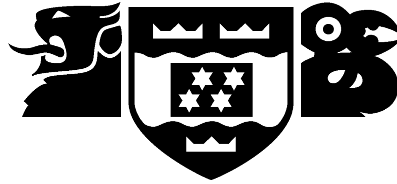
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

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

Genetic Programming: Evolving Simulated Human-Generated Programs with Loops and Control Structures

Daniel McGaughran

Supervisor: Mengjie Zhang

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

Abstract

This report describes research conducted in the research of Genetic Programming (GP), particularly in terms of evolving imperative programs that approach the expressive capabilities of human-generated code. The development of a formal structure based on statement chaining has proven effective, and allowed genetic programs to be output in C++. The introduction of additional genetic operators has shown promising results for greater evolutionary efficiency. A major focus of this work was to develop generic looping and control structures in a way that encourages more effective evolution. In particular, the identified issue of infinite loops has been mitigated by defining a 'safety limit' for the looping mechanisms such as the `While` loops, and using infinite loop statistics in the fitness function. An additional measure taken to reduce the likelihood of infinite loops occurring was the development of child constraints, though this yielded mixed results in terms of evolutionary performance. `For` loops were also engineered to make their loop index available dynamically within the entire scope of the loop body.

Acknowledgement

Contributions made in this research were implemented using the RMITGP genetic programming package, originally developed by Dylan Mawhinney at the Royal Melbourne Institute of Technology (RMIT).

I would like to thank Mengjie Zhang for his support throughout the course of this research.

Contents

1 Introduction and Motivation **1**
1.1 Motivation . . . 1
1.2 Goals . . . 2
1.3 Contributions . . . 3
1.4 Report Structure . . . 3

2 Background **5**
2.1 Overview of Genetic Programming . . . 5
2.2 Previous Work . . . 6
2.2.1 Related Work to Loop Structures . . . 6
2.2.2 Related Work to Genetic Operators . . . 6
2.2.3 Related Work to Local Variables . . . 7

3 New Developments in Genetic Program Representation **9**
3.1 Types Defined in Strongly Typed Genetic Programming . . . 9
3.2 The `Void` Return Type and Chaining . . . 10
3.3 Child Constraints . . . 11
3.4 Additional Genetic Operators . . . 12
3.4.1 Insertion . . . 13
3.4.2 Deletion . . . 13
3.4.3 Replacement . . . 14
3.4.4 Crossover 2 . . . 15
3.5 Dynamic Local Variables . . . 15
3.5.1 Design and Representation . . . 16
3.5.2 Surviving Genetic Operators . . . 17
3.5.3 Index Scaling . . . 18
3.6 Chapter Summary . . . 19

4 Loops **21**
4.1 The `While` and `Do-While` loops . . . 21
4.2 The `For` loop . . . 21
4.3 Dealing with Infinite Loops . . . 24

5 Test Case 1: The Artificial Ant **25**
5.1 Overview of the Test Case . . . 25
5.2 Experiment Design . . . 26
5.2.1 Experiment A: Koza Implementation . . . 27
5.2.2 Experiment B: Ciesielski & Li Implementation . . . 27
5.2.3 Experiment C: Level 1 Implementation with `For` Loops . . . 28
5.2.4 Experiment D: Level 2 Implementation with `For` Loops . . . 29

5.2.5 Experiment E: Level 3 Implementation with `For` Loops 30
5.2.6 Experiment F: Level 3 Implementation with `While` Loops 31
5.2.7 Experiment G: Level 1 Implementation with `For` and `While` Loops . . 31
5.3 Results 32
5.3.1 Existing Implementations versus Statement Chaining 32
5.3.2 Introducing Child Constraints 36
5.3.3 Introducing Additional Operators 37
5.3.4 `For` loops versus `While` loops 38
5.4 Chapter Summary 38

6 Test case 2: Factorial Calculation 39
6.1 Overview of the Test Case 39
6.2 Experiment Design 40
6.2.1 Experiment H: Level 1 Implementation with `For` Loops 41
6.2.2 Experiment I: Level 2 Implementation with `For` Loops 41
6.2.3 Experiment J: Level 3 Implementation with `For` Loops 41
6.2.4 Experiment K: Level 4 Implementation with `For` Loops, No 'temp' Variable 42
6.2.5 Experiment L: Level 4 Implementation with `For` Loops, One 'temp' Variable 42
6.2.6 Experiment M: Level 4 Implementation with `While` Loops, Two 'temp' Variables 44
6.3 Results 44
6.3.1 Introducing Additional Operators 44
6.3.2 Introducing Child Constraints 47
6.3.3 Introducing Dynamic Variables 48
6.3.4 Using the `While` loop 49
6.4 Chapter Summary 49

7 Conclusions 51
7.1 Specific Conclusions 51
7.2 Future Work 53
7.2.1 Prevention of Infinite Loops 53
7.2.2 Additional Operators 53
7.2.3 Restoring Balance 54
7.2.4 Intelligent Choice 54
7.2.5 Additional Control Structures 55
7.2.6 Structure Templates and Recursion 55
7.2.7 Dynamic Node Sets 55
7.2.8 Test Cases 56
7.3 Summary 56

List of Tables

4.1 Definitions of the `While` and `Do-While` loops. 22
4.2 Definitions of the two versions of `For` loop developed in this research. 23

5.1 Functions and terminals of the Artificial Ant problem, as originally defined by Koza. 27
5.2 Functions and terminals of the Artificial Ant problem, based on Ciesielski and Li's implementation [5]. 28
5.3 Functions and terminals of the Artificial Ant problem, using statement chaining. 29
5.4 Allocation of all genetic operators after introducing additional operators. (Artificial Ant problem) 30
5.5 Functions of the Artificial Ant problem with child constraints applied. 30
5.6 Additional functions and terminals used in the Artificial Ant problem, using `While` loops exclusively. 31
5.7 Functions and terminals used in the Artificial Ant problem, using both `For` and `While` loops. 33
5.8 Results of the Artificial Ant problem. 35

6.1 Functions and terminals of the Factorial problem, using statement chaining. . 41
6.2 Allocation of all genetic operators after introducing additional operators. (Factorial problem) 42
6.3 Functions and terminals of the Factorial problem, using dynamic variables. . 43
6.4 Functions and terminals of the Factorial problem, using `While` loops exclusively. 45
6.5 Results of the Factorial problem. 46

Figures

3.1 Definition of symbols used in diagrams of genetic programs 10
3.2 Demonstration of child constraints 12
3.3 Example of the Insertion operator 13
3.4 Example of the Deletion operator 14
3.5 Example of the Replacement operator 14
3.6 Example of the Crossover2 operator 16
3.7 Example of index scaling for Dynamic Variables 19

5.1 Visualisation of the Artificial Ant test case 26
5.2 Average best program fitness over 1000 generations for the Artificial Ant problem 34

6.1 Average best program fitness over 1000 generations for the Factorial problem 46

Chapter 1

Introduction and Motivation

Genetic Programming (GP) is a process of evolving computer programs to perform a specific task, and applying an evolutionary process with natural selection to further develop the 'surviving' programs so that they accomplish the task to a greater degree of competency. The motivation behind GP is to allow computer programs to be automatically-generated without requiring programmers to specify each program instruction. In effect, this tells the computer what to do, rather than how to do it [20, 22].

In GP the computer randomly generates a significantly-sized population of genetic programs and evaluates their fitnesses according to a fitness function. The fitness of a genetic program describes how well the program performs the assigned task. In subsequent generations, a new population is formed from programs in the previous generation selected to undergo Elitism (i.e. cloning of the 'fittest' programs), mutation or crossover (breeding) with a partner program [20]. Elitism, Crossover and Mutation are all referred to as genetic operators. Each generation undergoes the same genetic operations to produce subsequent generations of programs, with a view to the average fitness improving over time.

Koza [20] defines genetic programs as tree structures with leaf nodes selected from a set of 'terminals', and non-leaf nodes selected from a set of 'functions'. The terminals are used as basic inputs into the genetic programs. Functions use these terminals or the outputs of other functions and produce an output itself. Thus the program executes in a bottom-up fashion and produces an overall result at the output of the root node.

The greatest potential of GP is allowing computers to perform complex tasks towards a level of ability that rivals (or even excels) that of humans [2, 21]. These complex tasks would otherwise require an extremely large amount of design effort to solve by humans programming manually, and it is often likely that a more effective and/or efficient solution exists.

1.1 Motivation

Until recently, genetic programs have used mathematical expressions and operators for classification problems, or user-defined functions and terminals for programs that actively perform a task (such as controlling an agent's behaviour). The program constructs used in human-generated programs (e.g. loops, `If-then-else` blocks, etc.) have largely been incorporated into genetic programs only as user-defined functions and/or terminals specific to the problem being solved, not as generic structures. These constructs are typically tied to the particular problem being solved, and therefore lack genericity.

To illustrate this, the Artificial Ant problem introduced by Koza [19] and widely-studied [22, 5, 9] uses an `IfFoodAhead` function, which acts as a standard `If-then-else` construct but evaluates a specific condition specific to the problem. The problem with this is that if a second condition was needed, unnecessary duplication would arise from creating a second instance of the `If-then-else` construct to evaluate the specialised condition.

A related problem is that the evolved programs were originally evaluated in an iterated fashion by the program environment, hence a tight degree of control was maintained as to how often the program would iterate. Unfortunately, this control prevents the programs from having a more flexible flow of control with respect to looping. Only recently have attempts been made to use evolved loops instead of relying on the implicit looping in the program environment [5]. The Artificial Ant problem is described in more detail in Chapter 5.

Despite the use of specialised functions defined by the user for constructing imperative programs, the genetic programs evolved in previous work have been structured as expressions, rather than imperative programs. Such expressions have almost always been represented as LISP S-expressions for their close representation of the tree structure of genetic programs and ease at which they may be parsed, as evidenced by a wide variety of previous work [19, 5, 6]. In contrast, imperative programs are widely developed in declarative programming languages such as C/C++, and have the advantage of being able to present logical flow and structure more clearly.

A major success factor of GP is the ability to evolve programs that rival, or even exceed, those generated by humans for a variety of applications [2, 21]. In order to achieve this, one could argue that genetic programs should make use of the same ubiquitous language features as human-generated code. Such language features would include declaring and using local variables and using loop and control structures. This research focuses on engineering genetic programs to make use of these features, particularly loop and control constructs.

Because the major focus is on evolving loop constructs, language features are only covered within the scope of this research if they contribute to more efficient evolution of programs with loops and other control structures. Such an example of this is the definition and use of index variables in `For` loops, scoped to within the loop body. This requires a scheme to use local variables in GP, and is discussed in more detail in Section 3.5.

1.2 Goals

The high-level objective of this research project is to contribute to the evolution of genetic programs that use the capabilities of higher-level imperative languages such as C/C++. This would allow genetic programs to exhibit more powerful semantics than much of what has currently been implemented in GP, and therefore mimic human-generated code to a certain extent. The primary focus in this research is the implementation of generic loops and control structures in genetic programs.

As a whole, the project seeks to answer the following questions:

- How can control structures such as loops and `If` blocks be implemented and evolved effectively in genetic programs?
- How can infinite loops be avoided in loop constructs, particularly `While` loops?

- Can particular structures achieve better performance (in terms of effectiveness and efficiency of the evolved programs) than without them?
- What additional measures could be taken to accelerate the evolutionary process with these constructs implemented?

1.3 Contributions

In this research a new, formal structure of genetic programs is defined to more closely represent structure of imperative programs. This involves introducing a convention of 'chaining' nodes of return type `Void`, referred to as 'statement-chaining'. Statement chaining is used as a replacement of the `Prog2` and `Prog3` nodes used in [19] and [5].

Additional genetic operators have been defined to improve evolution efficiency of programs using statement-chaining. These are, in particular, Insertion, Deletion, Replacement and Crossover2. All of these additional operators are designed to minimise disruption to the rest of a subtree below the target node(s). The concept of child constraints is also introduced to support more efficient evolution through the ability to reduce a particular type of redundancy in loops and control structures.

Dynamic variables have also been introduced to support local variables in GP, with a particular focus on loop indices. This allows programs to evolve nested loops, with any loop index from the outermost loop to the innermost loop able to be used as a terminal within the body of the innermost loop. Dynamic variables also have the potential to be used for creating arbitrary variables for within the remainder of the scope of the current code block, similar to variable scope in a C++ function.

The `For` loop has been generalised to allow increments greater than 1, and to make index variables available for the entire scope within the loop body. A `While` loop has also been developed to follow similar semantics to C++. 'Safety limits' are used to detect and recover from infinite loops, with the count of infinite loops being made available to the fitness function. This allows the evolution to favour programs with a minimum number of infinite loops, and evolution is therefore able to proceed faster. The safety limits operate on a local (loop-based) level and global (program-based) level - the latter avoiding excessive nesting of loops that leads to slower program evaluations.

1.4 Report Structure

Chapter 2 provides a background of Genetic Programming and describes previous work relevant to this research. This describes the fundamental genetic operators and overall mechanism of GP in detail. The previous work is briefly summarised and evaluated in terms of how well they meet the goals of this research.

In Chapters 3 and 4 the contributions of this research are described and discussed in detail. A significant part of this discussion includes the design decisions made during the course of this research. Chapter 3 contains details of the contributions made, excluding the loops themselves. These contributions support the high-level goal and provide a level of functionality for the loops. The loops and their supporting mechanisms, specifically handling of infinite loops, are discussed in Chapter 4.

The test cases used and their experiments performed are described in Chapters 5 and 6. In each chapter, the corresponding test case is described in detail. This is followed by descriptions of the design of each experiment. Both chapters conclude with the presentation of experimental results and analysis of the findings.

Finally, Chapter 7 summarises the contributions made and their indivual effects on the performance of evolving solutions to particular problems. Ideas for future research are also discussed, relating to this field and where it may be heading.

Chapter 2

Background

2.1 Overview of Genetic Programming

Genetic Programming was conceived as a generalisation of Genetic Algorithms (GAs), which was inspired by natural biological evolution and survival of the fittest [22, 21, 20]. The general idea is to automatically evolve computer programs to solve a particular task, therefore the programmer is only required to specify what to do rather than how to do it. The key difference between GA and GP is that GA uses a bit-string representation for its entities (e.g. algorithms), and GP almost always uses tree structures to represent expressions or programs [21]. While genetic programs can be represented as linear forms and directed acyclic graphs [21], this research will focus on the tree-based representation.

For GP, a population of programs is randomly-generated and their 'fitness' is measured by way of a fitness function. It is the fitness function that is specified by the programmer. This tells the computer what the program should be doing and is used to benchmark programs against each other, and any 'correct' solution if applicable to the particular problem being solved. The initial population generated is then used to provide programs for undergoing genetic operations. The resulting programs are then promoted into the next generation, and their fitnesses are evaluated. In each subsequent generation, programs are selected for undergoing genetic operations and promotion to the next generation [21, 19, 20].

In standard Genetic Programming, the standard operators used are Elitism, Crossover and Mutation. Other variants, such as Hoist and Shrink, have also been developed. These variants are described in further detail by Langdon [21]. Each operator has a particular allocation of a destination population, in which to promote new programs after the operation. As a corollary, each program in a non-initial population is formed by exactly one of these operators. Thus one portion of the population is formed by Elitism, another by Crossover, and the remainder by Mutation.

Elitism is the simplest 'operator', as it simply copies a genetic program unaltered from one generation to the next. Mutation replaces a randomly-selected subtree with a new subtree randomly-generated using the Grow method, while Crossover operates on two subtrees simultaneously, swapping subtrees between the pair [20].

The evolutionary process continues until a termination condition is reached. This may be after a particular number of generations, or when a solution is found [20]. A solution may be determined by evaluating the fitness function and making a comparison against a given threshold.

2.2 Previous Work

2.2.1 Related Work to Loop Structures

In 2004 Ciesielski and Li [5] demonstrated using basic forms of `For` loops in Genetic Programs for controlling an agent in a modified Artificial Ant problem and sorting a 7-element array. This work was extended later that year to using a variant of these for-loops in classification problems [6].

In their implementation of `For` loops, Ciesielski and Li used Strongly-Typed Genetic Programming (STGP)[23] to ensure that the generated programs were syntactically correct. Even though STGP was formalised by Montana, the idea behind STGP was evident in Kinnear's work of sorting using GP [17]. Other constrained forms of GP, similar to STGP, have also been developed. These include Constrained GP (CGP) [14] and Adaptable Constrained GP (ACGP) [15], Grammatically-based GP [27] and Tree-Adjoining Grammar-based GP (TAG-based GP) [13].

Kinnear's work on sorting integers used an Automatically Defined Function (ADF) to implement the loop construct, and an 'index' variable as one of the terminals used for indexing the innermost loop. If the index was used outside a loop, it would simply adopt the value 0 [17]. This approach has two shortcomings: if a program uses two nested loops, the index of the outer loop cannot be used inside the innermost loop. If the program uses no loops whatsoever (unlikely for sorting), the 'index' terminal is still available for selection, despite being invalid.

In all cases of loops that have been examined in GP, the `For` loop construct was used and restricted to increment the index by one until it reached an upper bound. The present form of this cannot handle incrementing by any other value (including negative numbers), or multiplying the index by a calculated or terminal value. No evidence has been found of more general forms of the `For` loop, or `While` loops being used in Genetic Programs explicitly as functions. Other control structures yet to be implemented are `Case` statements and recursion.

2.2.2 Related Work to Genetic Operators

As mentioned in the introduction, there are many variants of genetic operators that have been developed. Langdon [21] describes several variations of mutation and crossover designed for standard GP, some of them bearing little resemblance to the standard operators. Some of these include Hoist, which isolates a subtree of a program as a new program in its entirety; Shrink, which replaces a subtree with a terminal; and Node Replacement, which replaces a node with another that requires the same number of children.

Since these are developed for standard GP, some of these (like Node Replacement) would prove impractical if used in STGP because of the need to enforce closure. Closure is defined by Koza [19, 21] as a property of programs being syntactically-correct after modifications, so that they may be executable. In the case of Node Replacement, it may be much harder to find a suitable replacement to satisfy closure since the types of each child need to be matched completely. Other forms of mutation by Hoai [13] include Insertion and Deletion, which are also desiged for standard GP.

It is worth mentioning that these Insertion, Deletion and Node Replacement operators are significantly different to the operators with the same name developed in this research. The

operators developed in this research are targeted towards STGP, and designed to be more flexible and less disruptive. These are described in detail in Section 3.4.

2.2.3 Related Work to Local Variables

Kirshenbaum [18] has developed some scheme to create local variable from within genetic programs, and ensure that local variable scope is properly maintained. His approach uses a form of labelling each variable in order to bind variables to programs upon modification by genetic operators. Each variable was maintained as a stack of recently-assigned values. The Crossover operator was modified to perform binding between variables by merging nodes from opposite parents, to ensure that variables were defined before their use.

This previous work relates to achieving the high-level goal of this research, however it seems to be targeted only at creating local variables on-the-fly in commands similar to the `Let` command in LISP, without considering the creation of loop indices by loop constructs. The scheme proves to be reasonably inflexible through its reliance on binding variables by name and from the nature of the bindings affecting where crossover points may be selected [18]. Nevertheless, the general idea of using dynamically-created local variables, particularly for loop indices, has prompted a new implementation to be designed as part of this research.

Section 3.5 describes the mechanism designed in this research, to support dynamic local variables without placing unnecessary constraints on genetic operators.

Chapter 3

New Developments in Genetic Program Representation

From the start of this research, the design of the loop and control structures was based on a representation that closely follows the structure of imperative programs written in languages such as C++. As part of this research, this representation was developed for imperative programs, while still providing the capability to support complex expressions, like those previously evolved as genetic programs. This provides direction towards fulfilling the high-level goal of genetic programs adopting the ubiquitous language features of higher-level languages and achieving a higher level of code quality that mimics human-generated code to an extent.

In order to implement the loops and control structures described as part of the research goals, a number of developments were necessary in order to allow an organised implementation of the constructs with positive results. This chapter discusses these developments in detail. In particular, the definition of types in Strongly Typed Genetic Programming (STGP) is discussed, along with particular consideration of the conventions used for functions and terminals of the `Void` return type. The last section describes the use of local variables that can be accessed directly by the genetic programs themselves, rather than rely on defining-functions and terminals specific to the problem being solved.

3.1 Types Defined in Strongly Typed Genetic Programming

The work performed as part of this research is based on an implementation of Strongly-Typed Genetic Programming (STGP) as presented by Montana [23]. STGP was chosen for its ability to support and distinguish between the various data / return types used in imperative programs. In this research the return types have been defined to reflect their C/C++ equivalents – `Void`, `Bool`, `Int` and `Double`. Neither the single-precision floating point type (`Float`), nor the long integer type (`Long`) are used so as to avoid double-up between similar data types. With the exception of the `Void` return type, each predefined type represents an explicit value of data passed upwards in the program tree, and functions of this type have no particular conventions to follow in terms of parameters. The definition of nodes are given in the next section.

It is worth noting that these return types are inherently defined as part of the GP paradigm used for evolving programs with control structures, yet additional types may be declared for specific applications. Examples of such additional types may include enumerated types,

collection objects such as vectors, and a subset of integers for indexing particular arrays or collections.
The rest of this chapter contains diagrams of genetic programs that depict aspects of the contributions made, particularly concerning child constraints and additional genetic operators. A guide to symbols used in these diagrams, covering node return types and operations performed on individual nodes, is presented in Figure 3.1.

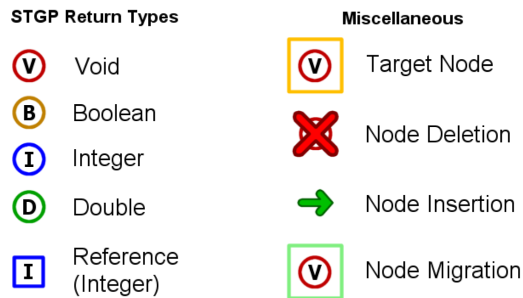


Figure 3.1: Symbols used in diagrams of genetic programs presented in this chapter. The left column shows the STGP return types, and the right column shows operations that may be performed on a particular node.

3.2 The `Void` Return Type and Chaining

The `Void` return type has been created for imperative statements that are executed in a similar way to C++ functions or statements. Such statements have no explicit return value, but perform a particular action. Furthermore, these statements are implemented as functions that are chained together in such a way that the first child node is the next `Void` statement to evaluate, which is evaluated last after all other children of that function.

To illustrate what constitutes a `Void` function, it is best to consider what the action of the node is. A call to an ADF results in an action being performed, though this ADF may or may not return a value. Consider the C++ operator '+' (overloaded for integers): its action is to add two integers (possibly the results of sub-expressions) together *and return the result*. Because a value is returned, the operator cannot be of type `Void`. However, consider the assignment operator '+=', as in '`a += (b + c);`': In this case the sub-expression `(b + c)` returns an integer to the '+=' operator, which then adds that result onto the variable '`a`'. The assignment node itself *does not return a value*, even though its sub-expressions do, therefore the assignment is of type `Void`. This becomes clearer if the operator is expressed as a standalone function in C++:

```
void operator+=(int &a, int b)
{
    a += b;
}
```

A node representing an equivalent function in C++ is always defined as a function in GP (i.e. not a terminal), regardless of whether the C++ equivalent function uses parameters or not. This is because `Void` statements always contain a child corresponding to the next function to evaluate. The terminals of type `Void` are the null-operation (`NOP`), and possibly the `break` and `continue` equivalent statements (where applicable). The `break` and

continue statements were not implemented in this research, and are therefore discussed in Section 7.2.7.

It becomes evident that the advantage of chaining Void functions is a more accurate representation of the C/C++ equivalent of genetic programs. This allows genetic programs to be structured in a similar manner to Control Flow Graphs [3, 7], but in tree-based form. The use of the Null Operation as the main terminal of this type also has an advantage of not requiring two versions of each Void function, which only differ in the inclusion or exclusion of a Void child node for the next statement to be executed. Hence a significant amount of redundancy is avoided.

There are also challenges presented as a result of adopting a chained approach. In particular, chaining causes the program trees to have a deeper structure than the structure used in the Artificial Ant Problem as presented by Koza [19] and Ciesielski and Li [5]. In their implementations, Void functions did not use a child node for chaining statements, hence statements without arguments were implemented as terminals. Statements were joined together by Prog2 and Prog3 functions (collectively referred to as ProgN functions), which were also of the Void return type and simply evaluated each of their child nodes sequentially. The Prog2 and Prog3 functions had two and three child nodes respectively, each of return type Void. Thus the representation encouraged a shallower tree structure. The Artificial Ant Problem is described in Section 5.1.

The chaining of the statements and the deeper tree structures also have implications on how the program structure is affected by the standard Crossover and Mutation operators. In particular, these operators affect the rest of the chain from the selected node downwards, and lead to greater disruption of the programs during evolution. This has led to the design of additional genetic operators that focus their operation on individual nodes, and preserve as many of the node's subtrees as possible. These operators are described in the following section.

3.3 Child Constraints

Child constraints allow functions to specify whether a particular child node may be either a function, terminal, or both (respectively Function, Terminal and Both). The constraints are usually Both by default, however the body of a loop node and the True branch of an If statement both have a child constraint allowing functions only (i.e. Function).

This forces loops to have a chain of at least one statement, not counting NOP, and is helpful in reducing the likelihood of infinite loops occurring in While loops, for example.

Figure 3.2 shows a sample fragment of a genetic program using the While loop, and demonstrates a legal and illegal example of how the subtree may be grown. The first two children ('next' statement and condition, respectively) both have no child constraint set (i.e. Both). The third child, representing the loop body, has a constraint set to functions only (i.e. Function). Therefore the NOP terminal may not be used directly as the root of the subtree originating from this child.

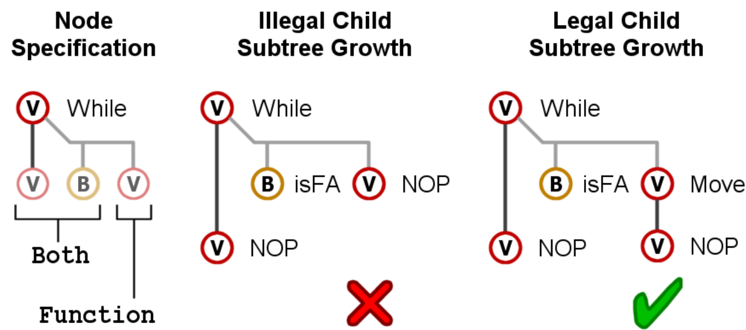


Figure 3.2: An example of how child constraints are enforced, using the `While` loop. The third child has a constraint of `Function`, whereas the other two allow either functions or terminals.

3.4 Additional Genetic Operators

In initial tests of genetic programs structured with statement chaining, it was found that the Crossover and Mutation operators were particularly destructive. In one of the test cases, there were a number of genetic programs that would require the deletion of one statement node, or replacement of another, to generate a correct solution. Instead, the Crossover and Mutation operators affected the entire subtree below the target node and made it much more difficult to evolve a solution. This prompted four additional genetic operators to be developed.

The additional operators introduced are the Insertion, Deletion, Replacement and Crossover2 operators. Each operator targets a single node in a genetic program, and is designed to preserve as many of the child nodes (and their subtrees) as possible. All operators maintain type and child constraints, and ensure that the overall depth of the resulting program does not exceed the depth limit. If a resulting program would exceed this limit, the attempt is aborted and a new attempt is made. After a certain number of unsuccessful attempts, the operator reverts to the standard Mutation operator.

Each operator is described in detail in the following sections. In the detailed descriptions that follow, there are a number of technical terms used. These are defined as follows:

- **Target Node**: The node randomly-selected for the genetic operation to be performed on. For Crossover, this is often referred to as the 'crossover point'.
- **Chainable**: A function is chainable if its return type is the same as at least one of its child nodes.
- **[Node] Specification**: A combination of return type and child constraint placed on the child of a function (or root node of a program), which the corresponding node must satisfy. For example, a specification of (`Void`, `Both`) allows both functions and terminals of type `Void`, but a specification of (`Void`, `Function`) only allows functions of type `Void`.
- **Surplus Child**: A subtree whose root is the child of the target node, and whose return type and child constraint does not satisfy at least one child specification of the new parent node.

- **Child Deficit**: A child deficit occurs where at least one child specification of a function cannot be satisfied by any of the existing child subtrees being repatriated to that function.

3.4.1 Insertion

This operator selects either a function or terminal as the target node. The inserted function is chosen so that one of its child nodes' return types matches that of the targeted node. As a result, the subtree from the targeted node is repatriated to the inserted node, which in turn is repatriated to the targeted node's original parent. Any surplus children of the inserted function are generated using the Grow program generation method.

Figure 3.3 demonstrates the effects of the Insertion operator. Note how the rest of the chain from the target node downwards becomes a child of the inserted node, and that all other children of the inserted node are randomly-generated.

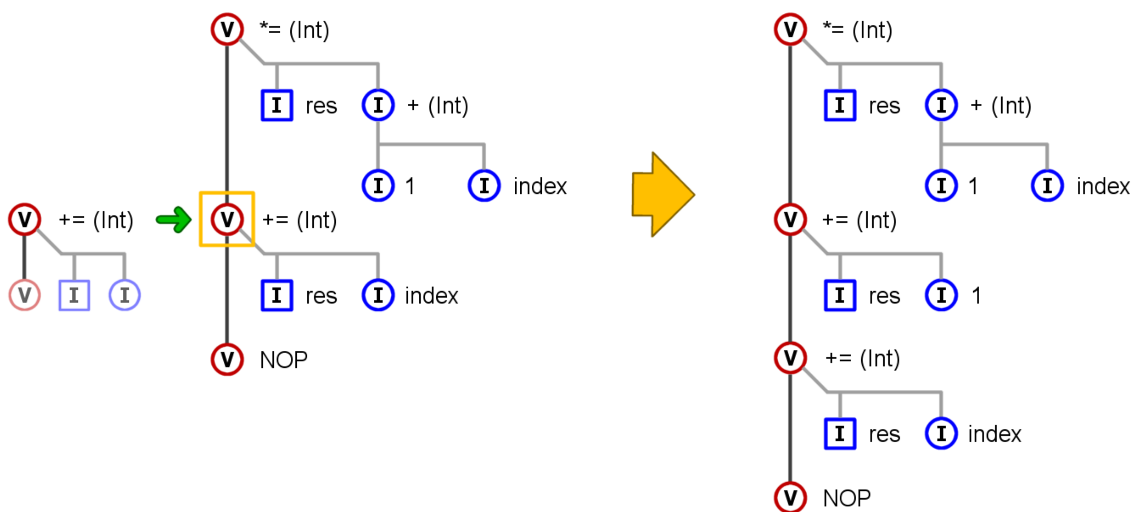


Figure 3.3: An example of the Insertion operator on a fragment of a genetic program.

The Insertion operator may make several attempts at finding a target node and a function to insert, so that the conditions mentioned are satisfied. If the number of unsuccessful attempts made has exceeded a set threshold, the operator reverts to the standard Mutation operator.

3.4.2 Deletion

The Deletion operator performs the opposite of the Insertion operator. This operator selects a chainable function as the target node. Assuming one such node is found, the first child with the matching return type is saved and repatriated to the target node's parent. The target node and its remaining child subtrees are deleted. If no suitable target node can be selected after a predetermined number of attempts, the operator reverts to the standard Mutation operator. An example of the Deletion operator is shown in Figure 3.4. Note that the rest of the statement chain below the deleted node is unaffected.

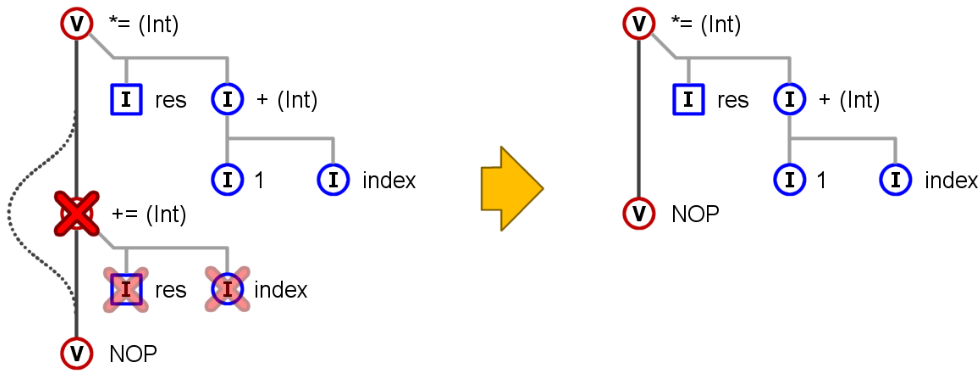


Figure 3.4: An example of the Deletion operator.

3.4.3 Replacement

This operator selects a function node for replacing with a different function, and is the most complicated of these new operators. A chainable function node is selected as the target node. All child subtrees of the target node are saved, and the target node is deleted. In its place a different chainable function is randomly-selected from the node table.

Once the target node and replacement node have been determined, the saved child subtrees are repatriated to the replacement node where the return type is satisfied. Surplus child subtrees are deleted as there are no child positions left on the replacement node. For a particular return type, a shortage of saved child subtrees for the replacement node are resolved by growing new subtrees.

Figure 3.5 shows an example of the Replacement operator. In particular, note that the rest of the chain is preserved, along with the `index` terminal below the deleted node. The loop body beginning with the `*=` assignment operator is randomly-generated as per mutation.

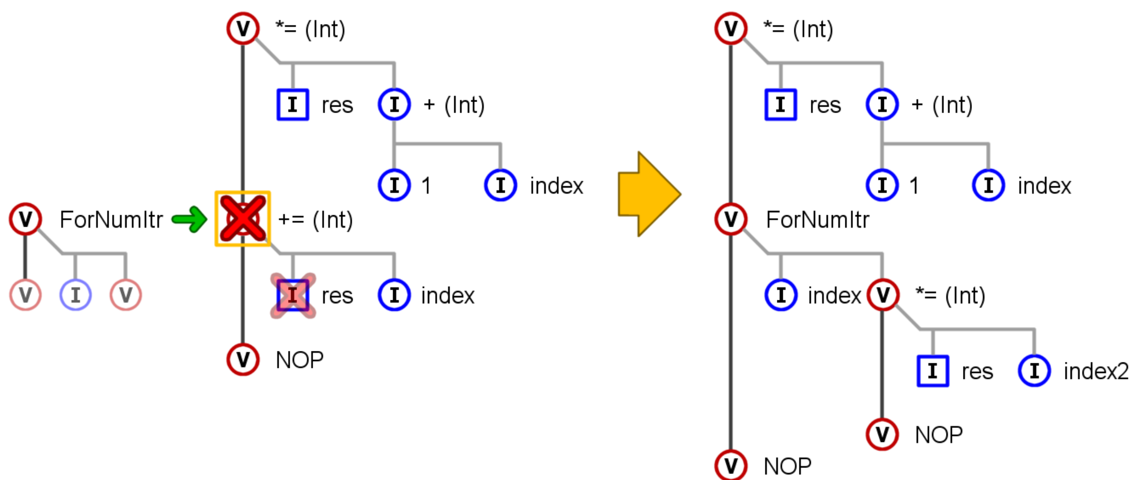


Figure 3.5: An illustration of the Replacement operator.

3.4.4 Crossover 2

The Crossover2 operator selects two programs as per standard crossover. Compatible target nodes are selected in each program, and the Replacement operator is effectively performed on each of them. A major distinction from the Replacement operator is that the replacement node for one program is the target node of the other. There is no random selection of chainable functions from the node table, nor are either of the target nodes deleted. Because no functions are deleted, all child subtrees are preserved. Any surplus subtrees are migrated with the target node to the other program to resolve the child deficit.

Once the target node and replacement node have been determined, the saved child subtrees are repatriated to the replacement node where the return type is satisfied. Surplus child subtrees are deleted as there are no child positions left on the replacement node. For a particular return type, a shortage of saved child subtrees for the replacement node are resolved by growing new subtrees.

Crossover2 is demonstrated in Figure 3.6. Note that this works in a different manner to standard Crossover. In particular, the rest of the statement chains below the target nodes in each program are unaffected – thus remaining in the same program. Standard Crossover would have migrated these with the target node to the other program. Also, note that the `res` reference terminal is migrated with the target node to the other program, since there is no compatible node specification in the incoming function, i.e. the `ForNumItr` function does not have an integer reference as one of its child return types. The index terminal is not migrated (as it would under standard Crossover), as it can be repatriated to the incoming `ForNumItr` function. As a result, no child subtrees are deleted and replaced by randomly-generated subtrees.

3.5 Dynamic Local Variables

In order to evolve loops with an index variable, a mechanism was devised for allowing local variables to be dynamically created, manipulated, and read within local scope of a genetic program. This mechanism allows local variables to be declared implicitly in the `For` loop and made available to the loop body in the genetic program as a read-only variable, so that the value can only be changed by the internal looping mechanism. The scope of the index variable is defined in a similar fashion to what is possible in C++.

The use of terminals has prompted additional return data types to be developed as references to the existing `Bool`, `Int` and `Double` data types. These reference types are `BoolPtr`, `IntPtr` and `DoublePtr` respectively. They may prove a useful starting point for implementing pointers in GP, although this is beyond the scope of this research. The actual variables are each associated with a "read-only" flag, which indicates whether the variable is able to be modified in the genetic program by assignment statements such as `+=`, `-=`, and `=`.

This mechanism for supporting local variables was designed beyond merely creating loop index variables in `For` loops, and can be used to dynamically create local variables in general. The variable creation scope is based on the model used in C++ (as opposed to ANSI C) to enable variables to be created anywhere within the program – not at the top of functions before any non-declaration statements. This makes the implementation much simpler than enforcing the ANSI C restriction, as a node that creates a variable defines the scope for the entire subtree of one or more children.

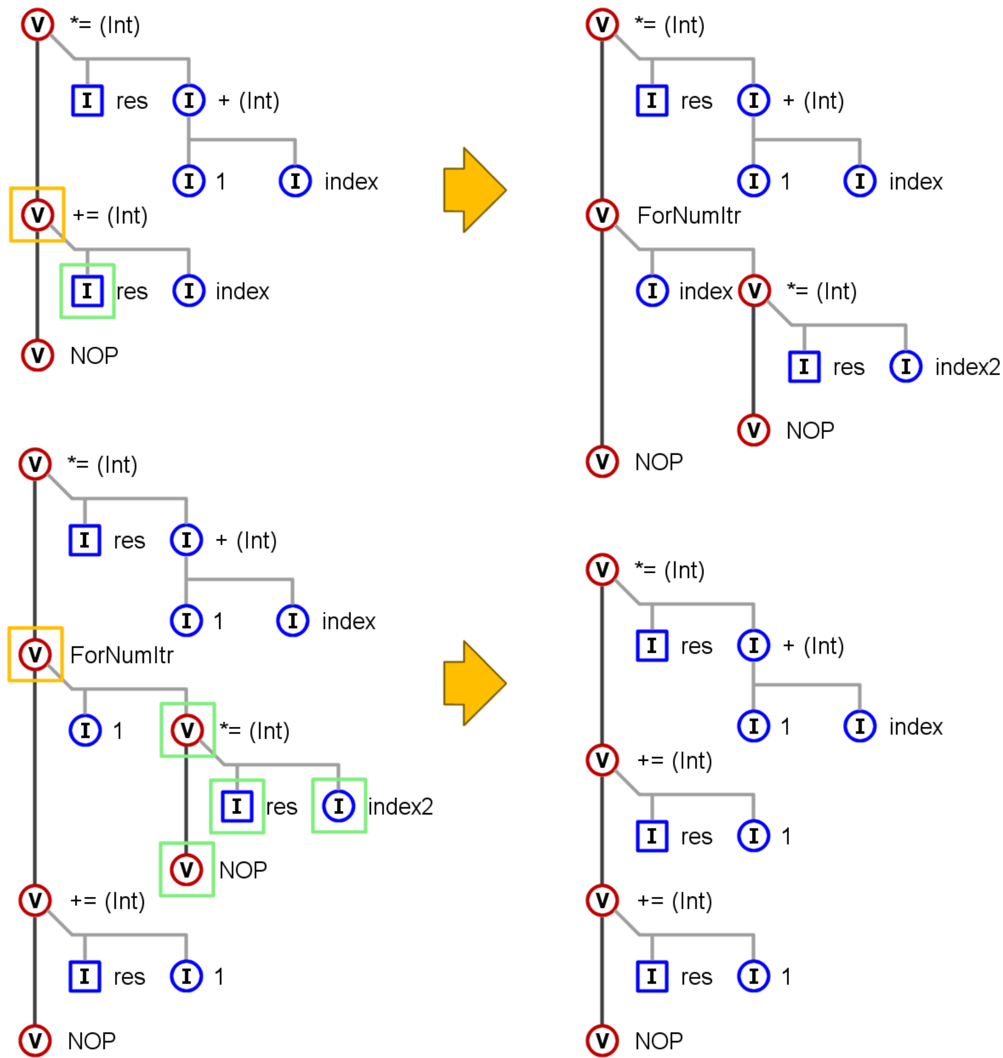


Figure 3.6: An example of the Crossover2 operator.

In particular, suppose there is a `Void` function `DeclareInt`, which creates a local integer variable. The variable's scope would last for the rest of the statement chain, which corresponds to the rest of the statement block in the C++ equivalent. This scope would involve the entire subtree from the '`Next`' child of `DeclareInt`, providing further justification for the chained statement structure introduced in Section 3.2. Under the previous model, which consisted of `Prog2` and `Prog3` nodes, there isn't a 'Next' child. As a result, it would be much more complicated to propagate the variable scope information to the rest of the nodes within a particular variable's scope.

3.5.1 Design and Representation

Local variables are aggregated in 'pools' according to their data type, and referenced by a numerical index. A terminal of each data type uses the implicit index to retrieve the corresponding variable from the pool for evaluation. Each data type has two pools which expand or contract as variables go in or out of scope, typically operating in a similar fashion

to stacks. One pool (**R**) is for all variables, whose values may be read within the genetic programs. The other (**RW**) is a subset, containing only those variables that may be overwritten. At this stage, only the `Int` data type has support for dynamic variables, although this can be easily adapted to the `Bool` and `Double` data types.

For each data type, there are two terminals that together access the **R** and **RW** pools. For the `Int` data type, these terminals respectively return an explicit integer value (`GetIntVal`) and an integer reference (`GetIntRef`).

The `GetIntVal` terminal may be used in any integer expression in the same manner as an explicit integer value randomly-selected on the node's instantiation. This terminal randomly selects an index into the **R** pool on the creation of each new instance and stores it locally with that instance. Upon evaluation, the index is used to retrieve the corresponding variable from the pool. Since the pool size varies to reflect changing program scope at different parts of the program, the pool size is actively tracked from the root node to a particular position in the program. This is to ensure the index is within the allowable range.

The `GetIntRef` terminal is used for statements that modify a variable's value, such as assignment statements. It is stressed that these are static references to variables, not pointers to a defined memory. Future research may investigate the feasibility of using this mechanism to represent actual pointers. The index itself is returned, so that the evaluating function may use it to look up the actual variable and modify its contents.

3.5.2 Surviving Genetic Operators

It may seem relatively straight-forward representing variables for individual genetc programs, but performing structure-altering genetic operations such as Crossover would have a likely potential to disrupt the indexing of variables that wouldn't exist afterwards. It is highly possible that after a modification, an index would either be out-of-range (and therefore invalid), or referencing a different variable altogether. If there are no variables in a pool, then every node that indexes a variable would be invalid. This section discusses the design decisions made to address these issues.

If a problem being solved by GP makes use of dynamic variables, then the corresponding pool is required to have at least one variable present at every node being evaluated. This guarantees that every terminal indexing into the pool is valid, and is enabled by index-scaling (described later). This strategy has an advantage of not requiring program analysis to remove the invalid nodes, but introduces a requirement of the GP environment that may be redundant for the problem being solved.

The decision was made on the basis of the disadvantage being relatively minor, compared to the alternative of expanding the scope of this research to include program analysis for maintaining the correctness of programs. Program analysis is a large field of computing aimed primarily toward error-correction and optimisation [1, 8], therefore the use of program analysis for optimising genetic programs and maintaining correctness would be suitable as a separate topic for future research. This implementation of dynamic variables is motivated by the need to allow proper use of loop indices in `For` loops, therefore the scope is kept to a basic level.

A particular reason why program analysis is not used in this research is that it makes the task of maintaining program correctness much more complicated. Suppose a genetic operation,

e.g. Crossover, has produced a program that contains a reference to an integer variable, with no local variables declared within its scope. Even though indexing is used to bind variables, the absence of variables in the scope means that any index dereferenced will be invalid. Therefore it would make the most sense to remove the variable reference. The trouble is that this node will require replacement by another node.

This would be more complex if the node in question is a variable reference, as these denote read-write variables used directly by a statement such as +=, and cannot be replaced by any other terminal since all reference data types interact with the dynamic variable pools. Therefore the parent node would need to be removed. Deletion would be unsuitable, as there is a possibility of conflict with child constraints. This would require Replacement to be used instead, assuming that there is another suitable node available. If neither approach would work, the parent's parent would need to be replaced.

3.5.3 Index Scaling

Index scaling is the technique used to bind variables after a genetic operation. There are two possibilities of how the dynamic variable pools and their respective index spaces would change after the change – expansion and compression. Expansion does not jeopardise the correctness of the resulting program, therefore no change is required.

With compression, there are likely to be indices left out of range of the resulting index space, therefore index-scaling is needed to shift the indices of an affected subtree downwards, so that the highest index of the subtree is within the legal index range. Any index shift is clipped so that the resulting index remains above or equal to zero (the first variable in the pool). As a result, all indices fall within the legal index range, thereby maintaining the correctness of the program.

Figure 3.7 shows how index scaling works, with an example of compression and expansion of index spaces. The red line depicts the 'cutoff' separating the indices of variables created prior to the target node of a genetic operation, and the indices of variables created in the subtree below the target node. The cutoff point in a program is the edge between a target node and its parent. Diamonds represent instances of terminals that access a particular variable. The cell in which a particular diamond is located corresponds to the index of the variable, stored in that particular terminal. Grey diamonds indicate variable accesses that are made in the path from the root of the entire program to just above the cutoff point, and coloured diamonds correspond to variable accesses made in the entire subtree below the cutoff point.

The colours of the diamonds indicate the transformation applied to a particular index reference made below the cutoff point, and merit further explanation. Blue diamonds correspond to variables created before the cutoff point, therefore the indices do not change. Orange and red diamonds fall within indices of variables created below the cutoff point, and are subject to scaling. For orange diamonds, the scaling is consistent and corresponds directly to the number of indices by which the index space shrinks (if it does shrink). Red diamonds cannot be scaled by the same amount as the indices would go out of range, therefore all such indices are set to zero.

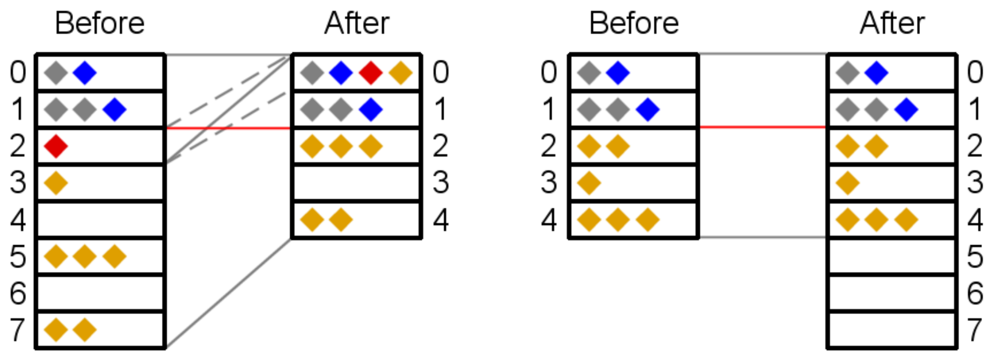


Figure 3.7: A sample demonstration of index scaling, showing compression (left) and expansion (right).

3.6 Chapter Summary

In this chapter a number of major contributions have been introduced and defined. Most of these are aimed toward the high-level goal of allowing genetic programs to adopt standard language features used commonly by human programmers. They are also intended to contribute to a more effective evolution of loop and control structures.

Particular developments made include the specification of statement chaining using Strongly-Typed Genetic Programming, definition of new genetic operators designed to minimise program disruption, introduction to child constraints and an outline of using dynamic variables. All these contributions, except the genetic operators, are directly used in the development of loops in the next chapter.

Chapter 4

Loops

In previous work, almost all implementations of loops have been simple `For` loops, which increment an internal index within a fixed integer range. For example, Ciesielski and Li [5] have implemented two versions of the `For` loop. The first version accepts a number of iterations and a loop body to evaluate. The second type accepts a start value and an end value in addition to the body, and iterates through the body once for each number between the start and end inclusive. In this project the `While` and `Do-While` loops have been developed to complement the existing `For` loops. Modifications have also been made to the `For` loops to encourage more effective evolution. This chapter will describe the implementation of these loops and discuss the strategies used to deal with infinite loops.

4.1 The `While` and `Do-While` loops

The `While` loop is one which executes the loop body as long as the Boolean condition is true, given that the condition is evaluated at the start of the loop. The `Do-While` loop is similar, except that the condition is evaluated at the end of the loop – thus the loop is guaranteed to execute at least once. In this discussion, both `While` loops and `Do-While` loops will be collectively referred to as `While` loops.

Unlike the `For` loops presented by Ciesielski and Li [5], these loops do not use integers to keep track of the number of iterations, therefore infinite loops are likely to occur as a result of evolving Boolean conditions. In light of this, the `While` loops are modified to provide an upper safety limit on the number of iterations. The use of safety limits is described in more detail in the context of dealing with infinite loops, in Section 4.3.

The definitions of the `While` and `Do-While` loops are given in Table 4.1. Full node specifications of the children are given. Child constraints are denoted by [`B`] for `Both` and [`F`] for `Function`.

4.2 The `For` loop

In languages such as C/C++, the `For` loop is a specific implementation of the While loop. The three components of this loop are:
This form of the loop does not explicitly require semantics of a counting or index variable to be initialised to a particular value, nor does it require the index to be incremented (or decremented) until the value reaches a particular number. Despite this genericity in such

<table>
<tr><th>Return Type</th><th>Node Name</th><th>Children</th></tr>
<tr><td rowspan="2">Void</td><td>While</td><td>Void next [B], Bool cond [B], Void body [F]</td></tr>
<tr><td colspan="2">Evaluates the loop body ‘body’ and condition ‘cond’ while ‘cond is true. The condition is evaluated before the body, therefore the body may not necessarily be evaluated at all.</td></tr>
<tr><td rowspan="2">Void</td><td>DoWhile</td><td>Void next [B], Bool cond [B], Void body [F]</td></tr>
<tr><td colspan="2">Evaluates the loop body ‘body’ and condition ‘cond’ while ‘cond is true. The condition is evaluated after the body, therefore the body is guaranteed to be evaluated at least once.</td></tr>
</table>

Table 4.1: Definitions of the While and Do-While loops.

languages, many other languages such as Basic use For loops for counting via index variables. Although a generic For loop as implemented in C/C++ may be implemented in Genetic Programming, the nature of statement chaining and the availability of a While loop makes such a generic implementation redundant. In light of this, the For loops in GP shall focus on integer counting.

Since the generic For loop is declared redundant, the most generic form of the loop is based on the second form presented in [5], but with the increment (delta) size able to be set to a different value than 1. The order of the start and end values determines whether the delta is added to or subtracted from the index. The first form of the For loop presented in [5] has also been reproduced in these experiments.

The generic nature of the For loop in C/C++ allows more complicated types of looping. This includes including multiplying or dividing the counter by a constant (e.g. 2) to suit exponential / logarithmic patterns. Another variant of the For loop in Java, also known as the ForEach loop in Visual Basic, uses iterators in a collection to determine the amount of looping. The former could be easily implemented by creating another implementation of For loop for multiplying the loop index by a constant, for example, however it would be more effective to specify a single For loop that can be evolved to perform addition, subtraction, multiplication or right-shifts on the index variable, in order to avoid duplication of effort. This, as well as the ForEach loop is a candidate for future research.

In this research, two different forms of the For loop were developed. The first is based on the basic version developed by Ciesielski and Li [5] and also presented by Montana as DoTimes [23], however there are a few differences.

Firstly, the version developed in this research contains an extra child for statement chaining, though this may seem trivial. The second change was to make its index variable available as a dynamic variable to the loop body, and only within the loop body. This scope extends to within the body of nested loops, and allows the loop to remain independent from any particular problem environment. This is much more significant, as previously loop indices were only made available by tailoring the loop implementation to the specific problem being solved.

For example, Kinnear copied the loop index into a single global variable, so that only the index of the innermost loop could be used when program execution was inside that loop [17]. It also becomes apparent from Ciesielski and Li’s work in sorting integers [5], that use of the loop index to access array elements was hard-coded into an ADF that combined the For loop and ‘test-then-swap’ function. In both instances, the implementation would

Return Type	Node Name	Children
`Void`	`ForNumItr`	`Void` next [`B`], `Int` numItr [`B`], `Void` body [`F`]
	Evaluates the loop body 'body' and 'numItr' while the index is less than 'numItr'.	
`Void`	ForRange	`Void` next [`B`], `Int` lowBnd [`B`], `Int` highBnd [`B`], `Int` inc [`B`], `Void` body [`F`]
	Initialises the index to 'lowBnd', and evaluates the loop body 'body', upper bound 'highBnd' and increment 'inc' while the index is less than 'highBnd'. The index is incremented by the most recent value of 'inc' evaluated.	

Table 4.2: Definitions of the two versions of `For` loop developed in this research.

have most likely been specialised to the problem being solved in order to make the index available as a variable.
The final change from previous implementations is the evaluation of the loop condition. In previous work [5, 17], the upper limit for the termination criteria is assumed to be constant over all iterations, therefore it is evaluated only once. Suppose the upper bound is given as the result of some algebraic function `fn(...)`. Then previous implementations would translate to C++ as:

```
for(int index = 0, int upper = fn(/* args... */); index < upper;
       index++)
{
   // Loop body
}
```

whereas the implementation developed in this research translates to C++ as:

```
for(int index = 0; index < fn(/* args... */); index++)
{
   // Loop body
}
```

Although it is not necessarily a good idea to change the upper limit during the loop execution, it is allowed in languages such as C/C++, it doesn't necessarily cause infinite loops, and it provides an alternative to modifying the loop index within the loop body. A disadvantage, however, is that it introduces the possibility of infinite loops, thus requiring enforcement of safety limits (Section 4.3).

The second version of the `For` loop developed increments the index through an integer range specified by a lower and upper bound, and provides a greater level of genericity through accepting arbitrary increments of the index variable. The closest to this implementation in previous work are the loops used by Ciesielski and Li [5, 6], and Kinnear [17]. In both cases the increment was fixed at one, and the evaluation of the upper bound was only performed once.

Table 4.2 defines both versions of the `For` loops developed for this research, including details of child constraints.

4.3 Dealing with Infinite Loops

Infinite loops are detected by using 'safety limits' on the number of iterations allowed by any particular loop, and rectified by breaking out of the loop. Thus, iterations are counted and compared against the safety limit statically-defined. If this count exceeds the safety limit, the looping mechanism will determine that the loop is infinite and break automatically.

In this scenario, the loop will be marked as being infinite so that the number of distinct infinite loops may be counted and used as part of the fitness function. Hence programs with fewer infinite loops would be more likely to be selected. It is worth mentioning that, if a program is generated with nested loops and the inner loop is found to be infinite, the marking and counting of this loop will ensure that the loop is counted only once overall – not once for each iteration of the outer loop.

Initial experiments with loops showed that even though safety limits reduced the overall evaluation time, the deep nesting of loops caused a major slowdown in performance. As a counter-measure, a global safety limit was defined, so that each iteration of every loop would contribute to the global iteration count. If the global count exceeds the global safety limit, all levels of nested loops are broken out of, and the program is flagged as being globally infinite. This flag is also available to the evaluation of a program's fitness to encourage evolution of more efficient programs. Since there are now two types of safety limit, the former, which applies to loops individually, will be hereafter referred to as local safety limits.

Chapter 5

Test Case 1: The Artificial Ant

This chapter describes the experiments performed to test the contributions made during the course of this research, using a variation of the Artificial Ant problem as the test case. Section 5.1 describes the test case in detail, and also outlines differences from the original Santa Fe implementation initially posed by Koza [19]. The experiment design of each individual experiment is then discussed in the following section. Finally, Section 5.3 presents the results and discusses the findings.

The implementations of the test cases presented in this chapter and the next chapter were performed on the RMITGP package originally developed at the Royal Melbourne Institute of Technology (RMIT). This package was chosen primarily for its support of STGP. The core GP engine was modified in order to implement most of the developments discussed in Chapter 3. A minor modification was also made to allow genetic programs to be output in C++ syntax, particularly for implementations designed to use statement chaining.

5.1 Overview of the Test Case

The Artificial Ant Problem was originally posed by Koza [19], and has since been adapted by Ciesielski and Li for their work on basic `For` loops [5]. This problem simulates an artificial ant on a grid, with the objective to navigate around the grid and eat all the food by moving into the corresponding cell containing each food item. The ant is limited to three basic actions - to move forward, turn left and turn right. The ant can also perceive whether the cell immediately in front of it contains food or not. This perception is used to influence a decision via branching in the program.

In much of the work in GP that uses the Artificial Ant problem as a test case, the food is laid out in what is known as the Santa Fe trail [19, 22]. This trail has no regular pattern contained within it, and food is distributed non-uniformly throughout the trail, i.e. there are numerous discontinuities of varying length within the trail. Ciesielski and Li [5] recognised that this layout was not conducive to using multiple loops, and therefore devised a more regular layout. This layout is shown in Figure 5.1. It is worth mentioning that in previous work, the ant trail is laid out on a toroidal grid, so that the ant wraps around to the opposite edge when it moves outside a given edge. In these experiments, however, the ant is assumed to remain on a square grid so that no wrap-around is possible.
The fitness function is defined as the number of cells with food still remaining after the ant has made its run [5, 22, 19]. Hence, a lower fitness value indicates fitter programs. A solution is identified when the fitness evaluates as zero.

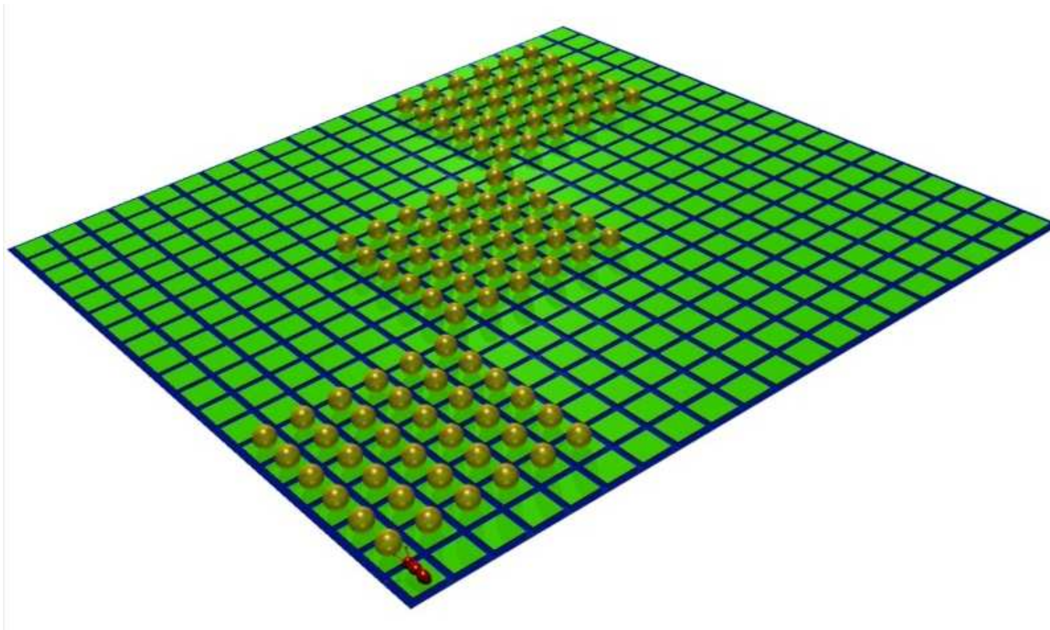


Figure 5.1: Visualisation of the Ant Trail problem, using the layout devised by Ciesielski and Li [5] and showing the ant in its starting position.

5.2 Experiment Design

In this section, each experiment for this test case is described in detail. The organisation of these experiments is designed to show a general progression of contributions included in the implementation. The first two serve as benchmarks, whereas the rest demonstrates the cumulative effects of each new contribution included.

These contributions are all based on using STGP and some kind of loop construct within the genetic programs. The progression of contributions included in the implementation is indicated by a series of 'levels'. The levels of implementation used in this test case are defined as follows:

- **Level 1**: STGP, Statement Chaining
- **Level 2**: Level 1 plus Additional Operators
- **Level 3**: Level 2 plus Child Constraints

The following sections describe each experiment in turn. These experiments correspond to the following implementations respectively: Koza, Ciesielski and Li, Level 1 with `For` loops, Level 2 with `For` loops, Level 3 with `For` loops, Level 3 with `While` loops, and Level 3 with both `For` and `While` loops.

Note that in each experiment, the evolution terminates after 1000 generations or when a solution is found. Each experiment uses the layout devised by Ciesielski and Li [5], shown in Figure 5.1.

<table>
<tr><th>Node Name</th><th>Children</th></tr>
<tr><td><code>Move</code></td><td>N/A</td></tr>
<tr><td colspan="2">Moves the ant forward one cell, eating any food present.
Assumes the ant is not at the edge of the grid, facing outwards.</td></tr>
<tr><td><code>TurnLeft</code></td><td>N/A</td></tr>
<tr><td colspan="2">Turns the ant 90° to its left.</td></tr>
<tr><td><code>TurnRight</code></td><td>N/A</td></tr>
<tr><td colspan="2">Turns the ant 90° to its right.</td></tr>
<tr><td><code>IfFoodAhead</code></td><td>trueBranch, falseBranch</td></tr>
<tr><td colspan="2">Evaluates 'trueBranch' if the cell immediately in front
contains food, otherwise evaluates 'falseBranch'.</td></tr>
<tr><td><code>Prog2</code></td><td>branch1, branch2</td></tr>
<tr><td colspan="2">Evaluates 'branch1', then 'branch2' sequentially.</td></tr>
<tr><td><code>Prog3</code></td><td>branch1, branch2, branch3</td></tr>
<tr><td colspan="2">Evaluates 'branch1', 'branch2' then 'branch3' sequentially.</td></tr>
</table>

Table 5.1: Functions and terminals of the Artificial Ant problem, as originally defined by Koza.

5.2.1 Experiment A: Koza Implementation

The default implementation posed by Koza [19, 22] generates programs without loops and runs them repeatedly in a static loop. Among the nodes used are `Prog2` and `Prog3` functions, which allow evaluation of multiple statements by evaluating their child subtrees sequentially. There is no generic implementation of the `If` construct. Instead, there is a specific `IfFoodAhead` function which evaluates whether the cell in front of the ant contains food, and evaluates the 'true' or 'false' subtree depending on the result. As a result, the full capabilities of STGP are not taken advantage of.

A full list of the functions and terminals used in this experiment are given in Table 5.1. Since this uses standard GP instead of STGP, no type information is given.

Many of the GP parameters of this experiment are based on those used by Ciesielski and Li [5] for a greater degree of consistency between experiments. In this experiment the population size was set to 200 programs. Each successive generation comprised of programs promoted by Elitism (2%), and created by Mutation (28%) and Crossover (70%). The maximum depth was set to 8. Programs in the initial population were generated using the Ramped Half-and-Half method.

This experiment aims to serve as a baseline against which all other experiments are evaluated. This allows a particular implementation to be evaluated according to how well it performs in relation to the Koza implementation, considering that human intervention was involved in specifying the looping in the GP environment. Therefore, implementations that perform better than this one would be considered to be doing exceedingly well.

5.2.2 Experiment B: Ciesielski & Li Implementation

This experiment also serves as a baseline to measure the remaining implementations against, except it becomes more important for implementations based on the contributions made in this research to perform better. The only differences from the Koza implementation are

Return Type	Node Name	Children
`Void`	`Move`	N/A
	Moves the ant into the cell in front, assuming the ant is not at the edge of the grid and facing that edge.	
`Void`	`TurnLeft`	N/A
	Turns the ant so that it is facing the cell that was on its left.	
`Void`	`TurnRight`	N/A
	Turns the ant so that it is facing the cell that was on its right.	
`Int`	`RandIntVal`	N/A
	Returns an explicit integer value determined on the node's creation.	
`Void`	`IfFoodAhead`	`Void` trueBranch, `Void` falseBranch
	The If construct. If 'cond' is true, then 'trueBranch' is evaluated; otherwise 'falseBranch' is evaluated.	
`Void`	`ForNumItr`	`Int` numItr, `Void` body
	Basic `For` loop, where the loop body is evaluated 'numItr' times.	
`Void`	`Prog2`	`Void` branch1, `Void` branch2
	Evaluates 'branch1', then 'branch2' sequentially.	
`Void`	`Prog3`	`Void` branch1, `Void` branch2, `Void` branch3
	Evaluates 'branch1', 'branch2' then 'branch3' sequentially.	

Table 5.2: Functions and terminals of the Artificial Ant problem, based on Ciesielski and Li's implementation [5].

the use of STGP for implementing the basic `For` loop, and the replacement of the implicit looping with the basic `For` loop as a function within the genetic programs.

Since this version of the `For` loop uses an integer to specify the number of iterations, a terminal returning an integer type is also included to maintain closure as required by STGP [23]. This returns an integer constant, which is randomly-selected when the node is created and initialised. In Ciesielski and Li's implementation, the range of values selected is defined to enforce a maximum number of iterations [5]. In this experiment, the range from which an integer constant is selected is given by the interval $[1, 20]$, since the maximum number of iterations would correspond to the length of a side of the grid.

The functions and terminals used in this experiment are redefined to be compatible with STGP, and listed in Table 5.2. The STGP return types used adopt the naming convention of the types defined in Section 3.1. All GP parameters are consistent with those in Experiment A, i.e. population size = 200, Elitism rate = 2%, Crossover rate = 70%, Mutation rate = 28%, maximum program depth = 8.

5.2.3 Experiment C: Level 1 Implementation with `For` Loops

This is the first experiment that uses statement chaining. The purpose is to allow the difference in performance between using statement chaining and using `ProgN` functions to be evaluated. In this implementation the `NOP` terminal is introduced, and all existing nodes of return type `Void` have been converted to functions by adding a child for the next statement in the chain (as described in Section 3.2). The `IfFoodAhead` function has been split into a generic `If` construct and a Boolean terminal, `IsFoodAhead`.

Table 5.3 lists all terminals and functions used in this implementation.

Return Type	Node Name	Children
`Void`	`NOP`	N/A
	Null operation. This is the only terminal of type `Void`.	
`Bool`	`IsFoodAhead`	N/A
	Returns true if the cell in front of the ant contains food.	
`Int`	`RandIntVal`	N/A
	Returns an explicit integer value determined on the node's creation. Values are selected from the range $[1, 20]$.	
`Void`	`IfCond`	`Void` next, `Bool` cond, `Void` truePart, `Void` falsePart
	The If construct. If 'cond' is true, then 'truePart' is evaluated; otherwise 'falsePart' is evaluated.	
`Void`	`ForNumItr`	`Void` next, `Int` numItr, `Void` body
	Basic `For` loop, where the loop body is evaluated 'numItr' times.	
`Void`	`Move`	`Void` next
	Moves the ant into the cell in front, assuming the ant is not at the edge of the grid and facing that edge.	
`Void`	`TurnLeft`	`Void` next
	Turns the ant so that it is facing the cell that was on its left.	
`Void`	`TurnRight`	`Void` next
	Turns the ant so that it is facing the cell that was on its right.	

Table 5.3: Functions and terminals of the Artificial Ant problem, using statement chaining. Refer to Section 3.2 for details of the 'next' child in functions of type `Void`.

All GP parameters in this experiment remain the same as in Experiments A and B, except for a slight modification to the fitness function. This modification involves adding a penalty variable D to the existing fitness function, based on whether the genetic program is empty (i.e. consisting of only a `NOP` terminal) or not. If the program is empty this variable has a value of 1, otherwise the value is set to zero. Thus, the updated fitness function becomes:

$$\text{fitness} = F + D$$

where F is the number of items of food left on the grid after the ant has finished its simulated run. This fitness function is also used in experiments D and E.

5.2.4 Experiment D: Level 2 Implementation with `For` Loops

In this experiment, additional genetic operators are introduced so that the effects of using such operators may be evaluated. The implementation is exactly the same as in Experiment C, except that the GP parameters have been adjusted to include the additional operators.

Since previous experiments used Elitism, Crossover and Mutation rates of 2%, 70% and 28% respectively, the additional operators are classified according to whether they are mutation-based or crossover-based and assigned a proportion of the original mutation and crossover rates respectively. Operations that are mutation-based do some form of random modification on a single program, without input from another program. Crossover-based operations use part of a second program as input into a modification performed on an individual. In this GP environment used, crossover-based operations mutually perform the operation on a pair of programs simultaneously.

Based on these definitions of mutation-based and crossover-based operators, most of the additional operators are classified as mutation-based operators. The mutation-based operators

Operator	Allocation Rate (%)
Elitism	2.0
Mutation	5.0
Insertion	7.0
Deletion	9.0
Replacement	7.0
Crossover	35.0
Crossover2	35.0

Table 5.4: Allocation of the genetic operators used in Experiment D.

<table>
<tr><th>Return Type</th><th>Node Name</th><th>Children</th></tr>
<tr><td rowspan="2">`Void`</td><td>`IfCond`</td><td>`Void` next [`B`], `Bool` cond [`B`], `Void` trueBranch [`F`], `Void` falseBranch [`B`]</td></tr>
<tr><td colspan="2">The If construct. Child constraints ensure that the true branch is non-empty.</td></tr>
<tr><td rowspan="2">`Void`</td><td>`ForNumItr`</td><td>`Void` next [`B`], `Int` numItr [`B`], `Void` body [`F`]</td></tr>
<tr><td colspan="2">Basic `For` loop, where child constraints ensure the loop body is non-empty.</td></tr>
</table>

Table 5.5: Functions of the Artificial Ant problem with child constraints applied. Child constraints are denoted by [`B`] and [`F`] for the `Both` and `Function` constraints respectively.

include Insertion, Deletion and Replacement. Crossover2, rather obviously, is a crossover-based operator.

Table 5.4 shows the rates of each genetic operator used, based on division of the original Crossover rate among all crossover-based operators, and division of the original Mutation rate among all mutation-based operators. Deletion was chosen to have a larger proportion of the population at the expense of Mutation, as an effort to reduce bloat in the genetic programs.

5.2.5 Experiment E: Level 3 Implementation with `For` Loops

This experiment aims to evaluate the effects of introducing child constraints, particularly for the `For` loop and `If` construct. The addition of child constraints aimed to serve two purposes – to simplify genetic programs by avoiding empty (and therefore redundant) loops and conditionals, and to improve the effectiveness of evolution so that solutions are found after fewer generations.

The functions with child constraints are shown in Table 5.5. All other functions and terminals are as defined in Experient C, with child constraints set to `Both` by default.

The GP parameters are the same as for Experiment D. To summarise, the genetic operators are distributed among the population with 2% Elitism, 5% Mutation, 7% Insertion, 9% Deletion, 7% Replacement, and 35% each for Crossover and Crossover2. Program depth remains limited to 8 levels deep, and the population size is fixed at 200 programs.

5.2.6 Experiment F: Level 3 Implementation with `While` Loops

In this experiment the use of the basic `For` loop is replaced with the `While` and `Do-While` loops (collectively referred to as `While` loops), and the GP environment is configured to deal with infinite loops. This involves modifying the fitness function to favour programs with fewer infinite loops, and not found to be globally infinite. Recall that the original fitness function used since Experiment C is:

$$\text{fitness} \;=\; F + D$$

where F is the number of items of food left on the grid after the ant has finished its simulated run, and D is the penalty of the program being empty. The new fitness used is given by:

$$\text{fitness} \;=\; F + 2I + 100G + D$$

where I is the number of loops in the program marked as infinite, and G is the penalty of the program being found to be globally infinite (1 if globally infinite, 0 otherwise).

Functions and terminals are similar to those used in Experiment E, except that the `ForNumItr` function and `RandIntVal` terminal have been removed. Apart from the introduction of the `While` loop, additional Boolean terminals have been defined. All new nodes introduced in this experiment are presented in Table 5.6.

<table>
<tr><th>Return Type</th><th>Node Name</th><th>Children</th></tr>
<tr><td rowspan="2">Bool</td><td>IsFoodLeft</td><td>N/A</td></tr>
<tr><td colspan="2">Returns True if there is uneaten food remaining on the grid.</td></tr>
<tr><td rowspan="2">Bool</td><td>CanMove</td><td>N/A</td></tr>
<tr><td colspan="2">Returns True if there is a cell in front of the ant, i.e. the ant is not at the edge of the grid and facing. outwards.</td></tr>
<tr><td rowspan="2">Void</td><td>WhileLoop</td><td>Void next [B], Bool cond [B], Void body [F]</td></tr>
<tr><td colspan="2">The While loop as presented in Section 4.1.</td></tr>
<tr><td rowspan="2">Void</td><td>DoWhileLoop</td><td>Void next [B], Bool cond [B], Void body [F]</td></tr>
<tr><td colspan="2">The Do-While loop as presented in Section 4.1.</td></tr>
<tr><td rowspan="2">Bool</td><td>LogAnd</td><td>Bool cond1 [B], Bool cond2 [B]</td></tr>
<tr><td colspan="2">Returns the logical AND of ‘cond1’ and ‘cond2’.</td></tr>
<tr><td rowspan="2">Bool</td><td>LogOr</td><td>Bool cond1 [B], Bool cond2 [B]</td></tr>
<tr><td colspan="2">Returns the logical OR of ‘cond1’ and ‘cond2’.</td></tr>
<tr><td rowspan="2">Bool</td><td>LogNot</td><td>Bool cond [B]</td></tr>
<tr><td colspan="2">Returns the logical inverse of ‘cond’.</td></tr>
</table>

Table 5.6: Additions to the set of functions and terminals used in Experiment F of the Artificial Ant problem. Child constraints are denoted by [B] and [F] for the `Both` and `Function` constraints respectively.

5.2.7 Experiment G: Level 1 Implementation with `For` and `While` Loops

The final experiment in this test case combines experiments E and F, thereby using both `For` loops and `While` loops. Since this experiment is based on a Level 3 implementation, it also makes use of child constraints, statement chaining and additional operators. The fitness

function is the same as that defined in Experiment F, in order to favour programs with minimal infinite looping. Table 5.7 presents the complete list of functions and terminals used in this experiment.

5.3 Results

This section presents the results of the experiments conducted and discusses the findings. Each experiment involved 20 test runs in order to evaluate a general trend. Table 5.8 gives a summary of the results of these experiments. This includes the mean fitness of the best program at Generation 1000, the success rate (i.e. proportion of test runs that yielded a solution), and the distribution of the number of generations evolved before a solution was found. The best program fitness of each experiment over 1000 generations, averaged across all 20 runs, is shown in Figure 5.2.
The following sections discuss comparisons between particular experiments in order to evaluate the effects of progressive changes made.

5.3.1 Existing Implementations versus Statement Chaining

From the results presented, it can be seen that the original Koza implementation is one of the most efficient configurations for solving this problem. However, this does not use loops as part of the genetic programs, rather it relies on human intervention to specify the looping in the program environment. This leads to the Koza implementation having an (unfair) advantage over implementations that evolved loops, but also defeats the purpose of genetic programming to minimise human specification of evolved programs. Nevertheless, it serves as an ideal benchmark to test the remaining implementations against.

The Ciesielski and Li implementation also serves as another benchmark for evaluating the utility gained by each successive contribution made as part of this research. It was interesting to note that this implementation was the least effective of all approaches, since the average best program fitness over all runs was the least fit when compared to all remaining implementations. This gives an indication that using statement chaining alone is an improvement on using `ProgN` statements and therefore creating bushy trees. A sample program evolved using Ciesielski and Li's method is as follows:

```
(ForNumItr 10
  (Prog2
    (ForNumItr 4
      (IfFoodAhead
        (ForNumItr 5
          (ForNumItr 12
            (IfFoodAhead move turnRight)))
        (Prog3
          (Prog3
            (Prog3 turnRight move move)
            (Prog3 move turnRight turnRight)
            turnRight)
          move
          (Prog3 move move
            (IfFoodAhead turnLeft turnRight)))))
```

Return Type	Node Name	Children
`Void`	`NOP`	N/A
	Null operation. This is the only terminal of type `Void`.	
`Bool`	`IsFoodAhead`	N/A
	Returns `True` if the cell in front of the ant contains food.	
`Bool`	`IsFoodLeft`	N/A
	Returns `True` if there is uneaten food remaining on the grid.	
`Bool`	`CanMove`	N/A
	Returns `True` if there is a cell in front of the ant, i.e. the ant is not at the edge of the grid and facing. outwards.	
`Int`	`RandIntVal`	N/A
	Returns an explicit integer value determined on the node's creation. Values are selected from the range $[1, 20]$.	
`Void`	`IfCond`	`Void` next [`B`], `Bool` cond [`B`], `Void` trueBranch [`F`], `Void` falseBranch [`B`]
	The If construct. If 'cond' is true, then 'truePart' is evaluated; otherwise 'falsePart' is evaluated.	
`Void`	`ForNumItr`	`Void` next [`B`], `Int` numItr [`B`], `Void` body [`F`]
	Basic `For` loop, where the loop body is evaluated 'numItr' times.	
`Void`	`WhileLoop`	`Void` next [`B`], `Bool` cond [`B`], `Void` body [`F`]
	The `While` loop as presented in Section 4.1.	
`Void`	`DoWhileLoop`	`Void` next [`B`], `Bool` cond [`B`], `Void` body [`F`]
	The `Do-While` loop as presented in Section 4.1.	
`Void`	`Move`	`Void` next [`B`]
	Moves the ant into the cell in front, assuming the ant is not at the edge of the grid and facing that edge.	
`Void`	`TurnLeft`	`Void` next [`B`]
	Turns the ant so that it is facing the cell that was on its left.	
`Void`	`TurnRight`	`Void` next [`B`]
	Turns the ant so that it is facing the cell that was on its right.	
`Bool`	`LogAnd`	`Bool` cond1 [`B`], `Bool` cond2 [`B`]
	Returns the logical AND of 'cond1' and 'cond2'.	
`Bool`	`LogOr`	`Bool` cond1 [`B`], `Bool` cond2 [`B`]
	Returns the logical OR of 'cond1' and 'cond2'.	
`Bool`	`LogNot`	`Bool` cond [`B`]
	Returns the logical inverse of 'cond'.	

Table 5.7: Functions and terminals used in Experiment G of the Artificial Ant problem.

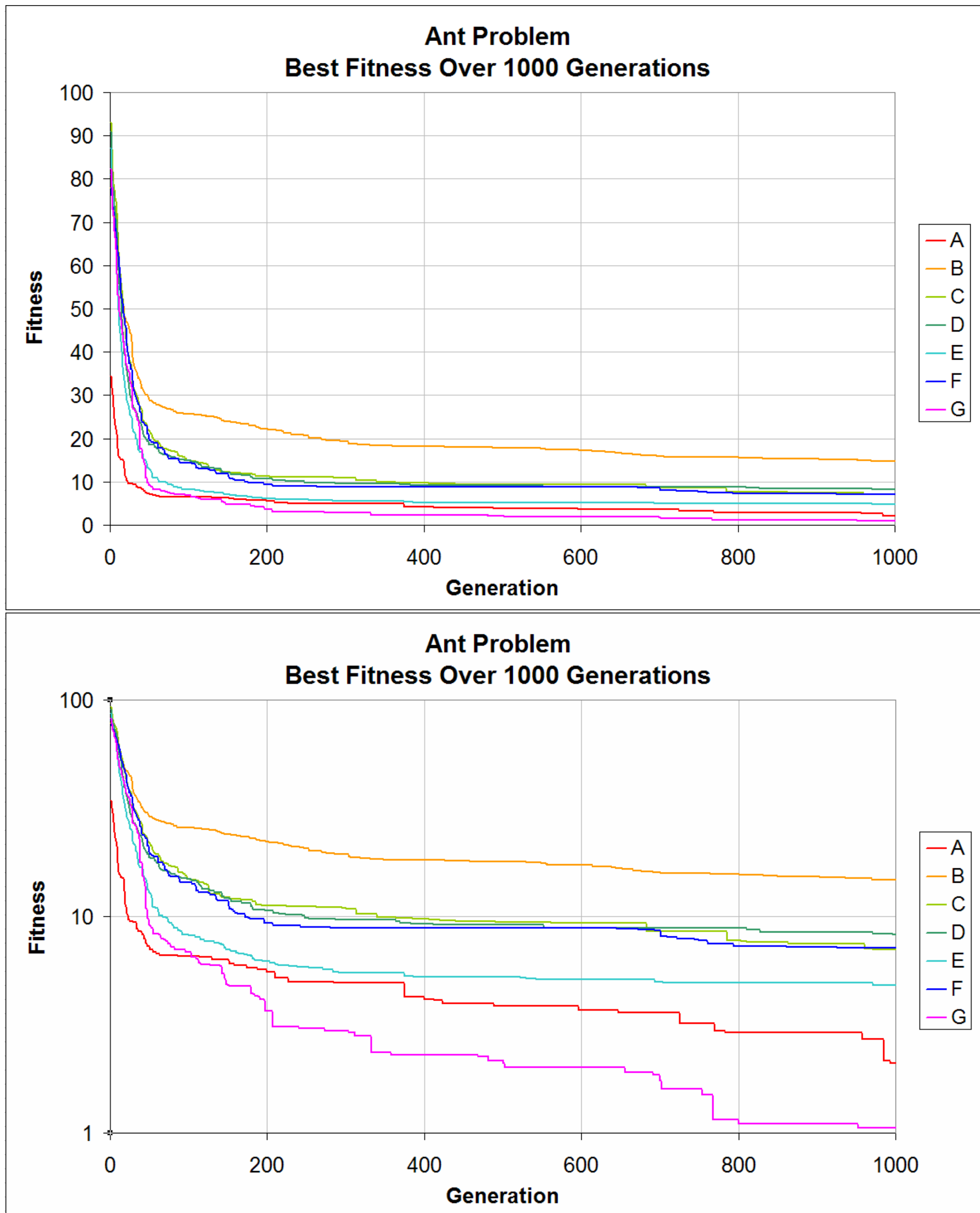


Figure 5.2: Average best program fitness over 1000 generations for the Artificial Ant problem. Refer to Table 5.8 for a brief explanation for the labels of the experiments.

Experiment	Mean Fitness, Gen. 1000	Success Rate (%)	#Gen of solutions	
			Mean	S.D. (n-1)
A: Koza	2.1	60	279	321.9
B: Ciesielski & Li	14.8	35	111	116.9
C: STGP, Lvl 1	7.1	25	244	321.4
D: STGP, Lvl 2	8.3	30	265	216.1
E: STGP, Lvl 3 (For)	4.8	30	115	101.1
F: STGP, Lvl 3 (While)	7.2	45	150	239.3
G: STGP, Lvl 3 (For + While)	1.1	75	197	216.8

Table 5.8: Results of the Artificial Ant problem.

```
(Prog3
  move
  (IfFoodAhead
    (IfFoodAhead turnRight (ForNumItr 1 move))
    (ForNumItr 1 move))
  turnLeft))}
```

In contrast, a sample program evolved using the statement chaining method developed in this research is presented here:

```
turnRight();
for(int index1 = 0; index1 < 20; index1++)
{
     for(int index2 = 0; index2 < 14; index2++)
     {
          for(int index3 = 0; index3 < 1; index3++)
          {
               for(int index4 = 0; index4 < 3; index4++)
               {
                    if(IsFoodAhead)
                    {
                         turnLeft();
                    } else
                    {
                         for(int index5 = 0; index5 < 12; index5++)
                         {
                         }
                    }
               }
               turnRight();
               if(IsFoodAhead)
               {
               } else
               {
               }
          }
          move();
     }
```

```
    move();
    move();
    turnLeft();
    for(int index2 = 0; index2 < 1; index2++)
    {
    }
}
if(IsFoodAhead)
{
} else
{
    turnLeft();
    for(int index1 = 0; index1 < 1; index1++)
    {
        if(IsFoodAhead)
        {
            turnLeft();
        } else
        {
            for(int index2 = 0; index2 < 14; index2++)
            {
            }
        }
    }
    turnRight();
    move();
}
```

5.3.2 Introducing Child Constraints

From the C++ example previously shown, it can be seen that there are many inefficiencies caused by empty loops and `If` constructs. This was the motivation behind introducing child constraints. After introducing child constraints, it was found that solutions were evolved more effectively. This is evident by the best programs on average (after 1000 generations) being significantly fitter with child constraints, than programs generated without using child constraints. From Table 5.8, the average number of generations required to find a solution using child constraints was lower than without child constraints. Furthermore, the distribution among test runs was more consistent.

An example of a program evolved using child constraints is as follows:

```
turnRight();
for(int index1 = 0; index1 < 15; index1++)
{
    for(int index2 = 0; index2 < 18; index2++)
    {
        for(int index3 = 0; index3 < 15; index3++)
        {
            if(IsFoodAhead)
            {
```

```
                move();
                move();
            } else
            {
            }
        }
        turnRight();
    }
    move();
    turnRight();
    turnRight();
}
for(int index1 = 0; index1 < 15; index1++)
{
    for(int index2 = 0; index2 < 15; index2++)
    {
        if(IsFoodAhead)
        {
            move();
        } else
        {
        }
    }
    turnRight();
}
move();
move();
turnRight();
```

5.3.3 Introducing Additional Operators

After comparing the results of experiments C and D, it can be seen that the use of additional genetic operators seemed to have little effect on the performance of evolution. While performance seemed to be worse than using only the standard genetic operators, it is likely that this is due to random variation. Little can be concluded about the distribution of generations required to find a solution, as the success rate was low.

It is likely that the allocation of each genetic operator was poor. This was designed to mimic the allocations of the standard operators in Experiment C, based on classification as either mutation-based or crossover-based operators. Despite this intention the low allocation of mutation-based operators did not give the Insertion, Deletion and Replacement operators enough influence on the successive populations. In hindsight, each of the rates of Insertion, Deletion and Replacement should have been significantly higher than the average of 7% of the population. As it stands, the effects of these additional operators need to be evaluated further to determine an optimum allocation of these operators. This is presented as an idea for future work in Section 7.2.

5.3.4 `For` loops versus `While` loops

It was noticed that the `While` caused a slowdown in program evaluation because of the presence of infinite loops, even though safety limits were set. At the start of this research a possibility was explored for using program analysis techniques to detect infinite loops and automatically correct them. This would require a scheme for engineering the loop body to eventually cause the condition to break out of the loop.

For this test case, such an implementation would be extremely difficult to implement, as the problem is defined in terms of user-specified functions specific to the Artificial Ant problem. It is therefore impossible to guarantee that executing such functions (e.g. `TurnLeft`) will cause the internal state to change so that the loop condition (e.g. `( !(IsFoodAhead()) )`) will eventually evaluate false, without requiring these functions to be analysed internally. Such measures to improve the chances of infinite loops being avoided, using program analysis techniques, would be an ideal topic for extended research (see Section 7.2).

5.4 Chapter Summary

In this chapter the Artificial Ant problem was defined, and particular variations from the original Koza implementation were discussed. A schedule of the experiments within this test case was presented along with particular GP parameter settings.

Results of the experiments were also presented, together with a discussion of each particular contribution progressively introduced. The overall findings showed that statement chaining and child constraints were particularly beneficial to this test case, when compared with existing implementations as benchmarks. The use of both the `For` and `While` loops resulted in the largest success, with evolutionary performance being significantly better than the original Koza implementation using implicit looping.

Chapter 6

Test case 2: Factorial Calculation

In this chapter, details of experiments conducted using the Factorial test case are presented. Section 6.1 gives an overview of the test case. In Section 6.2, the design of each experiment is described in turn. Finally, Section 6.3 presents the results and discusses the findings. This discussion also compares related findings from the Artificial Ant test case presented in Chapter 5.

6.1 Overview of the Test Case

Factorial Calculation is a regression problem where GP is used to evolve a function that calculates a factorial of a given integer. Each program is evaluated using the input range $[0, 5]$ in order to capture the factorial relationship as succinctly as possible, while providing enough distinction from an exponential relationship. For reference, the definition of a factorial of x is:

$$x! = \prod_{i=1}^{x} i$$

In this test case, two inherent variables, `res` and `temp`, are used for storing the result and a temporary value respectively. These are both reset to zero at the start of evaluation for each program. It is expected that the `temp` variable will become redundant once the loop index is made available as a dynamic variable (Experiment K). The input value is set by the program environment prior to evaluation of a particular factorial.

The fitness is evaluated by evaluating each genetic program on all values in the input range, and calculating the total sum of the squared proportional error over each output result. Hence fitter programs have a lower fitness value, and the problem is considered to be solved when the fitness reaches zero. The fitness can be expressed by the equation:

$$\text{fitness} \; = \; \sum_{i=0}^{5} (\frac{100(i! - f(i))}{i!})^2$$

where $f(i)$ is the result returned after evaluating the genetic program.

Since the `For` loops re-evaluate the termination case at each iteration, it is possible for infinite loops to occur. This is because the `res` and `temp` variables' values may increase inside the loop, and the re-evaluation would update the stopping criteria accordingly. Thus the

loop range would be ever-expanding, with the index variable being unable to 'catch up'. This prompted at least a local safety limit to be set. The fitness function was updated accordingly to now become:

$$\text{fitness} \ = \ 1000I + 100000G + \sum_{i=0}^{5}(\frac{100(i! - f(i))}{i!})^2$$

where I is the number of infinite loops detected, and G is the 'globally infinite' indicator, which adopts the value 1 if the program is found to be globally infinite (0 otherwise).

A limited range of input values was chosen for evaluation so that the basic factorial relationship could be distinguished from other functions (such as a polynomial) and applied to larger numbers. It would also avoid larger factorials reducing the contribution of smaller factorials to overall fitness. Initial tests showed that using a larger input range caused the evolution to focus on minimising the extremely large differences for the higher factorials, thus causing programs to be evolved so that fitting the function to the lower data points was overlooked.

6.2 Experiment Design

This section describes each experiment of this test case in detail. The organisation of these experiments is designed to show a general progression of contributions included in the implementation, in a similar fashion to the Artificial Ant test case (Chapter 5). Since no prior implementation of this test case has been found, these experiments will only consider the effects of each contribution progressively included in the implementation.

These contributions are all based on using STGP and some kind of loop construct within the genetic programs. The progression of contributions included in the implementation is indicated by a series of 'levels' based on those defined in Section 5.2. The levels of implementation used in this test case are defined as follows:

- **Level 1**: STGP, Statement Chaining
- **Level 2**: Level 1 plus Additional Operators
- **Level 3**: Level 2 plus Child Constraints
- **Level 4**: Level 3 plus Dynamic Variables

The following sections describe each experiment in turn. These experiments correspond to the following implementations respectively: Level 1 with `For` loops, Level 2 with `For` loops, Level 3 with `For` loops, Level 4 with `For` loops and no 'temp' variable, Level 4 with `For` loops and one 'temp' variable, and Level 4 with `While` loops and two 'temp' variables.

Note that in each experiment, the evolution terminates after 1000 generations or when a solution is found. The initial population of each experiment is generated using a ramped method, with 5% of the programs being generated via the Full program method. Each experiment involves 20 evolutionary runs and the average (mean) results are used in analysis.

6.2.1 Experiment H: Level 1 Implementation with `For` Loops

This experiment serves as a baseline implementation, against which later experiments are measured. As a consequence, the implementation only makes use of statement chaining. The population consists of 600 programs, with Elitism rate = 5%, Mutation rate = 45% and Crossover rate = 50%. The maximum depth was set to 6. Infinite loops were resolved by setting the local safety limit to 32 iterations, though no global safety limits were defined. Functions and terminals used in this experiment are presented in Table 6.1.

Return Type	Node Name	Children
`Void`	`NOP`	N/A
	Null operation. This is the only terminal of type `Void`.	
`Int`	`GetInput`	N/A
	Returns the value of the input.	
`Int`	`GetTemp`	N/A
	Returns the value of the 'temp' variable.	
`Int`	`GetRes`	N/A
	Returns the value of the 'res' variable.	
`Int`	`ZeroIntVal`	N/A
	Returns the number zero (0).	
`Int`	`OneIntVal`	N/A
	Returns the number one (1).	
`Void`	`SetTemp`	`Void` next, `Int` val
	Sets the temporary variable to the integer 'val'.	
`Void`	`AddTemp`	`Void` next, `Int` val
	Adds the integer 'val' to the temporary variable.	
`Void`	`SetRes`	`Void` next, `Int` val
	Sets the result to the integer 'val'.	
`Void`	`MultRes`	`Void` next, `Int` val
	Multiplies the result by the integer 'val'.	
`Void`	`ForNumItr`	`Void` next, `Int` numItr, `Void` body
	Multiplies the result by the integer 'val'.	

Table 6.1: Functions and terminals of the Factorial problem, using statement chaining. Refer to Section 3.2 for details of the 'next' child in functions of type `Void`.

6.2.2 Experiment I: Level 2 Implementation with `For` Loops

In this experiment, the effects of introducing the additional operators (Insertion, Deletion, Replacement and Crossover2) are investigated. Unlike the Artificial Ant problem, the allocation of these operators is more evenly split. Table 6.2 shows the allocation of each genetic operator for this experiment. All other parameters are the same as in Experiment H.

6.2.3 Experiment J: Level 3 Implementation with `For` Loops

The focus of this experiment is to evaluate the effects of adding child constraints. Child constraints were applied to the `For` loop in the same fashion as Experiment E (Section 5.2.5). In particular, child constraints of `Both` were used by default, except for the body of loops which used a child constraint of `Function`. All other GP parameters were consistent with those in Experiment I.

Operator	Allocation Rate (%)
Elitism	1.67
Mutation	15.00
Insertion	16.67
Deletion	16.67
Replacement	16.67
Crossover	16.67
Crossover2	16.67

Table 6.2: Allocation of the genetic operators used in Experiment I.

6.2.4 Experiment K: Level 4 Implementation with `For` Loops, No 'temp' Variable

From this experiment onwards, dynamic variables are able to be used within the genetic programs evolved. This has prompted a new set of functions and terminals to be defined to allow genericity, therefore many of the previous functions and terminals have become redundant. The functions and terminals used in this experiment are listed in Table 6.3. A significant point to note is that none of the functions or terminals is specific to the Factorial test case in particular, i.e. the functions and terminals are completely generic and independent of the implementation.

For this experiment, both versions of the `For` loop make their loop indices available as read-only dynamic variables to the rest of the genetic program within the loop body. This ensures that the loop index cannot be altered by the genetic program itself, but only by the loop internally. The `res` variable is assigned as the default read-write variable at reference index 0. By definition from Section 3.5, this variable is also assigned as the default variable in the read-only pool. Since no `temp` variable is defined, the rest of the reference indices are used by the loop indices of `For` loops. The removal of the `temp` variable is intended to encourage evolution of factorial functions that use a loop index.

Since the addition operator (+) is used, initial tests have showed that it introduces a greater deal of complexity into the genetic programs, thus causing a slow-down of the evolutionary process. To alleviate this, the population size was reduced to 200 programs. All remaining GP parameters are the same as in Experiments I and J.

The allocation of genetic operators is as defined in Table 6.2. Any error of allocation arising from truncation to integers is absorbed by the Elitism rate. Since the population size is 200, the errors from truncation result in the absolute allocation of programs being as follows: 5 from Elitism, 30 from Mutation, and 33 each from Insertion, Deletion, Replacement, Crossover and Crossover2.

6.2.5 Experiment L: Level 4 Implementation with `For` Loops, One 'temp' Variable

This experiment differs from Experiment K only by the inclusion of a read-write `temp` variable, assigned to index 1 of the read-write integer pool (and hence index 1 of the read-only pool). The implementation is otherwise identical to that of Experiment K. This shows how dynamic variables can lead to a more generic implementation that can be configured efficiently. Contrast this with the previous implementations up to Level 3 – each different variable required its own set of functions and terminals to read and modify its value, lead-

<table>
<tr><th>Return Type</th><th>Node Name</th><th>Children</th></tr>
<tr><td rowspan="2">Void</td><td>NOP</td><td>N/A</td></tr>
<tr><td colspan="2">Null operation. This is the only terminal of type Void.</td></tr>
<tr><td rowspan="2">Int</td><td>GetIntVal</td><td>N/A</td></tr>
<tr><td colspan="2">Returns the value of a dynamic integer variable, either read-only or read-write.
This terminal is displayed as the name of the variable in GP output.</td></tr>
<tr><td rowspan="2">IntPtr</td><td>IntVarRef</td><td>N/A</td></tr>
<tr><td colspan="2">Returns the reference of a dynamic integer variable from the read-write pool.
This terminal is displayed as the name of the variable in GP output.</td></tr>
<tr><td rowspan="2">Int</td><td>ZeroIntVal</td><td>N/A</td></tr>
<tr><td colspan="2">Returns the number zero (0).</td></tr>
<tr><td rowspan="2">Int</td><td>OneIntVal</td><td>N/A</td></tr>
<tr><td colspan="2">Returns the number one (1).</td></tr>
<tr><td rowspan="2">Void</td><td>AssignInt</td><td>Void next [B], IntPtr varRef [B], Int val [B]</td></tr>
<tr><td colspan="2">The = assignment operator. Stores the integer ‘val’ in the variable referenced by ‘varRef’.</td></tr>
<tr><td rowspan="2">Void</td><td>AddToInt</td><td>Void next [B], IntPtr varRef [B], Int val [B]</td></tr>
<tr><td colspan="2">The += assignment operator. Increments the variable referenced by ‘varRef’ by the integer ‘val’.</td></tr>
<tr><td rowspan="2">Void</td><td>MultByInt</td><td>Void next [B], IntPtr varRef [B], Int val [B]</td></tr>
<tr><td colspan="2">The *= assignment operator. Multiplies the variable referenced by ‘varRef’ by the integer ‘val’.
The product is stored in the variable referenced by ‘varRef’.</td></tr>
<tr><td rowspan="2">Void</td><td>ForNumItr</td><td>Void next [B], Int numItr [B], Void body [F]</td></tr>
<tr><td colspan="2">Evaluates the loop body ‘body’ and ‘numItr’ while the index is less than ‘numItr’.</td></tr>
<tr><td rowspan="2">Void</td><td>ForRange</td><td>Void next [B], Int lowBnd [B], Int highBnd [B], Int inc [B], Void body [F]</td></tr>
<tr><td colspan="2">The more generic form of For loop presented in Section 4.2.</td></tr>
<tr><td rowspan="2">Int</td><td>AddInt</td><td>Int val1 [B], Int val2 [B]</td></tr>
<tr><td colspan="2">Returns the sum of ‘val1’ and ‘val2’.</td></tr>
</table>

Table 6.3: Functions and terminals of the Factorial problem, using dynamic variables.

ing to a greater amount of redundancy. Therefore, if an additional variable was needed to be defined, another set of redundant functions and terminals would need to be created.

Aside from showing the benefits of keeping the implementation generic, this experiment aims to investigate the effects of introducing more functionality than is required. In this case, the extra functionality is the availability of an extra variable that wasn't present in Experiment K, as the factorial calculation could be performed using only loop indices.

6.2.6 Experiment M: Level 4 Implementation with `While` Loops, Two 'temp' Variables

In this final experiment, the `For` loops have been replaced with `While` loops (including `Do-While` loops) in order to compare the effects of one particular kind of loop over another. Unlike the Artificial Ant problem, this experiment is based on a Level 4 implementation in order to avoid the redundancy of adding new functions to access a second `temp` variable, `temp2`.

A second `temp` variable was included to keep the implementation similar to Experiment L, knowing that the `While` loop could be engineered to emulate a `For` loop. This means that one `temp` variable would be used as the index of a `For` loop, leaving a spare variable available in a similar fashion to Experiment L.

The functions and terminals for this test case are given in Table 6.4.

6.3 Results

This section presents the results of the experiments conducted and discusses the findings. As previously mentioned, each experiment involved 20 test runs in order to evaluate a general trend. Table 6.5 gives a summary of the results of these experiments. This includes the mean fitness of the best program at Generation 1000, the success rate (i.e. proportion of test runs that yielded a solution), and the distribution of the number of generations evolved before a solution was found. The best program fitness of each experiment over 1000 generations, averaged across all 20 runs, is shown in Figure 6.1.

In this test case, the experiments generally evolved much faster and more effectively than those in the Artificial Ant test case. With the exception of Experiment M, the experiments generally achieved a higher success rate of evolving solutions than similar experiments in Test Case 1. This could be attributed to the Factorial calculation being an easier problem.

The rest of this section presents a discussion of the differences between the results obtained. This is organised according to the progressive changes that were made to the implementations.

6.3.1 Introducing Additional Operators

Unlike the results of introducing additional operators into Test Case 1, the introduction of these operators for the Factorial problem was much more successful. This introduction led to the best performance overall, with all 20 runs finding a solution in under 200 generations. Experiment A had one run that could not find a solution after 1000 generations.

Return Type	**Node Name**	**Children**
`Void`	`NOP`	N/A
	Null operation. This is the only terminal of type `Void`.	
`Int`	`GetIntVal`	N/A
	Returns the value of a dynamic integer variable, either read-only or read-write. This terminal is displayed as the name of the variable in GP output.	
`IntPtr`	`IntVarRef`	N/A
	Returns the reference of a dynamic integer variable from the read-write pool. This terminal is displayed as the name of the variable in GP output.	
`Int`	`ZeroIntVal`	N/A
	Returns the number zero (0).	
`Int`	`OneIntVal`	N/A
	Returns the number one (1).	
`Void`	`AssignInt`	`Void` next [`B`], `IntPtr` varRef [`B`], `Int` val [`B`]
	The = assignment operator. Stores the integer 'val' in the variable referenced by 'varRef'.	
`Void`	`AddToInt`	`Void` next [`B`], `IntPtr` varRef [`B`], `Int` val [`B`]
	The += assignment operator. Increments the variable referenced by 'varRef' by the integer 'val'.	
`Void`	`MultByInt`	`Void` next [`B`], `IntPtr` varRef [`B`], `Int` val [`B`]
	The `*`= assignment operator. Multiplies the variable referenced by 'varRef' by the integer 'val'. The product is stored in the variable referenced by 'varRef'.	
`Void`	`While`	`Void` next [`B`], `Bool` cond [`B`], `Void` body [`F`]
	Evaluates the loop body 'body' and condition 'cond' while 'cond is true. The condition is evaluated before the body, therefore the body may not necessarily be evaluated at all.	
`Void`	`DoWhile`	`Void` next [`B`], `Bool` cond [`B`], `Void` body [`F`]
	Evaluates the loop body 'body' and condition 'cond' while 'cond is true. The condition is evaluated after the body, therefore the body is guaranteed to be evaluated at least once.	
`Bool`	`LTInt`	`Int` val1 [`B`], `Int` val2 [`B`]
	Returns True if 'val1' is less than 'val2'.	
`Bool`	`LTEInt`	`Int` val1 [`B`], `Int` val2 [`B`]
	Returns True if 'val1' is less than or equal to 'val2'.	
`Int`	`AddInt`	`Int` val1 [`B`], `Int` val2 [`B`]
	Returns the sum of 'val1' and 'val2'.	

Table 6.4: Functions and terminals of the Factorial problem, using `While` loops exclusively.

Experiment	Mean Fitness, Gen. 1000	Success Rate (%)	Mean #Gen of solutions	
			Mean	Std Dev.
H: Lvl 1	113.3	95	69	78.7
I: Lvl 2	0.0	100	65	51.3
J: Lvl 3	534.7	90	83	71.7
K: Lvl 4 (0 temp vars)	305.3	75	140	225.9
L: Lvl 4 (1 temp var.)	291.0	65	268	268.1
M: Lvl 4 (While)	1159.9	0	–	–

Table 6.5: Results of the Factorial problem.

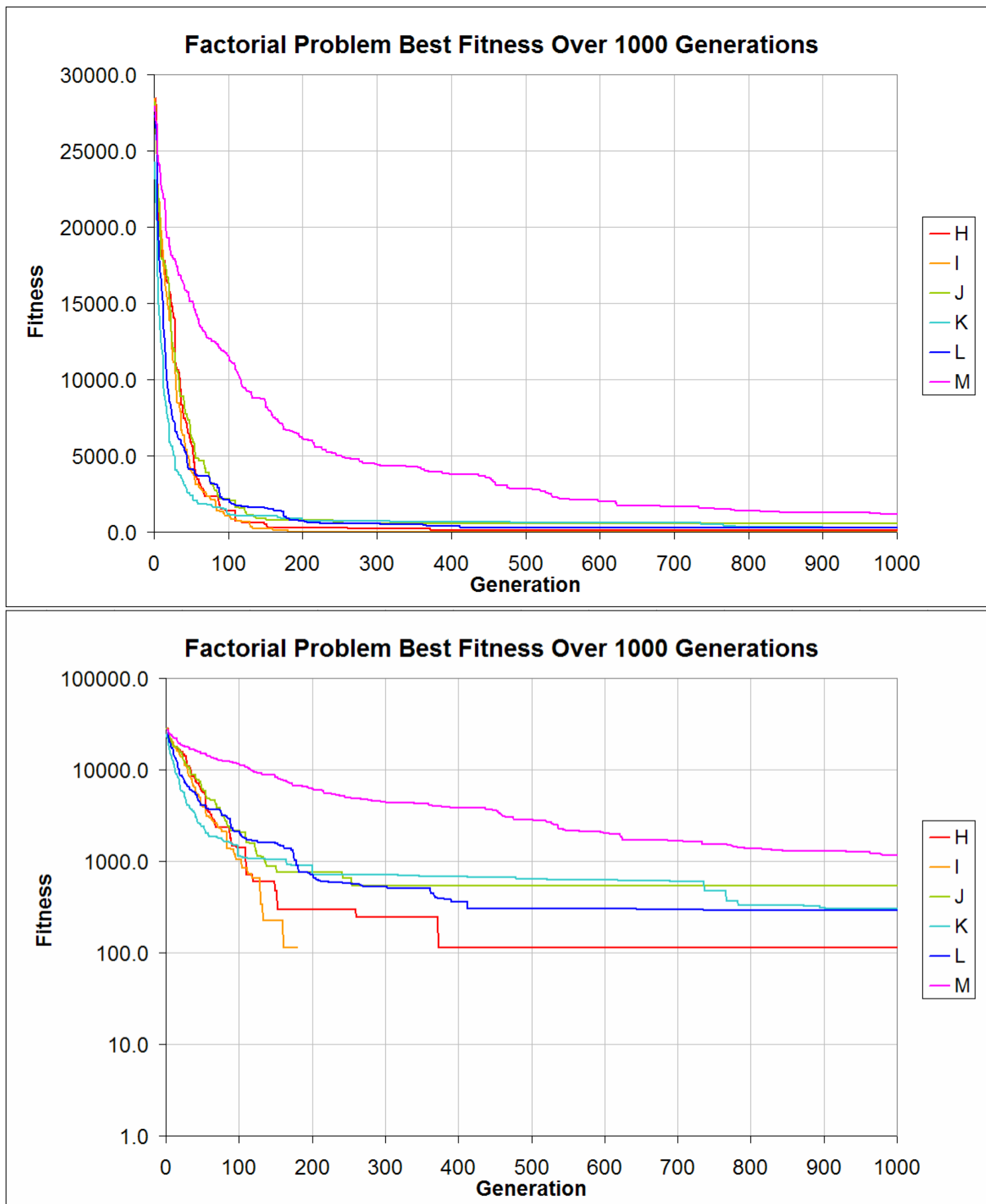


Figure 6.1: Average best program fitness over 1000 generations for the Factorial problem. Refer to Table 6.5 for a brief explanation for the labels of the experiments.

The comparative success of the introduction of additional operators producing a performance increase can be attributed also to a fairer distribution of operations. With the exception of Elitism, all remaining operators had a near-equal share. This allowed the additional mutation-based operators (Insertion, Deletion and Replacement) to have more impact than they did for Test Case 1.

6.3.2 Introducing Child Constraints

As mentioned in Test Case 1, the introduction of child constraints aimed to improve effectiveness of program evolution. This included raising the level of code quality by removing a particular type of redundancy. To illustrate this, a sample solution from Experiment I is presented as follows:

```
SetRes(1);
for(int index1 = 0; index1 < temp; index1++)
{
}
AddTemp(inputVal);
SetTemp(1);
for(int index1 = 0; index1 < inputVal; index1++)
{
    MultRes(temp);
    AddTemp(1);
}
MultRes(1);
```

In this example, it can be seen that there are redundancies arising from a lack of child constraints. Note that the first `For` loop is empty and therefore does not contribute to the solution. As a result, this unnecessarily makes the overall program deeper than it needs to be.

After introducing child constraints, the evolved solutions had no evidence of this kind of redundancy occurring. This can be seen in a sample solution output from Experiment J:

```
SetRes(1);
for(int index1 = 0; index1 < inputVal; index1++)
{
    AddTemp(1);
    MultRes(temp);
}
AddTemp(res);
SetTemp(res);
AddTemp(inputVal);
```

It was surprising, yet impressive, to see a program evolved without any redundancies whatsoever:

```
SetRes(1);
for(int index1 = 0; index1 < inputVal; index1++)
{
    AddTemp(1);
    MultRes(temp);
}
```

Despite the removal of a particular type of redundancy, the performance of evolution was actually worse than in Experiment I. This could be the result of random variation, since the distribution of generations taken to evolve a solution was more widespread. Another factor could be the relative difficulty involved in selecting suitable nodes in particular genetic operations. However, this result wasn't consistent with the performance increase in the Artificial Ant problem, therefore this could be either the result of random variation or the nature of the problem being solved.

6.3.3 Introducing Dynamic Variables

The introduction of dynamic variables allowed the evolved programs to be expressed in more natural C++. In particular, the use of generic functions based on arithmetic and assignment operators removed the need for specialised functions to be defined. A sample program evolved, also without any redundancies, is illustrated as follows:

```
res += 1;
for(int index = 0; index < inputVal; index++)
{
    res *= index + 1;
}
```

Since generic C++ operators are used, these output programs could be easily inserted into a wrapper function. A different example from Experiment L is shown embedded in a wrapper function:

```
int factorial (int inputVal)
{
    int res = 0;

    //GP output starts here

    res = inputVal;
    for(int index = inputVal; index <= res; index += 1 )
    {
        for(int index2 = 0; index2 < index; index2++)
        {
            res += 1;
        }
        res = 1;
    }
    for(int index = 1; index <= inputVal; index += 1 )
    {
        res *= index;
    }

    //End of GP output

    return res;
}
```

It was found that introducing dynamic variables yielded similar results to the previous implementation in Experiment J. Experiment L, which reinstated the `temp` variable, also

seemed to have little overall change in performance. However, both Experiments K and L had a lower success rate at evolving solutions within 1000 generations. Part of this could be a result of the generic operators allowing particular assignment operations to be performed on particular variables, which weren't possible using the function and terminal set of Experiment J. Such functionality included adding to `res` and multiplying any `temp` variables by a number.

6.3.4 Using the `While` loop

Using `While` loops exclusively showed how difficult it can be to co-ordinate the use of variables. As is evidenced by the results, no test run was able to find a solution. In addition to this, the performance was the worst out of all implementations in this test case. This result shows that `While` loops are not necessarily suited to all types of problem being solved, even though they are a generalisation of `For` loops. It therefore requires much more work to force Genetic Programming to specialise these loops, than to have this specialisation already defined.

6.4 Chapter Summary

In this chapter the Factorial problem was defined in terms of being a regression problem. A schedule of the experiments within this test case was presented along with their respective GP parameter settings.

Results of the experiments were also presented, together with a discussion of each particular contribution progressively introduced. The overall findings showed that introducing the additional genetic operators was able to improve evolutionary performance, however the use of child constraints eroded that advantage somewhat. The introduction of dynamic variables allowed a greater level of genericity in the functions and terminals, and hence improved output program representation. Finally, the use of `While` loops was shown to be disadvantageous in this test case as a result of the need to evolve loop specialisation to mimic `For` loops.

Chapter 7

Conclusions

In this research a formal structure has been devised for representing genetic programs as imperative programs, which now show a greater amount of resemblance to human-generated programs, especially when output as C++ programs. This was engineered through the concept of statement chaining and use of Strongly-Typed Genetic Programming. Furthermore, the Artificial Ant problem showed that this formal structure can lead to a more efficient evolution of solutions than previous implementations using `Prog2` and `Prog3` statements. Thus it becomes evident that the high-level goal was achieved.

Particular enablers of achieving this main goal were language features developed and enhanced, including `While` and `Do-While` loops, generic `If` statements and dynamic variables. The enforcement of child constraints added to a higher level of code quality. Changing the output representation of genetic programs to C++ also allowed similarities to be seen with a large range of human-generated programs.

The rest of this chapter discusses specific conclusions made in relation to the research questions posed, and outlines possibilities for future research in a number of related subject areas within Genetic Programming.

7.1 Specific Conclusions

The adoption of statement chaining has allowed an effective representation of control structures including loops and `If` blocks to be defined. This is due to the tree structure easily being able to be translated into languages such as C++, an imperative language that is widely used by human programmers. These control structures can be evolved more efficiently through the use of child constraints, as evidenced by results of the Artificial Ant problem, and through measures taken to deal with infinite loops. Furthermore, the use of STGP has allowed these constructs to be kept generic, and easily transportable between different GP test cases.

It was acknowledged that it can be extremely difficult to avoid infinite loops completely in genetic programs, though some measures can be taken to reduce the likelihood of infinite loops occurring. The specification of child constraints, particularly for the loop body of a `While` loop, eliminates one source of infinite loop – the empty `While` loop. This makes no assumptions about whether the evaluation of the condition results in state changes, which cause the loop condition to evaluate false and break out of the loop.

Another measure taken to discourage infinite loops was the detection of the infinite loop and its possible effects on program fitness. The detection of infinite looping was performed at a per-loop level and per-program level via the use of safety limits. These forced infinite loops to break, and contributed to statistics on infinite looping that were able to be taken advantage of by the fitness function. Therefore, the presence of infinite looping resulted in poorer program fitness, thus encouraging programs to evolve without infinite loops. Though it wasn't formally tested in experiments, this strategy led to faster evolution times, as expected, particularly with the introduction of the global safety limit.

The relative performance of a test case, based on inclusion and exclusion of a particular construct, was found to be dependent on the nature of the particular test case itself. For the Artificial Ant problem, implementations that used only `For` loops achieved reasonably better evolutionary performance than those using `While` loops exclusively, holding all other factors constant. Yet the implementation using both `For` and `While` loops had even better performance than the original Koza implementation. For the Factorial problem, exclusive use of the `For` loop was much more effective than using `While` loops exclusively, which in turn had the worst performance overall. These results show that the different types of loops are better-suited to particular environments. The `For` loop thrived on counting tasks, such as providing indexing for factorial calculation or counting cells to form a pattern on a grid for ant navigation. The `While` loop was better-suited to simple Boolean conditions that had a small search space, such as percepts of an artificial ant, than trying to evolve numeric comparisons for performing the job of a `For` loop.

The evolutionary process was accelerated by a number of factors arising from the contributions made in this research. Many of these have already been discussed, particularly concerning the handling of infinite loops. Other factors include attaining a greater level of genericity in genetic programs and minimisation of disruption to programs undergoing evolution. The Factorial problem, in particular, showed how reforming the test case to use dynamic variables allowed implementation-specific functions to be removed, and resulted in faster evolutionary convergence. However, it was found that physical running time was perceptibly slower when the addition operator was introduced. This was because deep expression trees were then generated for loops earlier in the program, restricted only by the depth of the entire program. Section 7.2.3 discusses how this could possibly be resolved.

Minimisation of disruption to genetic programs became another factor in accelerating the evolutionary process. This was achieved by defining additional operators designed to preserve as much of the existing genetic programs as possible – Insertion, Deletion, Replacement and Crossover2. While the introduction of these operators to the Artificial Ant test case did not show any conclusive trend, they dramatically improved the performance of the Factorial test case. Since these operators are new, future research would be needed to evaluate particular parameter settings that would accelerate evolution much further. This is discussed in Section 7.2.2.

7.2 Future Work

During the course of this research there were a number of ideas that presented themselves for future research. These arose from issues that presented themselves and from ideas formed from exploring possibilities. They were left out of this research due to being outside the scope of the focus on loops and control structures, and due to time constraints.

The following subsections describe these ideas according to their relevant contributions or existing aspects of GP.

7.2.1 Prevention of Infinite Loops

One of the developments made in this research was the definition of 'safety limits' to limit loop iterations and detect infinite loops. A drawback of this approach is that it works on 'curing' infinite loops rather than preventing them from occurring in the first place. While child constraints reduced the likelihood of infinite loops from occurring, particularly with `While` loops, it did not guarantee that infinite loops could not occur.

It was mooted that some form of program analysis be used when genetic programs are created or manipulated, to ensure that `While` loops would be made to eventually terminate. This would require either modifying the loop condition to eventually evaluate false, or modifying the loop internal to eventually make the loop condition false. A restriction would need to be placed on the condition to ensure that it would not be constant and therefore infinite, and program verification techniques would need to be used to ensure the loop terminates [28]. As the Artificial Ant test case would show, the latter would be much harder to implement as the variables specific to the fitness case are actually stored in the implementation environment, thus making them inaccessible directly from the GP mechanism.

Future research in this area would require a new representation of functions and terminals specific to the test case being solved. This would allow variables and statements to be accessed so that program verification could be performed. A possible solution may involve representing these specific functions in a similar manner to Automatically Defined Functions (ADFs) [21, 22, 20].

Alternatively, analysis of genetic programs could be based on heuristics that use relationships between pairs of nodes. For example, a set of 'Possibly Affects' relationships could include one defined between the `TurnLeft` and `IsFoodAhead` nodes of the Artificial Ant problem, meaning "`TurnLeft` possibly affects IsFoodAhead". However, these would not be reliable, since after four iterations of a `While` loop containing a single `TurnLeft` statement, the cyclic nature of the changing state and lack of a stopping condition would cause the loop to become infinite. These issues would need addressing in future work that looks into infinite loop prevention.

7.2.2 Additional Operators

The creation of additional genetic operators has shown promising results in the experiments, but they have been largely untested as a result of emphasis being placed on loops and control structures. The effects of different ratios between the operators used needs to be examined further, for example how many programs in the new population should be created via Insertion and Deletion versus Replacement.

There is also an opportunity to develop a third type of Crossover, which performs deletion on one program and insertion on the other. The 'deleted' node from the first program would be used as the node inserted in the second. Surplus children would be migrated as per Crossover2 to minimise disruption. It is worth noting that three of the four additional operators introduced (Insertion, Deletion and Replacement) describe Mutation operations. As a result, using all operators in an evolution would result in an imbalance between mutation and crossover operations – four mutation operations versus two crossover operations. The only issue seems to be a divergence from the biological model inspiring GP, but it would be interesting to investigate the usefulness of a third crossover operator introduced.

7.2.3 Restoring Balance

Despite the benefits of introducing the additional operators, it has become apparent that the single depth constraint imposed on the entire program results in early statements having deeper levels of nesting in their expressions than the last statements. This has invariably led to inefficiencies being introduced in genetic programs, making it very difficult to transform these into solutions.

One possible solution to investigate is splitting depth constraints among different conceptual levels. There would be three levels - the program level, construct level and statement level. The program level would define a depth similar to the existing program depth, but only include statements of type `Void` in the statement chain. The construct level would dictate the maximum length of a statement chain in a particular type of loop body or `If` construct. For example, the program environment could set a maximum depth for all `For` loops different to the maximum depth of the 'True' branch of all `If` constructs. Finally, the statement level would determine the maximum depth of any arithmetic expressions.

7.2.4 Intelligent Choice

A related theme in the evolutionary process is the nature of how genetic operations are administered. Currently, target nodes for genetic operations are selected completely at random as long as they satisfy closure (as defined by Koza [19, 21]). As a result, it is equally likely that a particular operator ends up disrupting a program as it is to improve fitness. The GP mechanism could be engineered to place weightings on potential target nodes according to potential gain from applying the operation there. Such weightings could be decided at program evaluation time, using simple and efficient program analysis techniques. The idea of these weights is to add some form of bias toward making potentially useful choices, but not eliminating the chance of making seemingly bad choices.

Similarly, weights could be assigned to particular functions and terminals when selecting new nodes to add to the program for Mutation, Insertion and Replacement. These weights would be dependent on the context of where the node is being added, which could be determined via heuristics or learning methods.

The overall benefit of this would be a more effective evolutionary process, by requiring fewer generations to generate a solution. This could also lead to fewer infinite loops being detected, and a more efficient evolution of loop and control structures.

7.2.5 Additional Control Structures

There are two control structures that were formalised in design, but were not implemented due to time constraints and technical challenges that needed to be overcome.

The first was the `ForEach` loop, which iterates over tems within a collection (e.g. set). This required a collection return type to be defined, and an architecture for storing and accessing its contents to be developed. Such a development would have at least similar complexity to the implementation of dynamic variables, and may require the handling of pointers in GP (as opposed to references). This is because dynamic collections such as vectors would need elements explicitly added. The easiest implementation would require collections to be created and populated prior to the genetic program running, thereby avoiding the need for pointers.

The second construct was the `Switch-Case` statement. This requires a special STGP return type, `Case`, to be defined. The `Case` type is almost the same as `Void`, but only used for chaining `Case` nodes directly below a `Switch` node. The key difference is that the integer value evaluated by the `Switch` node is propagated *down* the genetic program tree.

7.2.6 Structure Templates and Recursion

Another form of iteration often used in human-generated programs is recursion. Given that recursion is based on functions that make self-calls to enable iteration, this would be attainable in a basic sense from developments with ADFs [21, 22, 20].

A major issue to resolve is that recursive functions have the potential for infinite recursion, analogous to infinite loops for `While` loops. In computer science, it is a best practice for recursive functions to have a specific structure required to avoid infinite recursion [26]. This involves using conditional statements to evaluate a base (stopping) case, and the inductive case. The inductive case is required to induce a pattern that approaches the base case in all recursive calls.

To successfully evolve recursive functions, genetic programs would need to adopt a fixed high-level structure to ensure that the genetic functions all evaluate a base case separately from an inductive case. This can be provided by defining 'structure templates'. Structure templates are incomplete genetic program trees, requiring one or more children to be assigned subtrees in order to complete the tree structure. Particular nodes in these templates would be fixed in certain contexts to ensure that no genetic operation could disrupt the template. For recursive functions, this would guarantee that a decision would be made between the base case and inductive case.

7.2.7 Dynamic Node Sets

In order to implement a loop index whose scope was defined only within the corresponding loop's body, a mechanism was devised to support dynamic variables. This mechanism was also shown to be extensible to local variables in general. The major shortcoming of this is that if dynamic variables are able to be used in any part of a genetic program, at least one such variable must be globally defined so that every reference uses a valid index. Without such global variables, it would be possible for a genetic program to be formed with a node that accesses a variable in a scope where none has been defined.

The ideal solution would be to ensure that if no such variables have been defined within local scope of a particular node position, the genetic program will not allow any node that accesses that type of variable (as determined by return type and read-only status).

7.2.8 Test Cases

It was originally planned that a third test case would be run to further evaluate the contributions made in this research. This test case involved sorting arrays of integers, based on work conducted by Kinnear [17] and Ciesielski and Li [5]. All experiments in this test case were designed to use dynamic variables, so that loop indices would be properly scoped, even within nested loops.

Initial test runs could not evolve a solution, despite trying two different fitness functions. A likely cause of this is acknowledged by Ciesielski and Li [5], that sorting is a much harder task because of the need to co-ordinate loop indices. Some indication of this difficulty was evident in the Factorial problem, where `While` loops could not be evolved to perform the job of `For` loops, despite two temporary variables being allocated for the genetic program to use.

7.3 Summary

This research has presented a number of contributions, which both formalise a representation of imperative programs in Genetic Programming, and assist in more effective evolution of loop and control structures. These contributions have satisfied the overall goal of mimicking human-generated programs to a certain extent, through the use of core language features such as the loop and control structures and dynamic variables in genetic programs, and the ability to output these genetic programs as legal C++ code.

Test cases evaluated have shown evolutionary performance improvements and greater effectiveness from particular contributions introduced. In particular, converting the Artificial Ant problem to use statement chaining improved the performance significantly beyond the implementation by Ciesielski and Li [5]. Child constraints showed a clear improvement in effectiveness for the Artificial Ant problem, even though it had the opposite effect for the Factorial problem. Nevertheless, the quality of the programs output improved as a result of fewer redundancies. Additional genetic operators introduced showed promising results for the Factorial problem, though these need to be investigated further. Finally, the use of dynamic variables introduced a greater degree of genericity and led to less redundancy between functions and terminals defined. Based on the contributions made in this research, loops and control structures can now be evolved in Genetic Programming at a level of performance that rivals that of genetic programs without these constructs.

Bibliography

[1] Improving program robustness via static and dynamic analysis. Available from http://suif.stanford.edu/research/analysis.html (Accessed October 2005).

[2] Seven differences between genetic programming and other approaches to machine learning and artificial intelligence. Available from http://www.genetic-programming.com/sevendiffs.html (Accessed March 2005).

[3] ALLEN, F. E. Control flow analysis. In *Proceedings of a symposium on Compiler optimization* (1970), pp. 1–19.

[4] BACON, D. F., GRAHAM, S. L., AND SHARP, O. J. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4 (1994), 345–420.

[5] CIESIELSKI, V., AND LI, X. Experiments with explicit for-loops in genetic programming. In *2004 Congress on Evolutionary Computation* (July 2004).

[6] CIESIELSKI, V., AND LI, X. Using loops in genetic programming for a two-class binary image classification problem. In *17th Australian Joint Conference on Artificial Intelligence* (December 2004).

[7] COTA, B. A., FRITZ, D. G., AND SARGENT, R. G. Control flow graphs as a representation language. In *WSC '94: Proceedings of the 26th conference on Winter simulation* (1994), pp. 555–559.

[8] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1979), pp. 269–282.

[9] ESKRIDGE, B. E., AND HOUGEN, D. F. Memetic crossover for genetic programming: Evolution through imitation. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (2004), pp. 459–470.

[10] FRITZ, D. G., AND SARGENT, R. G. An overview of hierarchical control flow graph models. In *WSC '95: Proceedings of the 27th conference on Winter simulation* (1995), pp. 1347–1355.

[11] FUCHS, M. Crossover versus mutation: An empirical and theoretical case study. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (1998), pp. 78–85.

[12] GATHERCOLE, C., AND ROSS, P. Small populations over many generations can beat large populations over few generations in genetic programming. In *Genetic Programming 1997: Proceedings of the Second Annual Conference* (1997), pp. 111–118.

[13] HOAI, N. X. Softening the structural difficulty in genetic programming with tag-based representation and insertion/deletion operators. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (2004), pp. 605–616.

[14] JANIKOW, C. Z. A methodology for processing problem constraints in genetic programming. Tech. rep., University of Missouri, 1996.

[15] JANIKOW, C. Z. Adapting representation in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)* (2004), pp. 507–518.

[16] JOHN T. BAGWELL, J. Local optimizations. In *Proceedings of a symposium on Compiler optimization* (1970), pp. 52–66.

[17] KINNEAR, K. E. Evolving a sort: Lessons in genetic programming. In *1993 International Conference on Neural Networks* (1993).

[18] KIRSHENBAUM, E. Genetic programming with statically scoped local variables. Tech. rep., Hewlett-Packard Laboratories, 2000.

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

[20] KOZA, J. R. Genetic programming. In *Encyclopaedia of Computer Science and Technology* (1998), vol. 39, pp. 29–43.

[21] LANGDON, W. B. *Genetic Programming and data structures*. Kluwer Academic Publishers, 1998.

[22] LANGDON, W. B., AND POLI, R. *Foundations of Genetic Programming*. Springer-Verlag, 2002.

[23] MONTANA, D. J. Strongly typed genetic programming. *BBN Technical Report*, 7866 (March 1994).

[24] MOONEN, L. A generic architecture for data flow analysis to support reverse engineering. In *Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF'97)* (1997).

[25] SHIVERS, O. Control flow analysis in scheme. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation* (1988), pp. 164–174.

[26] STANDISH, T. A. *Data Structures in Java*. Addison-Wesley Longman Inc., 1997.

[27] WHIGHAM, P. A. Grammatically-based genetic programming. Tech. rep., University of New South Wales, 1995.

[28] ZHANG, J. A path-based approach to the detection of infinite looping. In *2nd Asia-Pacific Conference on Quality Software (APAQS 2001)* (2001).