluation time of an individual. How significant is this increase? Can it be minimised?

The first experiments are on a simple visit-every-square problem. This problem has been described in Section 3.8. In this chapter, a variation has been made in the grid size. Only 3×3 and 4×4 grids are investigated to facilitate comparison of the search spaces with and without

loops. The reason for this adjustment is to get a relatively smaller search space so that the programs can be enumerated for analysis. The second experiments are on the modified Santa Fe ant problem and a 10×10 visit-every-square problem. The modified Santa Fe ant problem is more complex and provides many opportunities for useful repetitive behaviours and has been described in Section 3.5. This chapter will examine some of the patterns evolved in the loop bodies and determine whether they are the kind of good building blocks, which, if executed a number of times, would facilitate solution of the problem.

Our expectations are that explicit *for-loops* can capture the good patterns and repeat them for a success, but we are not sure how the search space has changed with the introduction of these explicit loops and how significantly the change of the maximum number of iteration will affect the performance of the evolution.

5.3 Syntax and Semantics of the For-loops

In Chapter 3 and Chapter 4, a number of different variations of loops were explored. The focus of this chapter is on how explicit loops affect the performance of GP. So rather than investigating the complex formats of loops which are more domain biased, the experiments in this chapter only look into the simplest form to facilitate analysis. The loop syntax is

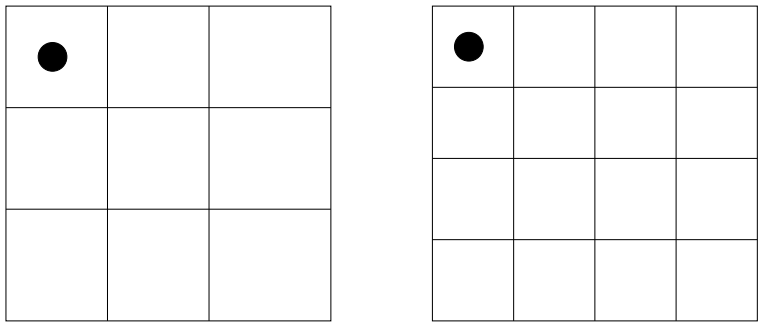
(*for-loop num-iterations body*)

and the semantics are as expected, *body* is executed *num-iterations* times. *Num-iterations* takes a value between 1 and a globally defined parameter, *max-iterations*.

Strongly typed genetic programming (STGP) is used to take care of multiple data types and enforce closure by only allowing parse trees which satisfy the type constraints. In this chapter, both *num-iterations* and *max-iterations* are of integer type. The *for-loop* function and the whole program return a dummy type. Nested loops are allowed.

5.4 Search Space Analysis — The Visit Every Square Problem

Experiments with the visit-every-square problem have been described in detail in Section 3.8. In that section, the task was to direct a robot to navigate through a 4×4 or 6×6 or 8×8 grid (see Figure 3.30). The robot begins in the top left square and needs to pass through every square. Each square can be visited more than once, but each square must be visited at least once in a



(a) 3×3 problem (b) 4×4 problem

Figure 5.1: The visit-every-square problem

solution.

In order to facilitate the search space analysis and make the enumeration of the candidate solutions possible, only 3×3 and 4×4 grids are used (see Figure 5.1).

5.4.1 Genetic Environment Settings

Functions and Terminals

Functions and terminals are the same as before. There are four terminals {*Left, Right, Up, Down*} for this problem. If a move through a border is required, for example, *Up* or *Left* from the positions shown in Figure 5.1, the robot takes no action. There is only one function *Prog2* for the no-loops approach and it takes two arguments and executes them sequentially. The loops approach uses the *for-loop* function described in Section 5.3.

The Fitness Function and Other Genetic Environment Settings

The fitness is the number of the non-visited squares. To simplify analysis of the search spaces, the maximum tree depth is set to 4. This allows at least one possible solution for the no-loops method in a 3×3 grid. Using domain knowledge *max-iterations* is assigned to 3 for the 3×3 and 4 for the 4×4 problem. The rest of genetic variable settings can be viewed in Table 5.1.

5.4.2 Experiments and Experimental Results

The GP program was run 100 times with and without loops for the 3×3, 4×4 visit-every-square problem with variable settings shown in Table 5.1.

Table 5.1: Parameter settings, the visit-every-square problem

PARAMETERS	**VALUES**
Population Size	100
Max. Generation	50
Mutation / Crossover / Elitism Rate	28% / 70% / 2%
Initialization Method	Ramped half-and-half
Selection Method	Proportional fitness
Termination Criteria	Successfully visited every square or 50 generations reached

Experimental Results

Figure 5.4 shows the cumulative probability of getting a successful solution for the 3×3 visit-every-square problem. All runs with loops found solutions within 40,000 evaluations, while only 65 solutions were obtained with the no-loops method by 50,000 evaluations. This difference is in good agreement with the previous work, that is, GP with looping constructs results in more successful solutions with fewer evaluations.

There are only 8 possible solutions for programs without loops with a maximum tree depth of 4 and it is not possible to get a solution that visits a square more than once within this tree depth. These solutions are shown in Figure 5.2. However, with looping constructs, there are 108 solutions, nearly 14 times as many. The numbers of possible solutions for both approaches were obtained by enumerating all of the possible programs and evaluating them. Figure 5.3 shows some solutions with loops.

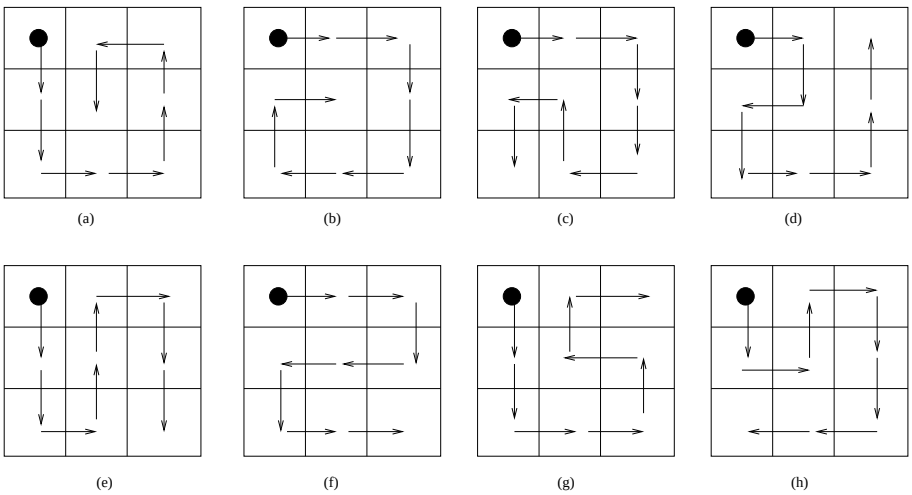


Figure 5.2: All possible solutions for no-loop programs, the 3×3 visit-every-square problem

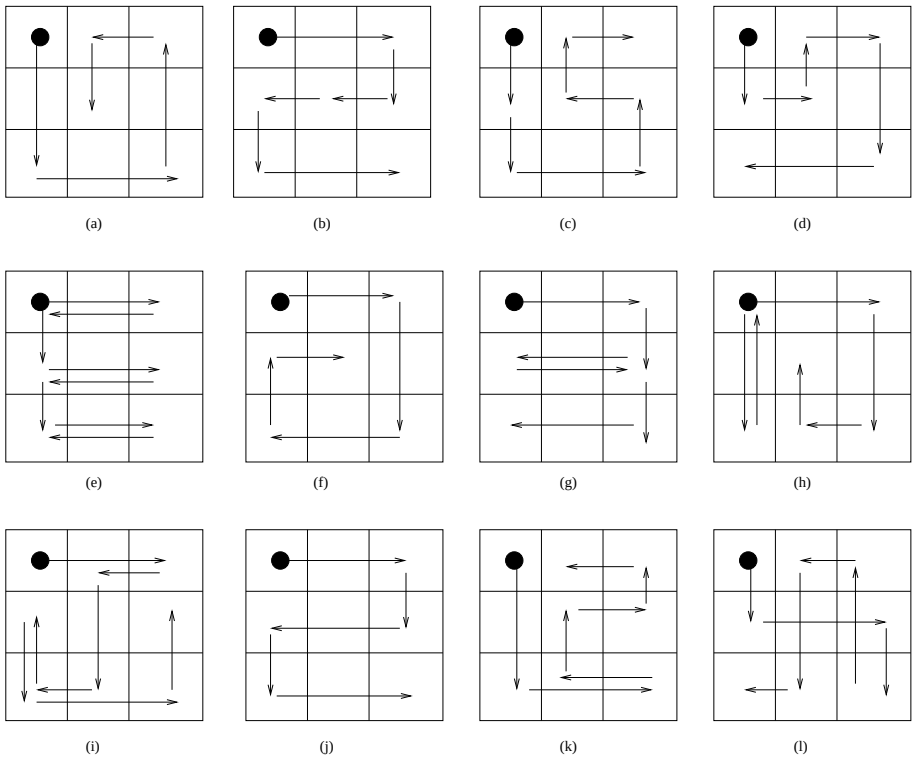


Figure 5.3: Some solutions for programs with loops, the 3×3 visit-every-square problem

Figure 5.5 shows the fitness of the best individual, averaged over 100 runs, for the loops and no-loops approaches for the 3×3 and 4×4 grids. The 4×4 problem with loops also showed a quicker improvement in fitness, using the same settings. Due to the tree depth limit, programs without loops cannot have fitness better than 7, because the maximum number of actions for a tree structure of depth 4 is 8.

5.4.3 Analysis of the Fitness Landscape

As noted above, all possible programs for the visit-every-square problem up to a tree depth of 4 were enumerated and the fitness evaluated. For the no-loops approach there are 163,220 possible programs in contrast to 1.2 million possibilities with the looping constructs. These two numbers were obtained by enumerating all tree shapes and then all labellings of each shape by functions and terminals. The number of possible programs with loops is considerably higher because there are more ways of labeling the nodes in the trees due to the range of possible values for the additional terminal *num-iterations* and function *for-loop*.

Figures 5.6 and 5.7 show the fitness distribution based on program length for the no-loops

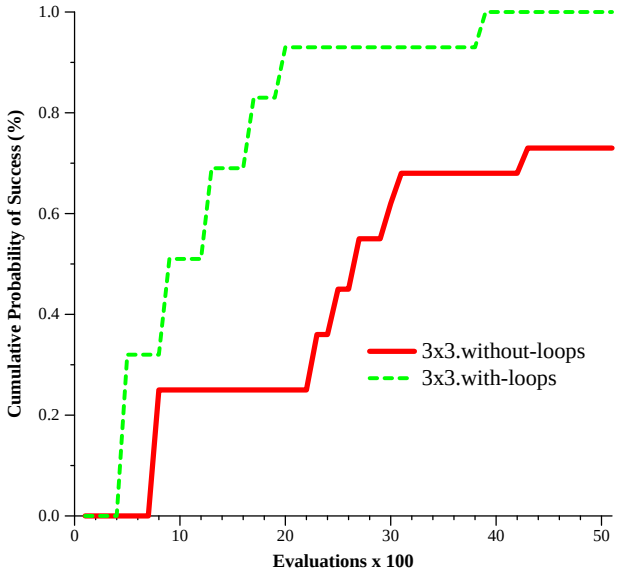


Figure 5.4: Cumulative probability of success, average of 100 runs, the 3×3 visit-every-square problem

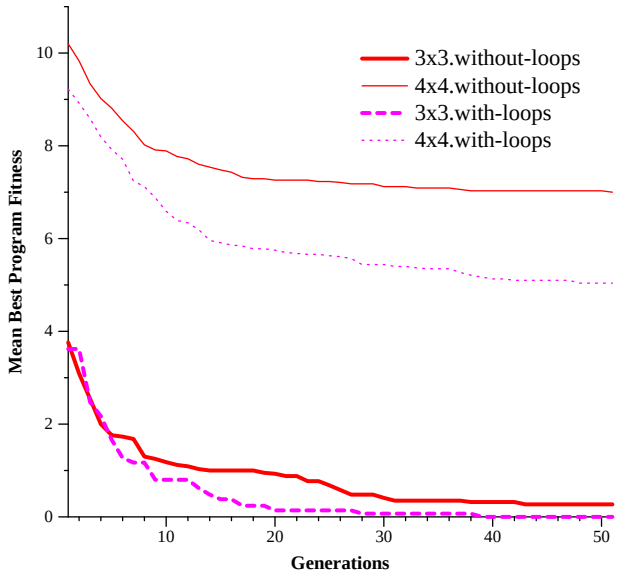


Figure 5.5: Mean best program fitness, average of 100 runs, the 3×3 and 4×4 visit-every-square problem

and loops approaches for the 3×3 visit-every-square problem. The x-axis reflects the fitness. The more squares visited, the better the program. The y-axis shows program length, that is, the number of nodes in the program. The number of programs is indicated by z-axis. To facilitate comparison, the scale of the z-axis is the same in both Figures 5.6 and 5.7. It can be seen that with the same depth constraint, there are many more programs with high fitness with loops than without. The 8 solutions for the no-loops approach and the 108 solutions for the loops approach are located in the square where the program-length is 14 and squares-visited is 9. However, they are not visible in these two graphs due to the scale of z-axis.

Figures 5.6 and 5.7 correlate with the higher fitness values for loop programs in Figure 5.5. However, they do not necessarily explain Figure 5.4. The proportion of solutions in the search spaces is roughly the same: 8/.16M is approx 80/1.6M which is the same order of magnitude as 108/1.2M. So why are the solutions with loops significantly easier to find? It would appear that solutions with loops are located in regions of high fitness and once the genetic search finds such a region, finding the actual solution is not difficult.

A solution from the no-loops approach (Figure 5.8) and one from the loops approach (Figure 5.9) are compared. In Figure 5.9 the left hand side symbol of the *for-loop* function indicates the number of iterations. For example, t2 in Figure 5.9 means repeat any operations in the

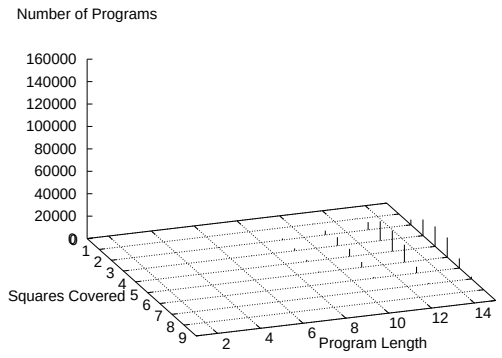


Figure 5.6: Fitness distribution based on program length, maximum depth constraint 4, the no-loops approach, the 3×3 visit-every-square problem

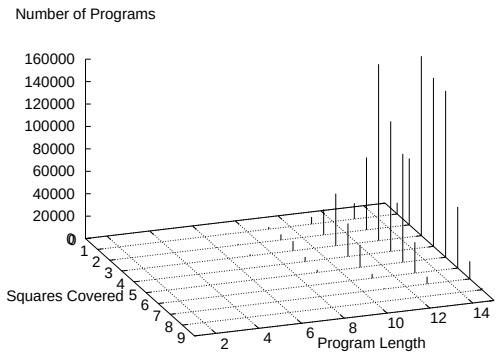


Figure 5.7: Fitness distribution based on program length, maximum depth constraint 4, the loops approach, the 3×3 visit-every-square problem

right hand branch two times. To visit all nine squares in a 3×3 problem requires at least 8 movements. A successful no-loops program needs to be a full tree with terminals in the correct order. In programs with loops, the consecutive identical actions are represented by (*for-loop num-iterations action*), which is not only a potential saving in length, but also makes repetition of good movement sequences more likely. For larger sized grids without loops a bigger tree with the terminals in the correct order needs to be discovered. For larger sized grids with loops, the same sized tree, but with different terminals, will be adequate. As we show in the next paragraph, requiring larger trees causes a massive explosion of the search space.

Consider the problem of traversing a row or column of the grid. This would be a useful partial solution of the visit-every-square problem. Suppose that the row/column lengths are 8, 16, 32, 64, 128 respectively. To visit to each square a no-loops solution needs to be a full tree of depth 4, 5, 6, 7, 8. The total numbers of different unlabeled binary trees for depths $d = 4, 5, 6, 7, 8$ are $21, 651, 457653, 2.10E + 11, 4.4E22$ respectively. This is a 'double exponential' sequence and after a depth of 5 the number of possible trees becomes massive (see Section 2.2.6, page 38). The probability of finding a no-loops program with the right shape by random search for the maximum tree depth setting of 4, 5, 6, 7, 8 is $1/21, 1/651, 1/457653, 1/2.10E + 11, 1/4.4E22$ respectively. The probability of finding the right shape with the right labels is even smaller. When using loops, a very short program: (*for-loop num-iterations right*) will complete the task if *num-iterations* is equal to or larger than the row/column length. This only needs a tree depth

of 2. The number of different tree shapes for a depth of 2 is 1. A search only needs to find the correct labeling of one tree.

By extending the above analysis from rows/columns to the entire grid, it is clear that as the size of the grid increases the chances of finding a no-loops solution are very greatly diminished because of the need to search through the massive number of possible tree shapes, while the chances of finding a loop solution decrease somewhat because of the need to search through alternate labellings of smaller trees.

Another possible reason that the loops approach works better is that when a good building block is found, there is a mechanism available to repeat it. If there are no loops, the building block needs to be discovered independently a number of times and correctly aligned with the other occurrences. In effect, an unrolled loop needs to be discovered.

A generic solution for all sizes of the visit-every-square problem is shown in Figure 5.10. This solution was evolved by accident and not by design. It was evolved with a larger tree depth (depth 5) constraint and {+, -} were added to the function set. The value of *max-iterations* was changed to reflect the size of the grid and was also available as a terminal. In this evolved solution the robot first moves down the first column and then back up to the starting square before moving one square to the left and repeating the down and up motion as shown in Figure 5.8. This action is repeated until every square is visited. The solution is of tree depth 5 and size 14. It repeats the pattern {*for-loop max-iterations - 1 down or up*} to complete the task. A program without loops to solve a 10×10 visit-every-square problem needs a tree with 100 leaves, which will require tree depth of at least 7. There are $2.10E + 11$ unlabeled binary trees of depth 7 and even more programs. This is somewhat like looking for a few needles in a very large haystack. With loops, crossover will help to pass good subtrees like (*max-iterations - 1*) to the descendants.

In this section, by comparing characteristics of the search spaces, we have shown why solutions with loops are much more likely to be found for problems like the visit-every-square problem and that programs with loops will scale better to larger instances of the problems.

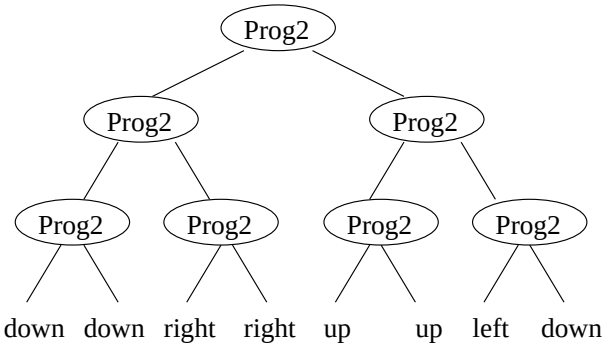


Figure 5.8: A solution from the no-loops approach, traversal pattern of the program shown in Figure 5.2 (a), the 3×3 visit-every-square problem

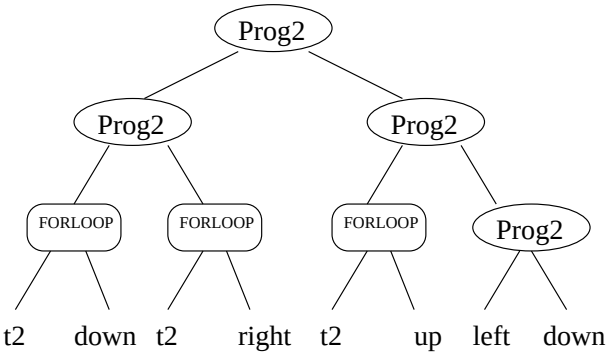


Figure 5.9: A solution from the loops approach, traversal pattern of the program shown in Figure 5.3 (a), the 3×3 visit-every-square problem

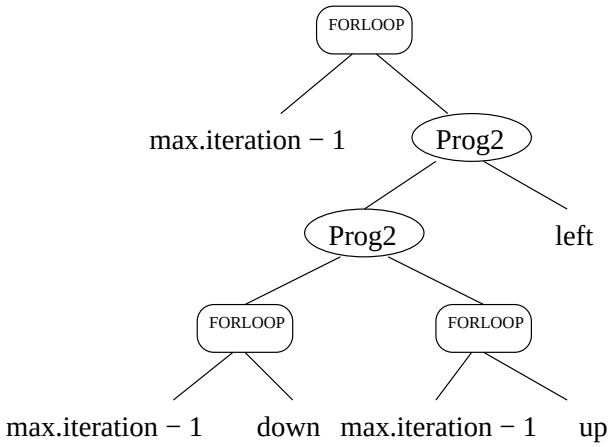


Figure 5.10: A generic solution of depth 5, the visit-every-square problem

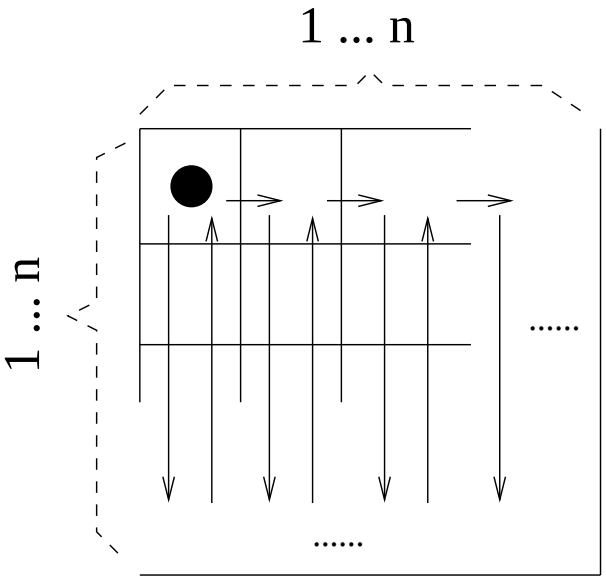


Figure 5.11: Traversal pattern of the generic solution shown in Figure 5.10, the visit-every-square problem

5.5 Pattern Analysis

5.5.1 The Modified Ant Problem

The modified ant problem was introduced in Section 3.5 and the intention was to investigate how explicit looping could be applied and be of benefit. The original ant problem (see Section 2.3.3) was modified to make it more amenable to programs with loops. The size of the grid was changed to 20×20 and 108 pieces of food were placed on the grid in three 6×6 blocks (see Figure 3.4). A small penalty was applied for the number of loops in an individual in order to encourage the evolution of programs with the smallest possible number of loops. At that time, the results showed that without loops (implicit or explicit), it was so hard to find a solution that out of 300 runs only one solution was found with a tree depth of 10. This solution is huge and contains 5,000 nodes. With explicit loops, many solutions were found. This is consistent with the results for the visit-every-square problem described above and the other problems investigated.

In this section, we investigate the kinds of patterns that frequently occur in the loop bodies of highly fit programs. Are there the kinds of patterns that, when repeated, will improve fitness?

Genetic Environment Settings

The same function and terminals are used (see Tables 3.1, 3.2). The population size is reset to 2,000 because we want as many solutions as possible for analysis. Other genetic parameter settings are the same (see Table 3.3).

Pattern Mapping Setting

To facilitate the analysis, the tree structures are converted to strings. Node names have been replaced by one character symbols (see Table 5.2). For example, the function *IfFoodAhead* is represented by "f". "Don't care" functions (e.g. Prog2 and Prog3) or terminals (e.g. RandNumber) are replaced by "#", because they are not important in this analysis. The brackets between functions have been removed with an intention to make the patterns more readable. The program "(Prog2 (Prog3 (IfFoodAhead Move Move) TurnLeft Move) Move)" will be converted to "##f11211". The objective of the conversion is to enable quick string matching, sorting and counting.

A pattern analysis program has been written to count the number of patterns of different

string lengths. For instance, an evolved tree represented by "##f11211" has "1" as the most frequently occurring pattern (MFP) of length of 1, since it occurs four times and has "11" as the most frequently occurring pattern of length 2, since it occurs twice.

Experiments

The *for-loop* format described in Section 5.3 is used. 200 runs were executed and 97 successful solutions were obtained. A successful solution means that the evolved program can direct the ant to eat all 108 pieces of food within 600 steps.

Table 5.2: Mapping table for functions and terminals, the modified ant problem

Nodes	Converted Symbol
Prog2	#
Prog3	#
IfFoodAhead	f
ForLoop	L
Move	1
TurnLeft	2
TurnRight	3
RandNumber	#

Table 5.3: Frequent pattern occurrences in the 97 evolved solutions and the corresponding program segments, the modified ant problem

Pattern	Number	Percent	Translation
L#	97	100%	ForLoop Num-Iterations
L#f1	65	67.01%	ForLoop Num-Iterations (IfFoodAhead Move
L#f13	33	34.02%	ForLoop Num-Iterations (IfFoodAhead Move TurnRight)
L#f12	27	27.84%	ForLoop Num-Iterations (IfFoodAhead Move TurnLeft)
L#L##f	22	22.68%	ForLoop Num-Iterations (ForLoop Num-Iterations (Prog2 (IfFoodAhead

Experimental Results

Table 5.3 displays the frequent patterns found in those 97 solutions. These patterns all have *for-loop* at the beginning and the actions within the loop body have been repeatedly executed.

Analysis of Results

The data in Table 5.3 reveals that every successful solution found contains looping constructs "L#" indicating that at least one loop is important in the solution. Actions captured in the loop include "L#f12" and "L#f13", that is, "(IfFoodAhead Move Left)" and "(IfFoodAhead Move Right)". This results in iteration of the basic behaviour: If there is food in the square you are facing, move into the square and eat the food, or else turn left/right. This is clearly the kind of behaviour that will follow a line of squares containing food. In addition, these patterns may occur a number of times in a solution and tend to have positive effects in solving the problem.

Figure 5.12 shows a solution using loops and Figure 5.14 shows the traversal pattern generated by executing this program. The traversal pattern for Figure 5.14 is shown in more detail in Figure 5.13 which shows the order in which the squares are visited. The pattern "f1" in the looping construct "L##f1" is executed 6 times at the position indicated by arrow (a) in Figure 5.14 and this definitely improves the fitness. Arrows (b) and (c) show the points at which the pattern "f13" or "f12" redirects the ant to traverse in a circle fashion by either turning left or turning right when there is no food ahead. These two patterns are within the looping construct "L#", so the ant need only make a turn when all of the pieces of food in a line are consumed. The root of the program is a *for-loop* function "L#" followed by another frequent pattern "L#L##f", see Table 5.3.

Table 5.4 shows the most frequent patterns from the last generations of 100 runs without loops at a maximum tree depth of 9. No solutions were found. In contrast to the loop patterns, there are no domain regularities here, just program control structure.

5.5.2 The Visit-Every-Square Problem

The visit-every-square problem was used for the search space analysis (see Section 5.4). It was found that GP with loops can generate more good solutions than without loops and many good solutions are evolved at a small tree depth setting. In this section, this problem is analysed by the same pattern analysis tools as the modified ant problem to find frequently occurring combinations. The size of the grid has been adjusted to 10×10 instead of 3×3 or 4×4 and the maximum number of iterations has been updated to 10.

(ForLoop times4 (ForLoop times6 (ForLoop times6 (Prog3 (IfFoodAhead Move (Prog3 Move TurnRight move)) (ForLoop times2 (IfFoodAhead Move TurnRight)) (ForLoop times3 (IfFoodAhead Move TurnLeft))))))
CONVERTED FORMAT:
L#L#L##f1#131L#f13L#f12

Figure 5.12: The smallest solution evolved with loops, the modified ant problem

	0	1	2	3	4	5	6	7	8
0		1	2	3	4	5	6	7	
1		14	13	12	11	10	9	8	
2		15	30	29	28	27	26		
3		16	31	38	37,39	36	25		
4		17	32	33	34,40	35,41	24		
5		18	19	20	21	22,42	23,43		
6							44	45	46

Figure 5.13: Detail of traversal pattern of Figure 5.14, numbers show order in which grid positions are visited, the modified ant problem

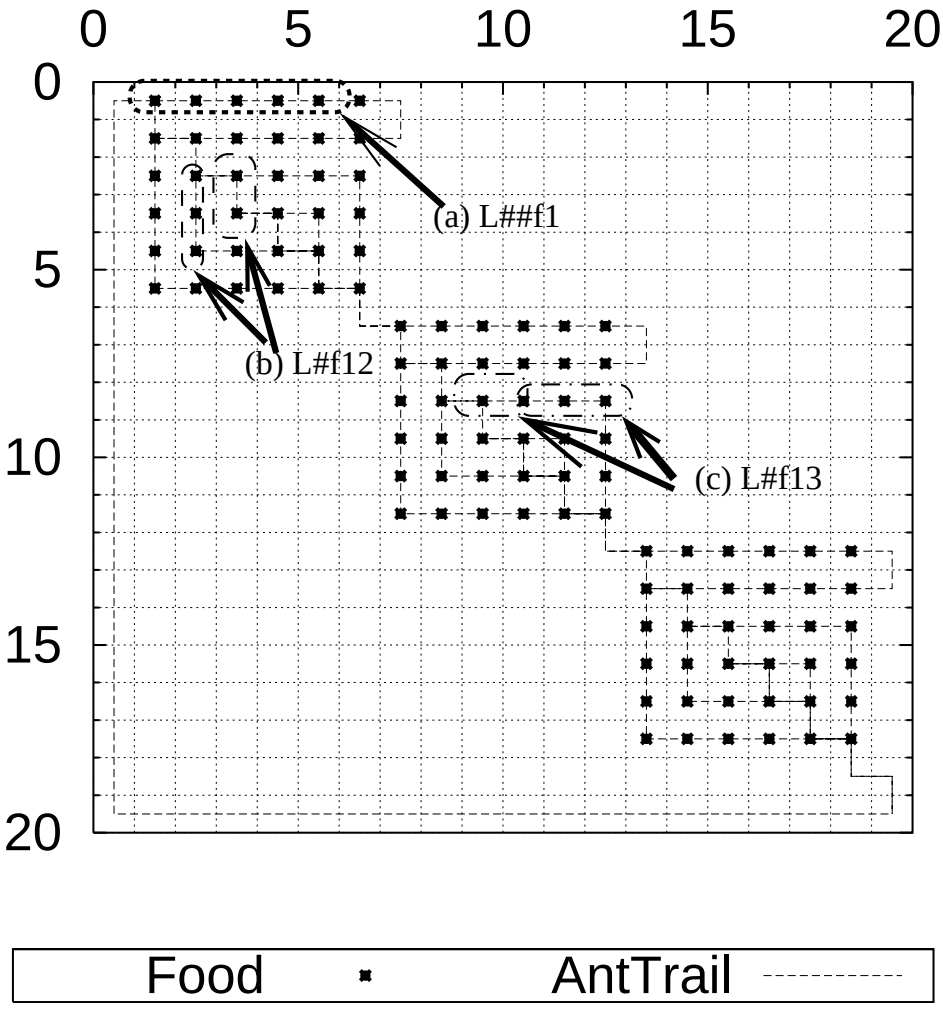


Figure 5.14: Traversal pattern of the program shown in Table 5.12, the modified ant problem

Table 5.4: Top 10 frequent patterns found in the no-loops program logs, the modified ant problem

Pattern	Translation
#	(Prog2|3
f	(IfFoodAhead
##	(Prog2|3 (Prog2|3
f#	(IfFoodAhead (Prog2|3
###	(Prog2|3 (Prog2|3 (Prog2|3
f##	(IfFoodAhead (Prog2|3 (Prog2|3
####	(Prog2|3 (Prog2|3 (Prog2|3 (Prog2|3
f##1	(IfFoodAhead (Prog2|3 (Prog2|3 Move
#####	(Prog2|3 (Prog2|3 (Prog2|3 (Prog2|3 (Prog2|3
f##13	(IfFoodAhead (Prog2|3 (Prog2|3 Move TurnRight

Table 5.5: Mapping table for functions and terminals, the visit-every-square problem

Nodes	Converted Symbol
Prog2	#
ForLoop	L
UP/RIGHT/DOWN/LEFT	1/2/3/4
RandNumber	#

Genetic Environment Settings

The maximum tree depth has been reset to 7 and the maximum allowed generations is reset to 100 instead of 50. Terminals and functions are the same as before (see Section 5.4.1). Other variables remain unchanged (see Table 5.1).

Pattern Mapping

The functions and terminals are converted to a symbol to feed into the analysing tool. The mapping is listed in Table 5.5. The terminals, {*up, right, down, left*}, have been assigned to "1, 2, 3, 4" respectively. The *for-loop* function has been assigned to "L". Others, {*Prog2, RandNumber*}, have been assigned to "#", because they are not important in this analysis.

Experiments

GP with the above settings has been run 200 times each for the programs with loops and without loops. The converted solutions have been put into the analysis tool to find the frequent patterns.

Table 5.6: Frequent pattern occurrences in the 134 evolved solutions and the corresponding program segments, the visit-every-square problem

Pattern	Number	Percent	Translation
L#	134	100%	ForLoop Num-Iterations
L#1	81	60.4%	ForLoop Num-Iterations UP
L#2	51	38.1%	ForLoop Num-Iterations RIGHT
L#3	96	71.6%	ForLoop Num-Iterations DOWN
L#4	45	33.6%	ForLoop Num-Iterations LEFT
L##L#	38	28.5%	ForLoop Num-Iterations (Prog2 (ForLoop Num-Iterations
L###L#	78	58.2%	ForLoop Num-Iterations (Prog2 (Prog2 (ForLoop Num-Iterations

```
PROGRAM :
(ForLoop t10 (Prog2 (ForLoop t7
   (ForLoop t7 down))
   (Prog2 (ForLoop t10 up)
          (ForLoop t1 right))))

CONVERTED FORMAT:
L##L#L#3#L#1L#2
```

Figure 5.15: A typical solution evolved using loops, the visit-every-square problem

Experimental Results

Table 5.6 shows the frequent patterns found in 134 solutions for GP with loops. The looping constructs exist in every solution and have captured all direction terminals {*up, right, down, left*}. These combinations allow the programs to traverse a line in any direction to complete the task. Every solution has "L" as the root node and has nested loops. This is because there are no restrictions for number of visits to a square, and programs with loops re-visit the squares freely to complete the task.

Figure 5.15 demonstrates a typical solution with loops. It follows the same pattern as the generic solution described in Section 5.4.3 (see Figure 5.11). The program moves down and up and then left and these actions are repeated until every square is visited.

None of the programs without loops got a successful solution in 200 runs even though it is possible to have solutions without loops at the depth setting of 7. The fittest program only visited 58 squares and left 42 squares unvisited.

In this section we have shown that good building blocks have been captured in the loop bodies

Table 5.7: Computation time in seconds, 100 runs each (50 for last row), the 6×6 visit-every-square problem

MAX-ITERATIONS	Total CPU Time(Sec)	Total Evaluations	Average per Eval	Number of successful runs	Average Time per solution
No-loops	92.35	389166	0.00023	92	1.004
1	214.04	872833	0.00024	37	5.785
2	145.26	590782	0.00024	80	1.816
3	59.74	250143	0.00023	98	0.610
4	35.84	157355	0.00022	100	0.358
5	22.71	100788	0.00022	100	0.227
6	20.76	92485	0.00022	100	0.208
10	33.01	144154	0.00022	100	0.197
20	40.62	174761	0.00023	100	0.406
50	16.08	58636	0.00027	100	0.161
100	36.24	80275	0.00045	100	0.362
200	1546.75	78237	0.01977	100	15.460
400	335882.65	49611	6.77000	50	6717.653

and they lead to improvement in fitness when repeated.

5.6 Computation Time Analysis — The Visit Every Square Problem

As stated in Section 5.3, the implementation of loops requires the value of *max-iterations* to be set at the beginning of the run. The value of *max-iterations* will clearly affect the CPU time required for an evaluation. In this section, we examine the effect of increasing values of *max-iterations*.

The computational complexity is $O(pn^k)$, where p is the population size, n represents increasing values of *max-iterations* and k is the maximum number of levels of nested loops.

For an empirical investigation, we use the visit-every-square problem described in Section 5.4 with some slight changes. A 6×6 grid is used and the maximum tree depth is 7. The maximum number of generations is 100 instead of 50. Other genetic parameter values are the same as in Table 5.1.

5.6.1 Experiments and Experimental Results

Experiments

There are known issues associated with the use of CPU time to compare the speed of algorithms as mentioned in Section 4.7.5. We have taken the same methods to minimise these effects as the CPU time measurements described in Section 4.7.5.

Experimental Results

Table 5.7 shows the computation times for the 1250 runs. The first column gives the value of *max-iterations*, the second gives the total CPU time for all of the runs, the third gives the total number of evaluations, the fourth gives the average time in seconds per evaluation (Total CPU/Total Evaluations), the fifth gives the total number of successful runs and the last gives the average time in seconds per solution (Total CPU/Total no of successful runs).

5.6.2 Analysis of the Results

Inspection of the fourth column of Table 5.7 shows that the average CPU time per evaluation does not increase noticeably until *max-iterations* reaches 100. The last three rows in the table are consistent with an $O(pn^k)$ complexity.

Perhaps more interesting are the last 2 columns. If we take the *no-loops* results as a baseline for comparison we find that the average time per solution increases for *max-iterations*=1,2. This is not surprising. A *max-iterations* value of 1 does not permit any repetitive behaviour, but additional terminals and functions have been introduced and the space of possible programs is increased without any benefit to the nature of a solution. In effect the *for-loop* function will act as noise and waste evolution time and space. A *max-iterations* value of 2 permits some repetition, but, based on domain knowledge, we know that this is not really significant. Once *max-iterations* reaches 5, a significant value from the point of view of domain knowledge, all runs are successful and the average times to a solution are considerably smaller than for no-loops. The wasted effort from having *max-iterations* too big becomes significant from 200.

5.7 Summary and Conclusion

The aim of this chapter was to find explanations for why using loops in genetic programs for problems with repetitive characteristics is superior to genetic programming without loops. To do this we asked and answered three questions:

1. Why is it easier to evolve good small-sized solutions with explicit for-loops? Can a comparison of the search spaces with and without for-loops reveal the reasons?

   For the visit-every-square problem we found that there are more higher-quality individuals in a search space with loops than without loops and the chance of finding a solution is higher. We found that as the problem size increases, a no-loops solution requires search through tree structures of greater depth and the number of possible tree shapes increases at a "double exponential" rate with depth. This drastically lowers the chance of finding a solution. However, for programs with loops, the search can be restricted to different labellings of smaller trees, which is only exponential.

2. Do looping constructs in programs capture useful patterns that help to improve the fitness and does repetition of these patterns lead to success?

   For the modified Santa Fe ant problem and the visit-every-square problem, we found the most frequently occurring patterns within loop bodies of solutions and other highly fit individuals. Analysis of these patterns showed that they are the kinds of basic building blocks which, if executed repeatedly, do improve fitness.

3. Since the evolved programs contain looping constructs, parts of the tree could be evaluated many times, thus increasing the evaluation time of an individual. How significant is this increase? Can it be minimised?

   In our runs, we have used a global parameter *max-iterations* to give an upper limit on the number of iterations of a loop. In the two problems used in our experiments, choices of this value based on domain knowledge led to good performance without significant increase in the evaluation time of an individual. However, the computational complexity is $O(pn^k)$, thus, our approach does not scale well to larger values. However, while the average cost of evaluating an individual is higher, the programs are fitter and solutions are found in a much smaller number of generations.

The major drawback of the use of loops is that the maximum number of iterations needs to be decided beforehand. This is in keeping with other work on loops in genetic programming where some kind of bound on the maximum computational effort in a loop is used (see Section 2.3.2).

Chapter 6

Conclusions

In this chapter we present the findings and the conclusions and provide suggestions for future work.

The broad aim of the thesis was to study how bounded loops, that is, a limited form of loops in which infinite loops are not possible, can be incorporated into tree-based genetic programming and to determine whether incorporating such loops delivers any major benefits. For this, we have developed a number of restricted *for-loop* formats. These restricted loops use domain information to restrict the possible number of iterations and can easily be utilised. We have demonstrated their advantages on a range of problems, from simple toy problems to more difficult image classification problems. We found the major benefits for runs with loops were that: (1) While the CPU time for evaluating an individual increased when loops were used, the CPU time for finding a solution was lower if relevant domain knowledge was used. (2) The evolved solutions with loops were more understandable because they were small in size and captured the repetitive patterns of the problem.

6.1 Conclusions Relating to Research Questions

The thesis addressed four main research questions (see Section 1.3.1, page 9):

1. *How can we restrict the syntax and semantics of for-loops in a way that avoids problems of infinite loops and still provides useful benefits for genetic programming?*

Format 1: *(for-loop1 num-iterations body)*

Format 2: *(for-loop2 start end method)*

We have composed two formats of *for-loops* in which infinite loops are not possible. In loop format 1, *body* is executed *num-iterations* times. *Num-iterations* is restricted by a user-supplied domain dependent *max-iterations* parameter. The *body* is a combination of terminals and functions. During evolution, both *num-iterations* and *body* undergo crossover and mutation. In loop format 2, *body* is executed once for each value of a counter between *start* and *end*. The *body* utilises an index variable for traversal of arrays or vectors. The increment value of the index is set to 1.

Infinite loops are avoided by setting a maximum number of iterations for loop format 1 and by setting the *start* and *end* values for loop format 2. These values are user supplied based on analysis of the problem domain. For example, when traversing a grid, the grid width or depth are good values for the maximum number of iterations; when sorting an array, the *start* and *end* values must be within the range of the array positions. These constraints together with the maximum tree depth settings naturally restrict the total number of iterations and make infinite iteration impossible.

Major benefits provided by loops are: Solutions can be obtained in fewer evaluations and these solutions are more understandable than without loops. The results of five experimental problems demonstrate that GP with these restricted explicit loops requires a much smaller number of evaluations to get fitter programs or solutions. Solutions evolved with loops were generally smaller than without loops and the loop bodies captured the repetitive patterns of the problem thus solutions were more easily understood.

It could be argued that, while we claim to be doing loops, we have created a number of complex functions. On the one hand, we did create a number of complex functions which performed iterations. On the other hand, iterations were rarely used in genetic programming and we have created relative simple loop formats and rules for the loops and provided an analysis to demonstrate that they are helpful in decreasing number of evaluations. This has not been achieved by others.

2. *Can GP with for-loops solve some problems that cannot be solved or are very difficult to solve without explicit loops?*

Yes, GP with *for-loops* restricted by domain information has solved the Santa Fe ant problem,

a modified Santa Fe ant problem, a visit-every-square problem and a sorting problem. In all of these problems, GP without loops always, or almost always fails.

In Chapter 3, we investigated *for-loop1* by using the Santa Fe ant problem and a variant, the modified Santa Fe ant problem. We required generated individuals to be called once, unlike the typical solution in which an individual is called repeatedly until a maximum number of steps is reached. The results showed that ants could not eat all the food without utilising loops, even though the tree depth was set large enough to allow no-loops solutions. The large tree depth setting exponentially increases the search space and dramatically decreases the chance of finding a successful solution. The same outcome occurred for *for-loop2* when it was applied to the sorting problem. In this problem, the no-loops approach did not get a successful solution in 100 runs of 4,000 evaluations. In contrast, with loops, solutions were consistently evolved within 4,000 evaluations. This is because with loops, the compare-and-swap pattern was captured in the loop bodies and was repeated to achieve the goal.

3. *Can for-loops be used in a difficult object classification problem with similar performance gains to those achieved on relatively simple artificial problems?*

Yes, *for-loops* can be used in a difficult object classification problem to distinguish noisy circles and squares. On this problem, the use of loops delivered the same performance gains as those achieved on relatively simple artificial problems.

The task of this object classification problem is to distinguish noisy circles and squares in binary images. The problem is reasonably difficult because pixels in objects have been randomly removed. In an extended version, these random-pixel-removed objects have been shifted in random directions to increase the difficulty. It is hard to find a classifier that has 100% accuracy on training and testing data to differentiate these two classes of objects in such images.

Our results demonstrate that with loop format 2 and its refinements, GP gave more successful classifiers with fewer evaluations than GP without loops.

However, this loop format can only be used for images in a one dimensional representation. Because of the layout of the pixels in the one dimensional representation, many pixels were used by classifiers with loops. Refinements of loop format 2 were used for images in a two dimensional representation. These refinements decrease the number of pixels used by the successful classifiers. The consequences of the refinements are: (1) The evolved classifiers utilise fewer pixels, thus like classifiers without loops, use less information from the images. When applied to other unseen

images, the classifiers containing less information are more likely to fail. (2) To construct loops for images in the two-dimensional representation, users need to decide in advance what traversal patterns will be permitted - lines, squares, rectangles etc. and then design appropriate formats to put looping in a function.

4. *How can the performance gains from using for-loops be explained?*

The performance gains can be explained by examination of fitness distributions, by analysing patterns captured in loop bodies and by analysing the relationships between the maximum number of iterations and the CPU time.

Two methods have been used to analyse why evolving programs with restricted forms of loops are superior to those without for the visit-every-square problem and the modified ant problem in Chapter 5.

The first method analysed the search space for a number of instances of the visit-every-square problem. Programs with and without loops up to a fixed maximum tree depth were enumerated and fitness distributions were plotted with the x-axis showing the fitness, y-axis program length and z-axis the number of programs. The results showed that there were many more good programs as well as solutions for GP with loops. The graph revealed that programs of good fitness were at the same region in the fitness distribution graph.

The second method was to analyse patterns captured by the loop bodies. The analysis found that every solution had looping constructs and that patterns captured by loops were useful. The experimental results show that both the visit-every-square problem and the modified ant problem have repetitive characteristics and GP with loops can always use them to improve the fitness and get solutions in fewer evaluations.

To find out the relationship between the maximum number of iterations and the cost of CPU time, we have experimented with the visit-every-square problem with different values of *max-iterations* and measured the overall CPU time for getting a successful solution. The results showed that: (1) There is a range of values for *max-iterations* in which GP with loops almost always finds a solution in fewer evaluations and shorter CPU time than GP without loops. (2) A too-small value for *max-iterations* does not help GP with loops in getting a solution quickly. (3) A too-large value for *max-iterations* still allows the evolutionary process to find a solution in fewer evaluations but greatly increases the CPU time.

While we have analysed two problems to this level of detail, we believe that the findings

would be similar for other problems with repetitive characteristics.

6.2 Interesting Findings

During the course of the investigation, there were a number of results that were not directly related to the research questions, but are worth stating.

1. *Favouring a smaller number of loops is better than just providing looping constructs in most problems.*

Loops are helpful for solving problems where repetitiveness exists. Although the cost of evaluating an individual increases, fewer evaluations are needed to get a solution than when loops are not used. This effect is even more evident when using the fitness function to favour programs with small numbers of looping constructs. With this setting, GP used even fewer evaluations to get a solution than without it. The loops tend to be positioned at the root of the program tree and each loop has a large body.

2. *A good choice for the maximum allowed number of iterations is a value between one and two times a domain parameter.*

The study in Chapter 5 found that increasing the maximum allowed number of iterations for loops will increase the evaluation time of an individual. For the visit-every-square problem, the average evaluation CPU time for a solution varied from 1 second to 6,717 seconds when GP allows 1 to 400 maximum iterations for a looping construct. There is a clear diminishing return between increasing the allowed number of iterations and the improvement because the slowness in evaluation outweighs the saving in finding solutions in fewer generations with the help of loops. The experiments found that the approximate maximum allowed number of iterations can be set at twice the known potential looping times. For example, the maximum allowed number of iterations for each loop can be between 6 and 12 for a 6×6 visit-every-square problem in order to get good solutions in fewer evaluations. While there were a few exceptions, in general, we found that setting the maximum number of iterations to one or two times a domain parameter was the best choice.

3. *Setting an appropriate maximum tree depth is important for GP with loops in order to get successful solutions in lower CPU time.*

The computational complexity of evolving programs with loops is $O(pn^k)$, where p is the population size, n represents values of *max-iterations* and k is the maximum number of levels

of nested loops. The value of k is closely related to the tree depth. If the maximum tree depth setting is too small, there will not be enough capacity to represent a successful solution. On the other hand, if the tree depth is too large, the computational complexity increases exponentially as does the corresponding CPU time. It is necessary to analyse the problem and set an appropriate maximum tree depth.

4. *For image classification using raw pixels, classifiers with loops for images represented by a two dimensional array are more economical in using the pixels, while for images represented by a one dimensional array classifiers use many more pixels, thus, use more information from the images.*

It is easier to find a classifier that can solve the problem and use many pixels in a one dimensional representation than to find a solution that utilises few pixels. Classifiers on images in a two dimensional representation can focus on interesting areas only. Due to the smaller number of pixels used, classifiers evolved in a two dimensional representation use less information and do not generalize as well to unseen objects. For image classification tasks using pixel statistics or high level features, further exploration is needed on how loops can be used.

6.3 Comparison to Previous Work on Loops

At the beginning of the thesis, we noted that others have used iterations implicitly and set a hard limit for the total number of iterations. Our work has tried to improve GP by allowing loops to be explicitly used and naturally incorporated into the evolved programs as in other programming languages. We have proposed a number of loop formats and created a heuristic to set the maximum number of iterations for each loop, or set a range by *start* and *end* instead of a global maximum number. In this section we compare our loops with other work in which there is a significant focus on loops.

Koza 1992, Santa Fe Ant Problem

Implicit loops were used in the Santa Fe ant problem described on page 54 of this thesis. We have composed explicit looping format 1 to solve this problem and successfully moved iterations from the environment into programs. In our implementation, the evolved solutions need only be called once instead of multiple times as in the original approach. Our looping format successfully captured the repetitive patterns and decreased the overall evolution time to get a successful solution.

Koza 1992, 9-Block Stack Problem

Explicit loops were used in the 9-block stack problem described on page 55 of this thesis. Our loop format 1 is similar in structure to this format. However, there are two major differences between Koza's loop format and our loop format 1. Firstly, Koza's loop format is highly customised. The DU and DO-ON-GOALS functions contain considerable domain information. To apply the same format to a different problem requires a major redefinition of the associated terminals and functions. Our loop format 1 only requires a small degree of customisation specifying the value of *max-iterations*. Secondly, the DU and DO-ON-GOALS conditions are not easy to evolve, thus, infinite loops frequently arise without an external restriction on the maximum number of iterations. We have called these kinds of loops *unbounded* loops (see page 8) and have stated at the beginning of the thesis that the study of unbounded loops is not our focus. Our looping format 1 is a kind of *bounded* loop (see page 8). Our loop format 1 avoids the infinite iterations problem by specifying the number of iterations explicitly, based on domain parameters.

Kinnear 1993, Sorting Problem

Explicit loops were used in the sorting problem described on page 58 of this thesis. We composed loop format 2 to solve a limited version of this problem. The major difference between Kinnear's approach and ours is in the method of restricting *max-iterations*. In his approach, the total number of iterations is restricted as well as the maximum number of iterations for each loop. We used the length of the array to restrict the possible values of *start* and *end*. Because we only tried to sort an array of limited length, setting the maximum number for *start* and *end* based on the array length was possible. Kinnear wanted to evolve a generalized sorting algorithm which is a much harder problem and is not our goal.

Finkel 2003, The Integer Factoring Problem

Explicit loops were used in the integer factoring problem described on page 60 of this thesis. As in Koza's 9-block stack problem, there are two major differences between Finkel's loops and ours. Firstly, Finkel used a do-while loop to solve this problem and this do-while loop is a kind of unbounded loop. To avoid infinite loops, he set a hard-coded maximum number of iterations. Because the condition was not easily satisfied, every loop in his approach reached the maximum limit and evaluation was very expensive, as he stated in his paper. Our loop format is a type of bounded loop and infinite loops are not possible. We have restricted the number of iterations

for each loop based on domain parameters and this decreased the overall evolutionary time to get a successful solution. Secondly, the do-while loop requires a condition as argument. The condition needs to be highly customised and is hard to evolve. Our loop only requires a small degree of customisation.

Chen and Zhang 2005, The Factorial and The Modified Ant Problem

Explicit loops were used by Chen and Zhang as described on page 60 of this thesis. They have used two loop formats - WhileLoop1 and WhileLoop2. Their WhileLoop1 is not really a "while" loop and is the same as our loop format 2 which was originally published in [43] in 2004. The WhileLoop1 function takes three arguments (*start*, *end* and *body*) and is a bounded loop. Their WhileLoop2 requires a condition as one of the arguments and is an unbounded loop. There are no major differences between this WhileLoop2 and the DU loop composed by Koza in 1992 or the do-while loop by Finkel in 2003. The condition of this WhileLoop2 was set to be *foodAhead* only and this is highly customised to the ant problem. The standard Santa Fe ant problem has gaps in the food trail. They modified the standard food trail and made the problem easier by placing the food continuously without gaps, while using our loop approach, it was possible to solve the original Santa Fe ant problem.

Koza 1999, Computing Average in A Vector Problem, Automatically Defined Functions and Automatically Defined Loops

Automatically defined functions (ADFs) and automatically defined loops (ADLs) were described on page 56 of this thesis. Koza has demonstrated their usage in a computing average problem and a number of other problems. Langdon has used them for his list structure problem. Our looping constructs are different to ADLs. Ours require no changes to the interpretation of a program tree while ADFs and ADLs requires different types of branches for the functions, loops and the main program. In ADFs and ADLs, the genetic operations need to take the different types of branches into consideration, so that crossover and mutation can correctly handle different situations to avoid nesting, and ensure that the main program can correctly reference the functions and loops. These operations are costly and increase search time.

Overall, the work described above, except ADLs, lacks a comparison with solving the problems without loops. The focus is on solving the problem at hand and it was taken for granted that loops were essential. Except for Koza, Chen and Zhang, they did not attempt to apply the same loop format to other problems. Our work provides comparisons for solving different

problems with and without loops and clearly demonstrates that iterations deliver major benefits to solve a number of problems.

6.4 Future Work

The investigation showed several directions which can be pursued further.

1. *Extend loop format 2 to have an increment value.*

In our loop format 2 (see Section 3.3.2), we do not have an explicit increment and we used 1 as the default value of the increment for the index in our experiments. Some problems need to use different values of the increment. For example, for a program with loops to sum up values of even numbered elements in an array, the value of the increment needs to be 2 instead of 1. Extending our loop format to include an increment terminal may improve our loop suitability to these problems.

2. *Extend the sorting problem to arrays of arbitrary size and extend the visit every square problem to grids of arbitrary size.*

It is known that a program without loops cannot sort an array of arbitrary size and cannot visit every square of a grid with arbitrary size. Extending the research on the sorting problem (see Section 3.6) and the visit every square problem (see Section 3.8) may help to determine whether such general solutions incorporating loops can be evolved. A number of the solutions for the 4*times*4 visit-every-square problem would have been solutions for any n if the 4 was replaced by n in the text of the program, so there is some reason for optimism here. If such solutions can be evolved, the question arises as to whether there are some procedures that are common to both problems, which one can follow for producing such generalised solutions in other problem domains.

3. *Extend the object classification problem to larger images and examine the effects of increasing image size.*

GP with loops has demonstrated its efficiency in terms of using fewer evaluations to get a solution than GP without loops in the object classification problem in this work (see Section 4.3). However, the images used were small in size and they are represented by a 16×16 grid. It will be useful to know what the effects are when we extend this problem to larger images. Will we get similiar or increased performance gain in larger images?

4. *Extend the object classification problem to classify grey level, real world objects instead of*

binary artificial objects.

The object classification problem is artificial (see Section 4.3). Furthermore, the problem was simplified by choosing binary pixel values to form the images. In real world images, pixel values are not binary and there are also many repeated patterns. An example of a repeating pattern in a real world image could be bricks in a wall. Intuitively, one may expect that repeated patterns in these images image may provide opportunities for the use of loops, because loops are useful functions for handling repetition efficiently.

5. *Extend the use of loops to other problems involving images.*

A characteristic of real world images is that pixel values tend to change gradually in a localised area. For example, an image of the ocean in a localised region would have variations of blue in that region. In a program that is building a model of the ocean, loops could be useful because they could change pixel values gradually. Changing pixel values in this manner would mirror the gradual continuity of pixel values in a localised region of a real world image.

6. *Explore GP with different genetic parameters for the tested problems.*

We have conducted a brief parameter sensitivity analysis (see Section 3.10). However the analysis has only been conducted for the modified ant problem. Even though the results convinced us that loops were the key factor that decreased the number of evaluations to get successful solutions, it would be more rigorous to extend these experiments to every problem in this thesis.

7. *Develop some mechanisms to allow an explicit index to be used in a loop.*

The loop index in this work is implicit, in that it does not appear in the loop body. Having an explicit index in the loop body, as those in programs written by a human, is desirable because the evolved programs with loops would be closer to many programming paradigms that people are familiar with today, such as loops in C and Java (see page 5). Many problems can be formulated by using loop indices implicitly; however, some problems will be hard to solve or need highly customised loop functions with the implicit use of indices. For example, to solve a matrix multiplication problem, multiple indices are needed to refer to different elements in different matrices. For GP with loops to allow explicit indices, mechanisms need to be developed to ensure that genetic operations follow the rule that each loop will have at least one index and to identify which loop an index belongs to, so that indices can be updated correctly in evaluation.

8. *Find better ways to visualize the search space with and without loops.*

It is difficult to analyse the search space in GP without understanding the relationship

between parents and descendants. It may be helpful to develop a new method, or investigate other methods from existing fitness landscape techniques (see page 42), in order to visualise the changes in the evolutionary process when loops are present or absent. To this end, we have conducted some experiments by using the fitness distribution method in Chapter 5. However, this work is limited in scope and further analysis methods should be investigated.

9. *Find new methods to avoid infinite iteration while not restricting the search for possible good programs.*

Most current work avoids infinite iteration by setting a maximum number of iterations for the whole program. We have used a maximum number of iterations for each loop or a maximum number for the value of *start* and *end*, together with the maximum program depth, in order to restrict the number of iterations for a program. However, these methods have limitations. They can result in a value for the maximum number of iterations which is too large, or too small. This can restrict the search because, if the maximum number of iterations is too large, the program takes too long to evolve. Similarly, if the maximum number of iterations is too small, a solution can never be found. It would be useful to have a method in which the maximum number of iterations is not set by the user, but is discovered by the evolutionary process.

10. *Apply loops to many more problems.*

Although the object classification problem in this thesis has been solved without loops (see Chapter 4), GP with loops has demonstrated that solutions with higher success rate can be evolved with less computational cost (see Chapter 3, 4). This raises the question about the performance of loop-based GP approaches to problems which have not been solved by GP, as well as to problems that have been solved by GP, but without loops. It is expected that loop based GP approaches will yield tangible benefits, as suggested by the work presented in this thesis.

This thesis explored the use of explicit loops with restricted syntax and semantics in genetic programming for a range of classic experimental problems and some variations as well as a difficult object classification problem. It showed that explicit loops with these restrictions can be beneficial to the evolutionary process by finding the repeating patterns which make the programs fitter and lead to solutions. The looping structures developed in this work are not as general as the ones written by a human. The loop index is not explicit and indices of two or more loops cannot be nested naturally. A maximum number of iterations still needs to be

used to restrict each loop to avoid infinite iterations. Researchers and practitioners of genetic programming have generally avoided loops, because of the difficulties of evolving consistent programs and the infinite loop problem. However, this work has shown that there is no need to fear loops. Restricted loops can avoid the above problems and deliver benefits. We hope that our work will encourage further research into loops in genetic programming.

Bibliography

[1] Y. L. Abdel-Magid and M. A. Abido. Optimal Multiobjective Design of Robust Power System Stabilizers Using Genetic Algorithms. *IEEE Transactions on Power Systems*, 18(3):1125–1132, Aug. 2003.

[2] J. C. Adams, W. S. Brainerd, J. T. Martin, B. T. Smith, and J. L. Wagener. *Fortran-95 Handbook — Complete ANSI/ISO Reference*. Scientific and Engineering Computation. MIT Press, Cambridge, MA, USA, 1997.

[3] A. Agapitos and S. M. Lucas. Evolving efficient recursive sorting algorithms. In *Proceedings of the Congress on Evolutionary Computation (CEC-2006)*, pages 9227–9234, Vancouver, Canada, 6-21 July 2006. IEEE Press.

[4] A. Agapitos and S. M. Lucas. Learning recursive functions with object oriented genetic programming. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2006)*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10-12 Apr. 2006. Springer.

[5] A. V. Aho and N. J. Sloane. Some doubly exponential sequences. *Fib. Quart*, 11, 1973. http://www.research.att.com/∼njas/doc/doubly.html.

[6] A. Almal, W. P. Worzel, E. A. Wollesen, and C. D. MacLean. Content diversity in genetic programming and its correlation with fitness. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 12, pages 177–190. Springer, Ann Arbor, 12-14 May 2005.

[7] L. Altenberg. The evolution of evolvability in genetic programming. pages 47–74, Cambridge, MA, USA, 1994. MIT Press.

[8] L. Altenberg. The Schema Theorem and Price's Theorem. In L. D. Whitley and M. D. Vose, editors, *Proceedings of Foundations of Genetic Algorithms 3*, pages 23–49, Estes Park, CO, USA, June 1995. Morgan Kaufmann.

[9] S. Amari, N. Murata, K.-R. Müller, M. Finke, and H. H. Yang. Asymptotic statistical theory of overtraining and cross-validation. *IEEE Transactions on Neural Networks*, 8(5):985–996, Sept. 1997.

[10] P. J. Angeline. Adaptive and self-adaptive evolutionary computations. In M. Palaniswami and Y. Attikiouzel, editors, *Computational Intelligence: A Dynamic Systems Perspective*, pages 152–163. IEEE Press, 1995.

[11] P. J. Angeline. Two self-adaptive crossover operators for genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 5, pages 89–110. MIT Press, Cambridge, MA, USA, 1996.

[12] W. Ashlock. Using very small population sizes in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2006)*, pages 1023–1030, Vancouver, Canada, 16-21 July 2006. IEEE Press.

[13] Y. Azaria and M. Sipper. GP-gammon: Genetically programming backgammon players. *Genetic Programming and Evolvable Machines*, 6(3):283–300, Sept. 2005.

[14] T. Bäck. Self-adaptation in genetic algorithms. In F. J. Varela and P. Bourgine, editors, *Proceedings of the 1st European Conference on Artificial Life*, pages 227–235, Cambridge, MA, USA, 1992. MIT Press.

[15] T. Bäck and H. P. Schwefel. Evolutionary computation: An overview. In *Proceedings of the Third IEEE Conference on Evolutionary Computation*, pages 20–29. IEEE Press, Piscataway, NJ, USA, 1996.

[16] W. Banzhaf. Genetic programming for pedestrians. In *Proceedings of the 5th International Conference on Genetic Algorithms*, page 628, San Francisco, CA, USA, 1993. Morgan Kaufmann.

[17] W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*. Morgan Kaufmann, 1998.

[18] J. Barnes. *Programming in Ada95*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[19] P. Bentley. An introduction to evolutionary design by computers. In P. J. Bentley, editor, *Evolutionary Design by Computers*, pages 1–73. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[20] Y. Bernstein, X. Li, V. Ciesielski, and A. Song. Multiobjective parsimony enforcement for superior generalisation performance. In *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 83–89, Portland, Oregon, USA, 20-23 June 2004. IEEE Press.

[21] S. Bleuler, M. Brack, L. Thiele, and E. Zitzler. Multiobjective genetic programming: Reducing bloat using SPEA2. In *Proceedings of the Congress on Evolutionary Computation CEC-2001*, pages 536–543, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[22] T. Blickle and L. Thiele. Genetic programming and redundancy. In J. Hopf, editor, *Genetic Algorithms within the Framework of Evolutionary Computation (Workshop at KI-94, Saarbrücken)*, pages 33–38, Im Stadtwald, Building 44, D-66123 Saarbrücken, Germany, 1994. Max-Planck-Institut für Informatik.

[23] T. Blickle and L. Thiele. A comparison of selection schemes used in genetic algorithms. TIK-Report 11, TIK Institut fur Technische Informatik und Kommunikationsnetze, Computer Engineering and Networks Laboratory, ETH, Swiss Federal Institute of Technology, Gloriastrasse 35, 8092 Zurich, Switzerland, Dec. 1995.

[24] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evolutionary Computation*, 4(4):361–394, 1997.

[25] W. Bohm and A. Geyer-Schulz. Exact uniform initialization for genetic programming. In R. K. Belew and M. Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, University of San Diego, CA, USA, 3-5 Aug. 1996. Morgan Kaufmann.

[26] Y. Borenstein and R. Poli. No free lunch, Kolmogorov complexity and the information landscape. In *Proceedings of the Congress on Evolutionary Computation (CEC-2005)*, Edinburgh, Scotland, UK, 2-5 Sept. 2005. IEEE Press.

[27] M. Brameier. *On linear genetic programming.* Dissertation, Fachbereich 4; Universität Dortmund, 13 Feb. 2004.

[28] M. Brameier and W. Banzhaf. *Linear Genetic Programming.* Number 1 in Genetic and Evolutionary Computation. Springer, 2006.

[29] J. Branke, M. Cutaia, and H. Dold. Reducing genetic drift in steady state evolutionary algorithms. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 1, pages 68–74, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[30] S. Brave. Evolving recursive programs for tree search. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 203–219. MIT Press, Cambridge, MA, USA, 1996.

[31] R. E. Bryant and D. R. O'Hallaron. *Computer Systems: A Programmer's Perspective.* Prentice Hall, 2003.

[32] E. Burke, S. Gustafson, and G. Kendall. Ramped half-and-half initialisation bias in GP. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U. M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, volume 2724 of *LNCS*, pages 1800–1801, Chicago, IL, USA, 12-16 July 2003. Springer-Verlag.

[33] E. K. Burke, S. Gustafson, and G. Kendall. Diversity in genetic programming: An analysis of measures and correlation with fitness. *IEEE Transactions on Evolutionary Computation*, 8(1):47–62, 2004.

[34] J. Busch, J. Ziegler, W. Banzhaf, A. Ross, D. Sawitzki, and C. Aue. Automatic generation of control programs for walking robots using genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2002)*, volume 2278 of *LNCS*, pages 258–267, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[35] M. V. Butz, K. Sastry, and D. E. Goldberg. Strong, stable, and reliable fitness pressure in XCS due to tournament selection. *Genetic Programming and Evolvable Machines*, 6(1):53–77, Mar. 2005.

[36] K. Chellapilla. Evolving computer programs without subtree crossover. *IEEE Transactions on Evolutionary Computation*, 1(3):209–216, Sept. 1997.

[37] G. Chen and M. Zhang. Evolving while-loop structures in genetic programming for factorial and ant problems. In S. Zhang and R. Jarvis, editors, *Proceedings of the Advances in Artificial Intelligence, 18th Australian Joint Conference on Artificial Intelligence, Proceedings*, volume 3809 of *Lecture Notes in Computer Science*, pages 1079–1085, Sydney, Australia, 5-9 Dec. 2005. Springer.

[38] W. S. Chung and R. A. Perez. The schema theorem considered insufficient. In *Proceedings of the Sixth IEEE International Conference on Tools with Artificial Intelligence*, pages 748–751, New Orleans, USA, 1994. IEEE Press.

[39] V. Ciesielski. The RMIT-GP genetic programming system. Available from http://www.cs.rmit.edu.au/∼vc.

[40] V. Ciesielski, A. Innes, S. John, and J. Mamutil. Understanding evolved genetic programs for a real world object detection problem. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 351–360, Lausanne, Switzerland, 30 Mar.–1 Apr. 2005. Springer.

[41] V. Ciesielski and X. Li. Pyramid search: Finding solutions for deceptive problems quickly in genetic programming. In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the Congress on Evolutionary Computation (CEC2003)*, pages 936–943, Canberra, Australia, 8-12 Dec. 2003. IEEE Press.

[42] V. Ciesielski and X. Li. Analysis of genetic programming runs. In R. I. Mckay and S. B. Cho, editors, *Proceedings of The Asian-Pacific Workshop on Genetic Programming (ASPGP-2004)*, Cairns, Australia, 6-7 Dec. 2004.

[43] V. Ciesielski and X. Li. Experiments with explicit for-loops in genetic programming. In D. B. Fogel and X. Yao, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 19–24, Portland, Oregon, USA, 20-23 June 2004. IEEE Press.

[44] V. Ciesielski and D. Mawhinney. Prevention of early convergence in genetic programming by replacement of similar programs. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2002)*, pages 67–72. IEEE Press, 2002.

[45] V. Ciesielski, D. Mawhinney, and P. Wilson. Genetic programming for robot soccer. In *Proceedings of the RoboCup 2001 International Symposium*, pages 319–324, Seattle, USA, July 2002. Springer.

[46] D. W. Corne and J. D. Knowles. No Free Lunch and Free Leftovers Theorems for Multiobjective Optimisation Problems. In C. M. Fonseca, P. J. Fleming, E. Zitzler, K. Deb, and L. Thiele, editors, *Proceedings of the Evolutionary Multi-Criterion Optimization. Second International Conference (EMO-2003)*, pages 327–341, Faro, Portugal, Apr. 2003. Springer.

[47] D. Costelloe and C. Ryan. Genetic programming for subjective fitness function identification. In M. Keijzer, U. M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2004)*, volume 3003 of *LNCS*, pages 259–268, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[48] N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In J. J. Grefenstette, editor, *Proceedings of an International Conference on Genetic Algo-*

rithms and the Applications, pages 183–187, Carnegie-Mellon University, Pittsburgh, PA, USA, 24-26 July 1985.

[49] E. F. Crane and N. F. McPhee. The effects of size and depth limits on tree based genetic programming. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*. Kluwer, Ann Arbor, 12-14 May 2005.

[50] J. M. Daida. Limits to expression in genetic programming: Lattice-aggregate modeling. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2002)*, pages 273–278. IEEE Press, 2002.

[51] J. M. Daida, T. F. Bersano-Begey, S. J. Ross, and J. F. Vesecky. Evolving feature-extraction algorithms: Adapting genetic programming for image analysis in geoscience and remote sensing. In *Proceedings of the 1996 International Geoscience and Remote Sensing Symposium*, pages 2077–2079. IEEE Press, 1996.

[52] J. Davis. Single populations v. co-evolution. In J. R. Koza, editor, *Artificial Life at Stanford 1994*, pages 20–27. Stanford Bookstore, Stanford, CA, USA, June 1994.

[53] I. De Falco, A. Della Cioppa, and E. Tarantino. Discovering interesting classification rules with genetic programming. *Applied Soft Computing*, 1(4):257–269, May 2001.

[54] E. D. de Jong and J. B. Pollack. Multi-objective methods for tree size control. *Genetic Programming and Evolvable Machines*, 4(3):211–233, Sept. 2003.

[55] E. D. de Jong, R. A. Watson, and J. B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 11–18, San Francisco, CA, USA, 7-11 July 2001. Morgan Kaufmann.

[56] K. Deb and D. E. Goldberg. An investigation of niche and species formation in genetic function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 42–50, San Mateo, CA, USA, 1989. Morgan Kaufman.

[57] M. Defoin Platel, M. Clergue, and P. Collard. Maximum homologous crossover for linear genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 194–203, Essex, UK, 14-16 Apr. 2003. Springer-Verlag.

[58] H. M. Deitel and P. J. Deitel. *Java How to Program*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[59] I. Dempsey. Constant generation for the financial domain using grammatical evolution. In F. Rothlauf, M. Blowers, J. Branke, S. Cagnoni, I. I. Garibay, O. Garibay, J. Grahl, G. Hornby, E. D. de Jong, T. Kovacs, S. Kumar, C. F. Lima, X. Llorà, F. Lobo, L. D. Merkle, J. Miller, J. H. Moore, M. O'Neill, M. Pelikan, T. P. Riopka, M. D. Ritchie, K. Sastry, S. L. Smith, H. Stringer, K. Takadama, M. Toussaint, S. C. Upton, and A. H. Wright, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, pages 350–353, Washington, D.C., USA, 25-29 June 2005. ACM Press.

[60] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, April 1997.

[61] D. J. Earl and M. W. Deem. Evolvability is a selectable trait. *Proceedings of the National Academy of Science*, 101(32):11531–11536, 2004.

[62] R. C. Eberhart and X. Hu. Human tremor analysis using particle swarm optimization. In *Proceedings of the Congress on Evolutionary Computation (CEC-1999)*, pages 1927–1930, Piscataway, NJ, USA, 1999. IEEE Press.

[63] M. Ebner. Evolution of a control architecture for a mobile robot. In M. Sipper, D. Mange, and A. Perez-Uribe, editors, *Proceedings of the Second International Conference on Evolvable Systems: From Biology to Hardware (ICES-1998)*, volume 1478 of *LNCS*, pages 303–310, Lausanne, Switzerland, 23-25 Sept. 1998. Springer Verlag.

[64] J. Eggermont. Evolving fuzzy decision trees with genetic programming and clustering. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2002)*, volume 2278 of *LNCS*, pages 71–82, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[65] J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. PhD thesis, Institute for Programming research and Algorithmics, Leiden Institute of Advanced Computer Science, Faculty of Mathematics & Natural Sciences, Leiden University, The Netherlands, 14 Sept. 2005.

[66] J. Eggermont, A. E. Eiben, and J. I. van Hemert. A comparison of genetic programming variants for data classification. In D. J. Hand, J. N. Kok, and M. R. Berthold, editors, *Advances in Intelligent Data Analysis, Third International Symposium, IDA-99*, volume 1642 of *LNCS*, pages 281–290, Amsterdam, The Netherlands, 9-11 Aug. 1999. Springer-Verlag.

[67] S. E. Eklund. Time series forecasting using massively parallel genetic programming. In *Proceedings of Parallel and Distributed Processing International Symposium*, pages 143–147, Nice, France, 22-26 Apr. 2003. IEEE Press.

[68] L. J. Eshelman and J. D. Schaffer. Crossover's niche. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 9–14, San Francisco, CA, USA, 1993. Morgan Kaufmann.

[69] C. Fernandes and A. Rosa. A study on non-random mating and varying population size in genetic algorithms using a royal road function. In *Proceedings of the Congress on Evolutionary Computation (CEC-2001)*, pages 60–66, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[70] J. E. Fieldsend and S. Singh. A Multi-Objective Algorithm based upon Particle Swarm Optimisation, an Efficient Data Structure and Turbulence. In *Proceedings of the 2002 U.K. Workshop on Computational Intelligence*, pages 37–44, Birmingham, UK, Sept. 2002.

[71] J. R. Finkel. Using genetic programming to evolve an algorithm for factoring numbers. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2003*, pages 52–60. Stanford Bookstore, Stanford, CA, USA, 4 Dec. 2003.

[72] D. B. Fogel. *Evolutionary Computation: Toward a New Philosophy of Machine Intelligence*. IEEE Press, NY, USA, 1995.

[73] D. B. Fogel. Applying Fogel and Burgin's 'Competetive Goal-Seeking through Evolutionary Programming' to coordination, trust, and bargaining games. In *Proceedings of the Congress on Evolutionary Computation (CEC-2000)*, pages 1210–1216, Piscataway, NJ, USA, 2000. IEEE Press.

[74] D. B. Fogel and A. Ghozeil. The schema theorem and the misallocation of trials in the presence of stochastic effects. *Lecture Notes in Computer Science*, 1447:313–322, 1998.

[75] L. J. Fogel. *On the organization of intellect.* PhD thesis, Los Angeles, CA, USA, 1964.

[76] G. Folino, C. Pizzuti, and G. Spezzano. Improving induction decision trees with parallel genetic programming. In *Proceedings 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, pages 181–187, Canary Islands, Spain, 9-11 Jan. 2002. IEEE.

[77] J. J. Freeman. A linear representation for GP using context free grammars. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 72–77, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[78] A. A. Freitas. Evolutionary computation. In W. Klosgen and J. Zytkow, editors, *Handbook of Data Mining and Knowledge Discovery*, chapter 32, pages 698–706. Oxford University Press, 2002.

[79] M. Fuchs. Large populations are not always the best choice in genetic programming. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1033–1038, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[80] S. Garcia Carbajal and F. G. Martinez. Evolutive introns: A non-costly method of using introns in GP. *Genetic Programming and Evolvable Machines*, 2(2):111–122, June 2001.

[81] C. Gathercole. *An Investigation of Supervised Learning in Genetic Programming.* PhD thesis, University of Edinburgh, 1998.

[82] C. Gathercole and P. Ross. Small populations over many generations can beat large populations over few generations in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 111–118, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[83] M. Glickman and K. Sycara. Reasons for premature convergence of self-adapting mutation rates. In *Proceedings of the Congress on Evolutionary Computation (CEC-2000)*, pages 62–69, Piscataway, NJ, USA, 2000. IEEE Press.

[84] A. Globus, J. Lawton, and T. Wipke. Graph crossover. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 761–771, San Francisco, CA, USA, 7-11 July 2001. Morgan Kaufmann.

[85] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[86] H. Gray. Genetic programming for classification of medical data. In J. R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, page 291, Stanford University, CA, USA, 13-16 July 1997. Stanford Bookstore.

[87] J. J. Greffenstette and J. E. Baker. How genetic algorithms work: A critical look at implicit parallelism. In J. D. Schaffer, editor, *Proceedings of the 3rd International Conference on Genetic Algorithms*, pages 20–27, George Mason University, June 1989. Morgan Kaufmann.

[88] B. Grosman and D. R. Lewin. Adaptive genetic programming for steady-state process modeling. *Computers & Chemical Engineering*, 28(12):2779–2790, 15 Nov. 2004.

[89] J. V. Hansen. Genetic programming experiments with standard and homologous crossover methods. *Genetic Programming and Evolvable Machines*, 4(1):53–66, Mar. 2003.

[90] R. Harper and A. Blair. A self-selecting crossover operator. In G. G. Yen, L. Wang, P. Bonissone, and S. M. Lucas, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2006)*, pages 5569–5576, Vancouver, 6-21 July 2006. IEEE Press.

[91] T. D. Haynes, D. A. Schoenefeld, and R. L. Wainwright. Type inheritance in strongly typed genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[92] R. B. Heckendorn, S. Rana, and D. L. Whitley. Test function generators as embedded landscapes. In W. Banzhaf and C. Reeves, editors, *Foundations of Genetic Algorithms 5*, pages 183–198. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[93] R. Hinterding, H. Gielewski, and T. C. Peachey. The nature of mutation in genetic algorithms. In L. J. Eshelman, editor, *ICGA*, pages 65–72. Morgan Kaufmann, 1995.

[94] N. X. Hoai, R. I. McKay, and D. Essam. Some experimental results with tree adjunct grammar guided genetic programming. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2002)*, volume 2278 of *LNCS*, pages 228–237, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[95] N. X. Hoai, R. I. McKay, D. Essam, and R. Chau. Solving the symbolic regression problem with tree-adjunct grammar guided genetic programming: The comparative results. In D. B. Fogel, M. A. E. Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC2002)*, pages 1326–1331. IEEE Press, 2002.

[96] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1992.

[97] W. Hordijk. A measure of landscapes. *Evolutionary Computation*, 4(4):335–360, 1997.

[98] H. Iba. Random tree generation for genetic programming. In H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature IV,*

Proceedings of the International Conference on Evolutionary Computation, volume 1141 of *LNCS*, pages 144–153, Berlin, Germany, 22-26 Sept. 1996. Springer Verlag.

[99] H. Iba and N. Nikolaev. Financial data prediction by means of genetic programming. In *Computing in Economics and Finance*, Universitat Pompeu Fabra, Barcelona, Spain, 6-8 July 2000.

[100] D. Jackson. Fitness evaluation avoidance in boolean GP problems. In D. Corne, Z. Michalewicz, M. Dorigo, G. Eiben, D. Fogel, C. Fonseca, G. Greenwood, T. K. Chen, G. Raidl, A. Zalzala, S. Lucas, B. Paechter, J. Willies, J. J. M. Guervos, E. Eberbach, B. McKay, A. Channon, A. Tiwari, L. G. Volkert, D. Ashlock, and M. Schoenauer, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2005)*, volume 3, pages 2530–2536, Edinburgh, UK, 2-5 Sept. 2005. IEEE Press.

[101] C. Z. Janikow and C. J. Mann. CGP visits the Santa Fe trail: effects of heuristics on GP. In *Proceedings of Genetic and Evolutionary Computation Conference (GECCO-2005)*, pages 1697–1704, New York, NY, USA, 2005. ACM Press.

[102] T. Jones. *Evolutionary Algorithms, Fitness Landscapes and Search.* PhD thesis, University of New Mexico, Albuquerque, NM, 1995.

[103] T. Jones. One operator, one landscape. In *Santa Fe Institute Technical Report 95-02-025*, Santa Fe Institute, 1399 Hyde Park Road, Santa Fe, NM, USA, 1995.

[104] K. A. D. Jong, M. A. Potter, and W. M. Spears. Using problem generators to explore the effects of epistasis. In T. Bäck, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms (ICGA-1997)*, San Francisco, CA, USA, 1997. Morgan Kaufmann.

[105] H. Juille and J. B. Pollack. Massively parallel genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 339–358. MIT Press, Cambridge, MA, USA, 1996.

[106] J. C. E. Kaiser, G. B. Lamont, L. D. Merkle, J. G. H. Gates, and R. Pachter. Polypeptide structure prediction: Real-value versus binary hybrid genetic algorithms. In *Proceedings of the 1997 ACM symposium on Applied computing*, pages 279–286, New York, NY, USA, 1997. ACM Press.

[107] M. Kantardzic. *Data Mining: Concepts, Models, Methods and Algorithms*. John Wiley & Sons, Inc., New York, NY, USA, 2002.

[108] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[109] W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2002)*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[110] S. A. Kauffman. Adaptation on rugged fitness landscapes. In e. D. Stein, editor, *Lectures in the Sciences of Complexity*, pages 527–618. Addison-Wesley, 1989.

[111] M. Keijzer. Efficiently representing populations in genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 13, pages 259–278. MIT Press, Cambridge, MA, USA, 1996.

[112] M. Keijzer. Alternatives in subtree caching for genetic programming. In M. Keijzer, U. M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2004)*, volume 3003 of *LNCS*, pages 328–337, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

[113] A. Kelley and I. Pohl. *A book on C*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1984.

[114] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of the IEEE International Conference on Neural Networks (ICNN-1995)*, volume 4, pages 1942–1947, Perth, Australia, Nov. 1995. IEEE Press.

[115] J. Kennedy and R. C. Eberhart. The particle swarm: Social adaptation in information-processing systems. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 379–387. McGraw-Hill, London, UK, 1999.

[116] T. M. Khoshgoftaar, Y. Liu, and N. Seliya. Genetic Programming-Based Decision Trees for Software Quality Classification. In *Proceedings of the Fifteenth International Conference on Tools with Artificial Intelligence (ICTAI-2003)*, pages 374–383, Los Alamitos, CA, USA, 3-5 Nov. 2003. IEEE Press.

[117] K. E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, CA, USA, 28 Jan. 1993. IEEE Press.

[118] K. E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms (ICGA-1993)*, pages 287–294, University of Illinois at Urbana-Champaign, 1993. Morgan Kaufmann.

[119] K. E. Kinnear, Jr. Fitness landscapes and difficulty in genetic programming. In *Proceedings of the 1994 IEEE World Conference on Computational Intelligence*, volume 1, pages 142–147, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[120] M. J. Kochenderfer. Evolving teleo-reactive programs for block stacking using indexicals through genetic programming. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2002*, pages 111–118. Stanford Bookstore, Stanford, CA, USA, June 2002.

[121] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of International Joint Conference on Artificial Intelligence (IJCAI-1995)*, pages 1137–1145. Morgan Kaufmann, 1995.

[122] M. Köppen and B. Nickolay. Design of image exploring agent using genetic programming. *Fuzzy Sets and Systems*, 2(103):303–315, 1999. Special Issue on Softcomputing.

[123] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, 1992.

[124] J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, 1994.

[125] J. R. Koza, F. H. Bennett III, D. Andre, and M. A. Keane. *Genetic Programming III; Darwinian invention and problem solving*. Morgan Kaufmann, 1999.

[126] J. R. Koza, F. H. Bennett III, J. L. Hutchings, S. L. Bade, M. A. Keane, and D. Andre. Evolving sorting networks using genetic programming and the rapidly reconfigurable Xilinx 6216 field-programmable gate array. In *Proceedings of the 31st Asilomar Conference on Signals, Systems, and Computers*. IEEE Press, 1997.

[127] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.

[128] K. Krawiec and B. Bhanu. Coevolution and linear genetic programming for visual learning. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, volume 2723 of *LNCS*, pages 332–343, Chicago, IL, USA, 12-16 July 2003. Springer-Verlag.

[129] R. A. Krohling, F. Hoffmann, and L. dos Santos Coelho. Co-evolutionary particle swarm optimization for Min-Max problems using Gaussian distribution. In *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 959–964, Portland, Oregon, USA, 20-23 June 2004. IEEE Press.

[130] T. Lai. Discovery of understandable math formulas using genetic programming. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2003*, pages 118–127. Stanford Bookstore, Stanford, CA, USA, 4 Dec. 2003.

[131] W. B. Langdon. Quick Intro to simple-gp.c. Internal Note IN/95/2, University College London, Gower Street, London, UK, 25 Apr. 1994.

[132] W. B. Langdon. Data structures and genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[133] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1092–1097, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[134] W. B. Langdon. Size fair tree crossovers. In E. Postma and M. Gyssen, editors, *Proceedings of the Eleventh Belgium/Netherlands Conference on Artificial Intelligence (BNAIC-1999)*, pages 255–256, Kasteel Vaeshartelt, Maastricht, Holland, 3-4 Nov. 1999.

[135] W. B. Langdon. Size fair and homologous tree genetic programming crossovers. *Genetic Programming and Evolvable Machines*, 1(1/2):95–119, Apr. 2000.

[136] W. B. Langdon. Convergence of program fitness landscapes. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U. M. O'Reilly, H. G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, volume 2724 of *LNCS*, pages 1702–1714, Chicago, IL, USA, 12-16 July 2003. Springer-Verlag.

[137] W. B. Langdon and B. F. Buxton. Genetic programming for mining DNA chip data from cancer patients. *Genetic Programming and Evolvable Machines*, 5(3):251–257, Sept. 2004.

[138] W. B. Langdon and R. Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer-Verlag, 23-27 June 1997.

[139] W. B. Langdon and R. Poli. Better trained ants for genetic programming. Technical Report CSRP-98-12, University of Birmingham, School of Computer Science, Apr. 1998.

[140] W. B. Langdon and R. Poli. Fitness causes bloat: Mutation. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 37–48, Paris, 14-15 July 1998. Springer-Verlag.

[141] W. B. Langdon and R. Poli. Why ants are hard. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo,

editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 193–201, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[142] W. B. Langdon and R. Poli. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[143] E. C. Laskari, K. E. Parsopoulos, and M. N. Vrahatis. Particle swarm optimization for integer programming. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2002)*, pages 1582–1587. IEEE Press, 2002.

[144] E. C. Laskari, K. E. Parsopoulos, and M. N. Vrahatis. Particle swarm optimization for minimax problems. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2002)*, pages 1576–1581. IEEE Press, 2002.

[145] J. Li, X. Li, and X. Yao. Cost-sensitive classification with genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC-2005)*, volume 3, pages 2114–2121, Edinburgh, Scotland, UK, 2-5 Sept. 2005. IEEE Press.

[146] X. Li. A non-dominated sorting particle swarm optimizer for multiobjective optimization. In E. Cantú-Paz, J. A. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. K. Standish, G. Kendall, S. W. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. A. Dowsland, N. Jonoska, and J. F. Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, volume 2723 of *Lecture Notes in Computer Science*, pages 37–48, Chicago, IL, USA, 12–16 July 2003. Springer.

[147] X. Li and V. Ciesielski. Using loops in genetic programming for a two class binary image classification problem. In G. I. Webb and X. Yu, editors, *Proceedings of the Australian Conference on Artificial Intelligence*, volume 3339 of *Lecture Notes in Computer Science*, pages 898–909. Springer, 2004.

[148] X. Li and V. Ciesielski. An analysis of explicit loops in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2005)*, pages 38–45, Edinburgh, Scotland, UK, 2-4 Sept. 2005. IEEE Press.

[149] J. D. Lohn, G. S. Hornby, and D. S. Linden. Rapid re-evolution of an X-band antenna for NASA's space technology 5 mission. In T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice III*. Kluwer, Ann Arbor, 12-14 May 2005.

[150] T. Loveard. Genetic programming with meta-search: Searching for a successful population within the classification domain. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 119–129, Essex, UK, 14-16 Apr. 2003. Springer-Verlag.

[151] T. Loveard and V. Ciesielski. Representing classification problems in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-2001)*, volume 2, pages 1070–1077, COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea, 27-30 May 2001. IEEE Press.

[152] T. Loveard and V. Ciesielski. Genetic programming for classification: An analysis of convergence behaviour. In B. McKay and J. Slaney, editors, *Proceedings of the Advances in Artificial Intelligence : 15th Australian Joint Conference on Artificial Intelligence*, volume 2557 of *Lecture Notes in Computer Science*, pages 309–320, Canberra, Australia, 2-6 Dec. 2002.

[153] P. B. Lubell-Doughtie. Using genetic programming to evolve a general purpose sorting network for comparable data sets. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2003*, pages 128–132. Stanford Bookstore, Stanford, CA, USA, 4 Dec. 2003.

[154] S. Luke. Code growth is not caused by introns. In D. Whitley, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 228–235, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[155] S. Luke. Two fast tree-creation algorithms for genetic programming. *IEEE Transactions on Evolutionary Computation*, 4(3):274–283, Sept. 2000.

[156] S. Luke. When short runs beat long runs. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke,

editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 74–80, San Francisco, CA, USA, 7-11 July 2001. Morgan Kaufmann.

[157] S. Luke, G. C. Balan, and L. Panait. Population implosion in genetic programming. In E. Cantú-Paz, J. A. Foster, K. Deb, D. Davis, R. Roy, U. M. O'Reilly, H. G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. A. Potter, A. C. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, volume 2724 of *LNCS*, pages 1729–1739, Chicago, IL, USA, 12-16 July 2003. Springer-Verlag.

[158] S. Luke, C. Hohn, J. Farris, G. Jackson, and J. Hendler. Co-evolving soccer softbot team coordination with genetic programming. In *Proceedings of the First International Workshop on RoboCup, at the International Joint Conference on Artificial Intelligence*, Nagoya, Japan, 1997.

[159] S. Luke and L. Panait. A survey and comparison of tree generation algorithms. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 81–88, San Francisco, CA, USA, 7-11 July 2001. Morgan Kaufmann.

[160] S. Luke and L. Panait. Fighting bloat with nonparametric parsimony pressure. In J. J. Merelo-Guervos, P. Adamidis, H.-G. Beyer, J.-L. Fernandez-Villacanas, and H.-P. Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VII*, number 2439 in Lecture Notes in Computer Science, LNCS, pages 411–421, Granada, Spain, 7-11 Sept. 2002. Springer-Verlag.

[161] S. Luke and L. Spector. A comparison of crossover and mutation in genetic programming. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 240–248, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[162] S. Luke and L. Spector. A revised comparison of crossover and mutation in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proceedings of the Third*

Annual Conference on Genetic Programming, pages 208–213, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[163] P. Machado, A. Dias, N. Duarte, and A. Cardoso. Giving colour to images. In A. Cardoso and G. Wiggins, editors, *Proceedings of AI and Creativity Symposium in Arts and Science*, Imperial College, United Kingdom, 2-5 Apr. 2002. Kluwer Academic.

[164] K. J. Mackin and E. Tazaki. Unsupervised training of Multiobjective Agent Communication using Genetic Programming. In *Proceedings of the Fourth International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technology*, volume 2, pages 738–741, Brighton, UK, 30 Aug.–1 Sept. 2000. IEEE Press.

[165] S. R. Maxwell III. Experiments with a coroutine model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 413–417, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[166] R. I. B. McKay. Fitness sharing in genetic programming. In D. Whitley, D. Goldberg, E. Cantu-Paz, L. Spector, I. Parmee, and H.-G. Beyer, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000)*, pages 435–442, Las Vegas, Nevada, USA, 10-12 July 2000. Morgan Kaufmann.

[167] N. F. McPhee and N. J. Hopper. Analysis of genetic diversity through population history. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1112–1120, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[168] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd ed.)*. Springer-Verlag, New York, NY, USA, 1994.

[169] Z. Michalewicz and M. Michalewicz. Evolutionary computation techniques and their applications. In *IEEE International Conference on Intelligent Processing Systems*, volume 1, pages 14–25, Beijing, China, 28-31 Oct. 1997. IEEE Press.

[170] J. F. Miller. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In W. Banzhaf, J. Daida, A. E. Eiben, M. H.

Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, volume 2, pages 1135–1142, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[171] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2000)*, volume 1802, pages 121–132, Edinburgh, Scotland, UK, 15-16 Apr. 2000. Springer-Verlag.

[172] M. Mitchell and C. E. Taylor. Evolutionary computation: An overview. *Annual Review of Ecology and Systematics*, 20:593–616, 1999.

[173] P. Monsieurs and E. Flerackers. Reducing bloat in genetic programming. In B. Reusch, editor, *Computational Intelligence : Theory and Applications*, volume 2206 of *LNCS*, pages 471–478, Dortmund, Germany, 1-3 Oct. 2001. Springer-Verlag.

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

[175] H. A. Montes and J. L. Wyatt. Cartesian genetic programming for image processing tasks. In *Neural Networks and Computational Intelligence*, pages 185–190. IASTED/ACTA Press, 2003.

[176] A. Moraglio and R. Poli. Topological interpretation of crossover. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, volume 3102 of *Lecture Notes in Computer Science*, pages 1377–1388, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[177] A. Moraglio and R. Poli. Geometric landscape of homologous crossover for syntactic trees. In *Proceedings of the Congress on Evolutionary Computation (CEC-2005)*, volume 1, pages 427–434, Edinburgh, Scotland, UK, 2-4 Sept. 2005. IEEE Press.

[178] E. M. Mugambi, A. Hunter, G. Oatley, and L. Kennedy. Polynomial-fuzzy decision tree structures for classifying medical data. *Knowledge-Based Systems*, 17(2-4):81–87, 2004.

[179] T. Nanduri. *Comparison of the Effectiveness of Decimation and Automatically Defined Functions*. Minor thesis, School of Computer Science and Information Technology, RMIT University, 2004.

[180] D. J. Newman, S. Hettich, C. L. Blake, and C. J. Merz. UCI repository of machine learning databases, http://www.ics.uci.edu/∼mlearn/mlrepository.html, University of California, Irvine, Department of Information and Computer Sciences, 1998.

[181] M. Nicolau and I. Dempsey. Introducing grammar based extensions for grammatical evolution. In G. G. Yen, L. Wang, P. Bonissone, and S. M. Lucas, editors, *Proceedings of the Congress on Evolutionary Computation (CEC2006)*, pages 2663–2670, Vancouver, 6-21 July 2006. IEEE Press.

[182] J. Niehaus and W. Banzhaf. Adaption of operator probabilities in genetic programming. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 325–336, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[183] P. Nordin. *Evolutionary Program Induction of Binary Machine Code and its Applications*. PhD thesis, der Universitat Dortmund am Fachereich Informatik, 1997.

[184] P. Nordin. AIMGP: A formal description. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Stanford Bookstore.

[185] M. O'Neill, A. Brabazon, and C. Adley. The automatic generation of programs for classification problems with grammatical swarm. In *Proceedings of the Congress on Evolutionary Computation (CEC-2004)*, pages 104–110, Portland, Oregon, USA, 20-23 June 2004. IEEE Press.

[186] M. O'Neill and C. Ryan. Under the hood of grammatical evolution. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-1999)*, Orlando, FL, USA, 13-17 July 1999. Morgan Kaufmann.

[187] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, 2001.

[188] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico. Crossover in grammatical evolution. *Genetic Programming and Evolvable Machines*, 4(1):67–93, Mar. 2003.

[189] U. M. O'Reilly. *An analysis of genetic programming.* PhD thesis, Carleton University, 1995.

[190] U. M. O'Reilly. Using a distance metric on genetic programs to understand genetic operators. In J. R. Koza, editor, *Proceedings of the 1997 Genetic Programming Conference*, pages 199–206, Stanford University, CA, USA, 13-16 July 1997. Stanford Bookstore.

[191] U. M. O'Reilly and F. Oppacher. An experimental perspective on genetic programming. In R. Manenner and B. Manderick, editors, *Proceedings of the 2nd International Conference on Parallel Problem Solving from Nature*, Holland, 1992.

[192] U. M. O'Reilly and F. Oppacher. The troubling aspects of a building block hypothesis for genetic programming. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 73–88, Estes Park, CO, USA, 31 July–2 Aug. 1994. Morgan Kaufmann. Published 1995.

[193] J. Page, R. Poli, and W. B. Langdon. Smooth uniform crossover with smooth point mutation in genetic programming: A preliminary study. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-1999)*, volume 1598 of *LNCS*, pages 39–49, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[194] A. D. Parkins and A. K. Nandi. Genetic programming techniques for hand written digit recognition. *Signal Processing*, 84(12):2345–2365, 2004.

[195] T. Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, FL, USA, 27-29 June 1994. IEEE Press.

[196] R. Poli. Schema theorems without expectations. In W. Banzhaf, J. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. Jakiela, and R. E. Smith, editors, *Proceedings of the Genetic*

and Evolutionary Computation Conference (GECCO-1999), volume 1, page 806. Morgan Kaufmann, 13-17 July 1999.

[197] R. Poli. On fitness proportionate selection and the schema theorem in the presence of stochastic effects. Technical Report CSRP-00-2, University of Birmingham, School of Computer Science, Feb. 2000.

[198] R. Poli. Tournament selection, iterated coupon-collection problem, and backward-chaining evolutionary algorithms. In *Proceedings of the Foundations of Genetic Algorithms Workshop*, LNCS, Aizu, Japan, 4 Jan. 2005. Springer. Forthcoming.

[199] R. Poli and S. Cagnoni. Genetic programming with user-driven selection: Experiments on the evolution of algorithms for image enhancement. In J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba, and R. L. Riolo, editors, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 269–277, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[200] R. Poli and W. B. Langdon. On the search properties of different crossover operators in genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 293–301, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[201] R. Poli and W. B. Langdon. A review of theoretical and experimental results on schemata in genetic programming. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-1998)*, volume 1391 of *LNCS*, pages 1–15, Paris, France, 14-15 Apr. 1998. Springer-Verlag.

[202] R. Poli and W. B. Langdon. Schema theory for genetic programming with one-point crossover and point mutation. *Evolutionary Computation*, 6(3):231–252, 1998.

[203] R. Poli and N. F. McPhee. General schema theory for genetic programming with subtree-swapping crossover: Part II. *Evolutionary Computation*, 11(2):169–206, June 2003.

[204] G. Poncin. Evolving 3D models of trees using genetic programming. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2003*, pages 153–162. Stanford Bookstore, Stanford, CA, USA, 4 Dec. 2003.

[205] K. V. Price. An introduction to differential evolution. In D. Corne, M. Dorigo, and F. Glover, editors, *New Ideas in Optimization*, pages 79–108. McGraw-Hill, London, UK, 1999.

[206] W. F. Punch, D. Zongker, and E. D. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 15, pages 299–316. MIT Press, Cambridge, MA, USA, 1996.

[207] C. A. P. Reyes and M. Sipper. Evolutionary computation in medicine: An overview. *Artificial Intelligence in Medicine*, 19(1):1–23, 2000.

[208] D. Rochat, M. Tomassini, and L. Vanneschi. Dynamic size populations in distributed genetic programming. In M. Keijzer, A. Tettamanzi, P. Collet, J. I. van Hemert, and M. Tomassini, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2005)*, volume 3447 of *Lecture Notes in Computer Science*, pages 50–61, Lausanne, Switzerland, 30 Mar. 2005. Springer.

[209] J. P. Rosca. Entropy-driven adaptive representation. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 23–32, Tahoe City, CA, USA, Sept. 1995.

[210] M. Rosskopf, U. Feldkamp, and W. Banzhaf. Classification of leukemia classes by GP-based DNA-chip analysis. In A. M. Barry, editor, *Proceedings of the Bird of a Feather Workshops, Genetic and Evolutionary Computation Conference (GECCO-2003)*, pages 81–82, Chigaco, 11 July 2003. AAAI Press.

[211] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In W. Banzhaf, R. Poli, M. Schoenauer, and T. C. Fogarty, editors, *Proceedings of the First European Workshop on Genetic Programming*, volume 1391, pages 83–95, Paris, France, 14-15 Apr. 1998. Springer-Verlag.

[212] C. Ryan and M. O'Neill. Grammatical evolution: A steady state approach. In J. R. Koza, editor, *Late Breaking Papers at the Genetic Programming 1998 Conference*, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Stanford Bookstore.

[213] M. Santini and A. Tettamanzi. Genetic programming for financial time series prediction. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2001)*, volume 2038 of *LNCS*, pages 361–370, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[214] K. Sastry, U. M. O'Reilly, and D. E. Goldberg. Population sizing for genetic programming based on decision making. In U. M. O'Reilly, T. Yu, R. L. Riolo, and B. Worzel, editors, *Genetic Programming Theory and Practice II*, chapter 4. Kluwer, Ann Arbor, 13-15 May 2004.

[215] H. P. Schwefel. *Evolutionsstrategie und numerische Optimierung*. PhD thesis, Technische Universitat Berlin, Berlin, Germany, May 1975.

[216] B. Skellett, B. Cairns, N. Geard, B. Tonkes, and J. Wiles. Maximally rugged NK landscapes contain the highest peaks. In *Proceedings of the Genetic and Evolutionary Computation (GECCO-2005)*, pages 579–584, New York, NY, USA, 2005. ACM Press.

[217] V. Slavov and N. I. Nikolaev. Inductive genetic programming and superposition of fitness landscapes. In T. Back, editor, *Proceedings of the Seventh International Conference on Genetic Algorithms*, pages 97–104, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[218] J. Smith and T. C. Fogarty. Self adaptation of mutation rates in a steady state genetic algorithm. In *Proceedings of the International Conference on Evolutionary Computation*, pages 318–323. IEEE Press, 20-22 May 1996.

[219] P. W. H. Smith. Controlling code growth in genetic programming. In R. John and R. Birkenhead, editors, *Advances in Soft Computing*, pages 166–171, De Montfort University, Leicester, UK, 2000. Physica-Verlag.

[220] A. Song, T. Loveard, and V. Ciesielski. Towards genetic programming for texture classification. In M. Stumptner, D. Corbett, and M. Brooks, editors, *Proceedings of the 14th*

International Joint Conference on Artificial Intelligence AI 2001: Advances in Artificial Intelligence, volume 2256 of *Lecture Notes in Computer Science*, pages 461–472, Adelaide, Australia, 10-14 Dec. 2001. Springer-Verlag.

[221] T. Soule and J. A. Foster. Removal bias: a new cause of code growth in tree based evolutionary programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-1998)*, pages 781–186, Anchorage, Alaska, USA, 5-9 May 1998. IEEE Press.

[222] T. Soule, J. A. Foster, and J. Dickinson. Code growth in genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 215–223, Stanford University, CA, USA, 28-31 July 1996. MIT Press.

[223] W. M. Spears. Crossover or mutation? In L. D. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 221–237. Morgan Kaufmann, San Mateo, CA, USA, 1993.

[224] W. M. Spears. Adapting crossover in evolutionary algorithms. In J. R. McDonnell, R. G. Reynolds, and D. B. Fogel, editors, *Proceedings of the Fourth Annual Conference on Evolutionary Programming*, pages 367–384, San Diego, CA, USA, 1-3 Mar. 1995. MIT Press.

[225] W. M. Spears, K. A. D. Jong, T. Bäck, D. B. Fogel, and H. de Garis. An overview of evolutionary computation. In P. B. Brazdil, editor, *Proceedings of the European Conference on Machine Learning (ECML-1993)*, volume 667, pages 442–459, Vienna, Austria, 1993. Springer Verlag.

[226] L. Spector. Genetic programming and ai planning systems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1329–1334, Menlo Park, CA, USA, 1994. AAAI Press.

[227] M. Srinivas and L. M. Patnaik. Adaptive probabilities of crossover and mutation in genetic algorithms. *IEEE Transactions on Systems, Man, and Cybernetics*, 24(4):656–667, 1994.

[228] C. R. Stephens and H. Waelbroeck. Effective degrees of freedom in genetic algorithms and the block hypothesis. In T. Bäck, editor, *Proceedings of the 7th International Conference on Genetic Algorithms*, pages 34–40, San Francisco, CA, USA, 19-23 July 1997. Morgan Kaufmann.

[229] C. R. Stephens, H. Waelbroeck, and R. Aguirre. Schemata as building blocks: Does size matter? In W. Banzhaf and C. Reeves, editors, *Foundations of Genetic Algorithms 5*, pages 117–133. Morgan Kaufmann, San Francisco, CA, USA, 1999.

[230] S. Stepney. Book review: Evolutionary Electronics: Automatic design of electronic circuits and systems by genetic algorithms. *Genetic Programming and Evolvable Machines*, 5(4):395–396, Dec. 2004.

[231] W. A. Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, USA, 1994.

[232] K. C. Tan, A. Tay, T. H. Lee, and C. M. Heng. Mining multiple comprehensible classification rules using genetic programming. In D. B. Fogel, M. A. El-Sharkawi, X. Yao, G. Greenwood, H. Iba, P. Marrow, and M. Shackleton, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2002)*, pages 1302–1307. IEEE Press, 2002.

[233] I. Tanev. Evolutionary algorithms for intelligent software design. In *Proceedings of the Sino-Japanese Symposium of Kansei and Artificial Life*, pages 60–66, Beijing, China, 2004.

[234] E. Tchernev. Stack-correct crossover methods in genetic programming. In E. Cantú-Paz, editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 443–449, New York, NY, USA, July 2002. AAAI Press.

[235] A. Teller. Evolving programmers: The co-evolution of intelligent recombination operators. In P. Angeline and K. Kinnear, editors, *Advances in Genetic Programming II*, pages 45–68. MIT Press, Cambridge, MA, USA, 1996.

[236] A. Teller and M. Veloso. Algorithm evolution for face recognition: What makes a picture difficult. In *International Conference on Evolutionary Computation*, pages 608–613, Perth, Australia, 1–3 Dec. 1995. IEEE Press.

[237] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[238] M. Terrio and M. I. Heywood. Directing crossover for reduction of bloat in GP. In W. Kinsner, A. Seback, and K. Ferens, editors, *IEEE CCECE 2003: IEEE Canadian Conference on Electrical and Computer Engineering*, pages 1111–1115. IEEE Press, 12-15 May 2002.

[239] I. V. Tetko, D. J. Livingstone, and A. I. Luik. Neural network studies, 1. comparison of overfitting and overtraining. *Journal of Chemical Information and Computer Sciences*, 35(5):826–833, 1995.

[240] J. Townsend. Search in grid world using genetic programming and genetic algorithms. In J. R. Koza, editor, *Genetic Algorithms and Genetic Programming at Stanford 2000*, pages 407–414. Stanford Bookstore, Stanford, CA, USA, June 2000.

[241] K. Uesaka and M. Kawamata. Synthesis of low-sensitivity second-order digital filters using genetic programming with automatically defined functions. *IEEE Signal Processing Letters*, 7(4):83–85, Apr. 2000.

[242] T. Van Belle and D. H. Ackley. Uniform subtree mutation. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2002)*, volume 2278 of *LNCS*, pages 152–161, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[243] L. Vanneschi, M. Clergue, P. Collard, M. Tomassini, and S. Vérel. Fitness clouds and problem hardness in genetic programming. In K. Deb, R. Poli, W. Banzhaf, H.-G. Beyer, E. Burke, P. Darwen, D. Dasgupta, D. Floreano, J. Foster, M. Harman, O. Holland, P. L. Lanzi, L. Spector, A. Tettamanzi, D. Thierens, and A. Tyrrell, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, volume 3103 of *Lecture Notes in Computer Science*, pages 690–701, Seattle, WA, USA, 26-30 June 2004. Springer-Verlag.

[244] L. Vanneschi, M. Tomassini, P. Collard, and M. Clergue. Fitness distance correlation in structural mutation genetic programming. In C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, editors, *Proceedings of the European Conference on Genetic Programming (EuroGP-2003)*, volume 2610 of *LNCS*, pages 455–464, Essex, UK, 14-16 Apr. 2003. Springer-Verlag.

[245] V. K. Vassilev, D. Job, and J. F. Miller. Towards the automatic design of more efficient digital circuits. In J. Lohn, A. Stoica, and D. Keymeulen, editors, *Proceedings of the Second NASA and DoD workshop on Evolvable Hardware*, pages 151–160, Palo Alto, CA, USA, 13-15 July 2000. IEEE Press.

[246] S. Verel, P. Collard, and M. Clergue. Where are bottleneck in NK fitness landscapes? In R. Sarker, R. Reynolds, H. Abbass, K. C. Tan, B. McKay, D. Essam, and T. Gedeon, editors, *Proceedings of the Congress on Evolutionary Computation (CEC-2003)*, pages 273–280, Canberra, Australia, 8-12 Dec. 2003. IEEE Press.

[247] E. D. Weinberger. Fourier and taylor series on fitness landscapes. In *Biological Cybernetics*, volume 65:5, pages 321–330. Springer, Sept. 1991.

[248] P. A. Whigham. Grammatically-based genetic programming. In J. P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, CA, USA, 9 July 1995.

[249] P. A. Whigham. A schema theorem for context-free grammars. In *Proceedings of the Congress on Evolutionary Computation (CEC-1995)*, volume 1, pages 178–181, Perth, Australia, 29 Nov.–1 Dec. 1995. IEEE Press.

[250] P. A. Whigham. *Grammatical Bias for Evolutionary Learning*. PhD thesis, School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, Canberra, Australia, 14 Oct. 1996.

[251] P. A. Whigham and G. Dick. GP and bloat: Absorbing boundaries and spatial structures. In T. L. Pham, H. K. Le, and X. H. Nguyen, editors, *Proceedings of the Third Asian-Pacific workshop on Genetic Programming*, pages 1–12, Military Technical Academy, Hanoi, Viet-Nam, 2006.

[252] D. Whitley. The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best. In J. D. Schaffer, editor, *Proceedings of the Third International Conference on Genetic Algorithms*, San Mateo, CA, USA, 1989. Morgan Kaufman.

[253] D. Whitley. A genetic algorithm tutorial. *Statistics and Computing*, 4:65–85, 1994.

[254] S. M. Winkler, M. Affenzeller, and S. Wagner. Using enhanced genetic programming techniques for evolving classifiers in the context of medical diagnosis - an empirical study. In S. L. Smith, S. Cagnoni, and J. van Hemert, editors, *Proceedings of MedGEC 2006 GECCO Workshop on Medical Applications of Genetic and Evolutionary Computation*, Seattle, WA, USA, 8 July 2006. ACM Press.

[255] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, April 1997.

[256] M. L. Wong and K. S. Leung. Applying logic grammars to induce sub-functions in genetic programming. In *Proceedings of the Congress on Evolutionary Computation (CEC-1995)*, volume 2, pages 737–740, Perth, Australia, 29 Nov.–1 Dec. 1995. IEEE Press.

[257] M. L. Wong and K. S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[258] S. Wright. The roles of mutation, inbreeding, crossbreeding and selection in evolution. In *Proceedings of the Sixth Annual Congress of Genetics Conference*, pages 356–366, 1932.

[259] T. Yu and C. Clack. Recursion, lambda abstractions and genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. Riolo, editors, *Proceedings of the Third Annual Conference on Genetic Programming*, pages 422–431, University of Wisconsin, Madison, WI, USA, 22-25 July 1998. Morgan Kaufmann.

[260] M. Zhang, V. B. Ciesielski, and P. Andreae. A domain-independent window approach to multiclass object detection using genetic programming. *EURASIP Journal on Applied Signal Processing*, 2003(8):841–859, July 2003. Special Issue on Genetic and Evolutionary Computation for Signal Processing and Image Analysis.

[261] M. Zhang and W. Smart. Multiclass object classification using genetic programming. In G. R. Raidl, S. Cagnoni, J. Branke, D. W. Corne, R. Drechsler, Y. Jin, C. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. D. Smith, and G. Squillero, editors, *Applications of Evolutionary Computing, EvoWorkshops2004: EvoBIO, EvoCOMNET, EvoHOT,*

EvoIASP, EvoMUSART, EvoSTOC, volume 3005 of *LNCS*, pages 369–378, Coimbra, Portugal, 5-7 Apr. 2004. Springer Verlag.

[262] F. Zhao, Q. Zhang, D. Yu, X. Chen, and Y. Yang. A hybrid algorithm based on PSO and simulated annealing and its applications for partner selection in virtual enterprise. In D. Huang, X. Zhang, and G. Huang, editors, *Proceedings of Advances in Intelligent Computing, International Conference on Intelligent Computing (ICIC-2005)*, volume 3644, pages 380–389, Hefei, China, 23-26 Aug. Springer.

[263] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson. Evolving accurate and compact classification rules with gene expression programming. *IEEE Transactions on Evolutionary Computation*, 7(6):519–531, Dec. 2003.

[264] D. Zongker and B. Punch. Lilgp 1.01 user's manual. Technical report, Michigan State University, USA, 26 Mar. 1996.