EVOLVING HIGH-LEVEL IMPERATIVE PROGRAM TREES WITH GENETIC PROGRAMMING

A THESIS SUBMITTED TO

THE UNIVERSITY OF KENT

IN THE SUBJECT OF COMPUTER SCIENCE

FOR THE DEGREE

OF DOCTOR OF PHILOSOPHY

By

Thomas Anthony Castle

June 2012

Abstract

Genetic Programming (GP) is a technique which uses an evolutionary metaphor to automatically generate computer programs. Although GP proclaims to evolve computer programs, historically it has been used to produce code which more closely resembles mathematical formulae than the well structured programs that modern programmers aim to produce. The objective of this thesis is to explore the use of GP in generating high-level imperative programs and to present some novel techniques to progress this aim.

A novel set of extensions to Montana's Strongly Typed Genetic Programming system are presented that provide a mechanism for constraining the structure of program trees. It is demonstrated that these constraints are sufficient to evolve programs with a naturally imperative structure and to support the use of many common high-level imperative language constructs such as loops. Further simple algorithm modifications are made to support additional constructs, such as variable declarations that create new limited-scope variables. Six non-trivial problems, including sorting and the general even parity problem, are used to experimentally compare the performance of the systems and configurations proposed.

Software metrics are widely used in the software engineering process for many purposes, but are largely unused in GP. A detailed analysis of evolved programs is presented using seven different metrics, including *cyclomatic complexity* and Halstead's *program effort*. The relationship between these metrics and a program's fitness and evaluation time is explored. It is discovered that these metrics are poorly suited for application to improve GP performance, but other potential uses are proposed.

Acknowledgements

I have many people to thank for the part they have played in helping me throughout the PhD process. The first thank you goes to my supervisor, Colin Johnson, for the exceptional supervision and guidance, without which I am sure the journey would have been even more testing. I would also like to thank Alex Freitas and David Barnes, for the formal role they have played on my panel and for the questions they have asked, which have on more than one occasion helped to guide my research to areas of higher fitness!

Throughout my studies at Kent I have been very fortunate to share an office with some genuinely wonderful people. So, to the original occupants of S109B - Lawrence, Laurence, Rob and Ahmed - I thank you for initially making me so welcome and for the ever friendly and productive environment. Thank you also to Patrick, for keeping me sane during my brief time in S15. Special thanks to both Lawrence and Fernando, who have both at times been like unofficial mentors. Thank you for the guidance, stimulating conversation and of course the ongoing collaboration on the EpochX software.

Thank you, also, to all my family and friends, for putting up with me during the low points, high points and the long periods of self-imposed reclusion. In particular, to my parents who played a key role in my own evolutionary tale, not to mention the continual support and encouragement they have always provided. Also, thanks go to Mike for his willingness to read much of this thesis. Finally, a huge thank you to Trish, for her unending patience - I am finished now, I promise!

Contents

List of Tables

List of Figures

List of Algorithms

From so simple a beginning endless forms
most beautiful and most wonderful have been,
and are being, evolved.

CHARLES DARWIN

Chapter 1

Introduction

Genetic Programming (GP) [32, 87] is a technique which uses an evolutionary metaphor to automatically generate computer programs. Biological evolution has demonstrated itself to be an excellent optimisation process, producing structures as diverse as a snail's shell and the human eye, each life form filling a niche to which they are remarkably well adapted. Evolutionary algorithms aim to replicate this success, to produce solutions to a specified problem. The GP evolutionary algorithm uses a population of individuals that represent computer programs, with a well defined encoding, which are progressively improved by applying operations that are based on biological reproduction. Although GP proclaims to evolve computer programs, historically it has been used to produce code which more closely resembles mathematical formulae than the well structured programs that programmers aim to produce. The objective of this thesis is to explore the use of GP in generating high-level imperative programs and to present some novel techniques to progress this aim.

Many of today's most commonly used programming languages can be described as both high-level and imperative, such as C, PHP and Javascript. Even object-oriented languages such as Java and C++ have an imperative core. However, evolving high-level imperative programs with GP is challenging. High-level code is required to abide by strict and often complex structural rules. Furthermore,

programs typically make use of constructs such as iteration and variable declarations which add additional complexity. Despite these difficulties, the potential reward of being able to automatically generate code that is comparable to that produced by human programmers make this a worthwhile direction for research.

In this thesis, it is demonstrated that a set of simple extensions to a commonly used variant of GP can allow it to support the evolution of program trees with a high-level imperative program structure. Experiments are conducted which explore the performance advantages and the impact on the search-space of these extensions. Further modifications are also presented with the aim of supporting program constructs that declare limited-scope variables. A number of new types of non-terminal node are proposed which make use of this, including some that represent loops which more closely resemble iteration as used by human programmers. The use of these new nodes is experimentally compared to non-declarative alternatives and there is some discussion of the advantages.

As the scale and complexity of the programs that GP can evolve increases, the more human programming methods become relevant to the GP algorithm. Similarly, the evolution of high-level imperative programs raises the possibility of using tools and techniques associated with these languages. In this thesis, the application of one such tool is considered: software metrics. Software metrics have found many applications throughout the software development process and it is possible that they may have applications in the automatic development of software with techniques like GP. Previous research in GP has only made very limited use of simple measures such as program length or depth. In this thesis, a detailed analysis of programs evolved with GP is conducted using a series of popular software metrics and potential applications are discussed.

1.1 Motivations for Evolving High-Level Imperative Programs

Koza listed seven reasons for choosing to evolve programs in the Lisp programming language [87, chapter 4.3]. The first six of these reasons all focus on aspects of Lisp that make it a convenient choice and easier to implement. They include reasons such as a Lisp program being equivalent to its own parse tree and point to features such as the built-in `EVAL` and `PRINT` functions, which make it simple to evaluate a Lisp program that was created and print it presentably. Choosing to use Lisp because of the ease of implementation is a reasonable reason for early work with genetic programming, but it is not necessarily the best choice for all applications now.

The primary motivation for evolving high-level imperative programs is their popularity. All of the top ten programming languages listed by the TIOBE Index as the most popular can be described as imperative [156]. The software development industry overwhelmingly favours imperative programming languages. For many applications of GP, this is irrelevant. But, for the development of software, the advantages of being able to produce code that is comparable to that produced by human programmers is significant. In a scenario where GP is used to generate fragments of code or complete modules of a larger software system, this would allow automatically generated code to more easily interact with existing modules and to be tested and maintained by human programmers alongside their existing codebase. This point is well demonstrated by the success of recent work using GP to perform tasks such as bug-fixing with commercial imperative programs [164]. Koza's seventh reason for choosing Lisp was that a large range of programmer tools were available for it. The popularity of imperative languages such as Java, C/C++, PHP and Javascript, means that all have received substantial investment, with a vast range of tools available and so this reason very much applies to these imperative languages as well.

Even without applications to software development, there are still motivations for researching the evolution of imperative programs. Often, the Lisp programs produced with tree-based GP are deeply nested and can be incomprehensible. But the clearer structure and the wide familiarity with imperative programs could potentially make them easier to read and reason about. There is also the possibility that the more structured programs could result in performance advantages and the use of high-level imperative programming constructs, such as for-loops and for-each-loops, have much potential for finding smaller and more general solutions.

1.2 Terminology

Throughout this thesis, a number of terms are used which may be unfamiliar or ambiguous for the reader. The following definitions are used:

- *Node* – programs are represented as trees composed of nodes, where each node represents a programming construct.
- *Arity* – the number of child nodes a node has or requires. A node of arity 0 is a terminal or leaf node.
- *Terminal set* – the set of nodes supplied to the system that require no inputs and so will have no child nodes (arity 0).
- *Non-terminal set* – the set of nodes supplied to the system that require one or more inputs (arity >0). This term is preferred to the widely used term, *function set*, because in most cases the elements of the set are components of an imperative style, not functions.
- *Syntax* – there is often no need to distinguish between the terminal and non-terminal sets, so we frequently use the term *syntax* to refer to the union of these two sets.

1.3 Contributions

The following major contributions are made by this thesis:

- Extensions to an existing GP system to evolve programs with stricter constraints. Current versions of Strongly Typed Genetic Programming support only data-type constraints. The proposed modifications to this system support a high degree of structural constraint in the classic tree representation.

- Application of new structural constraints to evolve high-level imperative program trees. Demonstrations are made of how a high-level imperative program structure can be enforced on programs evolved with a tree-based representation and how standard imperative programming constructs can be supported.

- Support for the evolution of programs with limited-scope variable declarations. Modifications are proposed to allow a dynamic syntax, where the available terminals and non-terminals are modified by a program. New program constructs are proposed that create and add variables to the available syntax and these are experimentally tested.

- Simple performance enhancement for when evolving programs with a high-level imperative structure. A trivial modification is made to the fitness evaluation procedure which provides a significant performance improvement under specific circumstances. The extent of the improvement is experimentally tested.

- Analysis and application of software complexity metrics to evolved program code. A detailed analysis is presented of programs generated with GP using software complexity metrics. Seven different metrics are examined and compared. Potential applications of software metrics in GP are discussed.

1.4 Thesis Overview

Chapter 2 gives the background to the evolution of high-level imperative programs. It provides an overview of the genetic programming algorithm and discusses some of the key issues in current GP research. The chapter also outlines some of the common program representations in use and in particular covers existing research into evolving imperative programs in each of the representations.

Chapter 3 outlines the common methodologies used throughout this thesis. A set of six test problems are described, as used in the experimental work reported in this thesis, along with a listing of the training and test data used. Some existing attempts at solving each of the problems are reviewed. The chapter also describes the general approach used to conduct experiments and present the results.

Chapter 4 describes a novel method for introducing structural constraints into a tree-based GP system. It is demonstrated how this system can be used to support the evolution of programs with new programming constructs and how the shape of the program trees can be constrained to a natural high-level imperative structure. The performance impact of using this structure is experimentally tested and a comparison is made to an equivalent setup without structural constraints. The use of the structural constraints is shown to reduce the search-space and to have a mostly positive impact on performance.

Chapter 5 outlines two extensions to the original system that was described in chapter 4 to specifically target the evolution of high-level imperative programs. The first adds support for a dynamic syntax which allows the evolution of programs with constructs that declare limited-scope variables. This is experimentally shown to have a problem dependent impact upon performance, but discussion focuses on other benefits, such as reducing the need for insight into the solution space. The second extension is a simple method to improve the efficiency of fitness evaluation by evaluating multiple variants of the same program. This technique is shown to significantly improve the performance of the algorithm where it can be applied.

Chapter 6 provides a detailed analysis of evolved programs with software metrics. Results are presented which compare seven common software metrics, including complexity metrics such as cyclomatic complexity and commonly used GP program measures such as program depth. The relationship between these metrics and and a program's fitness and evaluation duration is explored and potential applications of software metrics in GP are considered. It is found that these metrics are of little value for application to improve the success rates of GP.

Chapter 7 summarises the results of this thesis and the contributions made. Conclusions are drawn based upon analysis of the presented research and areas for future work are discussed.

Appendix A provides a complete listing of source code templates to convert a program tree evolved with the system described in chapter 4 to valid Java syntax.

1.5 Publications

During the course of this research a number of contributions were made to the genetic programming literature.

Peer-Reviewed Conference Papers

- T. Castle and C.G. Johnson. Positional Effect of Crossover and Mutation in Grammatical Evolution. In *Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010)*, pages 26–37. Lecture Notes in Computer Science 6021, Springer, April 2010.

- T. Castle and C.G. Johnson. Evolving High-Level Imperative Program Trees with Strongly Formed Genetic Programming. In *Proceedings of the 15th European Conference on Genetic Programming (EuroGP 2012)*, pages 1–12. Lecture Notes in Computer Science 7244, Springer, April 2012.

– T. Castle and C.G. Johnson. Evolving Program Trees with Limited Scope Variable Declarations. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2012)*, pages 1–8. IEEE Press, June 2012.

Workshop Contributions

– F.E.B. Otero, T. Castle and C.G. Johnson. EpochX: Genetic Programming in Java with Statistics and Event Monitoring. In *Proceedings of the 14th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO 2012)*, pages 93–100. ACM Press, July 2012.

– L. Vaseux, F.E.B. Otero, T. Castle and C.G. Johnson. Event-based Graphical Monitoring in the EpochX Genetic Programming Framework. In *Proceedings of the 15th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO 2013)*. ACM Press, July 2013.

Chapter 2

Background

2.1 Introduction to Genetic Programming

Evolutionary Algorithms (EA) are a class of optimisation algorithm that use the metaphor of Darwinian evolution to generate solutions to a predefined problem. A varied population of potential solutions to a problem is maintained and progressively improved by a process of selection, modification and reproduction. The use of Evolutionary Algorithms to evolve computer programs using a flexible tree representation was first suggested by Cramer [32]. This work was extensively expanded by Koza [85,87], who initiated and popularised the term *Genetic Programming* (GP). Since Koza's initial work, a vast range of modifications and extensions have been applied to his algorithm, some of which will be described in the following sections. We follow the trend of using the term *Genetic Programming* to refer to all examples of evolutionary algorithms applied to computer programs, regardless of program representation. To distinguish Koza's specific example of GP, we refer to it as *standard* GP. Standard GP represents individual programs as trees and follows the basic algorithm shown in the flowchart in Figure 2.1 to evolve the population. The GP algorithm is non-deterministic and is not guarenteed to find an optimal solution.

The GP algorithm is controlled by a number of configuration parameters which guide its progress and determine its capability to solve a given problem. The principal parameters used are:

- *Population size* – the number of individuals to use within each generation of the algorithm.
- *Maximum generations* – termination criterion based upon the number of iterations of the algorithm performed.
- *Initialisation* - the random program construction procedure to use for generating the first population of individuals.
- *Selection* – the selection mechanism for choosing individuals to undergo genetic operators.
- *Genetic operators* – one or more operators able to produce new program trees based upon one or more existing programs.
- *Operator probability* – the probability of performing each genetic operator.
- *Maximum depth* – the maximum depth allowable for program trees created by the initialisation procedure and genetic operators (sometimes a separate *initial-maximum-depth* setting is used for the first population).
- *Syntax (terminal and non-terminal sets)* – the available components that programs may be composed of.
- *Elites* – whether the practice of elitism should be used and the number of elites to be automatically placed into the next population each generation.

An initial population of computer programs is randomly generated, with each program in the population composed of components taken from the available syntax supplied in the terminal and non-terminal sets. Each of these programs is evaluated according to some quantitive measure of quality and assigned a fitness

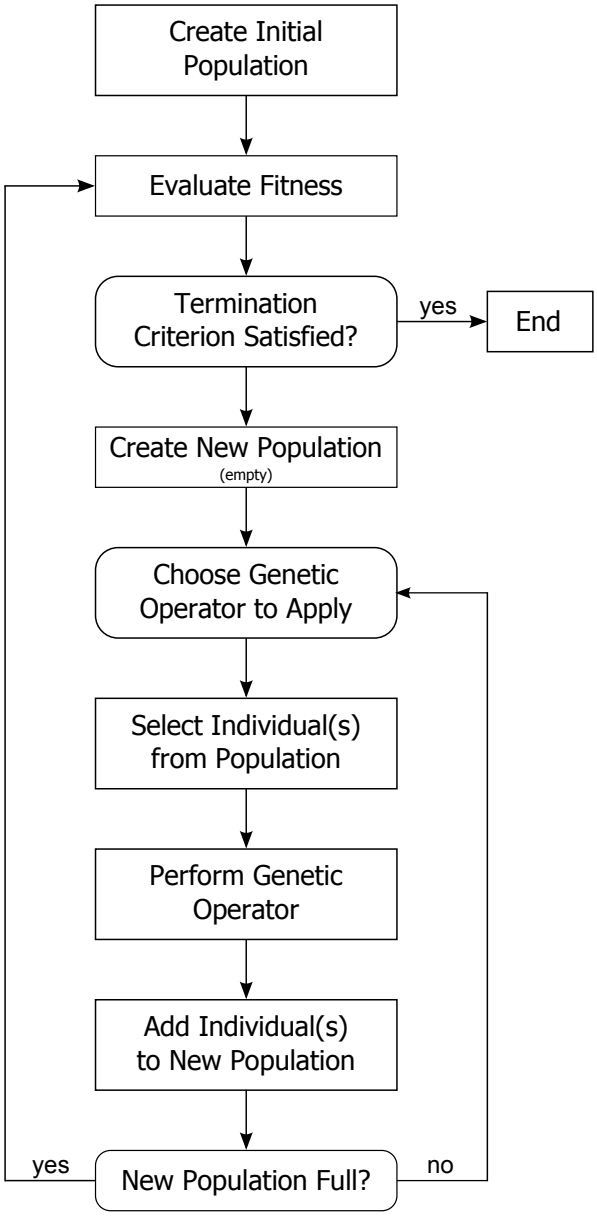


Figure 2.1: Flowchart illustrating the main steps of the genetic programming algorithm

score. The fitness score is typically a numeric value which must either be maximised or minimised by the algorithm[1]. A new population of computer programs is created from the previous population, by applying a number of genetic operators to the best programs, selected probabilistically based upon their fitness. These steps are repeatedly performed until some termination criterion is satisfied, such as a solution is found, or a maximum number of generations has been reached. Since the algorithm is inherently stochastic, no guarantee can be made about the quality of its results. However, this random quality is the source of its key strength, that it is potentially capable of escaping local optima to provide a good heuristic method for searching complex solution spaces.

2.1.1 Representation

Standard GP represents each individual candidate solution as a program tree, describing Lisp S-expressions. Although programs in virtually all programming languages can be described using trees, the simple structure of Lisp S-expressions are very naturally represented as a tree. Each leaf node in a program tree represents a variable or a constant and each non-leaf node represents a function. Koza's terminology refers to these as terminals and functions, but this could lead to confusion when discussing imperative programs, so the terms terminal and non-terminal are preferred in this thesis and the term *syntax* is used to apply to the union of these two sets of nodes. Figure 2.2 shows an example program tree which represents the expression $((2 + x) * (3 - x))$, or $(* \ (+ \ 2 \ x) \ (- \ 3 \ x))$ when expressed using the prefix notation more typical for S-expressions.

The evaluation of a program tree occurs with a depth-first traversal of the tree. Starting with the root node, each node's arguments must first be evaluated, down to terminal nodes which when evaluated return their value. Each non-terminal node uses its arguments, performs some operation and then returns a value as

[1] minimisation is used in all cases throughout this thesis.

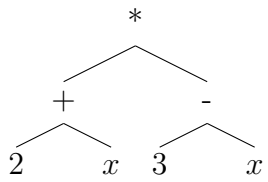


Figure 2.2: Example GP program tree representing the arithmetic expression $((2+x)*(3-x))$. It can be evaluated with a depth-first traversal of the tree, with each node performing the associated operation upon the results of evaluation of its subtrees.

input to its parent node. The result of evaluating the root node is returned as the result of the program tree.

Other representations have been used and gained some popularity. Linear GP [17, 113] represents individuals as a sequence of instructions, graph-based GP [145,153] uses potentially cyclical graph structures and grammar-guided GP [105, 121, 166] uses concrete syntax trees derived from a language grammar. Some of these representations are described in detail later in this chapter.

2.1.2 Initialisation

The initial population of individuals is constructed using an initialisation procedure that is capable of producing random tree constructions from the available syntax specified in the terminal and non-terminal sets. The most commonly used initialisation method is *ramped-half-and-half*, as introduced by Koza [87]. It uses two tree generation methods, *full* and *grow*, which are alternately employed to construct programs to a maximum depth that is gradually increased from some minimum value up to the maximum allowable depth. The full initialisation method constructs full program trees where all branches extend to the same maximum allowable depth. Program trees that are grown, have branches that may extend to any depth within the maximum allowable setting. The ramped-half-and-half technique is intended to increase population diversity and Koza reinforced this by ensuring that no syntactically duplicate programs are added to the initial population.

Numerous alternative initialisation procedures have been proposed and are often claimed to be superior to ramped-half-and-half [14,72]. However, a comparison of different initialisation procedures by Luke [96] found little if any fitness advantage was gained by using these alternative approaches. One key advantage that ramped-half-and-half holds in practice is simplicity, but it does provide little control over the tree construction in comparison to these other methods. Ultimately, different tree generation methods perform better on different problems [130].

Studies into the effect of diversity in GP [20,57] imply that increased diversity in the exploratory phase of the algorithm, including initialisation, is important for evolutionary progress. This does supply some support for the ramped-half-and-half measure, but it is important to be aware that population diversity can be measured in many different ways. In particular, it can be based upon either syntactic or semantic traits. Ramped-half-and-half ensures a degree of syntactic variety, but it is entirely possible that many of the programs generated may be semantically identical. Other authors have sought to increase behavioural diversity in the initial population [12] and found this to be beneficial to both success rates and the required computational effort.

2.1.3 Fitness Evaluation and Selection

Having been randomly constructed with no intelligent thought to solving the given problem besides in the selection of the control parameters, the initial population is likely to contain programs which are very poor solutions. However, it is expected that some individuals may perform slightly less poorly than others. The quality of programs are evaluated using a fitness function and typically allocated a numeric fitness score. In GP this commonly involves evaluating each program on multiple training cases in order to judge the program's ability to correctly process a range of inputs.

It is widely reported that program evaluation is the most time consuming element of the GP algorithm [62,100,134]. This is unsurprising since it is not unusual

for several million program evaluations to be required in the course of identifying one complete solution. Therefore, program evaluation has frequently been identified as a key step to undergo performance tuning. Teller and Andre [152] proposed a sophisticated algorithm they called Rational Allocation of Trials, for identifying the minimum number of fitness cases per program that are required. Other techniques include caching of subtree evaluations, where subtrees are likely to reappear again and again throughout a population [28] and fitness approximation, where only a small proportion of individuals are evaluated in the normal way, while others are assigned a fitness which is estimated based upon more cheapily calculated metrics such as the average fitness of its parents [74].

One of the strengths of the GP algorithm is that it is particularly well suited to parallelisation. It is normally the case that each program is evaluated independently from all others (exceptions to this include co-evolutionary approaches [86]) and so the evaluation of an entire population may be split across multiple processors [127], machines [150] or even continents [26]. A recent trend is the application of the many processors often found in graphics processing units (GPUs) to scientific computation and a number of studies have looked to apply mass market GPUs to improve the performance of GP evaluation [11, 25, 62].

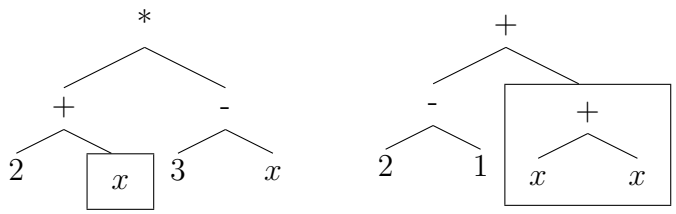
As with biological evolution, the best individuals are allowed to reproduce, creating new individuals, while the weaker individuals die out. Some form of selection method is used to probabilistically choose individuals from the population to undergo genetic operators with a bias towards those of better fitness. The extent of this bias determines the degree of selection pressure. Too much selection pressure degrades the evolutionary algorithm into a simple hill-climber, but too little selection pressure leads to an undirected search of the search-space. Some of the more established selection methods include *fitness-proportionate*, *rank* and *tournament* selection [87, chapter 6.4]. Tournament selection works by randomly plucking x individuals from the population to take part in a tournament, with the best individual in that tournament selected. Varying the value of x provides a

simple mechanism for modifying the level of selection pressure. Also, since tournament selection only considers whether one individual is better than another, not the degree to which it is better, it provides a constant selection pressure that cannot be excessively biased towards one substantially better individual. This is important to avoid the population being overwhelmed by one individual.

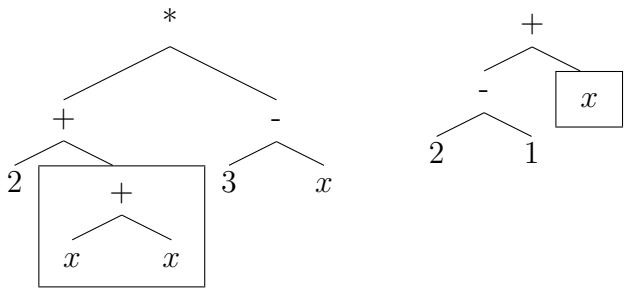
2.1.4 Genetic Operators

Two main genetic operators are used: *crossover* and *mutation* based on sexual and asexual reproduction respectively. Crossover operators typically require two parent individuals to be selected from the existing population and produce two new child programs to be inserted into the next population, while mutation operators involve the manipulation of just one program. Wide variations in the implementation of genetic operators are possible and they are heavily dependent upon the representation in use. *Subtree crossover* is commonly used with tree representations. Two individuals are selected from the population using some selection measure, then undergo an exchange of genetic material. A node is then randomly selected in each of the individuals' program trees and the subtrees rooted at those two points are swapped.

The justification for subtree crossover is that some subtrees may encapsulate useful behaviour, which when transferred to a new individual may benefit that program [107]. This is based upon the idea of *building blocks*, adopted from genetic algorithms [54, 69]. However, it has been suggested that crossover may in fact be little more than a form of macro-mutation [7]. It is certainly true that crossover operations, as with mutation operations, are largely destructive [75, 116]. That is, a high proportion of genetic operations result in reduced fitness from the parents to their children, or otherwise have no impact upon fitness. The destructive effect of crossover can be reduced with context-aware crossover [101], which replaces the random selection of crossover points with a more measured approach, whereby a subtree is inserted into all possible locations to identify the best position. The



(a) Parent programs.

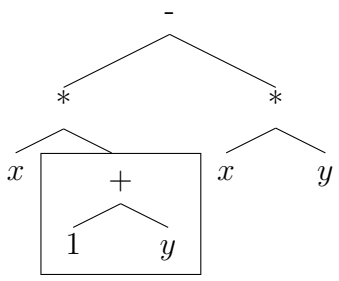


(b) Child programs.

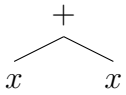
Figure 2.3: Example illustrating subtree crossover. Two parent programs are selected from the population using a selection method which favours fitter individuals. Subtrees are randomly selected in both parent programs (as highlighted) and are exchanged to produce two new child programs.

authors claim that this sufficiently improves the performance of the algorithm to warrant the additional computational expense.

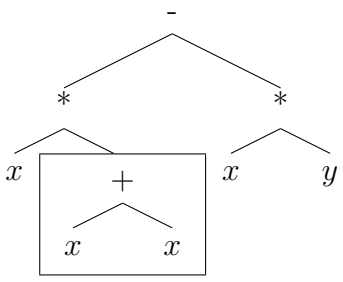
Subtree mutation operates similarly to subtree crossover. One individual selected from the population undergoes the operation, with a node in its program tree selected at random. The subtree rooted at this node is then replaced with a newly generated subtree, constructed using some form of initialisation procedure. Different rates of each genetic operator may be used, so that more of a population is produced by one operator than another. Common practice in the GA literature is to predominately use crossover. Koza advocated similar practice. In fact, much of Koza's work uses no mutation operator at all. A thorough comparison of subtree crossover and subtree mutation was performed in [98], which demonstrated that subtree crossover performed better when larger population sizes were used, but that subtree mutation outperformed subtree crossover in some cases



(a) Parent program.



(b) Random subtree.



(c) Child program.

Figure 2.4: Example illustrating subtree mutation. One parent program is selected from the population using a selection method which favours more fit individuals. A subtree in that program is randomly selected (as highlighted) and is then replaced with a randomly generated subtree to produce a new child program.

with smaller populations. Variations of subtree crossover and subtree mutation are used in much of the work throughout this thesis.

A desirable trait of genetic operators in all evolutionary algorithms is that they should exhibit high locality; small genotypic changes should result in similarly small changes in the phenotype and the fitness of the individuals [137, 138]. This allows the algorithm to more smoothly navigate the search space. This is more difficult to enforce in systems that use separate genotype and phenotype representations such as Grammatical Evolution [21, 23, 139]. Other operators have been designed that seek to increase locality [158] and these have been demonstrated to improve performance and reduce bloat. The term *bloat* is used to refer to the often seen characteristic of GP runs where the average size of individuals

increases as the run progresses and is an issue that has received substantial interest in the GP literature. Discussions of bloat revolve around the appearance of unused or unnecessary fragments of code known as *introns*. Techniques for reducing bloat are varied, but many focus on the use of genetic operators designed for the purpose [90, 97, 154].

2.1.5 Data-types, Closure and Sufficiency

Because of the random nature of the construction and manipulation of program trees, some consideration must be given to the available syntax, to ensure that all program trees are valid. The syntax may include functions such as arithmetic operations ($+$, $-$, $*$, $/$), boolean operations (AND, OR, NOT) and conditional operators (IF, $IF-ELSE$). If all possible combinations of these functions are to be allowed, then some form of procedure needs to be in place to handle the potential mismatch of data-types. Koza's solution was the *closure* property, which requires each function node in the syntax to be designed so that it can work with any possible set of inputs that it may receive. For example, a function node which performs a greater-than comparison of two integer values can be defined to return +1/-1 instead of true/false boolean values, so its output would be valid input to an arithmetic function. Some functions may also require protection from specific values, for example division by zero has no sensible result, so some integer value (typically +1 or 0) is designated as the result of this expression.

The use of the closure property requires some substantial engineering of the syntax in many cases. Alternative solutions include discarding or penalising individuals that do not evaluate correctly, but these are rarely used unless the closure property is difficult to impose. More elegant solutions have since been proposed to overcome many of the issues by enforcing data-type constraints to maintain valid programs with mixed data-types [13, 109, 177]. These will be discussed in more detail in section 2.3, in relation to their use in evolving imperative programs.

A second consideration in the selection of terminals and non-terminals for the syntax is the *sufficiency* property. The search space is comprised of all possible

program trees that can be created by compositions of the available syntax. So, if no solution can be described with the components available in the syntax, then no solution can ever be found; the syntax is insufficient. Designing the syntax to be sufficient to solve a given problem requires a certain level of insight into what a solution may look like. The difficulty of this task is problem dependent and not always trivial. A related issue is that of *extraneous* components, where the syntax contains variables or non-terminals that are not required for solving the problem. Each additional component supplied increases the size of the search space exponentially and so the effect is significantly reduced success rates and increased computational effort, although the degradation is found to be linear rather than exponential [87].

2.2 Evolving Imperative Programs

The GP literature is dominated by the generation of functional Lisp programs. Koza lists seven reasons for his choice of using Lisp [87], all of which revolve around the idea that the features of the Lisp language make it *easier* to evolve than the alternatives. But an easier implementation, does not necessarily mean better or more useful programs. In contrast, the vast majority of computer programs written by human programmers today are written in high-level imperative programming languages such as C/C++, Java and Python. So, for an automatic programming system to be useful for the development of software, it should be capable of generating high-level imperative code. Most languages described as imperative share five features: assignment, variable declaration, sequence, test and loop [40]. However, there are some substantial challenges involved with evolving programs with these features using GP, including:

- Handling mixed data-types
- Supporting limited scope variables
- Supporting complex constructs such as loops

– Maintaining a sequential structure

The rest of this section will discuss some of the existing work into evolving imperative programs using some of the more common program representations, with reference to how they approach some of these challenges.

2.2.1 Tree-Based Genetic Programming

Although the majority of the work presented by Koza [87] made use of a complex nesting of Lisp S-expressions, he did experiment with the use of a more imperative style, such as in solving the Artificial Ant problems. On these problems, evolved programs were composed of instructions strung together with `ProgN` nodes, where each instruction enacts a side-effect upon external elements (an ant within a 2-dimensional environment), rather than returning a value. Although using an imperative style, this work neglects to use mixed data-types or any of the standard high-level programming constructs such as loops. The imperative structure is also rather superficial, with `ProgN` nodes introducing a sequential ordering without a control structure that corresponds to any standard imperative construct [104].

In general, the benchmark problems that have been widely used in the GP literature have been expressed so as to avoid the need for imperative constructs. For example, the Artificial Ant problems used by Koza and since by many others, prescribe that the program controlling the ant should be executed multiple times until a set number of time steps is used up. This builds the looping concept into the problem, rather than requiring it of the solution. Another technique commonly used, is to supply specially crafted functions, such as an *if-food-ahead* node, which will conditionally execute a number of instructions if food is in the facing cell. No such construct exists in any general purpose programming languages, but in this case the problem of mixed data-typing is side-stepped.

Koza [87] suggested an approach for adding additional constraints for handling multiple data types with what he described as *constrained syntactic structures*. These constraints were imposed with a set of rules defined for each non-terminal

stating which other terminals or non-terminals may be used as its children. The generation of new individuals and genetic operators were modified to support the rules throughout the algorithm. These extensions were demonstrated as a way of finding solutions to symbolic regression problems with multiple dependent variables, where the result can be returned as an ordered list.

Other researchers have proposed more powerful mechanisms for overcoming the closure requirement. The most notable of these is Strongly Typed Genetic Programming (STGP) [109] which introduces explicit data-types that each terminal and non-terminal are required to declare and which an enhanced algorithm is able to enforce. STGP is of special relevance to this thesis so will be described in greater detail in section 2.3. Other similar efforts include PolyGP [30, 177], which provides a polymorphic typing system using a parse tree syntax based upon λ-calculus. The authors assert that PolyGP is superior to STGP because it does away with the lookup table required for tree creation and because it is able to support higher-order functions which they claim STGP is unable to (in fact Montana describes how STGP can evolve and use higher-order functions in his paper). Binard and Felty [13] proposed a similar system, System F, also based upon λ-calculus which was intended to improve upon PolyGP by removing the need for a type unification algorithm by annotating terms in place with all necessary information. They claim that System F has better support for recursion and is able to evolve new types alongside other program elements. Both PolyGP and System F use an expression-based approach which makes them inherently functional in nature and unsuitable for representing the high-level imperative programs with which this thesis is concerned. The Strongly Typed Evolutionary Programming System (STEPS) [78] is another modified form of STGP that was proposed for the generation of functional logic programs in the Escher programming language. It has primarily been used by Kennedy for tackling concept learning problems. Specialised genetic operators ensure the evolved programs are variable consistent as well as type consistent. Local variables are given restricted scope and are required to be used once quantified.

The data-type constraints enforced by strongly typed evolutionary systems go some of the way to supporting the restrictions required for evolving high-level imperative programs, but none of the systems mentioned above include any explicit mechanism for constraining the structure of program trees in the way that is necessary. One particularly popular way of constraining the structure of solutions in an evolutionary system is with the use of grammars. Grammar guided approaches to GP are discussed in 2.2.3. Techniques based upon the more standard tree-based GP are less established. McGaughran and Zhang [104] used a tree based representation to generate imperative (non-object oriented) C++ programs. They enforced an imperative structure by chaining statements together to form a sequential ordering for execution.

2.2.2 Linear Genetic Programming

Some of the earliest attempts at evolving imperative programs were with a linear representation [17, 113]. In linear GP, programs are comprised of a sequence of either machine code or interpreted higher-level instructions that manipulate the value of machine registers. At the lowest level, almost all computer architectures represent programs as sequences of instructions that are executed consecutively, so intuitively it is a sensible form to represent programs under evolution. Programs are initialised as random constructions from the target processor's instruction set and manipulated by genetic operators that exchange fragments of code between programs. The operators would normally be constrained to ensure crossover points only occur at word boundaries and that only a restricted set of instructions from the instruction set may appear, in the same way that only a restricted syntax is made available in standard GP.

The primary incentive for using linear GP is faster execution speed, since the programs may often be executed directly on hardware with little or no interpretation. In order to achieve this, each instruction should consist of a machine code instruction for a real computer. Crepeau [34] generated code for the Z80 and

Nordin [113] used his CGPS system to evolve RISC code for the SUN SPARC architecture and later CISC code on Intel's x86 [115]. CGPS (later AIM-GP), was shown to be approximately 1000 times faster than interpreted GP representations at evaluating individuals.

Although beneficial in terms of speed, the concern with evolving machine code instructions is that the solution programs are closely tied to that specific architecture. However, there is some suggestion that using an interpreted form of linear GP can still be more efficient than an interpreted tree-based system [130]. A linear program is also potentially easier to analyse for purposes such as identifying and removing ineffectual instructions (introns) [16]. In other studies [67, 99], byte code has been evolved for the Java virtual machine, making greater platform independence possible.

2.2.3 Grammar-Guided Genetic Programming

The term *grammar-guided genetic programming* refers to a number of different techniques for introducing language grammars into the evolutionary algorithm, such that the syntactic structure of programs may be constrained [105]. This makes them very suitable for introducing both data-type constraints and the necessary structural constraints required by high-level imperative programs. In the rest of this section, some of the most popular grammar-guided GP approaches will be outlined, as well as those attempts at evolving programs with an imperative structure that have used such approaches.

Whigham proposed *context-free grammar genetic programming* (CFG-GP) [166], which makes use of grammars using the Backus-Naur Form (BNF) notation. BNF grammars are context-free, so are unable to contain any of the formal semantic constraints of a language. An example BNF grammar is shown in Algorithm 2.1. Whigham's modifications from the standard GP algorithm, construct solutions which are represented as parse trees by stepping through the grammar, randomly selecting from the available set of productions in each rule. All solutions are thus created valid according to the grammar. This syntactic validity is maintained by

genetic operators which replace subtrees with parse trees rooted at the same non-terminal, either randomly generated in the same way as the initial construction (for mutation) or copied from another individual in the population (crossover). An additional benefit of the use of grammars that Whigham explores in his work is modifying the grammar to bias specific constructs.

Algorithm 2.1 Example grammar in backus-naur form (BNF) notation, expressing the syntax of simple arithmetic expressions

⟨*expr*⟩ ::= '(' ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩ ')'
| ⟨*var*⟩
| ⟨*literal*⟩

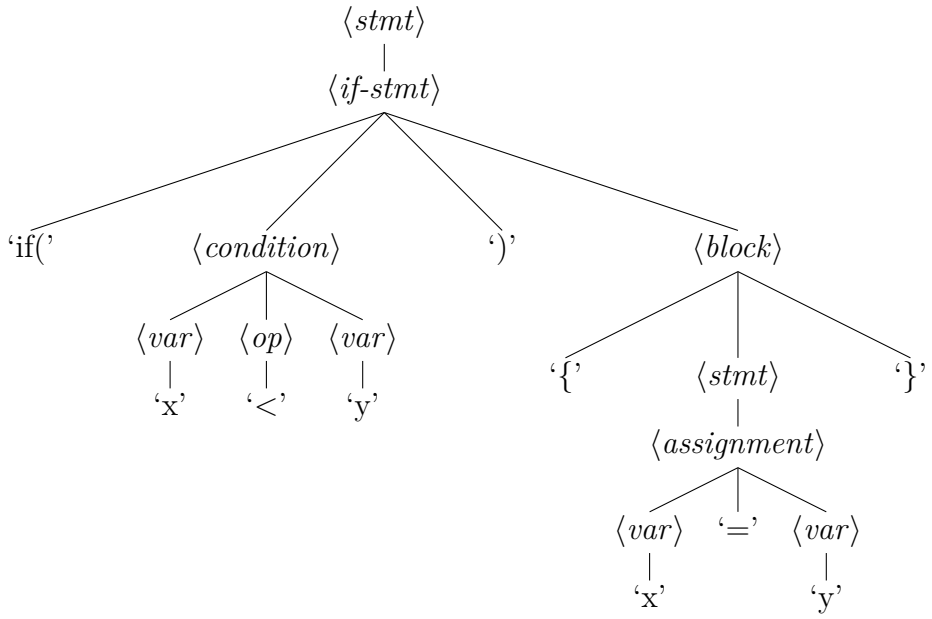
⟨*op*⟩ ::= '+' | '-' | '*'

⟨*var*⟩ ::= 'x' | 'y'

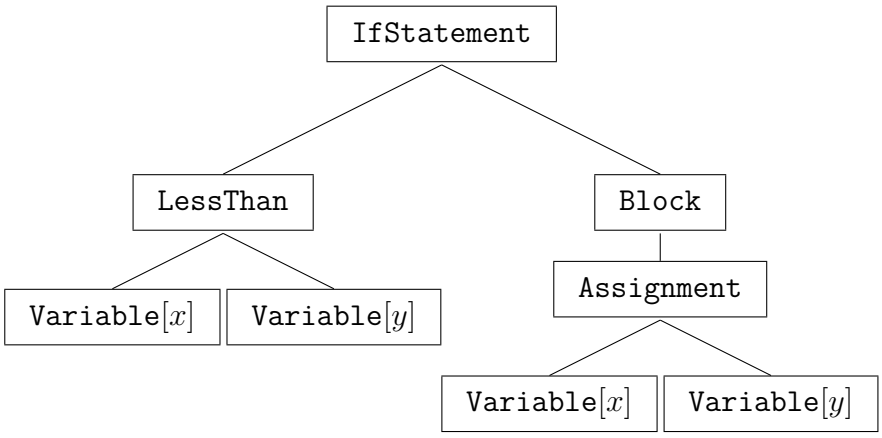
⟨*literal*⟩ ::= '1.0' | '2.0'

Grammatical Evolution (GE) [122] is an alternative grammar based approach which uses a separate genotype and phenotype representation. The genotype representation, which is the representation modified by the genetic operators, is a simple sequence of codons, where each codon is an integer (or bit string representing an integer). During evaluation a mapping operation is used, whereby the phenotypic parse tree representation is constructed from the grammar by selecting productions in the grammar rules according to the value of codons. One of the key advantages of GE over other grammar-guided approaches is that the simple linear genotypic representation is very simple and inexpensive to modify by genetic operators. However, GE has undergone some criticism on the topic of locality [21, 23, 139], because its search operators appear to exhibit low locality, where a small modification in the genotype results in a large change to the phenotype and the resulting fitness of individuals.

The use of grammars provides a powerful mechanism for constraining the structure of solutions that can be used for introducing a more naturally imperative control structure. This was demonstrated in O'Neill and Ryan's work on evolving



(a) Concrete Syntax Tree



(b) Abstract Syntax Tree

Figure 2.5: Example computer program represented as both a *concrete syntax tree*(CST) and an *abstract syntax tree*(AST). In the CST, the actual syntax of the program can be read from left to right in the leaf nodes. The AST uses a higher level of abstraction to represent the semantics of the program.

multi-line C programs in GE to generate caching algorithms [119] and to solve the Santa Fe ant trail problem [120]. It has been suggested that complete BNF grammars for languages such as C can be "easily plugged in to GE" [121]. But no known attempts of this appear in the literature. It seems likely that in practice it is far from easy to use GE with such large and complex grammars. One difficulty is that the context-free grammars used by GE and CFG-GP lack the expressiveness to describe the semantic constraints associated with many high-level programming constructs. For example, variable declarations needing to precede any use of a variable. Other authors [31, 37, 125] have described extensions to GE that do use context-sensitive grammars, but none have used the extensions to evolve imperative programs.

Other grammar-based approaches have been designed to make use of context-sensitive grammars, such as DCTG-GP [136] and LOGENPRO [172, 173] which uses definite clause grammars to induce programs in a range of languages, including imperative C programs. Definite clause grammars allow symbols to include arguments that can be used to enforce context-dependency. Wong and Leung demonstrate using the additional context information to enforce data-type constraints [173] and to evolve recursive structures that solve the general even-n-parity problem [174].

The parse trees used to represent programs in CFG-GP, GE and other grammar-based systems are *concrete syntax trees*(CST), as opposed to the *abstract syntax trees* (AST) used in other GP representations. Concrete syntax trees contain explicit elements of a language's syntax, while abstract syntax trees are abstracted from the specific syntax of any one language and instead model the semantic constructs themselves. In this way, ASTs are a higher level abstraction than CSTs and they can be interpreted to represent any of a number of syntactic structures. Which means that a computer program represented as an AST can be very simply converted to source code in any programming language that supports the same programming constructs, regardless of the syntax used to express them. Figures

2.5a and 2.5b show comparable CST and AST representations of the same program.

2.3 Strongly Typed Genetic Programming

Strongly Typed Genetic Programming (STGP) [109] was introduced as an alternative to maintaining the closure property, by supporting explicit data-type constraints. This was demonstrated to make GP applicable to a wider range of problems but also shown to improve performance, which Montana suggested is due to the reduced search space associated with constraining which nodes may be joined together. The operation of STGP is of some significant relevance to this thesis, since in later chapters extensions upon STGP will be introduced, so the STGP algorithm will be described in some depth here. Montana describes two forms of STGP. Basic STGP and a more advanced form supporting generic functions, which we shall refer to as *polymorphic* STGP. Since the basic form is essentially a simplified form of the polymorphic version, only the full polymorphic version shall be described here. But note that any further references to *basic* STGP are referring to STGP without support for generic functions.

STGP requires all terminal nodes (variables and literal values) to be assigned a data-type, and all non-terminal nodes are required to define a data-type for each argument and for its return value. Modifications to the algorithm enforce these type constraints to ensure all programs are formed valid, such that all program trees have a root node that returns a value of the data-type required by the problem and that all other nodes return a value which matches the data-type required by its parent node. Modifications must be made to the tree creation procedure and any genetic operators.

The responsibility of the initialisation operator is to create new program trees that abide by the given type restrictions, that is, all nodes have a return value of the data-type expected by their parent's argument or the required data-type for the problem, in the case of the root node. Initialisation is also responsible for

ensuring that each program tree is valid according to the *max-depth* parameter. These requirements are achieved with the use of a lookup table which is constructed in advance based upon the available syntax. The lookup table contains one row for each possible depth from 1 to *max-depth*. Each row contains a list of all those data-types that are valid return types from a subtree of that depth. Constructing a valid program tree using the lookup table is achieved by recursively selecting each node from the set of nodes that are able to return the required data-type given inputs of any combination from the lookup table at $depth - 1$. Some generic functions may have a one-to-many relationship between a return type and sets of argument types. That is, more than one set of argument types may produce the same return type. In these cases, one of the set of argument types must be selected from at random to determine what the required return types of the child nodes will be.

The lookup table is constructed only once for a given syntax. Row 1 contains the data-types of all terminal nodes only, since they are the only valid subtrees that may be constructed within a depth of 1. The process for filling all other rows from 2 to *max-depth* is to check the return type of each non-terminal with all possible combinations of argument types, as taken from the table at row $depth - 1$. In the case of a grow initialisation procedure, the data-types of any terminal nodes should also be added to each row, since branches are not required to extend right down to the maximum depth allowable. Therefore, a ramped-half-and-half initialisation procedure, which makes use of both full and grown trees, will require two separate lookup tables.

The mutation operator can use the same initialisation algorithm to generate a new subtree that can replace an existing (randomly selected) subtree from the program. The only modification from a standard subtree operator is that the new subtree is constructed to return the same data-type as the original deleted subtree. Similarly, the crossover operator requires only a minor modification from a standard subtree crossover operator. The first crossover point is still selected at random from all nodes in the first parent, but the crossover point in the second

parent is selected only from those subtrees that return the same data-type as the subtree from the first parent. These two subtrees are exchanged as normal. In the case that there are no subtrees of the same data-type in the second parent, then Montana advises returning the parents or nothing (and presumably selecting an alternative genetic operator).

STGP has been widely used and a number of extensions have been suggested. Haynes et al [68] added type inheritence and Harris [66] explored uses for STGP in constraining hierarchical structure.

2.4 Search-Based Software Engineering

A growing area of research is the application of search-based techniques, including genetic programming, to software engineering tasks. A detailed survey of such research in 2009 [65], found that of more than 500 papers on search-based software engineering (SBSE), over 60 used some form of GP technique. In this section some of the more interesting or significant applications of GP to SBSE will be explored.

Most aspects of the traditional software development process have received some attention from research on using metaheuristic algorithms, but by far the greatest focus of their use has been with application to testing. GP is particularly well adapted for the generation of test cases, where the order of method calls is of significance. Emer and Vergilio [42, 43] used a grammar-guided GP approach to generate valid imperative C code to use as mutants in a mutation[2] testing strategy. More recently, the testing of object-oriented programs has received attention [132, 133, 141, 162, 163].

Arcuri [9] evolved programs that conformed to a given specification while simultaneously evolving a population of unit-tests that tested a program's conformity to that same specification. A co-evolutionary approach to fitness evaluation was used whereby individuals in the testing population were rated according to

[2]The term 'mutation' is used here in reference to the well known software testing strategy, with no association to the mutation genetic operator implied.

their ability to make programs fail, while individuals in the program population were evaluated according to the number of unit tests they were able to pass. A similar co-evolutionary technique was used to automatically fix bugs [8, 10], where a specification guides both unit tests and programs towards an improved solution, with the addition of an aspect of the fitness being associated with structural difference from the original (defective) program. Automatic bug-fixing of real bugs discovered in commercial software has been demonstrated by Weimer et al. [48, 164] using a novel GP technique where existing statements taken from elsewhere in the same program supply the genetic material. They also make full use of any available unit tests to identify a path of execution that contains the bug and therefore substantially reduce the search space by isolating the evolutionary modification to that path.

Genetic programming has also been used to refactor existing correct code to produce a semantically equivalent program with an improvement to some non-functional property of the code. Ryan [140] explored the use of GP for automatic parallelisation of sequential programs. The improvement of other properties of software, such as power consumption and memory usage was tackled by White et al [168], with the use of multi-objective optimisation. They demonstrated their approach with application to pseudo-random number generators, in particular for embedded systems. Jensen and Cheng [73] applied GP to produce substantial refactorings of object-oriented software to apply standard design patterns.

The SBSE field also contains a substantial bulk of work on topics of only marginal interest to this thesis, including the use of metaheuristic algorithms for project management tasks such as software development effort estimation [19, 39, 142] and quality classification [45, 80, 81, 94].

2.5 Summary

This chapter has introduced the genetic programming algorithm and reviewed the related literature. The problems associated with evolving high-level imperative

programs with GP have been outlined, along with a discussion of the existing methods for tackling them. Several of the more common alternative program representations were described, including linear GP and grammar-guided GP, with reference to their strengths and weaknesses, particularly in relation to the task of evolving imperative programs. Finally, an overview of the developing search-based software engineering field was presented, where search-based optimisation algorithms are used to identify solutions to software engineering tasks.

Chapter 3

Methodology

Throughout the work outlined in the following chapters, a number of experiments are carried out to analyse various properties of the systems and operations under discussion. This chapter outlines the common properties of the experimental set ups as used in these experiments.

3.1 General Configuration

Some of the GP configuration options are set consistently throughout this thesis for all problems and on all experiments. These are listed in Table 3.1. All other control parameters are specified for each experiment and are listed in parameter tableau for each problem. These parameter values were chosen arbitrarily. It is likely that better parameters could be chosen experimentally.

All experiments are conducted over 500 evolutionary runs on each problem, with a different random seed used for each run. This is more runs than is typically used in GP, but a large sample size like this allows us to produce much narrower confidence intervals and more statistically significant results. This is only practical because the GP system and extensions used are sufficiently fast to perform each run in several seconds in most cases. Each run continues until either the maximum number of generations is reached or a solution is found which successfully solves all training and all test cases.

Table 3.1: Listing of the control parameter settings that are used for all of the six test problems

Number of runs:	500
Population size:	500
Maximum generations:	50
Elites:	1
Selection:	Tournament selection, size 7
Crossover probability:	0.9
Mutation probability:	0.1

3.2 Software and Hardware

In all cases, experiments were conducted using the open source EpochX evolutionary framework [22, 126], with extensions implemented according to the specifications defined within this thesis. Both EpochX and our extensions are written in the Java programming language and where applicable make use of Java's primitive data-types. Where the duration of program evaluation is recorded, it is measured using Java's `System.nanoTime()` method. According to the Java documentation, this method has nano-second precision, but not necessarily nano-second accuracy. It is therefore important that all timings for comparison are carried out using the same Java Virtual Machine, on the same physical machine. Large sample sizes are also particularly valuable to reduce the impact of this potential lack of accuracy on our conclusions. In most cases, statistical calculations are based upon tens of thousands of programs or more. All non-deterministic behaviour was controlled by an implementation of the Mersenne twister pseudo-random number generator, as supplied in the EpochX framework.

3.3 Presentation of Results

The results of those experiments that are intended to demonstrate the performance of an algorithm or configuration option are presented in a consistent manner throughout this thesis. A table which summarises the results is given which includes information about the success rates and required computational effort to solve the problem. For an example of a typical results table, see Table 4.14. The

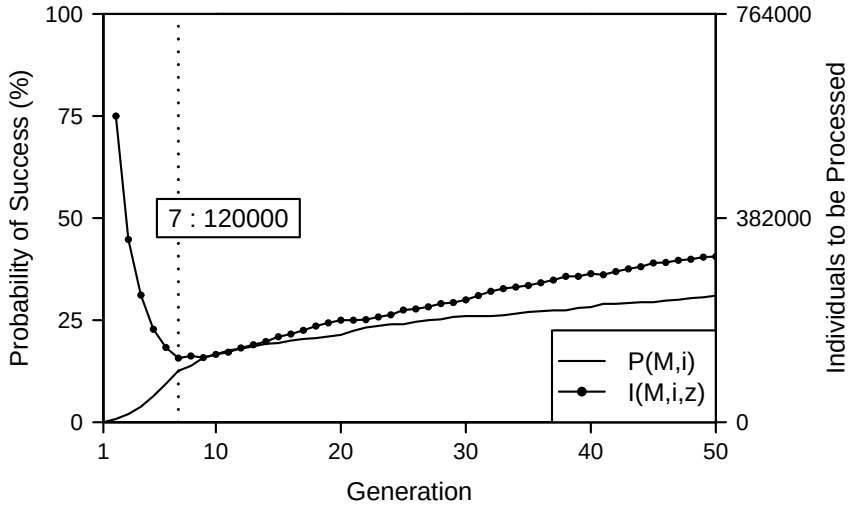


Figure 3.1: Example performance curves for a set of runs. The $P(M, i)$ curve shows the cumulative success rate and the $I(M, i, z)$ curve shows the number of individuals that must be processed to find a solution with 99% confidence.

Train% column lists the proportion of runs that produced at least one program which correctly solved all the training cases for the problem, while the *Test%* column lists the proportion of runs that produced at least one solution which solved all the test cases as well as the training cases. The *Effort* column describes the computational effort to solve each problem. Computational effort is a calculation of the number of individuals that must be processed in order to produce a solution with 99% confidence. It is calculated in the manner of Koza [87, chapter 8] but with the ceiling operator removed, as recommended by Christensen and Oppacher [27]. Confidence intervals are supplied for the computational effort, in the *95% CI* column, which are calculated using the Wilson 'score' method [159]. The final *Evals* column gives the total number of program executions that are required to find a solution, which is a product of the computational effort and the number of training inputs. Where multiple experiments are under comparison and listed in the same table, the entry with the highest success rates and lowest computational effort for a problem are listed in bold.

In addition to the results tables, performance curves are displayed which show the relationship between the number of generations and both the probability of success and the number of individuals that must be processed to yield a solution. Figure 3.1 shows an example of a performance curve. This type of graph is commonly found throughout the literature and provides a useful portrait of the evolutionary progress on a problem. The $P(M, i)$ curve shows the proportion of successful runs by generation, this is effectively a cumulative listing of the *Test%* column over the generations. The $I(M, i, z)$ curve follows the required computational effort at each generation. A vertical line is plotted on the chart to intersect the curves at the generation with the minimum computational effort. The line is labelled with the generation and the number of individuals that must be processed to find a solution with 99% confidence, which is the minimum required computational effort.

We consider computational effort to be a more important characteristic of the results than probability of success. Success rates can be a misleading gauge of quality since they are so easily manipulated by increases to the population size and the number of allowable generations. In contrast, the computational effort incorporates this information and a low value indicates that few computational resources were required to produce each solution.

3.4 Test Problem Set

In each experiment we are concerned primarily with the generation of high-level imperative programs and so the problem test suite is composed of problems requiring use of standard imperative programming constructs, including loops, arrays and variable declarations. Each of the problems have also been used elsewhere in the literature, as described below.

3.4.1 Factorial

The task to be solved here is an implementation of the factorial function. One input is provided, which is the integer variable i, where $i!$ is the expected result. The first 20 factorials (0! – 19!) were used to evaluate the quality of solutions, with a normalised sum of the error used as an individual's fitness score. The fitness function is defined in (3.1), where n is the size of the training set, i is the ith training case, $f(i)$ is the correct result for training case i and $g(i)$ is the estimated result for training case i returned by the program under evaluation. Each individual which successfully handles all training inputs (fitness of 0) is tested for generalisation using a test set consisting of elements 21 to 50 of the sequence.

$$Fitness = \sum_{i=0}^{n} \frac{|f(i) - g(i)|}{|f(i)| + |g(i)|} \tag{3.1}$$

Solutions to the factorial problem have been evolved using various representations. Object-oriented GP (OOGP) [4] uses a constrained tree structure to evolve object-oriented Java programs. OOGP produced solutions to the factorial problem with 74% probability, but they rely on large population sizes of 7,000-12,000 and so the required computational effort remains high where 600,000 individuals must be processed to find a solution with 99% confidence. The authors also note that their approach, which relies upon Java's Reflection mechanism, is computationally expensive, requiring several hours to perform just one evolutionary run. Wang et al [161] used a novel system called Function Sequence Genetic Programming (FSGP), with a linear representation and reported success rates of 75%. However, that figure is based upon only 20 runs and required 200,000 generations, so it seems likely that the required computational effort will be very high, even with a population of just 100. Other attempts have similarly required substantial computational resources, with the graph based system, GRAPE, achieving a success rate of 69% with 59% generalising, but requiring 2.5 million evaluations [145]. All these attempts have made use of recursive structures, but Wan et

al [160] compared several different loop structures within a tree GP system. Their experiments found only one loop construct that was able to produce any solutions to the factorial problem and still with only 14 of 75 runs successful.

3.4.2 Fibonacci

The Fibonacci problem is posed in a similar form as factorial, with an integer variable input `i` and an expected output which is the ith element of the Fibonacci sequence. Two further inputs are also provided in the form of variables containing the value of the first two elements of the sequence; 0 and 1. The same function (3.1) is also used to determine an individual's fitness, with the training inputs comprised of the first 20 elements of the Fibonacci sequence. A test set made up of elements 21 to 50 of the sequence is used to test the generalisation of successful programs. The Fibonacci sequence begins:

```
1 1 2 3 5 8 13 21 34 55 89 144 ...
```

Previous attempts at evolving recursive structures that can generate the Fibonacci sequence include Harding et al [63], who used Self-Modifying Cartesian GP to generate both the first 12 and first 50 elements of the sequence with success rates up to 90.8% and up to 94.5% of those able to generalise to 74 elements of the sequence. In [170], a Linear GP system was used to achieve success rates up to 92% and solutions which generalised in 78% of cases. Another linear GP system that evolved machine language programs required over 1 million evaluations, using elements 1 to 10 of the sequence as training inputs, but with all solutions shown to generalise to the infinite series [71]. OOGP was also applied to the Fibonacci problem with success rates of only 25% on the first 10 elements of the sequence and requiring a minimum computational effort of 2 million. Weaker results still were presented in [145], with a success rate of only 6% on their test set.

3.4.3 Even-n-parity

The boolean parity problems are widely used as a benchmark task in the GP literature [17, 23, 87]. However, they have only occasionally been tackled in the general form; for all values of `n`. A program which successfully solves the even-n-parity problem, must receive as input an array of booleans, *arr*, of unknown length and must return a boolean `true` value if an even number of the elements are `true`, otherwise it must return `false`. All eight boolean arrays of length 3 are used for training data. The fitness of an individual is then a simple count of how many of inputs are incorrectly classified. It is perhaps surprising that such a restricted set of inputs is able to produce solutions to the general problem. However, the same set of test inputs were successfully used by Wong and Leung [174] and they require considerably less evaluations per program than larger sets of inputs comprised of multiple sizes. A test set consisting of all possible input arrays of lengths 4 to 10 is used to test the generalisation of solutions that successfully solve the training cases (fitness of 0).

The even-parity problems are considered to be difficult problems for GP to solve [51,64]. Koza's experiments required 1,276,000 individuals to be processed to yield a solution to just the 4-bit version of the problem and was unable to solve the problem with any higher number of bits without the use of automatic functions. Other research has tackled the general even-n-parity problem. OOGP [4] required 680,000 individuals to be processed, where they used all the inputs for the even-2-parity and even-3-parity problems as training data. In contrast, Wong and Leung [174] used LOGENPRO, their logic grammar-guided system, with just the 3-bit training inputs and found its required computational effort to be only 220,000 individuals.

3.4.4 Reverse List

A solution to the list reversion problem must receive as input a list of any length and return a list of the same length, with the order of the elements reversed. In the

experiments in this thesis a list of characters is used, but any element data-type could equally have been used. The same five randomly generated lists of lengths 9..10 elements are used as the training inputs in all experiments:

```
[U,V,B,L,N,U,G,D,A,H] [X,I,D,L,O,I,R,P,W] [I,A,D,B,E,G,K,U,D]
[C,R,T,U,U,U,P,W,N,M] [U,E,Q,W,G,U,O,M,O]
```

A further 30 randomly constructed lists of lengths 10..20 are used to test generalisation. The fitness of an individual is calculated as the sum of the Levenshtein distances [92], between the returned lists and the expected reverted lists. The Levenshtein distance is calculated as the minimum number of edits needed to transform one string into another, where insertion, deletion and substitution are the allowable transformations.

The list reversion problem has been extensively used as a test problem in the Inductive Programming (IP) field, where computer programs are derived from specifications. However, it has only rarely been attempted with Genetic Programming, most likely because it requires much the same approach as sorting a list, but it lacks the general appeal of sorting algorithms, which are more widely seen in the GP literature. Shirakawa et al [145] used GRAPE with training lists of lengths 5..10 and found programs that were able to correctly reverse these lists in 71% of runs and in 65% a solution generalised to correctly reverse a test set made up of lists of lengths 11..15. They allowed each run to progress to 2.5 million evaluations. Another use of list reversion is found in [171], where the authors test a new representation with a separate linear genotype and statement based phenotype. Their results found solutions in 44 out of 50 runs, in an average of 117 generations. PushGP [148] has also been used to successfully evolve list reversing programs, but unfortunately the authors do not report their success rates or required computational effort.

3.4.5 Sort List

The task of sorting a list involves arranging the elements of a given list into order. Sorting algorithms attract much attention within computer science and are widely studied. Many different algorithms are known and established, with different compromises made with regards to complexity, run-time and memory usage. Some of the more well known sorting algorithms include bubble sort, quicksort, merge sort and insertion sort. The problem as we propose it for GP, requires the sorting of a list into ascending order. The quality of a solution is judged in a similar way as used for the reverse list problem. Five randomly generated training lists of lengths 9..10 elements are used, with the Levenshtein distance between the returned list and the correctly sorted list used as the fitness score. Generalisation is determined based upon a further 30 random lists of lengths 10..20 elements. Lists of characters (A..Z) are used, although any data-type with a natural ordering could equally have been used. To ensure consistency and fair comparisons, the same five lists are used in all experiments:

```
[T,E,L,K,R,D,B,O,M,L] [U,C,L,B,A,E,R,D,E] [B,K,Q,E,D,O,R,H,Q,K]
[U,U,Z,T,Q,P,R,Q,K] [C,O,F,R,N,X,T,B,D,I]
```

The evolution of sorting algorithms have been attempted on numerous occasions in the literature [2, 3, 5, 82, 83, 123, 124, 144, 148]. Many of the earliest attempts had limited success. O'Reilly and Oppacher [123] were unable to find any solutions that correctly sorted all their training cases, with their runs suffering from premature convergence. Although they suggest the problem was with the available syntax, it seems likely that their results were at least in part caused by the use of a rather coarsely grained fitness function that only rewarded for elements that were correctly positioned. This view is validated by Kinnear's [82, 83] work which used a more subtle measure based upon the disorder of the sequence and was able to reliably produce sorting programs that could generalise to his test set of 300 random lists of lengths up to 40 elements. However, he does use some problem specific operations such as *order*, which swaps two elements if the

first is larger than the second. Abbott and Parviz [2] criticise this approach, although they do use an *insertAsc* method (which inserts an element in the correctly sorted position) in their own experiments. They justify the use of this method by demonstrating that the OOGP system they use is capable of evolving this method separately.

More recent efforts have focused on improving the efficiency of the evolved sorting programs, by moving away from bubble sort like algorithms based upon simply swapping elements, or other $O(n^2)$ naïve sorting algorithms. In [3] a sorting algorithm of $O(n \times log(n))$ time complexity was evolved. They used a recursive rather than an iterative approach and supplied a *filter* method, which is a higher-order function used in the implementation of quicksort. In a second set of experiments they co-evolved the filter method. Their success rates were up to 46% with a minimum $I(M, i, z)$ of 3,360,000. All solutions were shown to generalise against a test set of 200 random lists of lengths up to 100. The work that we present here uses sorting as a test problem and does not use the efficiency or time complexity of the algorithm as an objective. We do however consider it important to avoid supplying problem specific components.

3.4.6 Triangles

This problem is based upon an exercise from an introductory programming textbook [56]. It is an appealing notion to try to learn solutions to the sorts of problems that novice human programmers begin with. One integer input, n, is supplied which identifies the height and width of the triangle that should be produced. The program is then required to construct a string which when printed would form a triangle of the correct dimensions. To our knowledge, this problem has not been attempted with GP previously. The correct responses for values of n from 1 to 5 would be:

```
*          *          *          *          *
           **         **         **         **
                      ***        ***        ***
                                 ****       ****
                                            *****
```

The fitness function for the triangles problem is based upon the number of incorrect rows and the number of incorrect characters in each row. The total fitness score is a sum of the score obtained on each of the training cases, obtained using the function defined in (3.2). e is the estimated triangle produced by the program under evaluation for this training case and r is the correct result. The function $m(x, y)$ is the maximum number of rows in triangles x and y, where rows are defined as being separated by new line characters. x_i refers to the ith row of x and $len(x_i)$ is the length, or number of characters in row x_i.

$$Training\ case\ score = \sum_{i=1}^{m(e,r)} |len(e_i) - len(r_i)| \tag{3.2}$$

Earlier fitness functions that were attempted included treating the outputs as a simple string and simply counting the number of incorrect characters or using the Levenshtein distance. However, these approaches led to simple programs (using only one level of iteration) that produced solutions of the correct length being rewarded excessively. There was little incentive towards inserting new line characters in the correct positions, which is the most challenging aspect of the problem. The fitness function that was used rewards based upon both the number of rows and the length of those rows.

Chapter 4

Strongly Formed Genetic Programming

4.1 Introduction

In this chapter, we introduce Strongly Formed Genetic Programming (SFGP), a novel approach to constraining the structure of the program trees evolved with GP. Currently, the most reliable way of constraining the structure of programs evolved with GP is with a grammar-guided approach, where a syntax grammar defines the allowable syntax. However, one of the disadvantages of grammar-based systems is that they require a grammar to be provided for each problem. This is a particular weakness if the aim is automatic generation of software, since it merely shifts the problem of writing a program to one of writing a grammar. Tree-based systems avoid this issue as the components only need to be written once and for each problem it is simply a case of choosing which components to include. However, no techniques currently exist for supporting a similar level of structural constraint in the classic tree representation as are found in grammar GP. We propose SFGP as a solution to this.

SFGP extends previous work by Montana, with Strongly Typed Genetic Programming [109] and combines it with constraints similar to those used by Koza in his work on *constrained syntactic structures* [87]. The addition of structural

constraints opens a range of possibilities, including the support of more powerful iterative programming constructs and the enforcement of a program structure that corresponds to a programming paradigm other than the usual functional style. Its use for these purposes will be explored throughout this chapter, with emphasis on evolving programs with a naturally imperative structure and supporting common high-level imperative constructs such as loops, arrays and variable assignment. A series of experiments will also be conducted to investigate the performance impact of these modifications.

4.2 The Algorithm

Strongly Formed Genetic Programming (SFGP) is a technique for evolving program trees that conform to strict structural constraints. It inherits strong data-typing restrictions from Montana's Strongly Typed Genetic Programming (STGP) system, which it extends. STGP provides a mechanism for constraining the data-type of each non-terminal node's inputs. However, no limitation may be placed on which terminal or non-terminal is attached as the child node that provides that input. This is most easily explained with an example. Consider a type of node that performs the variable assignment operation. Any non-trivial imperative program is likely to require such a node. This `Assignment` node will require two children: a variable and an expression, which returns a value of the same data-type which is to be assigned to that variable. STGP can easily constrain these two children to be of the same data-type, but requires additional constraints to limit the first child to be a `Variable` node, rather than any other node of that data-type. Similarly, it is not possible to constrain a code-block construct to contain only statements, or loop constructs that require a variable to update with an index or element. This is the issue that SFGP provides a solution to.

STGP imposes a requirement of all terminals and non-terminals to define the data-type of their return value and a further requirement of all non-terminals to define the required data-type of each of their arguments. SFGP has the same

requirements, with one addition: all non-terminals must also define the required *node-type* for each of their arguments. The node-type property of an argument is defined as being the required terminals or non-terminals that can be a child node at this point, which when evaluated will return the value of the specified data-type. These constraints are then satisfied throughout the evolutionary process by modifications to the initialisation, mutation and crossover operators, as will be described in the rest of this section. This provides a mechanism for both ensuring certain constructs have access to the components they require and for imposing an explicit structure upon the program trees that are generated. In the case of the `Assignment` example, these constraints are sufficient to state that the first child must not only be of the same data-type as the second, but must specifically be a `Variable` node.

4.2.1 Initialisation

SFGP uses a grow initialisation procedure [87, chapter 6.2] to construct random program trees. Each node is selected at random from those with a compatible data-type and node-type required by its parent (or the problem itself for the root node). Montana's grow initialisation operator [109] made use of lookup tables to check whether a data-type is valid at some depth, but the addition of a second constraint excessively complicates these tables. The alternative is to allow the algorithm to backtrack when no valid nodes are possible for the required constraints. At each step, if no valid nodes are possible within the available depth, then the function returns an error and if the construction of a subtree fails with an error then an alternative node is chosen and a new subtree generated at that point. This approach is simpler than the one taken by Montana, but it does reduce the algorithm's potential to support generic functions. This is discussed further in section 4.4.1. The algorithm ensures that all program trees that are generated satisfy all data-type and node-type limitations and that each tree is within the *maximum-depth* parameter.

Pseudo-code for the grow initialisation algorithm is listed in Algorithm 4.1. The `generateTree` function is initially called with a `dt` parameter that is the required return type for the problem and an `nt` parameter which defines the node-type required for the root of the program tree. A full initialisation procedure [87, chapter 6.2] would also be possible, by adapting the grow initialisation algorithm to select only non-terminal nodes, if available, until the *maximum-depth* -1 is reached. The grow initialisation procedure is preferred here for its simplicity and its tendency to produce a more diverse range of depths to the full method.

Algorithm 4.1 High-level pseudocode of the initialisation procedure in SFGP. dt, nt and $depth$ are the required data-type, node-type and maximum depth. The $filterNodes(S, dt, nt, depth)$ function is defined to return a set comprised of only those nodes in the available syntax, S, with the given data-type and node-type and with non-terminals removed if $depth = 0$.

```
1:  function GENERATETREE(dt, nt, depth)
2:      V ← filterNodes(S, dt, nt, depth)
3:      while V not empty do
4:          r ← removeRandom(V)
5:          for i ← 1 to arity(r) do
6:              dti ← required data-type for ith child
7:              nti ← required node-type for ith child
8:              subtree ← generateTree(dtr, nti, depth − 1)
9:              if subtree ≠ err then
10:                 attach subtree as ith child
11:             else
12:                 break and continue while
13:             end if
14:         end for
15:         return r                          ▷ Valid subtree complete
16:     end while
17:     return err                            ▷ No valid subtrees exist
18: end function
```

As an example, consider the syntax in table 4.1, with one `A` non-terminal, two `B` non-terminals and three terminal `C` nodes. The initialisation procedure would then construct individuals in the following way, with the partial program tree at each stage shown in Figure 4.1. The root node in this example is required to have a node-type of `A` and a data-type of `Integer`.

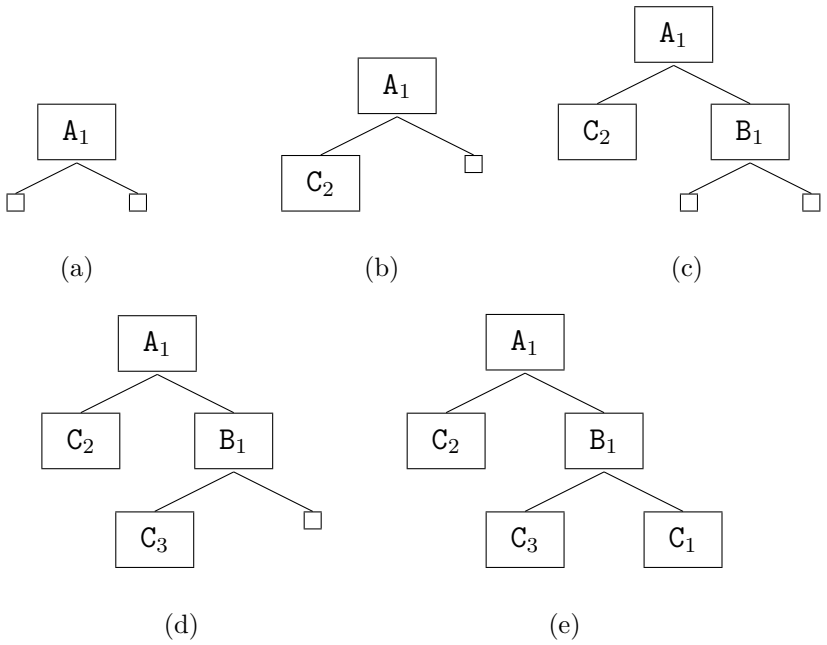


Figure 4.1: Example illustrating the steps of the SFGP grow initialisation procedure, using the syntax from table 4.1. The small, empty nodes indicate nodes yet to be filled by the algorithm.

(a) The initialisation procedure starts by choosing a root node. Only one of the five nodes in the syntax has the required node-type and data-type, so `A` is selected as the root. It requires two child nodes.

(b) The type list shows that the first child of $\texttt{A}_1$ must be of node-type `C` and data-type `Integer`. Three nodes have an appropriate data-type, but only $\texttt{C}_2$ and $\texttt{C}_3$ also have the required node-type. One of these two possible nodes is selected at random, in this case $\texttt{C}_2$. It requires no child nodes.

(c) The initialisation procedure returns to the root, to fill the second child. This node must have a node-type of `B` with a `Boolean` data-type. There are two nodes in the syntax, $\texttt{B}_1$ and $\texttt{B}_2$, that match these type requirements. One of these is randomly chosen and set as the second child of the root. In this case $\texttt{B}_1$ has been selected. It requires two child nodes.

(d) The first child of all $\texttt{B}_1$ nodes must have a node-type of `C` and a data-type of `Integer`. These are the same requirements as in step (b) and the same two

nodes from the syntax match the required types. This time the $\texttt{C}_3$ node is randomly chosen. It requires no child nodes.

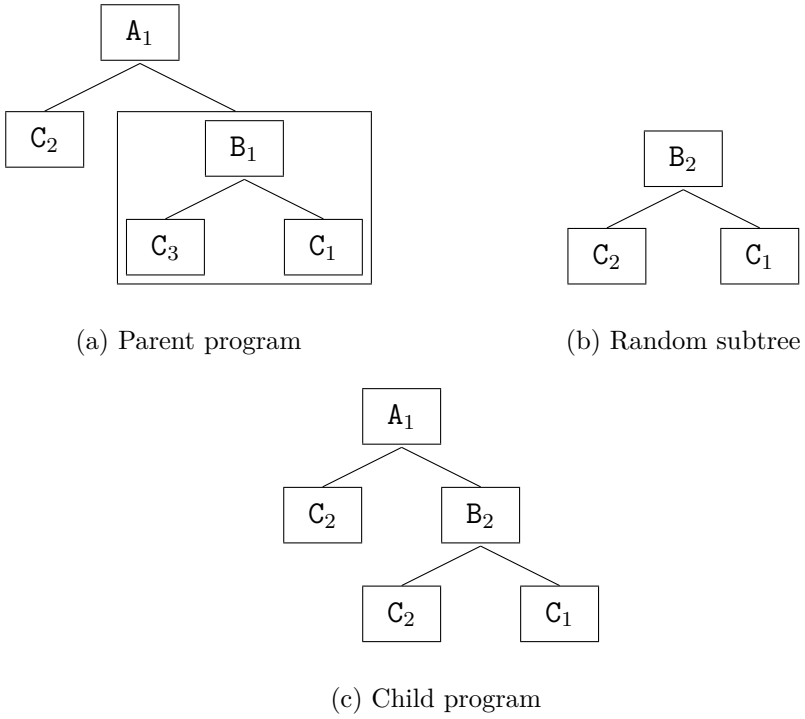
(e) Finally, the second child of the $\texttt{B}_1$ node is selected. The type list shows that it must be a `Boolean` node with a node-type of `C`. There are three nodes in the syntax with the required data-type, $\texttt{B}_1$, $\texttt{B}_2$ and $\texttt{C}_1$, but only the last of these has the required node-type, so $\texttt{C}_1$ is set as the second child. It requires no child nodes and so the initialisation procedure returns the completed program.

4.2.2 Mutation

The mutation operator employs the initialisation algorithm to grow new subtrees of the same data-type and node-type as an existing randomly selected node in a program tree. This node is then replaced with the newly generated subtree. Assuming the set of available nodes is unchanged, then it will always be possible to generate a legal replacement subtree for any existing node, but it is possible that the replacement is syntactically or semantically identical to the existing subtree. It is possible that this could lead to a high degree of neutral mutation if the syntax contains little variety, which may mean an inefficient search of the fitness landscape. In the example, shown in Figure 4.2, the second child of the root $\texttt{A}_1$ node has been selected to be replaced. The list of type constraints for the example

Table 4.1: Type list for an example syntax, showing the data-type and node-type constraints for each type of node

Node	*Data-type*	*Child data-types*	*Child node-types*
$\texttt{A}_1$	`Integer`	`Integer` `Boolean`	`C` `B`
$\texttt{B}_1$	`Boolean`	`Integer` `Boolean`	`C` `C`
$\texttt{B}_2$	`Boolean`	`Integer` `Boolean`	`C` `C`
$\texttt{C}_1$	`Boolean`		
$\texttt{C}_2$	`Integer`		
$\texttt{C}_3$	`Integer`		



(a) Parent program

(b) Random subtree

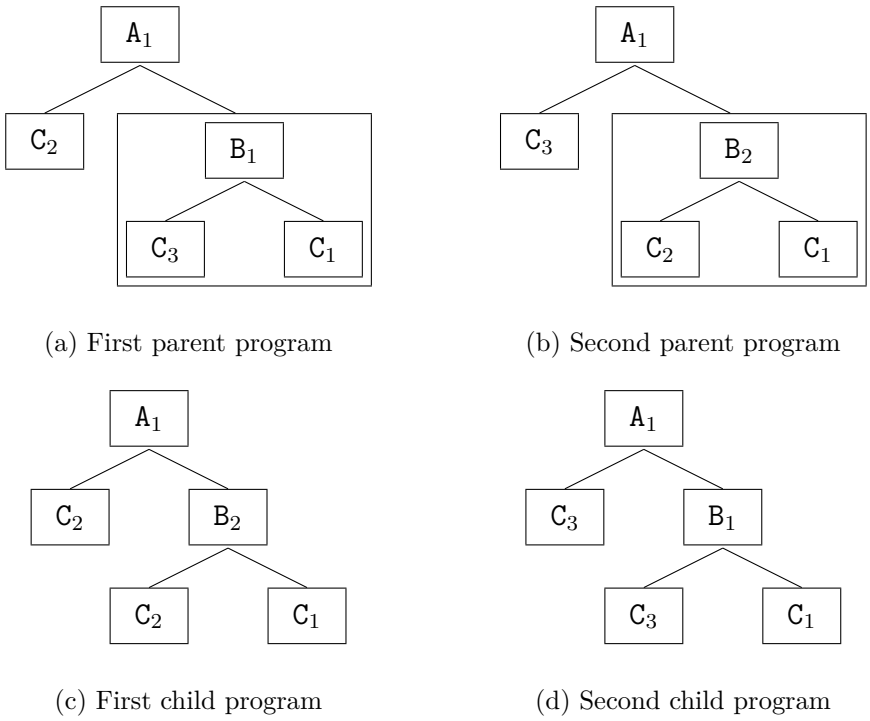
(c) Child program

Figure 4.2: Example illustrating the SFGP subtree mutation operator

syntax, shown in Table 4.1, shows that the second child of an $\texttt{A}_1$ node is required to have a node-type of `B` and a data-type of `Boolean`. The initialisation procedure is used to randomly produce a subtree which is rooted at a node with these type properties. The original subtree is then substituted with this new one, to create the new child. Had the $\texttt{C}_1$ node been selected as the mutation point, the resulting child program would have been identical to the parent, because there is only one possible node in the syntax with the required data-type and node-type.

4.2.3 Crossover

The subtree crossover operator has been modified to maintain the node-type constraint while exchanging genetic material between two program trees. A node is selected at random in one of the programs. Then a second node is chosen at random from those nodes in the other program that are of the same data-type and node-type as the first node. The subtrees rooted at these two selected nodes are then exchanged. Those resultant child programs that have depths that exceed the



(a) First parent program

(b) Second parent program

(c) First child program

(d) Second child program

Figure 4.3: Example illustrating the SFGP subtree crossover operator. The crossover point in the first parent is selected at random and the crossover point in the second parent is selected from those with a compatible data-type and node-type.

maximum depth parameter are discarded. Figure 4.3 shows an example crossover, performed on two programs formed from the syntax in Table 4.1. Once the $\mathtt{B}_1$ node is chosen as the crossover point in the first parent, there is only one valid crossover point in the second parent, which has the required data-type and node-type. It is possible that the second parent in a crossover may not have *any* compatible crossover points, in which case an alternative crossover point is selected from the first parent. Once there are no further crossover points to attempt, the crossover is discarded and new individuals are selected.

4.3 Enhanced Iteration and Variable Assignment

Repetition of a process is a fundamental concept within computer programming. However, the use of repetition in GP has often been avoided due to the problem of

potentially infinite loops. Due to the undecidable nature of the halting problem, this is ensured to be a non-trivial task. Wijesinghe and Ciesielski [169] propose that there are two approaches to using repetition in GP: *implicitly* and *explicitly*. They explain that implicit repetition is integrated into the problem and involves a solution being executed multiple times. This is the typical approach taken for artificial ant problems such as the Santa Fe trail [87], where programs are repeatedly executed until a set number of timesteps are used up. This completely avoids the need for any looping constructs undergoing the evolutionary process, but is only appropriate for a limited number of problems and it imposes the same looping structure on all programs. On the other hand, an explicit approach puts the repetitive behaviour into a node that may be harnessed by the algorithm. In this research we consider only the explicit approach.

There are examples of both recursion and loops being used in GP, with various approaches taken to ensure all programs terminate. The more common techniques include limiting the number of iterations of each loop [29, 47, 88] or the total number of iterations in a program [83]. In other cases, restrictions are not used, but programs are penalised or even removed from the population entirely, if their execution time is excessive [70, 174]. Maxwell [102] experimented with unbounded iteration, where programs were evaluated in parallel and assigned partial fitness based upon their progress in relation to other programs. This allowed programs with infinite loops to not only be tolerated but to participate in the evolutionary process based upon their partial fitness score. Recursion is a more typical method for introducing repetition to functional programs, so has been the preferred approach for a number of studies [4, 18, 167, 174]. However, according to Brave [18], recursion is difficult to evolve not just because of the possibility of infinite recursion, but also because the recursive structure suffers from low locality, where small variations to it can result in a large fitness change. This fragility is reiterated by Moraglio et al. [110], who describe an approach where a non-recursive solution is evolved first and used as scaffolding to produce an optimum recursive solution.

Because of the emphasis on evolving imperative programs in this thesis, iterative techniques are of more relevance here than recursive ones.

Reproducing even the bounded iterative constructs from modern imperative programming languages is not simple. There is no mechanism for a GP node which represents a loop to declare its own variable, that it can use for supplying an index or element. In some cases this has led to very simplified loop forms being used, which just repeatedly evaluate an expression a set number of times, without supplying access to context information such as the index or element [109]. Elsewhere, an improved approach has been used, where an existing variable in the syntax is assigned to be used by the loop for providing the relevant context [29, 83, 88]. Although a substantial improvement, this can be rather intricate in more complex cases involving multiple nested loops and the variable used by a loop structure is not subject to evolution. A comparable situation exists for variable assignment.

In his paper on STGP [109], Montana explains how a `SET-VAR-`x operator can be supplied for each variable, x, to make that variable updateable. This was an extension of the `SET-SV` function suggested by Koza [87, chapter 18.2]. Having to supply an additional operator per variable is a little unwieldy. A preferable situation would be if the operator could be disconnected from the variable, so that it could be used with any variable. This would be even more useful in a situation where new variables can be declared by the evolved programs, an issue which will be tackled in chapter 5 of this thesis. With structural constraints enforced by SFGP, this is at least partially solvable. One of the child nodes of an assignment operator can be constrained to be a `Variable` node of the same data-type as the value to be assigned to it. It is then a simple process for the value of that variable to be updated with the value to be assigned. The same solution applies to loop constructs, where one of the children can be restricted to being a `Variable` of an integer data-type in order to hold an index, or the element data-type of the array to be iterated over. That `Variable` can then be updated upon each iteration.

New iteration and assignment constructs based on these ideas will be outlined in the following section and used in the experimental work throughout this thesis. In section 4.6, an experimental study will be also be performed which compares SFGP with these operators to the alternative approach where assignment operators are associated with specific variables.

4.4 Evolving High-Level Imperative Programs

High-level imperative programs can be represented as trees and often are represented as trees as part of the parsing and compilation/interpretation process. So, evolving program trees that represent high-level imperative programs should be possible with tree-based genetic programming. However, the many structural rules that must be abided by make them difficult to evolve. It is for this reason that functional programs comprised solely of nested expressions are the norm for tree-based genetic programming representations. However, the mechanism for enforcing structural constraints that have been presented in this chapter make it possible to evolve programs that abide by the necessary rules.

We consider the main structural components of an imperative program to be statements, blocks and sub-routines. A sub-routine is composed primarily of a block, which may or may not need to return a value, depending on whether it is a function or a procedure. A block is a sequential list of one or more statements and a statement is an instruction with a side-effect. A statement may also contain one or more nested blocks. In all the work presented here, an individual represents a sub-routine[1]. In this section, the SFGP nodes used to represent these components and achieve the imperative structure are presented.

[1]where used, the term 'program' to refer to an individual can be considered to be synonymous with 'sub-routine' and 'individual'

4.4.1 Polymorphism and Generic Functions

SFGP supports a simple form of polymorphism for both the data-type and node-type constraints. Figure 4.4 shows the basic structure of all imperative programs generated in the work presented in this thesis. The `ReturnBlock` node is shown to have a sequence of children with a `Statement` node-type. In an object-oriented implementation, this could be interpreted as any object that is an instance of the `Statement` class, or any sub-class. Nodes such as `Assignment`, `IfStatement` and `ForLoop` may then be implemented as sub-classes of `Statement` and may all appear in this position. In fact, it makes little sense to create a node of the type `Statement` itself, it is merely used to maintain the hierarchy of node-types. We refer to such node-types as *abstract* node-types. `Expression` is also an abstract node-type, as is `Node`. `Node` is the parent type of all nodes and so can be used to specify that there is no node-type constraint to enforce. Data-type constraints can make use of the same polymorphic properties. If `Integer` and `Float` are both sub-classes of a class called `Number`, then either may appear where a required data-type of `Number` is specified. Note that the object-oriented approach we refer to here is a property of the implementation, rather than of the evolved programs, which are not themselves object-oriented.

In contrast to this, STGP supports full generic functions which are able to handle multiple sets of input data-types and return values of a variety of data-types. This is made possible with the use of a lookup table that defines the data-types that can potentially be returned given a certain depth. Constructing such a table that also incorporates information about the node-types that can provide each data-type is possible, but it substantially complicates the process. There would certainly be some value in considering this in future work, but at this stage we consider the basic level of polymorphism supplied by class inheritence to be sufficient for the initial aims of this work. Furthermore, the extensions that are presented in chapter 5 would not be possible with a system that relies on a pre-processing step to generate lookup tables.

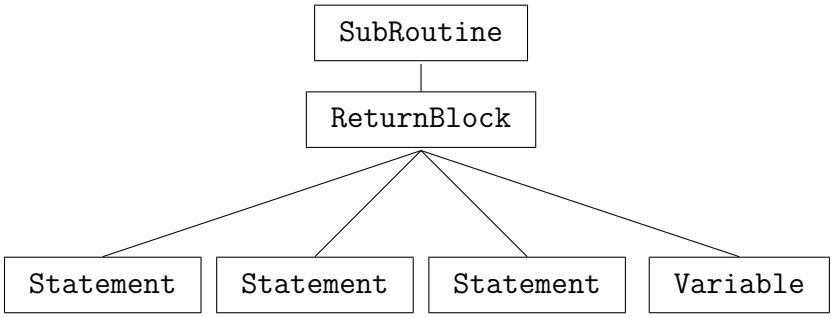


Figure 4.4: The imperative structure imposed on all program trees. All experiments used `CodeBlocks` requiring 3 statement arguments, except where otherwise stated.

4.4.2 Syntax

This section itemises the list of nodes that will be used to evolve high-level imperative programs in SFGP, along with their data-type and node-type requirements. The required root node-type for all the imperative programs that are evolved here is `SubRoutine`, which models a sub-routine with a block of statements and a return value. This means that all programs that are generated have the same basic imperative structure, as shown in Figure 4.4. Table 4.2 specifies the type constraints for the nodes that model the structural components which have the following semantics:

- `SubRoutine` - upon evaluation, the return-block child is evaluated, with the result returned as the result of the sub-routine.
- `ReturnBlock` - semantically the same as a code-block, but with an additional child `Variable` which supplies a return value.
- `CodeBlock` - consists of a series of statements which are evaluated in sequence. The number of statements is modifiable, but was arbitrarily chosen to be three in all cases in this thesis (except where the impact of this setting is examined in section 4.7.1).

These structural nodes are supplied for all problems to support the desired imperative structure. But, the actual computational work is performed by statements.

As already mentioned, `Statement` is an abstract node-type, of which many concrete node-types exist. The set of statements provided to SFGP defines the range of actions that can be performed. The statements that are used in this work are largely based upon standard programming constructs, that are basic features of most modern imperative programming languages. The only possible exception to this is the `SwapElements` statement, which we consider to be a sufficiently general component for it to be provided by a common library. The following list details the semantics of all the statements that are made use of in this chapter, with the related type constraints outlined in Table 4.3.

- `Loop` - the first child expression is evaluated to provide a number of iterations to perform (which is capped at 100) and then the code-block child is evaluated the specified number of times. No variables are manipulated by this loop construct.

- `ForLoop` - the second child is evaluated to provide an integer, which is used as the number of iterations to perform (capped at 100). Then the third child is evaluated this number of times. The index of the iteration is assigned to the variable, which is set as the first child, starting at 1.

Table 4.2: Type list for the *structural* nodes of the imperative syntax, showing the data-type and node-type constraints for each type of node. d indicates a pre-specified data-type and a `Void` data-type indicates that no value is returned.

Node-type	*Data-type*	*Child data-types*	*Child node-types*
`SubRoutine`	d	d	`ReturnBlock`
`ReturnBlock`	d	`Void`	`Statement`
		`Void`	`Statement`
		`Void`	`Statement`
		d	`Variable`
`CodeBlock`	`Void`	`Void`	`Statement`
		`Void`	`Statement`
		`Void`	`Statement`

- `ForEachLoop` - the second child is evaluated to provide an array. The third child is evaluated once per element of that array, with the current element assigned to the variable supplied as the first child.
- `IfStatement` - the code-block child is conditionally evaluated only if the expression evaluates to true.
- `Assignment` - both inputs are required to have the same data-type specified upon construction. Upon evaluation, the expression is evaluated and the result is assigned as the value of the variable.
- `ElementAssignment` - the variable should be of some pre-specified array data-type. The second child supplies an integer which is used as an index into the array, with the value of that element assigned to the value of the third child. The index is protected from being out of the bounds of the array. If it is less than zero then zero is used and if it is greater than the largest element in the array then length-1 is used.
- `SwapElements` - the two integer arguments are treated as indexes and on evaluation, the elements of the array (which is given as the first argument) at the two integer indexes are exchanged. The indexes are protected as for `ElementAssignment`.

Many statements make use of expressions which perform some calculation and return a value. These are modelled as subtypes of an abstract `Expression` node-type and each have a non-void data-type. The type constraints for the expressions used are given in Table 4.4 and the semantics are listed below.

- `Add`, `Subtract`, `Multiply`, `Divide` - each performs the relevant arithmetic operation and returns the result. Division is protected against a divisor of zero and returns a zero value.
- `And`, `Or`, `Not` - perform the relevant boolean operator and return the boolean result.

Table 4.3: Type list for the `Statement` nodes of the imperative syntax, showing the data-type and node-type constraints for each type of node. d indicates a pre-specified data-type and $d[]$ indicates an array of elements of the data-type d. A `Void` data-type indicates that no value is returned.

Node-type	*Data-type*	*Child data-types*	*Child node-types*
`Loop`	`Void`	`Integer` `Void`	`Expression` `CodeBlock`

`ForLoop`	`Void`	`Integer`
		`Integer`
		`Void`
			`Variable`
			`Expression`
			`CodeBlock`

`ForEachLoop`	`Void`	d
		$d[]$
		`Void`
			`Variable`
			`Expression`
			`CodeBlock`

`IfStatement`	`Void`	`Boolean` `Void`	`Expression` `CodeBlock`
`Assignment`	`Void`	d d	`Variable` `Expression`

`ElementAssignment`	`Void`	$d[]$
		`Integer`
		d
			`Variable`
			`Expression`
			`Expression`

`SwapElements`	`Void`	$d[]$ `Integer` `Integer`	`Expression` `Expression` `Expression`

- `GreaterThan` - the two input data-types must be comparable. The data-type is pre-specified, but it is only used with a character data-type here and only characters A–Z are used. It returns a boolean value which will be true if the first input is strictly larger than the second, where for alphabetic characters 'Z' is considered larger than 'A'.
- `ArrayLength` - it returns an integer value which is the total number of elements in the given array.
- `ArrayElement` - returns the element at the specified index. The array data-type must be specified on construction. The indexes are protected as for `ElementAssignment`.
- `Concat` - the returned string is the received string with the given character appended.
- `Literal` - holds a fixed literal value of a given data-type.
- `Variable` - holds a value of a given data-type which may be modified (by assignment) throughout evaluation. The data-type of a variable is fixed at construction.

As stated, a number of these node-types require protection from invalid values in a way that is a departure from the functionality of any standard programming language. This is necessary to avoid the need for exception handling which is currently outside the scope of this work. Although these protected versions of operations are not generally found in the programming languages themselves, they can easily be supplied as a common library. It is the assumption that this is the case with the source examples that are presented.

Table 4.4: Type list for the `Expression` nodes from the imperative syntax, showing the data-type and node-type constraints for each type of node. d indicates a pre-specified data-type and $d[]$ indicates an array of elements of the data-type d.

Node-type	*Data-type*	*Child data-types*	*Child node-types*
`Add`	`Integer`	`Integer` `Integer`	`Expression` `Expression`
`Subtract`	`Integer`	`Integer` `Integer`	`Expression` `Expression`
`Multiply`	`Integer`	`Integer` `Integer`	`Expression` `Expression`
`Divide`	`Integer`	`Integer` `Integer`	`Expression` `Expression`
`And`	`Boolean`	`Boolean` `Boolean`	`Expression` `Expression`
`Or`	`Boolean`	`Boolean` `Boolean`	`Expression` `Expression`
`Not`	`Boolean`	`Boolean`	`Expression`
`GreaterThan`	`Boolean`	d d	`Expression` `Expression`
`ArrayLength`	`Integer`	$d[]$	`Expression`
`ArrayElement`	d	$d[]$ `Integer`	`Expression` `Expression`
`Concat`	`String`	`String` `Character`	`Expression` `Expression`
`Literal`	d		
`Variable`	d		

4.4.3 Converting to Source

As with standard GP and STGP, programs in SFGP are represented as abstract syntax trees (ASTs), where each node in the tree represents some language construct. Where the program tree is representing a functional LISP program, there is a very direct relationship between the structure of the tree and the syntax of the program. Indeed, this is one of the reasons Koza chose to use LISP. But, this need not be the case and the syntax required to express the concept represented by a node may be something far more complex. By using nodes which represent very general high-level programming concepts, an individual in SFGP may then be expressed in the syntax of any number of different imperative programming languages. Given a *code template* for each possible node-type, which describes the structure of the source code to express a node of that type in a given language, it is a trivial process to convert from a program represented as an AST to syntactically valid source code in some language. As an example, consider the AST in Figure 4.5. Using the source code templates in Tables 4.5, 4.6 and 4.7 this program fragment can be converted to Java, Pascal or Python respectively. The template for a node-type is used by starting at the root node and recursively replacing the placeholders with the source code for the relevant child, where the ⟨*child-1*⟩ placeholder is the first child and ⟨*child-n*⟩ is the nth. The result is the source code listed in Algorithms 4.2, 4.3 and 4.4.

Throughout this thesis, example solutions are listed using Java syntax, but could equally have been represented using any number of other imperative programming languages. A complete listing of Java code templates for all the node-types used is given in appendix A.

Algorithm 4.2 Java source code generated from the AST in Figure 4.5 using the source code templates for the Java programming language, listed in Table 4.5.

```
if (x < y) {
    y = x;
}
```

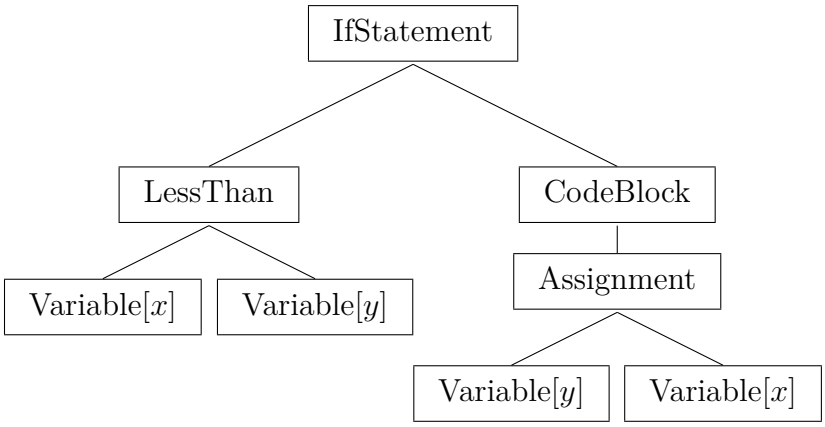


Figure 4.5: Example abstract syntax tree representing a conditional statement as it would be represented in SFGP

Algorithm 4.3 Pascal source code generated from the AST in Figure 4.5 using the source code templates for the Pascal programming language, listed in Table 4.6.

```
if x < y then
    y := x;
```

Algorithm 4.4 Python source code generated from the AST in Figure 4.5 using the source code templates for the Python programming language, listed in Table 4.7.

```
if x < y:
    y = x
```

Table 4.5: Example source code templates for the Java programming language, where ⟨*child-n*⟩ is replaced by the source code for the node's nth child. A complete listing of templates for the Java programming language is given in appendix A.

IfStatement	`if(` ⟨*child-1*⟩ `)` ⟨*child-2*⟩
LessThan	⟨*child-1*⟩ `<` ⟨*child-2*⟩
CodeBlock	`{` ⟨*child-1*⟩ ⟨*child-2*⟩ ⟨*child-n*⟩ `}`
Assignment	⟨*child-1*⟩ `=` ⟨*child-2*⟩ `;`

Table 4.6: Example source code templates for the Pascal programming language, where ⟨*child-n*⟩ is replaced by the source code for the node's nth child.

IfStatement	`if` ⟨*child-1*⟩ `then` ⟨*child-2*⟩
LessThan	⟨*child-1*⟩ `<` ⟨*child-2*⟩
CodeBlock	⟨*child-1*⟩ ⟨*child-2*⟩ ⟨*child-n*⟩
Assignment	⟨*child-1*⟩ `:=` ⟨*child-2*⟩ `;`

Table 4.7: Example source code templates for the Python programming language, where ⟨*child-n*⟩ is replaced by the source code for the node's nth child.

IfStatement	`if` ⟨*child-1*⟩`:` ⟨*child-2*⟩
LessThan	⟨*child-1*⟩ `<` ⟨*child-2*⟩
CodeBlock	⟨*child-1*⟩ ⟨*child-2*⟩ ⟨*child-n*⟩
Assignment	⟨*child-1*⟩ `=` ⟨*child-2*⟩

4.5 Imperative Experiments

To test the ability of SFGP to generate high-level imperative programs, experimental runs were conducted on six non-trivial problems, each requiring the use of branching and iterative programming constructs.

4.5.1 Experimental Setup

500 runs were performed for each of the six test problems: factorial, Fibonacci, even-n-parity, reverse list, sort list and triangles, as specified in section 3.4. The SFGP grow initialisation procedure and subtree crossover and mutation operators were defined as described earlier in this chapter, in section 4.2. The default control parameters listed in table 3.1 were used. All other control parameters were problem dependent and are outlined in Tables 4.8 to 4.13. The maximum depth parameter was set using an educated guess based on the perceived difficulty of the problem and the required complexity of a solution. The implications of setting this parameter too low are that the problem may be difficult or even impossible to solve. For example, all problems with the exception of factorial and Fibonacci require a maximum depth greater than 6, because a solution will require at least two nested constructs (loops or conditional statements) which is only possible within a node depth of 7 or greater. However, setting an unnecessarily high value for the maximum tree depth is likely to produce larger, more bloated programs, which would increase evaluation times. According to [135] smaller programs are also more likely to generalise. This highlights how the GP algorithm requires the user to have some insight into possible solutions.

Some care was taken to choose terminal and non-terminal sets that satisfied the sufficiency property with limited additional extraneous components. In particular it was considered important to supply only general-purpose components that could not be considered to be providing a key part of the required program logic. For example, the `SwapElements` non-terminal which exchanges two array elements is deemed acceptable for the sort-list problem, since it encapsulates a programming

Table 4.8: Listing of the control parameter settings used for SFGP on the factorial problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `Assignment`, `Add`, `Subtract`, `Multiply`
Terminals:	i, $loopVar$, 1

Table 4.9: Listing of the control parameter settings used for SFGP on the Fibonacci problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `Loop`, `Assignment`, `Add`, `Subtract`
Terminals:	i, $i0$, $i1$

task which is applicable to many problems, as exemplified by its use on the reverse-list problem too. In contrast, an intelligent swap, such as has been used elsewhere in the literature [83], which only exchanges two array elements if the first is larger than the second, is rejected as having only a very narrow range of problems that it is applicable for.

4.5.2 Results

A summary of the results are presented in Table 4.14, which lists the success rates, generalisability and required computational effort found in the experiments. Performance curves showing the progression of success rates and computational

Table 4.10: Listing of the control parameter settings used for SFGP on the even-n-parity problem

Root data-type:	`Boolean`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForEachLoop`, `IfStatement`, `Assignment`, `And`, `Or`, `Not`
Terminals:	arr, $loopVar$, $resultVar$, `true`, `false`

Table 4.11: Listing of the control parameter settings used for SFGP on the reverse list problem

Root data-type:	`Character[]`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `ArrayLength`, `Subtract`, `Divide`, `SwapElements`
Terminals:	arr, $loopVar1$, $loopVar2$, `1`, `2`

Table 4.12: Listing of the control parameter settings used for SFGP on the sort list problem

Root data-type:	`Character[]`
Root node-type:	`SubRoutine`
Max. depth:	10
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `IfStatement`, `ArrayLength`, `ArrayElement`, `GreaterThan`, `SwapElements`
Terminals:	arr, $loopVar1$, $loopVar2$

Table 4.13: Listing of the control parameter settings used for SFGP on the triangles problem

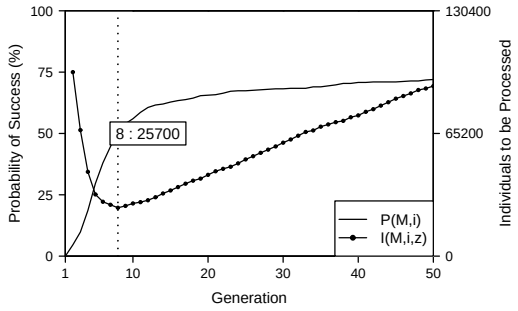
Root data-type:	`String`
Root node-type:	`SubRoutine`
Max. depth:	8
Non-terminals:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `IfStatement`, `Assignment`, `Concat`
Terminals:	n, $resultVar$, $loopVar1$, $loopVar2$, `'*'`, `'\n'`

effort over the generations, for each of the sets of runs, are displayed in Figure 4.6. Solutions which solve all training cases and test cases are found for all problems. The results show that the similar problems of factorial and Fibonacci are both solved with little difficulty and it is little surprise that the Fibonacci results exhibit both lower success rates and higher computational effort. Fibonacci is known to be a more difficult problem which requires second-order recursion when solved with a recursive approach [170]. However, as seems to be the case with many of these problems, the use of a sensible iterative approach seems to have been beneficial. These computational effort values are considerably lower than those reported from the other research that was reviewed in section 3.4, but of course this was not a controlled experimental comparison.

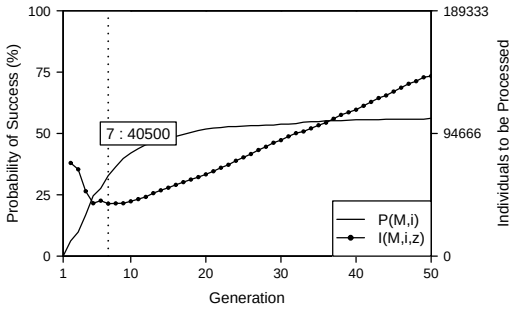
Table 4.14: Summary of the results of using SFGP to solve each of the test problems with high-level imperative programs. *Train%* is the percentage of success on the training cases (as used for fitness) and *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort to find a solution with 99% confidence and *95% CI* is its confidence interval. *Evals* is the number of program evaluations required to find a solution with 99% confidence. The approach used to calculate each of these values is described in detail in section 3.3.

	Train%	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	72.8	72.0	25,700	22,700 - 29,200	514,000
Fibonacci	59.0	56.2	40,500	34,800 - 47,400	810,000
Parity	90.2	80.0	29,500	26,000 - 33,700	236,000
Reverse	78.6	77.0	29,200	25,900 - 33,000	146,000
Sort	75.0	71.2	65,200	55,900 - 76,300	326,000
Triangles	69.6	69.6	15,900	13,900 - 18,200	95,400

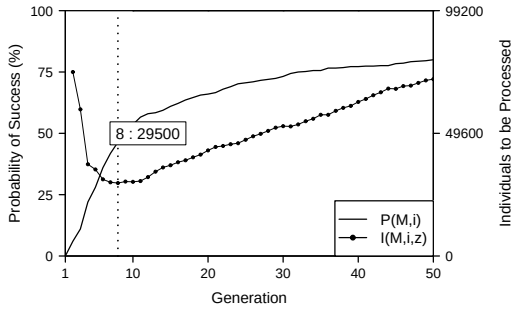
The results suggest that only 29,500 individuals need be processed to identify one solution to the general even-n-parity problem. This is in contrast to the more than 1.2 million individuals Koza's work required to yield a solution to just the 4-bit version of the problem. With over 80% of runs resulting in a solution to the even-3-parity training cases that were able to also solve the general even-n-parity problem, the decision to use just the 3-bit inputs as training data appears to be vindicated. However, it should be noted that this fact is put under question by further results presented in section 4.6. The primary reason for SFGP's greater



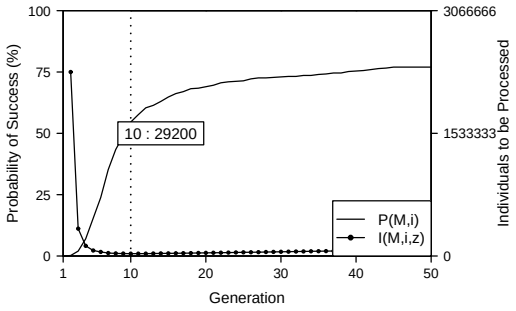
(a) Factorial



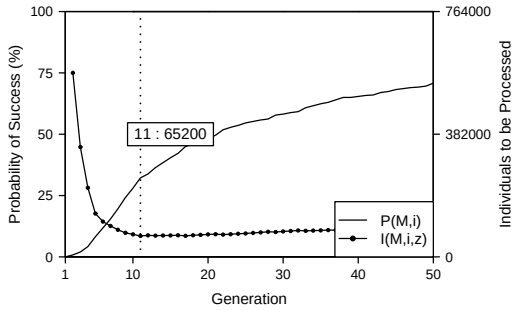
(b) Fibonacci



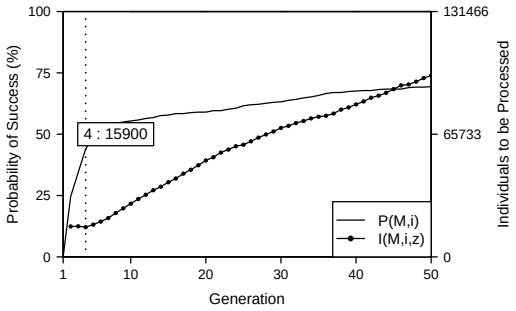
(c) Even-n-parity



(d) Reverse list



(e) Sort list



(f) Triangles

Figure 4.6: Performance curves for each of the test problems, where a high-level imperative structure was enforced with SFGP. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

performance on the even-n-parity problem is likely to be the result of using the `ForEachLoop` node, which encapsulates the necessary behaviour of performing an operation on each element of the array. The other studies described in chapter 3 relied on complex recursive structures developing through evolution.

Solutions were more readily found for the reverse-list problem than the sort-list problem. Given the components available, it is perhaps surprising that the sort-list problem is not more easily solved. A solution need only put together two nested loops with a swap, to produce a full sort. However, the intricacies of setting up the loops with sensible bounds and different variables proved to be more difficult than the issues to be overcome to solve list reversion. One of the main difficulties involved with evolving solutions to the reverse-list problem is that a near solution which iterates over the whole list, swapping the element at i with those at $(length - 1) - i$, will be given a very poor fitness score, since the resulting list will ultimately be returned to its original condition. Despite being only one small mutation away from a complete solution, such a program is highly biased against in the selection procedure. Characteristics such as this lead to a rugged fitness landscape that is difficult to navigate and are generally less suited to solving with an evolutionary algorithm. The high success rates suggest this did not prove to be a significant obstacle though.

Some trial and error (< 10 trials) was used in determining the fitness measure and training cases for the list-reversion problem. It was discovered that the choice of training lists was crucial to the performance of the algorithm. Lists of lengths 7 or less were found to lead to a high degree of solutions that did not generalise and use of a combination of shorter and longer lengths produced low success rates. The magic number of 7 is due to the use of code-blocks that support 3 statements, each of which could be a `SwapElements` statement, to reverse lists up to length 7 without the use of a loop (with an odd length, the middle element of a list need not be swapped). The use of lists of lengths 9 and 10 forced solutions to make use of loops, this contributed to the high rates of generalisation that are seen.

4.5.3 Example Solutions

Solutions to all six of the test problems were identified using the SFGP system. A few of the typical features of the solutions found will be highlighted in this section. As was described in section 4.4.3, the program trees that are produced by SFGP can be easily expressed in the syntax of any programming language which supports the required constructs. The constructs that were used throughout these experiments were chosen to be general programming constructs found in most high-level imperative programming languages, either as standard or easily provided through code libraries. The example programs in this section will be expressed using Java syntax, constructed using the source code templates shown in appendix A.

The following correct solution to the factorial problem was found in generation 23 of one run. Note that for clarity this source code is displayed using Java's `long` data-type, but in practice a `BigInteger` data-type would be more suitable, due to the maximum value of the `long` data-type only being sufficiently large to store values up to the 21st factorial before overflow occurs. The semantics of the solution are simple. A loop iterates up to the given argument, multiplying the index of the loop by a running total. Many of the solutions found to the factorial problem used a very similar approach. Lines 3–9 are all part of the loop, which requires additional Java statements to replicate the semantics of our `ForLoop` construct. In particular, the loop structure contains the necessary infrastructure to ensure the index variable is updated but the bounds remain immutable to avoid any chance of an infinite loop occuring. This program has not undergone any post-processing, but it could be simplified by static analysis. Most obviously, lines 6, 7 and 10 could all be removed, but the structure of the loop could potentially be simplified too.

```
1.   public long getFactorial(long i) {
2.       loopVar = 1;
3.       long upper = i;
4.       i = 1L;
```

```
5.       for (long x = 1L; x <= upper; x++, i = x) {
6.           i = i;
7.           loopVar = loopVar;
8.           i = (loopVar * i);
9.       }
10.      loopVar = loopVar;
11.      return i;
12.  }
```

The list reversion problem does not require nested loops, but because there is sufficient depth provided, many of the solutions make use of them. The following solution was found in generation 11 during one of the runs. Much simpler solutions are possible that just iterate over half the input array, but most of the solutions that are discovered take a far more complex approach. This highlights the need in any practical application for the complexity of solutions to be considered, with regards to both code complexity and time complexity. This point will be addressed further in the work with complexity metrics in chapter 6.

```
1.   public char[] reverseList(char[] input) {
2.       Utilities.swap(input, (loopVar2 - 2), 1);
3.       int upper1 = input.length;
4.       loopVar1 = 1;
5.       for (int x = 1; x <= upper1; x++, loopVar1 = x) {
6.           Utilities.swap(input, 2, 1);
7.           int upper2 = loopVar2;
8.           loopVar2 = 1;
9.           for (int y = 1; y <= upper2; y++, loopVar2 = y) {
10.              Utilities.swap(input, loopVar2, 1);
11.              Utilities.swap(input, 1, 2);
12.              Utilities.swap(input, 1, loopVar2);
13.          }
14.          Utilities.swap(input, (loopVar1 - loopVar1), 2);
15.      }
16.      Utilities.swap(input, 1, 2);
17.      return loopVar;
18.  }
```

4.6 A Reduced Search-Space

Montana suggested that one of the benefits of his STGP system is improved performance due to the potentially reduced search-space, courtesy of the type constraints. If that is the case, then it could be expected that SFGP, which introduces even tighter constraints, would reduce the search space even further. Of course, one of the concerns is that reducing the search-space may reduce the number of solutions within that space, or make them more difficult to locate due to a less smooth or disconnected fitness landscape. It is therefore feasible that performance could be either improved by, or degraded by, the addition of structural constaints. In order to test this experimentally, a further set of 500 runs was performed on each of the same problems as in section 4.5, but with node-type constraints removed. Where possible, identical control parameters were used, as described in Tables 4.8–4.13. However, with all node-type constraints removed, a couple of further modifications are unavoidable:

- Loops without node-type constraints are not able to define that they require a variable as a child and so loop implementations are used where their variable is predefined. One such loop node is added to the syntax for each applicable variable in the syntax. The logic of the loops used are identical.

- Similarly, `Assignment` nodes are unable to specify that the first argument should be a variable. Instead, `SET-VAR-`x nodes are supplied for each variable in the same manner as Montana [109].

Removing the node-type constraints removes the imperative structure that is imposed, so that any node may appear anywhere within a program tree that its data-type allows. This version of the algorithm is directly equivalent to the basic form of STGP. This change increases the search-space as it allows any node of the correct data-type to appear where previously only a node of that data-type *and* a specific node-type was allowed. Therefore, the search-space of SFGP can be shown to be a subset of the search-space of STGP. The results from this set of runs

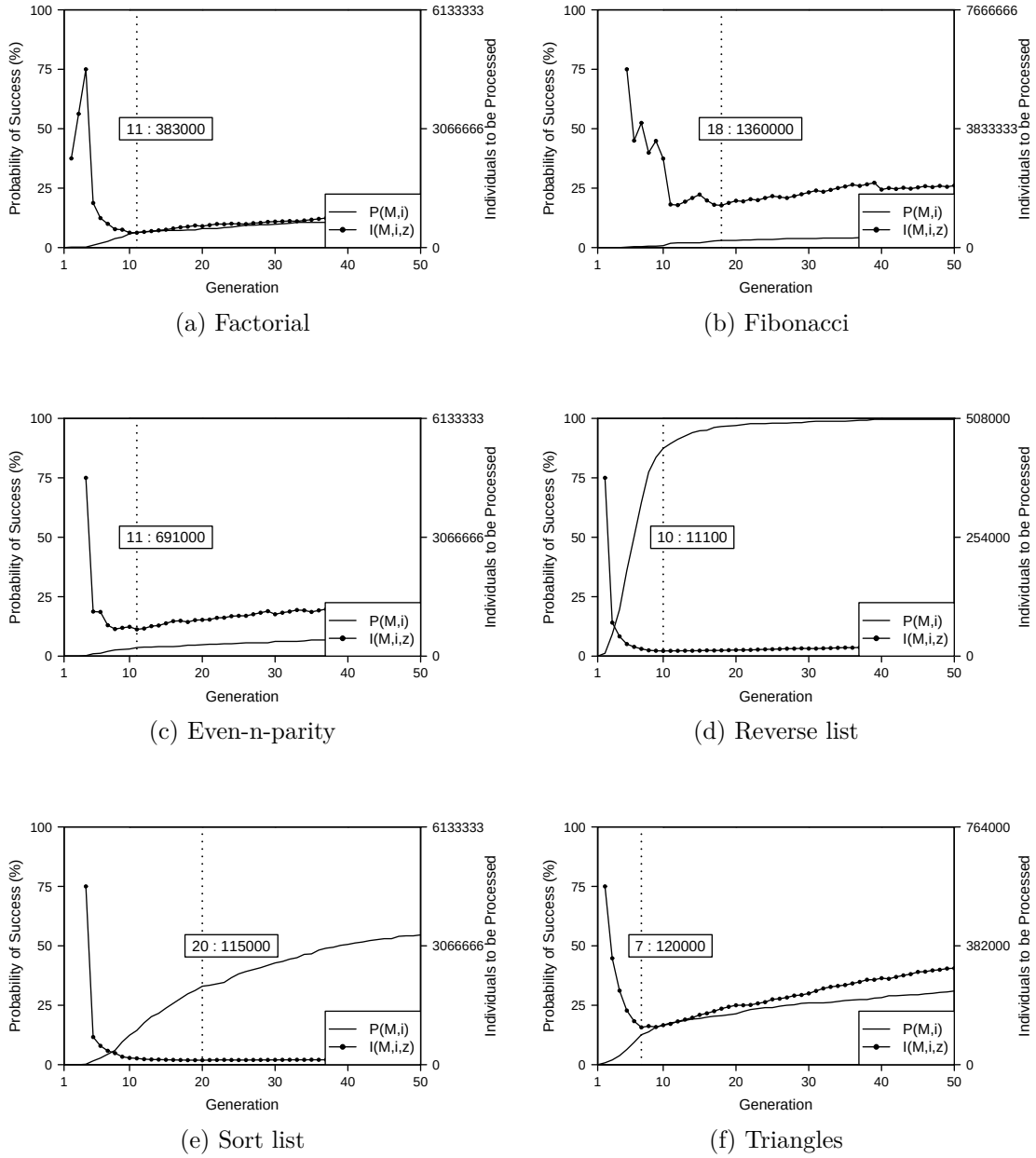
is presented in Table 4.15. For comparison, the results from section 4.5.2, where structural constraints are used, are reproduced here in the rows with 'SFGP' listed in the experiment column. Performance curves are displayed in Figure 4.7.

Table 4.15: Summary of the results comparing SFGP to a system without node-type constraints, shown in the rows labelled *STGP*. *Train%* is the probability of success on the training cases (as used for fitness) and *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort to find a solution with 99% confidence and *95% CI* is its confidence interval. *Evals* is the number of program evaluations required to find a solution with 99% confidence. The approach used to calculate each of these values is described in detail in section 3.3.

	Exp.	*Train%*	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	STGP	11.2	11.2	383,000	272,000 - 542,000	7,660,000
	SFGP	**72.8**	**72.0**	**25,700**	22,700 - 29,200	514,000
Fibonacci	STGP	6.6	6.0	1,360,000	827,000 - 2,249,000	27,200,000
	SFGP	**59.0**	**56.2**	**40,500**	34,800 - 47,400	810,00
Parity	STGP	20.2	9.0	691,000	438,000 - 1,094,000	5,528,000
	SFGP	**90.2**	**80.0**	**39,500**	26,000 - 33,700	236,000
Reverse	STGP	**99.6**	**99.6**	**11,100**	9,990 - 12,500	55,500
	SFGP	78.6	77.0	29,200	25,900 - 33,000	146,000
Sort	STGP	**69.0**	55.0	115,000	98,900 - 134,000	575,000
	SFGP	75.0	**71.2**	**65,200**	55,900 - 76,300	326,000
Triangles	STGP	31.2	31.2	120,000	93,700 - 153,000	720,000
	SFGP	**69.6**	**69.6**	**15,900**	13,900 - 18,200	72,000

It would be unfair to make performance comparisons between SFGP and STGP based on these results, as STGP may well be able to make better use of alternative syntax and perform better with different control parameters. However, it serves to illustrate the potential impact of the reduced search space and some of the advantages of SFGP. Computational effort is significantly lower where node-type constraints are used on five of the test problems, but is significantly higher on one of them, the reverse-list problem. This matches our expectations that the reduced search space may impact performance either positive or negatively, depending on the problem and the available syntax.

One of the key reasons for the better performance where the node-type constraints were used is that solutions were effectively forced to contain a certain level of complexity. The root structure (from Figure 4.4) that is enforced, ensures that all programs contain at least three statements which are highly likely to contain



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse list

(e) Sort list (f) Triangles

Figure 4.7: Performance curves for each of the test problems, where structural constraints are omitted. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

some degree of program logic. In contrast, the populations where only data-type constraints were used, contained many solutions that were comprised of very little complexity and they were unable to make proper use of the constructs available. In this situation, resources are wasted on non-sensical programs that are unlikely to contribute anything towards a correct solution. The reverse-list and sort-list problems suffer much less from this issue, because on these problems the data-type constraints are nearly sufficient to enforce the same root structure. The syntax for each of these problems contains only two nodes which are able to return a value of the correct data-type required for the problem. One of those is the input array variable, which will be unchanged and so any programs using this as the root node will receive a very poor fitness score. The other is `SubRoutine`, which would result in the same root structure as enforced by the node-type consraints. In our experiments, the proportion of individuals in the population using the input array as the root node was approximately half after initialisation, but on most runs it had dropped to less than 2% of the population after just 3 generations. From this point, the runs can progress in a very similar manner to those where the node-type constraints are used, so the performance does not differ as substantially as on the other problems.

In most cases the level of generalisation has been maintained, but notably the even-n-parity problem shows a substantial drop, with less than 50% of the runs that solved the training cases also solving the test cases. This may be a result of the removal of the enforced nodes in the first couple of levels of the program trees (`SubRoutines` and `CodeBlock`). This allows a greater depth of nested boolean expressions to be used, which are then capable of expressing a solution to the even-3-parity problem which is used for training, without any use of iteration. It seems likely that this would be less of a problem if a greater range of training inputs were used. This is a suggestion as to why use of just the 3-bit inputs are sufficient with SFGP; because there is barely sufficient depth available for a non-iterative solution. Had a greater maximum depth been used for the SFGP experiments, our results may have suffered.

4.7 Alternative Parameter Settings

Some of the constructs that have been used in this section have relied upon non-standard control parameters which were arbitrarily set. In this section, the impact of these setting values will be briefly examined by comparing the results already presented to alternative parameter settings. Other standard control parameters, such as the maximum depth, have already been widely studied in GP [33] so will not be considered any further here.

4.7.1 Code Block Size

`CodeBlock` and `ReturnBlock` nodes require a pre-specified number of `Statement` children. In all other experiments in this thesis a code-block size of 3 has been used, so that three statements are required for each block. Changing this value directly modifies the maximum allowable size of the program trees. Reducing the code-block size, reduces the maximum number of statements that may appear in the whole program, because the depth of the program trees is constrained. It may be that for some problems, a reduced program size will leave insufficient room for the sequence of statements required to solve the problem, while on others, there may be some benefit in reducing the number of possible statements. It may also be the case that a setting of 3 is already too restrictive for some of the problems.

To explore the impact of the code-block size, 500 runs were performed on each of the six test problems using alternative code-block sizes of 2 and 4. All other control parameters were set as used in section 4.5.1. The results of these runs are presented in Table 4.16, along with the results of using a code-block size of 3, which are reproduced here for comparison. These results suggest that a smaller code-block size of 2 is preferable for performance on five of the six problems studied, while the remaining problem performed better with a code-block size of 3. On only the Fibonacci problem are success rates significantly lower with a code-block size of 2 than the larger settings tested. It would be interesting to extend this study to consider a code-block size of 1, particularly as this would

result in programs without sequentially ordered statements, which is the essential property of imperative programs.

As with all of the GP control parameters, setting the perfect code-block size is not straightforward. However, the results do imply a degree of robustness, as solutions are found to all problems despite non-perfect code-block settings. An alternative approach, which has not been considered here, is to use a mix of code-block sizes to cover a sensible range of values. It would be interesting to explore the impact of this in future work and in practice it may help to relieve the burden of having yet another control parameter to set.

Table 4.16: Summary of the results comparing code-block sizes of 2, 3 and 4. The *Size* column lists the number of statements to a code-block. *Train%* is the percentage of success on the training cases (as used for fitness) and *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort to find a solution with 99% confidence and *95% CI* is its confidence interval. *Evals* is the number of program evaluations required to find a solution with 99% confidence. The approach used to calculate each of these values is described in detail in section 3.3.

	Size	*Train%*	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	2	**78.2**	**78.0**	**21,500**	19,200 - 24,200	430,000
	3	72.8	72.0	25,700	22,700 - 29,200	514,000
	4	54.4	53.4	47,100	40,100 - 55,500	942,000
Fibonacci	2	43.4	43.0	97,600	83,300 - 115,000	1,952,000
	3	**59.0**	**56.2**	**40,500**	34,800 - 47,400	810,000
	4	56.0	52.8	44,700	38,500 - 52,100	894,000
Parity	2	**97.0**	**91.0**	**15,700**	14,100 - 17,600	125,600
	3	90.2	80.0	29,500	26,000 - 33,700	236,000
	4	74.2	59.2	68,400	56,600 - 83,000	547,200
Reverse	2	**94.2**	**93.2**	**13,600**	12,200 - 15,200	68,000
	3	78.6	77.0	29,200	25,900 - 33,000	146,000
	4	50.4	48.6	70,000	60,400 - 81,400	350,000
Sort	2	**84.0**	**82.6**	**58,500**	52,300 - 65,900	292,500
	3	75.0	71.2	65,200	55,900 - 76,300	326,000
	4	68.2	62.4	92,900	80,300 - 108,000	464,500
Triangles	2	**77.4**	**77.4**	**12,100**	10,600 - 13,800	72,600
	3	69.6	69.6	15,900	13,900 - 18,200	95,400
	4	41.8	41.8	42,000	35,100 - 50,300	252,000

4.7.2 Maximum Loop Iterations

In order to ensure all loops are bounded to terminate within a reasonable evaluation time, all indexed loop constructs were restricted to performing a maximum of 100 iterations. This value was arbitrarily chosen and is used consistently on all problems. However, it may be the case that this value is overly restrictive for some problems, or indeed some solutions may actually rely upon this bound to function correctly. Alternatively, this upper bound could be too generous and may be unnecessarily allowing unfit programs to waste valuable evaluation time without any benefit for good program solutions. To test the impact of the maximum loop iterations on the performance of the system, 500 runs were performed on each of five test problems using alternative settings of 50 and 150. Only five of the six test problems are used, because a `ForEach` loop was used for the even-n-parity problem, which is not constrained by this setting and so the problem is omitted. The values of 50 and 150 were selected as being simple multiples of the original setting and they are also both larger than the largest training and test case index on all problems.

Performance results from these runs are presented in Table 4.17. The results show little variation between the success rates of the three maximum iteration settings on all the problems tested and the overlapping confidence intervals suggest that none of the computational effort results are statistically significant. This suggests that the solutions do not have a strong dependency upon the maximum iteration parameter being set specifically at 100, as used in this thesis. However, there are some small variations worth mentioning. In particular, on the sort-list problem the probability of success on the training cases was highest where the iterations parameter was set at 150. Yet the proportion of runs finding a generalisable solution was lowest with that same setting. Although these differences are not statistically significant, they do highlight that a potential implication of setting this parameter too high or too low is that the level of generalisation may be reduced.

Table 4.17: Summary of the results comparing different maximum iteration settings, as listed in the *Its.* column. *Train%* is the percentage of success on the training cases (as used for fitness) and *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort to find a solution with 99% confidence and *95% CI* is its confidence interval. *Evals* is the number of program evaluations required to find a solution with 99% confidence. The approach used to calculate each of these values is described in detail in section 3.3.

	Its.	*Train%*	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	50	72.4	**72.2**	26,300	23,200 - 29,900	526,000
	100	**72.8**	72.0	**25,700**	22,700 - 29,200	514,000
	150	70.4	70.0	27,800	24,600 - 31,500	556,000
Fibonacci	50	59.0	56.6	44,000	37,900 - 51,300	880,000
	100	59.0	56.2	40,500	34,800 - 47,400	810,000
	150	**62.4**	**60.6**	**37,700**	32,500 - 43,900	754,000
Reverse	50	78.2	76.2	**27,700**	24,700 - 31,400	138,500
	100	**78.6**	**77.0**	29,200	25,900 - 33,000	146,000
	150	77.6	76.4	28,700	25,600 - 32,300	143,500
Sort	50	73.8	70.4	67,200	57,200 - 79,300	336,000
	100	75.0	**71.2**	**65,200**	55,900 - 76,300	326,000
	150	**76.6**	69.4	70,400	60,200 - 82,900	352,000
Triangles	50	**71.6**	**71.6**	14,400	12,700 - 16,500	86,400
	100	69.6	69.6	15,900	13,900 - 18,200	95,400
	150	69.8	69.8	**14,300**	12,600 - 16,400	85,800

Given the comparable performance results, it may be preferable to use a lower maximum iterations setting to potentially reduce evaluation times. Table 4.18 shows the mean fitness evaluation time with each of the three maximum iteration settings over these runs. As expected, these results mostly show a positive correlation between the maximum iteration setting and evaluation time, so this justifies an approach of choosing a smaller number of maximum iterations where possible. On these problems, which require repetition to solve, there must be a minimum maximum iterations setting at which the problem is still solvable. It would be interesting to identify this point for each of the problems and test the performance impact of values around this point. This remains for future work.

Table 4.18: Comparison of the mean time required to evaluate an individual with maximum iterations settings of 50, 100 and 150

	Mean Evaluation Time (ns)		
	50 Iterations	*100 Iterations*	*150 Iterations*
Factorial	79753 ± 32	109592 ± 52	139911 ± 79
Fibonacci	51577 ± 18	63909 ± 29	72372 ± 38
Reverse	81375 ± 41	91035 ± 42	93888 ± 46
Sort	542304 ± 228	724187 ± 333	675877 ± 278
Triangles	16087 ± 23	15823 ± 13	16628 ± 30

4.8 Summary

This chapter has introduced a novel mechanism for adding structural constraints to a tree-based GP representation and has established that these constraints are sufficient to impose a naturally high-level imperative structure upon the evolved programs. It was demonstrated how these programs could be converted to the source code of a number of different high-level imperative programming languages by using *code templates*. The solutions which were produced to each of the problems were found with high success rates and generalised well to wider test inputs, while using a relatively low amount of computational resources in comparison to those studies reviewed in chapter 3. The reduced size of the search space caused by the additional structural constraints was investigated and shown to be beneficial on five out of the six test problems, but there is a warning of the potential to damage success rates in the reduced performance on the reverse-list problem. Finally, the impact of two new control parameters, code-block size and maximum iterations, was explored. The conclusion was that the ideal code-block size is problem dependent but with a preference for a smaller value. The maximum iterations setting was discovered to have little impact on results as long as it is higher than some unknown threshold to solve a given problem, but also that a lower value does in most cases reduce the average evaluation time, as may be expected.

Chapter 5

High-Level Imperative Extensions

5.1 Introduction

It was demonstrated in chapter 4 that SFGP provides a general mechanism for constraining the structure of program trees and that those constraints are sufficient to evolve programs with an imperative structure using standard high-level imperative programming constructs such as loops and arrays. However, more can be achieved with some additional modifications to the algorithm to specifically target the evolution of these imperative programs. This chapter will propose two such extensions. The first adds the feature of a dynamic syntax, which may be updated by a program to allow new limited scope variables to be declared. The second is a simple method for improving the evaluation performance of programs with the imperative structure, by considering multiple variables as candidates for supplying the return value of a subroutine. The modifications necessary for each of these extensions will be described and some experimental results will be presented that demonstrate their impact, along with a discussion of the potential benefits.

5.2 Limited Scope Variable Declarations

Variables are a fundamental component of computer programs. However, rarely has a GP system been given the power to construct new variables. Without variable declarations, all variables must be supplied as inputs to the system, including any auxiliary variables required for the computation process that are not part of the specified inputs or outputs. With complex programs, this can require a considerable degree of insight into the solution space. By supporting the evolution of variable declarations, the aim is to lighten this burden without excessively degrading performance.

It has already been described how the Strongly Formed Genetic Programming (SFGP) variant of GP can be used to enforce a high-level imperative structure upon evolved program trees. With a series of simple modifications, SFGP can include support for allowing operators to declare new limited scope variables. Limited scope variables are commonly found in modern high-level imperative programming languages, but are particularly challenging to incorporate into an evolutionary system. Each variable must not be used prior to being declared, nor beyond the extent of its scope. Neglecting the limited scope aspect of variable declarations may simplify the problem. However, this is inconsistent with the way local variables are used by human programmers and produces programs reliant on global variables [175].

One of the frequently mentioned issues with genetic programming is the difficulty in evolving iteration or recursion [4, 29]. If a mechanism for supporting variable declarations is used, iterative constructs that more closely resemble those used in high-level imperative programming languages become simple to implement. These constructs can supply indices or elements through variables that they declare. Such constructs are commonly used in human written code and it seems likely that they could help to expand the range and scale of problems to which GP can be applied.

5.2.1 Related Work

Variables are widely used in applications of GP for a variety of purposes. The inputs for programs in a GP population are typically supplied using variables, with the set of inputs defined by the GP practitioner and would normally be the same for all programs in a population. There is often no facility for the value of these variables to be altered. However, Koza [87, chapter 18.2] did propose a mechanism for assigning the value of a global variable using a `SET-SV` operator. He suggested that the use of a settable variable like this was beneficial for the evolution of building blocks, since the variable provided a way of labelling a useful computation so that it could be used elsewhere in the program. Koza's approach not only treated all variables as global, but also required them to be defined in advance; no variable declarations here.

Linear GP variants [17, 114] commonly make use of defined memory registers which can be both assigned to and have values retrieved from them. The number of available registers is defined in advance to include registers for each input, plus additional registers for facilitating calculations. Brameier and Banzhaf [17, chapter 2.1] make the point that it is important for a sufficient number of registers to be provided to avoid valuable information being overwritten. However, too many registers may spread the computation too widely and make it difficult to build a solution. As Oltean and Grosan [118] put it, "The number of supplementary registers depends on the complexity of the expression being discovered. An inappropriate choice can have disastrous effects on the program being evolved".

Stack-based GP systems [129,149] provide an alternative approach to memory, where the result of expressions are pushed onto a stack and popped off as inputs are required. As such, they are able to support an expandable memory allocation (within some reasonable bounds). PushGP [149] evolves programs in the specially designed Push programming language. Push provides a `NAME` data type, which maintains its own stack of variable labels, upon which values may be pushed by a program to define a new variable.

There has been some limited use of variable declarations with tree-based GP approaches. The authors of OOGP [1] imply their existence by stating that "new local variables may occur within block statements". But they fail to give any additional details. A far more thorough explanation is given by Kirshenbaum [84], in his work with *statically scoped local variables*. He describes a method for supporting Lisp's `LET` expression, which is able to define variable bindings with limited scope. This is achieved by adding each `LET` expression's bindings to the set of available operators for each of the expression's subtrees as they are generated. We take a similar approach to Kirshenbaum, but must deal with a slightly more complicated scenario, to cater for an imperative structure based on statements and blocks. The scope of a variable in an imperative program should not just descend into the children of the operator that declares it, but should also be accessible to sibling operators (for example, statements following a declaration, within the same block).

5.2.2 Syntax Updates

There are two forms of variable declaration that we wish to support, each requiring different scope, consistent with modern imperative languages such as C/C++, Java and Python:

- Standard declarations create a new variable and assign it a value according to some expression. The variable's scope extends from the statement following the declaration, up until execution leaves the block the declaration is contained within. The variable is not in-scope for the declaration's own subtrees, but is available at any level of nesting for the following statements, up until it is removed from scope.

- Some more advanced statement types, such as loops, may declare variables for use only within the body of a child block. This is the case for loop constructs that declare a new variable to be updated on each iteration with

the index or element. These variables are available at any level of nesting within the loop statement that declared them, but not beyond.

To support these types of declaration, we introduce *syntax updates*. A syntax update is an opportunity for a node to modify the terminal and non-terminal sets. These syntax updates can be applied as the initialisation procedure progresses in order to change the available syntax for the construction of a node's subtrees or any following nodes. A syntax update may involve the addition or removal of nodes from the syntax. For the purpose of supporting variable declarations, the emphasis here is on modifying the available variables, but the same infrastructure could be used for other purposes, such as the declaration of extra sub-routines. Each node is able to define $arity + 1$ syntax updates, which when the tree is traversed depth-first, are applied before and after each of its child nodes are processed. Figure 5.1 illustrates this, with the dotted branches indicating the points of each syntax update, which are labelled with the order they would be applied. In this example, the syntax updates 4, 6 and 8 would all be defined by the $\texttt{B}_1$ node. Both forms of limited scope variable declarations that have been highlighted can be achieved using this system.

- The $\texttt{B}_1$ node can declare a new variable just to be available for its child subtrees by adding the variable in syntax update 4 and then removing it in syntax update 8. It can also restrict the variable to just one of its two child subtrees by using the syntax update directly before and after that child.

- The $\texttt{B}_1$ node can declare a new variable to be available only for following nodes, by adding the new variable in syntax update 8.

With some small modifications to the initialisation, crossover and mutation operators, programs can be evolved that make use of this dynamic syntax to declare new variables. The necessary modifications are described in the following sections.

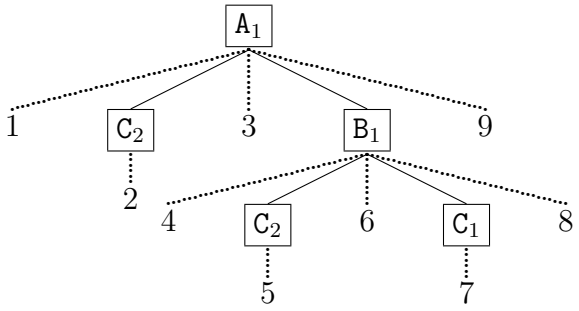


Figure 5.1: Example illustrating the position of syntax updates, which are shown as dotted branches labelled with the order they would be applied. At each syntax update, the available syntax can be modified for the following nodes when traversed depth-first.

5.2.3 Modified Initialisation

Section 4.2.1 describes the grow initialisation procedure used by SFGP to construct random program trees, where each node is selected at random from those with a compatible data-type and node-type required by its parent (or the problem itself for the root node). The only modification necessary is to apply any syntax updates that are defined for a node as the tree is built. This will ensure that at each point of the initialisation procedure, the available syntax contains only those variables that are in-scope at that point. Algorithm 5.1 shows the updated initialisation algorithm, including the syntax update step on lines 6 and 16. As an example, consider the program tree in Figure 5.1. Once the root node $\texttt{A}_1$ is selected, the initialisation will proceed as follows:

1. The $\texttt{A}_1$ node's 1$^{\text{st}}$ syntax updates, labelled 1 in the figure, are applied.
2. A recursive call to the initialisation procedure is made to construct a subtree as the first child of the $\texttt{A}_1$ node.
3. The $\texttt{A}_1$ node's 2$^{\text{nd}}$ syntax updates, labelled 3 in the figure, are applied.
4. A recursive call to the initialisation procedure is made to construct a subtree as the second child of the $\texttt{A}_1$ node.
5. The $\texttt{A}_1$ node's 3$^{\text{rd}}$ syntax updates, labelled 9 in the figure, are applied.

Algorithm 5.1 High-level pseudocode of the SFGP initialisation procedure with modifications to support variable declarations. dt, nt and $depth$ are the required data-type, node-type and maximum depth. The $filterNodes(S, dt, nt, depth)$ function is defined to return a set comprised of only those nodes in S with the given data-type and node-type, and with non-terminals removed if $depth = 0$. The function $updateSyntax(S, r, i)$ performs the task of updating the available syntax, S, as defined for the ith position of the node-type r.

```
1:  function generateTree(dt, nt, depth)
2:      V ← filterNodes(S, dt, nt, depth)
3:      while V not empty do
4:          r ← removeRandom(V)
5:          for i ← 1 to arity(r) do
6:              S ← updateSyntax(S, r, i)
7:              dti ← required data-type for ith child
8:              nti ← required node-type for ith child
9:              subtree ← generateTree(dti, nti, depth − 1)
10:             if subtree ≠ err then
11:                 attach subtree as ith child
12:             else
13:                 break and continue while
14:             end if
15:         end for
16:         S ← updateSyntax(S, r, arity(r))
17:         return r                          ▷ Valid subtree complete
18:     end while
19:     return err                            ▷ No valid subtrees exist
20: end function
```

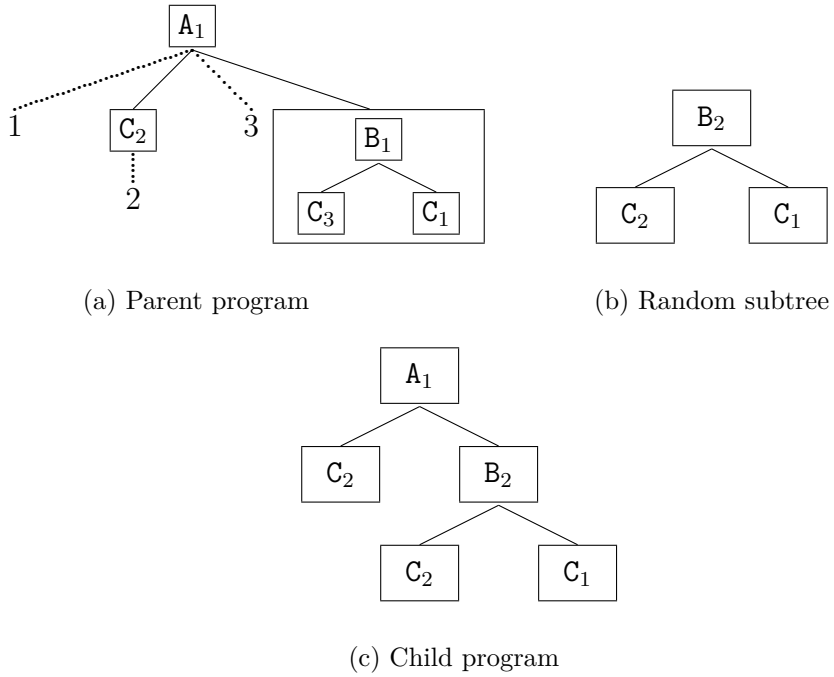
5.2.4 Modified Mutation

In the basic form of SFGP, a program tree undergoing subtree mutation has a node randomly selected and replaced with a newly generated subtree with a compatible data-type and node-type. The main addition required, in order to support a dynamic syntax, is for the newly generated subtree to be constructed from the available syntax at the mutation point after all syntax updates up to that point have been applied. The available syntax at the mutation point is easily obtained by performing a partial traversal of the program tree, up to the mutation point, applying each node's syntax updates. A subtree can then be constructed from the syntax at this point, using the initialisation procedure. In Figure 5.2, the $\texttt{B}_1$ node has been selected as the mutation point. Before a replacement subtree is generated, all syntax updates prior to this point (1, 2 and 3) are applied to the syntax. The initialisation procedure can then construct the subtree using this updated syntax.

There is one further problem that needs to be overcome. No restrictions are in place upon which subtree may be selected for replacement by the mutation operator. So, a node which performs a variable declaration could be replaced, potentially leaving *dangling* variables. A dangling variable, in this case, is a use of a variable without an associated declaration. To resolve this issue, a repair operation is performed, which is described in section 5.2.6. An alternative to repairing the dangling variables is to simply disallow any mutation operation on a node which will leave dangling variables. The problem with this approach is that program trees are liable to accumulate variable declarations which perform no fitness enhancing functionality without some further mutation that makes use of the variable.

5.2.5 Modified Crossover

Subtree crossover in SFGP operates on two program trees, by randomly selecting a node in one program and swapping the subtree rooted at that node with another



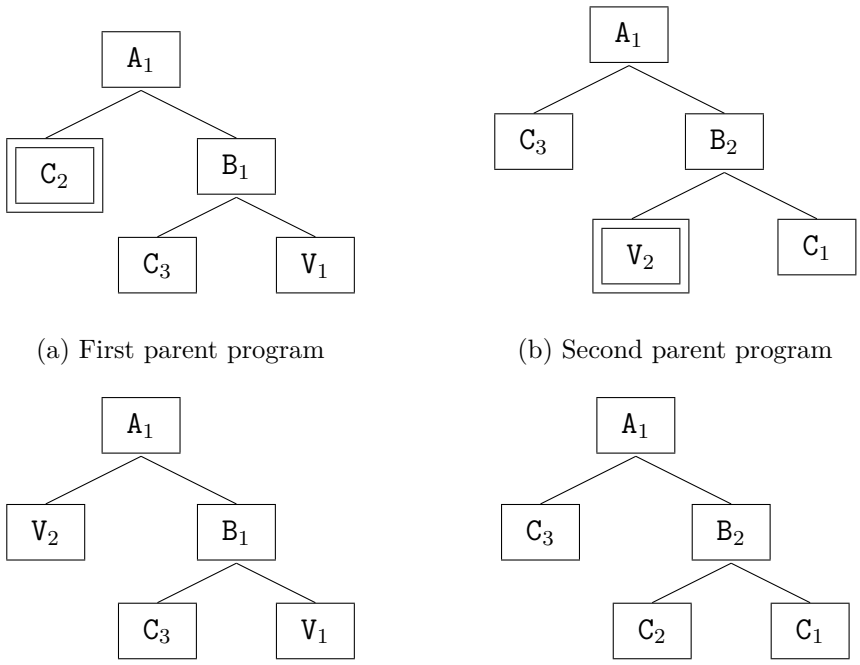
(a) Parent program

(b) Random subtree

(c) Child program

Figure 5.2: Example subtree mutation where a dynamic syntax is supported. The syntax updates prior to the mutation point are applied to construct the syntax from which the subtree is created.

from the other program tree, randomly selected from those with compatible data-type and node-type. No modifications to the basic operation of the crossover operator are necessary to support a dynamic syntax. However, there are two specific scenarios that must be handled for variable declarations to be evolved. (1) As with mutation, the subtree that is removed may contain the declaration for variables that are used elsewhere in the program tree, so these will be left orphaned as dangling variables. (2) The subtree that is swapped into the program tree may also contain dangling variables that were previously supported by declarations that were not part of the genetic material transferred. Both of these situations are resolved with the same repair operation, described in section 5.2.6. As with mutation, an alternative is possible; crossovers that would lead to a situation of dangling variables could be prevented from occurring, but it seems unlikely that the algorithm will be able to take advantage of declarations if they are prevented from being exchanged.



(a) First parent program

(b) Second parent program

(c) First child program

(d) Second child program

Figure 5.3: Example subtree crossover where a dynamic syntax is supported, with crossover points highlighted in the parent programs. $\texttt{C}_2$ declares the $\texttt{V}_1$ variable and $\texttt{C}_3$ declares the $\texttt{V}_2$ variable. The first child program is left with 2 dangling variables because the declaration for $\texttt{V}_1$ is moved to the second program and the $\texttt{V}_2$ variable is inserted from the second program without its associated declaration.

To demonstrate how these dangling variables can occur, Figure 5.3 displays an example crossover. The `V` nodes are variables of node-type `C`, with a data-type of boolean and integer for $\texttt{V}_1$ and $\texttt{V}_2$ respectively. If the $\texttt{C}_2$ node in the first parent is responsible for declaring the variable $\texttt{V}_1$, then removing it from the program will leave the variable dangling. This is an example of scenario (1) described above. Scenario (2) occurs when the $\texttt{V}_2$ variable is swapped in to the first program, where it is not supported by a declaration. This leaves the first child program with two dangling variables which must be fixed by the repair operation.

5.2.6 Repair Operation

To remove all dangling variables introduced by the crossover and mutation operators, a repair operation is applied to each program after undergoing one of these

Table 5.1: Type list for the declarative `Statement` node-types, showing the required data-type and node-type for each type of node. *d* indicates a pre-specified data-type and a `Void` data-type indicates no value is returned.

Node-type	Data-type	Child data-types	Child node-types
`ForLoopDecl`	`Void`	`Integer` `Void`	`Expression` `CodeBlock`
`ForEachLoopDecl`	`Void`	*d*[] `Void`	`Expression` `CodeBlock`
`Declaration`	`Void`	*d*	`Expression`

genetic operators. The repair operation replaces any dangling variables with an in-scope variable of a compatible data-type. To do this the program tree is traversed, with each node checked to see if it is a dangling variable. A node is defined as a dangling variable if it has a `Variable` node-type and if that variable does not exist in the updated syntax at that point. To ensure the syntax includes all in-scope variables, all syntax updates must be applied as the tree is traversed. If a dangling variable is identified, then an alternative variable is selected at random to replace it, from those in the updated syntax with the correct data-type. If no suitable alternative variables exist in the syntax then the repair operation fails and the related genetic operator that was performed must be discarded and reattempted.

5.2.7 New Syntax

The dynamic syntax allows for the support of additional node-types that declare new variables. The following new node-types are added, which use variables which are limited in scope as specified. They are all subtypes of the abstract `Statement` node-type and have `Void` data-types. Table 5.1 lists the type constraints, while the semantic operation of the constructs is specified below:

– `ForLoopDecl` - adds a new variable, *i*, of an integer data-type to the available syntax on the second syntax update and the same variable is removed from the syntax on the third syntax update. On evaluation, the first child is evaluated, with the result used as the number of iterations to perform

(capped at 100). The child code-block is evaluated the given number of times, with the value of the variable i set to the current index, starting at 1.

– `ForEachLoopDecl` - adds a new variable, e, of the same data-type as the elements of the array input, to the available syntax on the second syntax update. The same variable is removed from the syntax on the third syntax update. On evaluation, the first child is evaluated to obtain an array to be iterated over. The second child is evaluated once per element in the array, with the value of e set to the current element prior to each evaluation.

– `Declaration` - adds a new variable of the same data-type as the input from the only child to the syntax in the second syntax update. The value of the variable is set as the result of evaluating the child expression. Removal of the variable is left to the code-block the declaration is contained within.

To ensure the scope of variables declared by the `Declaration` node-type are limited to the code-block within which they are defined, the `CodeBlock` and `ReturnBlock` node-types are modified. They record the state of the syntax at the first syntax update and revert the syntax to that state at the final syntax update, after its last child `Statement` has been processed. This results in all variables declared within that block being removed from the syntax.

5.2.8 Experiments

A series of experimental runs were performed to test the impact of the introduction of variable declarations. Three different scenarios were compared, with the labels SFGP, LOOP and DECL used to distinguish them. Each one used an identical set of control parameters, as used in section 4.5.1, with the exception of the terminal and non-terminal sets. These are listed in Tables 5.2–5.7.

– **SFGP** - For five of the six test problems, the results in this experiment are simply reproduced from the SFGP experiments in chapter 4. The exception

is the Fibonacci problem, which made use of a `Loop` node in chapter 4. As the `Loop` node-type does not use a variable, there is no equivalent declarative form. To enable a fair comparison, a new set of runs was performed for this problem with the `Loop` node replaced with a `ForLoop` node.

– **LOOP** - Used the same terminal and non-terminal sets as the SFGP experiment, but each loop node was replaced with the equivalent declarative form. For instance, `ForLoop` was replaced with `ForLoopDecl`, which operates according to the same semantics, excepting that it declares its own variable for storing the iteration index.

– **DECL** - Used the same terminal and non-terminal sets as the LOOP experiment, but with the addition of a `Declaration` operator on each problem.

Any auxiliary variables required for the SFGP experiment setup were also supplied for the LOOP and DECL experiments, even where not required. This was in order to keep the setups constant, other than the constructs under examination.

5.2.9 Results & Discussion

The results summary in Table 5.8 lists all the success rates and required computational effort for each problem in the three experiments. The impact of the

Table 5.2: Listing of the control parameter settings used for SFGP with variable declarations on the factorial problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `Assignment`, `Add`, `Subtract`, `Multiply`, i, $loopVar$, 1
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Assignment`, `Add`, `Subtract`, `Multiply`, i, $loopVar$, 1
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Declaration`, `Assignment`, `Add`, `Subtract`, `Multiply`, i, $loopVar$, 1

Table 5.3: Listing of the control parameter settings used for SFGP with variable declarations on the Fibonacci problem

Root data-type:	`Integer`
Root node-type:	`SubRoutine`
Max. depth:	6
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `Assignment`, `Add`, `Subtract`, $loopVar$, i, $i0$, $i1$
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Assignment`, `Add`, `Subtract`, $loopVar$, i, $i0$, $i1$
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Declaration`, `Assignment`, `Add`, `Subtract`, $loopVar$, i, $i0$, $i1$

Table 5.4: Listing of the control parameter settings used for SFGP with variable declarations on the even-n-parity problem

Root data-type:	`Boolean`
Root node-type:	`SubRoutine`
Max. depth:	8
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForEachLoop`, `IfStatement`, `Assignment`, `And`, `Or`, `Not`, arr, $loopVar$, $resultVar$, `true`, `false`
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForEachLoopDecl`, `IfStatement`, `Assignment`, `And`, `Or`, `Not`, arr, $loopVar$, $resultVar$, `true`, `false`
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForEachLoopDecl`, `IfStatement`, `Declaration`, `Assignment`, `And`, `Or`, `Not`, arr, $loopVar$, $resultVar$, `true`, `false`

Table 5.5: Listing of the control parameter settings used for SFGP with variable declarations on the reverse list problem

Root data-type:	`Character[]`
Root node-type:	`SubRoutine`
Max. depth:	8
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `ArrayLength`, `Subtract`, `Divide`, `SwapElements`, arr, $loopVar1$, $loopVar2$, `1`, `2`
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `ArrayLength`, `Subtract`, `Divide`, `SwapElements`, arr, $loopVar1$, $loopVar2$, `1`, `2`
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Declaration`, `ArrayLength`, `Subtract`, `Divide`, `SwapElements`, arr, $loopVar1$, $loopVar2$, `1`, `2`

Table 5.6: Listing of the control parameter settings used for SFGP with variable declarations on the sort list problem

Root data-type:	`Character[]`
Root node-type:	`SubRoutine`
Max. depth:	10
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `IfStatement`, `ArrayLength`, `ArrayElement`, `GreaterThan`, `SwapElements`, arr, $loopVar1$, $loopVar2$
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `IfStatement`, `ArrayLength`, `ArrayElement`, `GreaterThan`, `SwapElements`, arr, $loopVar1$, $loopVar2$
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Declaration`, `IfStatement`, `ArrayLength`, `ArrayElement`, `GreaterThan`, `SwapElements`, arr, $loopVar1$, $loopVar2$

Table 5.7: Listing of the control parameter settings used for SFGP with variable declarations on the triangles problem

Root data-type:	`String`
Root node-type:	`SubRoutine`
Max. depth:	8
SFGP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoop`, `IfStatement`, `Assignment`, `Concat`, n, $resultVar$, $loopVar1$, $loopVar2$, `'*'`, `'\n'`
LOOP syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `IfStatement`, `Assignment`, `Concat`, n, $resultVar$, $loopVar1$, $loopVar2$, `'*'`, `'\n'`
DECL syntax:	`SubRoutine`, `ReturnBlock`, `CodeBlock`, `ForLoopDecl`, `Declaration`, `IfStatement`, `Assignment`, `Concat`, n, $resultVar$, $loopVar1$, $loopVar2$, `'*'`, `'\n'`

declarative constructs varies across the problems. On four out of the six problems, the LOOP variant produced the best result, with a lower computational effort than the SFGP version. In two of those cases the results were statistically significant[1]. However, on the remaining two problems, the lowest computational effort was required by the SFGP setup, with one of those cases being statistically significant[1]. This problem dependence is, at least in part, explained by the higher number of variables that are available to programs with the LOOP and DECL setups. Having extra variables available is beneficial on some problems, to hold partial calculations, but where they are not useful they only serve to increase the search space. Because of this, if auxiliary variables had not been supplied for the LOOP and DECL setups, then it is likely that the performance would have been better on some problems, but worse on others. There are also problem specific scenarios that have an impact. For example, on some problems there is a benefit to using the index variable beyond the loop it is used by. This seems to be the case on the factorial problem. It is difficult to know, in general, on which problems variable declarations will be helpful and on which they will be harmful. This point is well illustrated by the vast difference between the results for the similar problems of factorial and Fibonacci.

The results for the DECL setup, with `Declaration` nodes, show a significantly worse computational effort on half the problems, but is comparable on the remaining half. The main problem with `Declaration` nodes, which only perform the single task of adding new variables, is that they do not themselves contribute to fitness. After a mutation which introduces a declaration, the variable that is added by that declaration will remain unused. For the variable to be utilised by the program towards solving the problem, a further mutation is necessary that introduces a reference to the variable. Until that point, the declaration cannot contribute to the fitness and so is effectively 'junk' code. This problem is most likely to be an issue when tight size bounds are used, because any `Declarations`

[1]Statistical significance here is determined by non-overlapping confidence intervals for the computational effort.

that occur do so at the expense of other potentially useful statements. This seems to be the case with our experimental results, where the worst impact by declarations was on problems where tighter size constraints were used. For example, it is shown in chapter 6, that the reverse-list and sort-list problems can both be solved without using the maximum available depth that is allocated in this study and these problems are not as substantially impacted by declarations. This is unlikely to be the whole story, but it suggests that using a less restrictive size constraint could help declarations to be used productively.

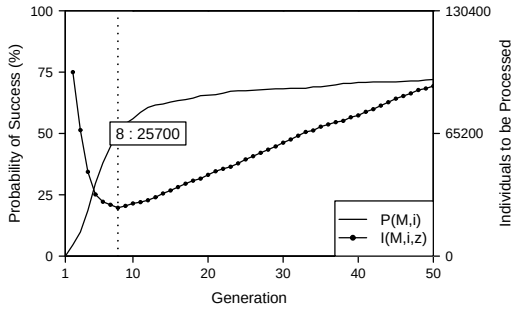
Even where performance is degraded by using variable declarations, generalisable solutions are still discovered with some reliability. This is important since the declaration and loop constructs do bring other benefits. The burden of knowing which auxiliary variables to supply in addition to the inputs is removed. It is always beneficial to provide a perfect set of components, but the degree of insight this requires in to the solution space is sometimes impractical. It could also be argued that variables are used in a manner which is more consistent with how high-level programming languages are used by human programmers, with loops that provide their own variables, limited in scope to just the body of that loop. This could be significant in some scenarios, such as in a software development application where the resultant program is to be used as just a fragment of a much larger human written computer program.

One potential concern with the modifications made to the genetic operators to support the dynamic syntax is that the repair operation could be damaging. Table 5.9 lists the proportion of genetic operations that required a repair operation to be performed to resolve dangling variables. It is interesting to note that only a very small proportion of both crossovers and mutations require the repair operation to be applied, so any impact upon the performance is likely to be minimal. The table shows that the number of mutations that introduce dangling variables is zero on all problems of the LOOP experiment. This is because any mutations that remove a declarative loop will also remove all references to their variables, since they are contained within the body of the loop that is being removed. The factorial and

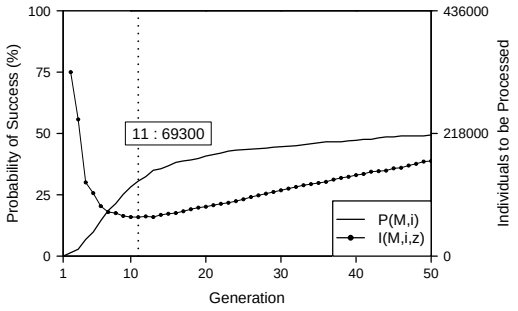
Fibonacci problems require a much lower proportion of programs to be repaired, possibly because a lower maximum depth has been used on this problem, so the smaller programs are likely to contain fewer declared variables.

Table 5.8: Summary of the results comparing SFGP with and without variable declarations, where the *Exp.* column is the experimental setup used. *Train%* is the percentage of success on the training cases (as used for fitness) and *Test%* is the percentage of runs that found a solution that generalised to the test set. *Effort* is the required computational effort to find a solution with 99% confidence and *95% CI* is its confidence interval. *Evals* is the number of program evaluations required to find a solution with 99% confidence. The approach used to calculate each of these values is described in detail in section 3.3.

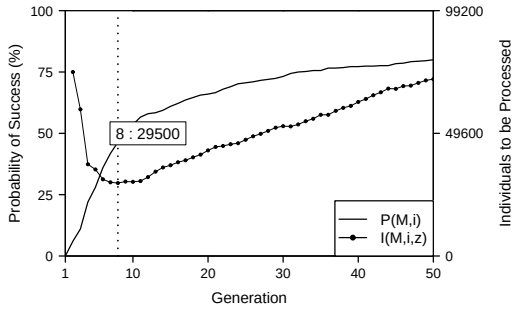
	Exp.	*Train%*	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	SFGP	**72.8**	**72.0**	**25,700**	22,700 - 29,200	514,000
	LOOP	21.2	21.0	134,000	102,000 - 177,000	2,680,000
	DECL	7.8	7.8	519,000	360,000 - 752,000	10,380,000
Fibonacci	SFGP	51.6	49.6	69,300	59,300 - 81,500	1,386,000
	LOOP	**62.2**	**60.2**	**56,600**	49,200 - 65,500	1,132,000
	DECL	41.0	40.2	167,000	141,000 - 198,000	3,340,000
Parity	SFGP	90.2	80.0	29,500	26,000 - 33,700	236,000
	LOOP	**99.4**	**95.6**	**11,200**	10,000 - 12,500	89,600
	DECL	98.2	93.4	16,600	14,900 - 18,600	132,800
Reverse	SFGP	78.6	77.0	29,200	25,900 - 33,000	146,000
	LOOP	**88.4**	**87.4**	**19,500**	17,500 - 21,900	97,500
	DECL	87.2	86.6	20,200	18,100 - 22,700	101,000
Sort	SFGP	**75.0**	**71.2**	**65,200**	55,900 - 76,300	326,000
	LOOP	70.8	61.2	86,200	73,800 - 101,000	431,000
	DECL	62.6	57.4	93,500	80,900 - 109,000	467,500
Triangles	SFGP	**69.6**	**69.6**	15,900	13,900 - 18,200	95,400
	LOOP	59.6	59.6	**14,200**	12,300 - 16,400	85,200
	DECL	25.8	25.8	45,600	34,500 - 60,600	273,600



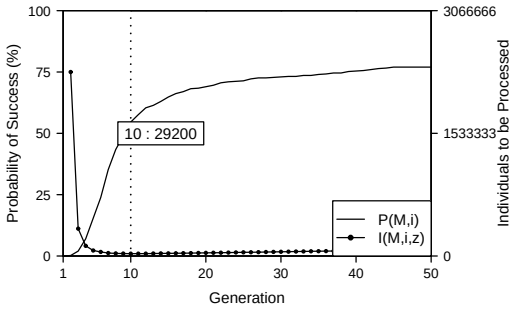
(a) Factorial



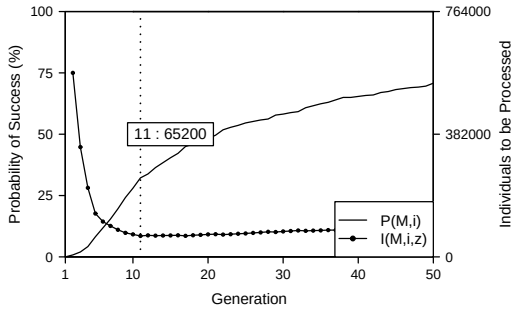
(b) Fibonacci



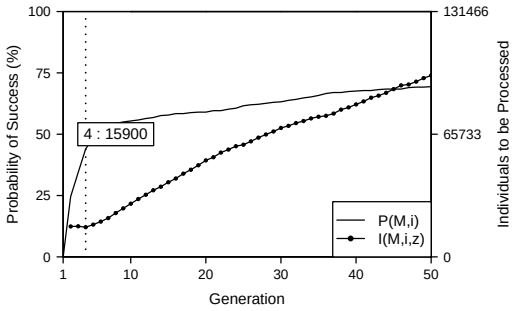
(c) Even-n-parity



(d) Reverse list

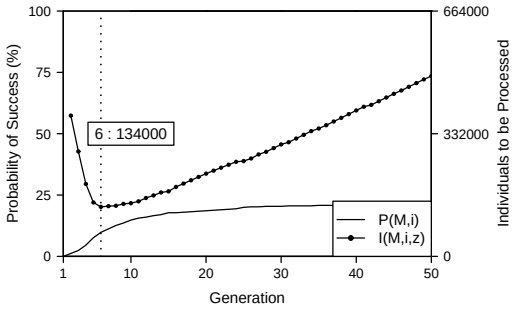


(e) Sort list

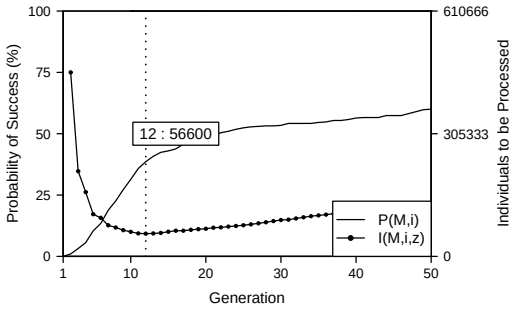


(f) Triangles

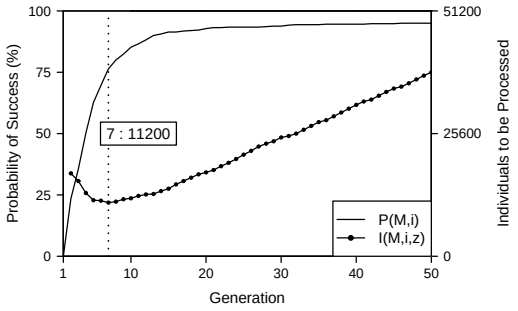
Figure 5.4: Performance curves for each of the test problems in the SFGP experiment. With the exception of the curves for the Fibonacci problem, these are reproduced from chapter 4. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.



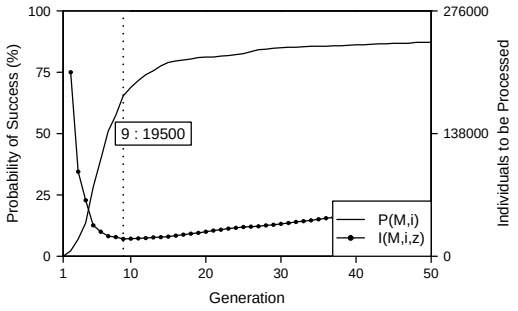
(a) Factorial



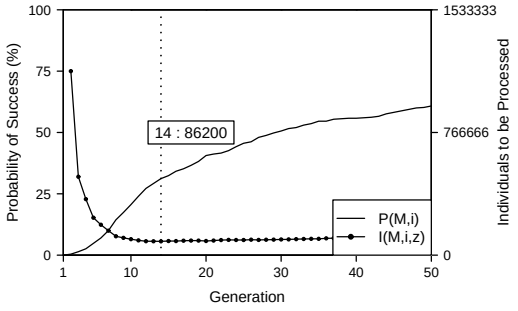
(b) Fibonacci



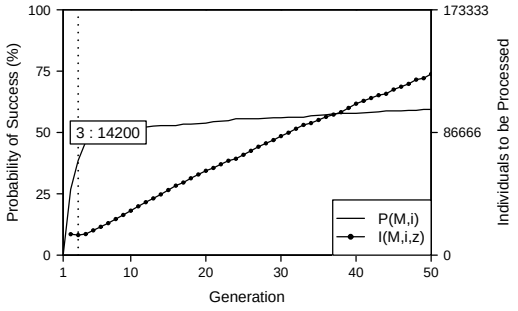
(c) Even-n-parity



(d) Reverse list

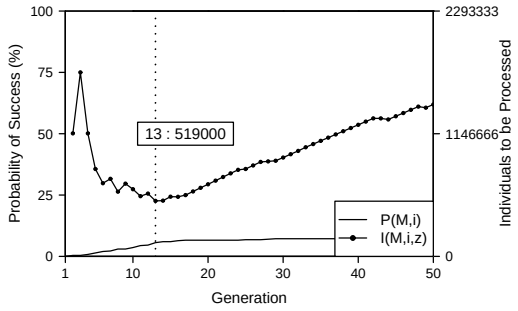


(e) Sort list

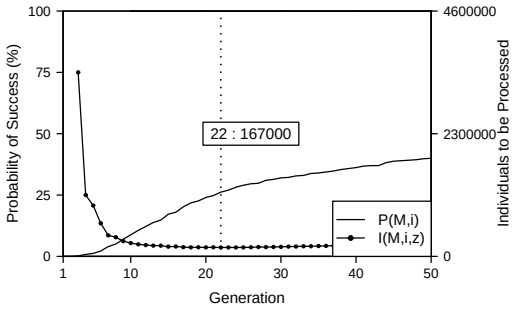


(f) Triangles

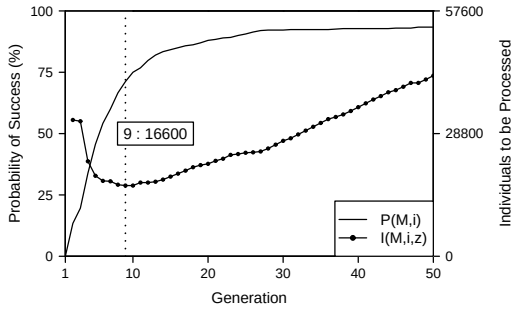
Figure 5.5: Performance curves for each of the test problems in the LOOP experiment. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.



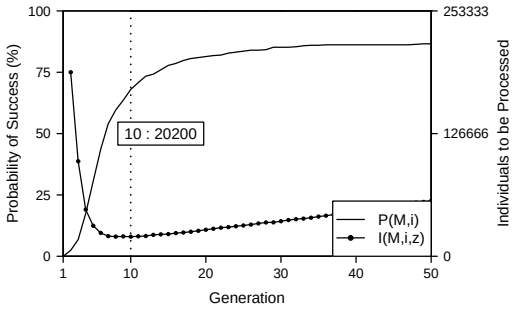
(a) Factorial



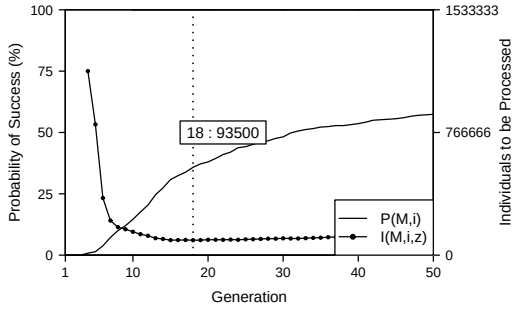
(b) Fibonacci



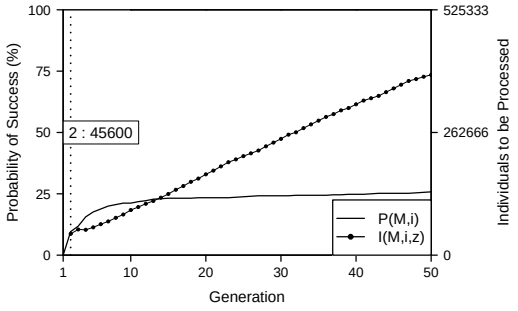
(c) Even-n-parity



(d) Reverse list



(e) Sort list



(f) Triangles

Figure 5.6: Performance curves for each of the test problems in the DECL experiment. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

Table 5.9: Summary of repair operations, showing the proportion of program trees produced by each of the genetic operators that required the repair operation to fix one or more dangling variables

	Exp.	Repair Operations	
		Crossover	*Mutation*
Factorial	LOOP	5.7%	0.0%
	DECL	4.8%	0.6%
Fibonacci	LOOP	7.3%	0.0%
	DECL	6.8%	0.6%
Parity	LOOP	16.3%	0.0%
	DECL	16.3%	1.8%
Reverse	LOOP	12.1%	0.0%
	DECL	13.3%	1.1%
Sort	LOOP	18.3%	0.0%
	DECL	20.4%	1.7%
Triangles	LOOP	0.6%	0.0%
	DECL	6.5%	1.6%

5.3 Multi-Variable Return

As discussed in section 2.1.3, fitness evaluation is generally the most time consuming phase of the GP algorithm. Any technique that can improve the efficiency of fitness evaluation can have a substantial impact. In this section, we describe a simple method that enhances the evaluation procedure when applied to programs that are constrained to have the high-level imperative structure that has been used throughout this thesis.

Given a program with the structure in Figure 4.4, it is possible to evaluate multiple versions of this program without any additional executions. The value that is returned from the program is defined by the variable found as a child to the `ReturnBlock`. The constraints ensure that the data-type of this variable is consistent with the required data-type for the problem. Even where variable declarations are not in use, there may be several variables of the correct data-type in the syntax which could potentially be positioned at this point. As with any other node in the tree, the specific variable that is used is initially randomly selected by the initialisation procedure and is then subject to evolution. However, there is no need for the evolutionary process to specify just one variable. With little additional expense, the evaluation procedure can consider all compatible

variables as potential return values. This effectively evaluates multiple versions of the same program, just differing by the variable to be returned. No further executions are required, as the value of each of the variables will have been set with just one execution for a given set of inputs. We call this technique *multi-variable return* (MVR).

In MVR, the fitness evaluation procedure executes each program tree once per training case as usual; no additional executions are necessary. However, for each individual, x fitness scores are calculated, where x is the number of variables of the correct data-type that are in-scope at the point of return. On each training case, each of the variables is used to update its own separate fitness score, according to the fitness function for the problem. Once all training cases have been handled, the fitness for the individual is assigned to be the minimum of the candidate fitnesses and the associated variable is considered to be the variable that is returned by the program. This technique has some similarity to Multi Expression Programming [117] and the related ME-CGP technique, described by Cattani and Johnson [24], for improving the efficiency of evaluation in Linear and Cartesian GP variants. Although MVR is described here in context of the SFGP system, the same idea could be applied wherever one of multiple available values is designated as the return value of a program. For example, Linear GP representations often designate one memory register as a program's result.

5.3.1 Experiments

The expectation is that the MVR method of fitness evaluation should improve performance wherever the imperative root structure is used and multiple variables of the correct data-type exist. The extent of the improvement may be correlated with the number of candidate return variables. To test this experimentally, two sets of evolutionary runs were performed with the MVR technique enabled:

1. Without variable declarations. To enable comparison, the same setup was used as for the SFGP experiments from chapter 4 (listed in Tables 4.8 to 4.13), but with the MVR technique enabled

2. With variable declarations. The DECL setup from section 5.2.8 (listed in Tables 5.2 to 5.7) is used, but with the MVR technique enabled.

The results from each of these are to be compared against the related results where the MVR method was not enabled. There is no need for the terminal and non-terminal sets to be changed to support MVR. However, the variable that is assigned to the subroutine's `ReturnBlock` by the evolutionary process is no longer be used to supply the return value, so is ignored. The use of `ReturnBlock` could be replaced with a simple `CodeBlock`. This approach is not taken here to enable fair comparison with results without MVR.

Experiment 1

The MVR evaluation procedure relies upon there being multiple potential return variables of the correct data-type that are in-scope at the point of return. Where only one variable of the right data-type exists there can be no performance benefit and the MVR technique is semantically identical to the alternative approach of evolving the choice of return variable. In the first experiment, no constructs supporting variable declaration are used, so the only available variables are those that are supplied as inputs to the system. The number of variables matching the return data-type on each of the six test problems are:

Factorial:	2
Fibonacci:	3
Even-n-Parity:	2
Reverse List:	1
Sort List:	1
Triangles:	1

Only the factorial, Fibonacci and even-n-parity problems have more than one potential return variable, all others are unable to benefit and so are excluded. Table 5.10 summarises the results for the runs on these three problems with MVR enabled. The related results from chapter 4 are reproduced for comparison. Performance curves showing the progression of the success rate and computational effort over the generations are displayed in Figure 5.7. As expected, the results where MVR was used show higher success rates and lower required computational effort on all problems, in comparison to the results where MVR was not used. The non-overlapping confidence intervals suggest the reduced computational effort is a statistically significant result on all three problems studied.

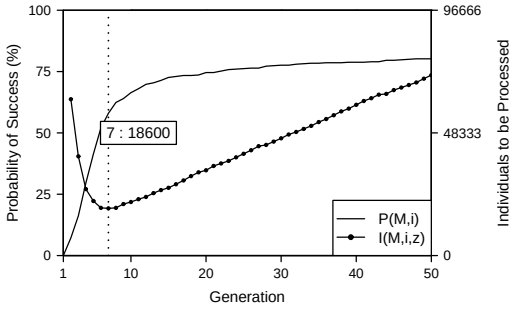
Table 5.10: Summary of the results of using the MVR method of fitness evaluation. Rows labelled SFGP+MVR are where the MVR method was used, while the SFGP rows show the results where MVR was not used for comparison (the setup was otherwise identical). The approach used to calculate each of these values is described in detail in section 3.3.

	Exp.	*Train%*	*Test%*	*Effort*	*95% CI*	Evals
Factorial	SFGP	72.8	72.0	25,700	22,700 - 29,200	514,000
	SFGP+MVR	**80.4**	**80.2**	**18,600**	16,500 - 21,000	372,000
Fibonacci	SFGP	51.6	49.6	69,300	59,300 - 81,500	1,386,000
	SFGP+MVR	**70.8**	**69.2**	**24,600**	21,500 - 28,200	492,000
Parity	SFGP	90.2	80.0	29,500	26,000 - 33,700	236,000
	SFGP+MVR	**96.8**	**92.4**	**17,400**	15,500 - 19,500	139,200

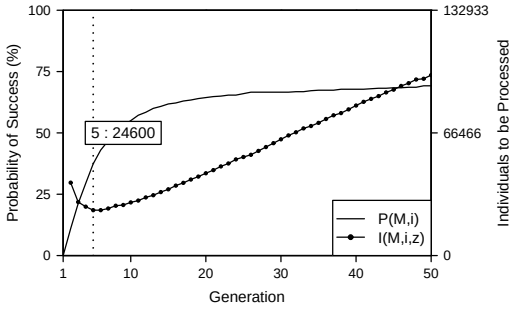
Experiment 2

As described in the first experiment, the use of the MVR technique is only of benefit where multiple potential return variables are possible. In the second experiment, variable declarations are enabled, so new return variables can potentially be declared. However, on only the factorial, Fibonacci, even-n-parity and triangle problems is the data-type of the declarations the same as the required return value. The remaining two problems are excluded as they are unable to take advantage of the MVR method with this setup.

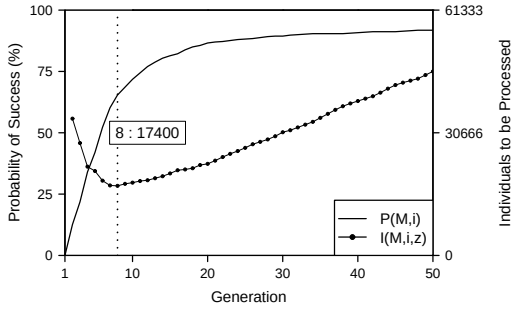
A summary of the results for experiment 2 is given in Table 5.11 along with a listing of the DECL results without MVR enabled for comparison. The related



(a) Factorial



(b) Fibonacci



(c) Even-n-parity

Figure 5.7: Performance curves for each test problem where the SFGP+MVR experimental setup is used and MVR is enabled. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

performance curves showing the progression of the success rates and computational effort are displayed in Figure 5.8. The success rates where the MVR evaluation procedure is used are substantially improved on *all* test problems studied, in comparison to where MVR is not used. The results for the triangles problem are of particular interest, because all additional return variables must have been the result of a variable declaration. On the triangles problem, there is a large but not statistically significant reduction in computational effort, when MVR is enabled. Certainly not sufficient to overcome all of the performance disadvantage that is seen when using `Declaration` nodes.

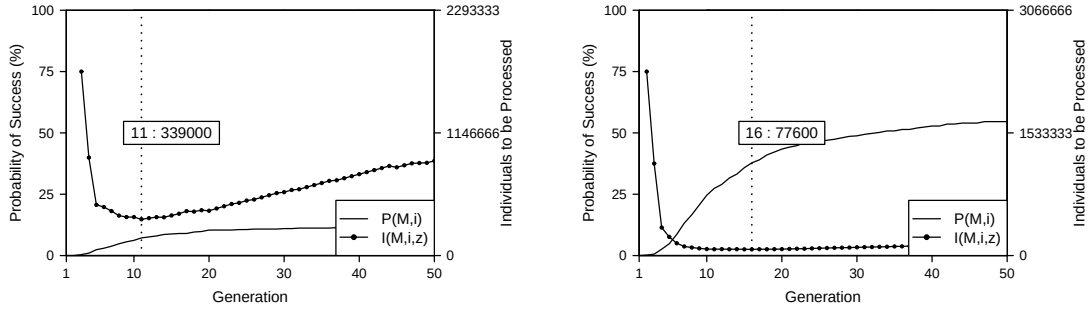
Table 5.11: Summary of the results of using the MVR method of fitness evaluation when variable declarations are used. Rows labelled DECL+MVR are where the MVR method was used, while the DECL rows show the results where MVR was not used for comparison (the setup was otherwise identical). The approach used to calculate each of these values is described in detail in section 3.3.

	Exp.	*Train%*	*Test%*	*Effort*	*95% CI*	*Evals*
Factorial	DECL	7.8	7.8	519,000	360,000 - 752,000	10,380,000
	DECL+MVR	**12.2**	**12.2**	**339,000**	245,000 - 470,000	6,780,000
Fibonacci	DECL	41.0	40.2	167,000	141,000 - 198,000	3,340,000
	DECL+MVR	**55.4**	**54.8**	**77,600**	67,400 - 89,800	1,552,000
Parity	DECL	98.2	93.4	16,600	14,900 - 18,600	132,800
	DECL+MVR	**100.0**	**98.4**	**8,980**	8,080 - 10,100	71,840
Triangles	DECL	25.8	25.8	45,600	34,500 - 60,600	273,600
	DECL+MVR	**29.6**	**29.6**	**38,800**	29,900 - 50,400	232,800

Table 5.12: Listing of the average number of possible return variables per individual and the percentage of computational effort where MVR is enabled by comparison with the case where MVR is not enabled. A percentage of 75 indicates that the computational effort where MVR was enabled was three quarters of what it was where MVR was not enabled.

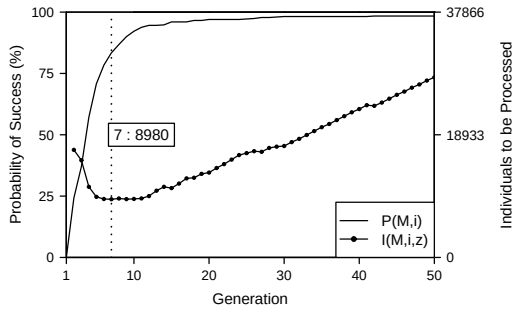
	Return Vars	*%Effort*
Factorial	2.75	65.3
Fibonacci	4.48	46.5
Parity	2.67	54.1
Triangles	1.86	85.1

It is clear that the MVR technique can improve performance and this is not surprising, since it increases the area of the solution-space that is searched in a manner that the original solution space is contained within it. It was suggested

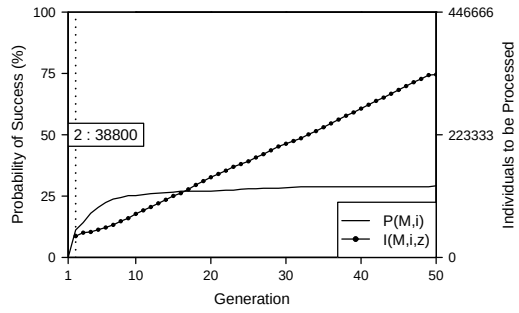


(a) Factorial

(b) Fibonacci



(c) Even-n-parity



(d) Triangles

Figure 5.8: Performance curves for each test problem where the DECL+MVR experimental setup is used with variable declarations and MVR is enabled. $P(M, i)$ is the success rate and $I(M, i, z)$ is the number of individuals to process to find a solution with 99% confidence.

early in this section, that the magnitude of the performance improvement due to MVR could be expected to be correlated with the number of potential return variables. This seems to be a reasonable conjecture. Table 5.12 shows the mean number of potential return variables that were available in each program. Comparing these values against the proportional amounts of computational effort with respect to the baseline case where MVR is not enabled shows some support for this idea. The results for the Fibonacci problem, which has by far the largest number of potential return variables, shows the largest proportional drop in computational effort of all the problems and the triangles problem shows the smallest drop with the smallest number of return variables. However, on the factorial and even-n-parity problems, where a similar number of potential return variables existed, there was quite a considerable difference in the change in computational effort. Part of this disparity may be explained by the high success rates on the even-n-parity problem, leaving less room for improvement in the computational effort.

The MVR technique is a very simple way of improving performance with little computational expense. However, its practical application has the assumption that the execution of a program tree is computationally expensive, while the calculation of a fitness score from the result is cheap. If there are applications where the inverse is true, then the practical benefit of MVR would be reduced or even negated. This is because it introduces more fitness scoring and in this scenario it may be more efficient to use a larger population size or perform additional runs.

5.4 Summary

Two enhancements to the SFGP algorithm have been proposed in this chapter. One for the purpose of supporting a new range of program constructs and the other for improving the evolutionary performance, but both target the evolution of high-level imperative programs. The concept of a dynamic syntax has been introduced with the aim of supporting the evolution of programs with new limited

scope variables. The dynamic syntax is made possible with the use of *syntax updates*, which allow nodes to modify the available syntax for their child subtrees or for successive nodes in the tree. Experiments were conducted which showed the performance impact of variable declarations could be beneficial on some problems, but the key motivations for supporting variable declarations are the generation of programs with more standard programming constructs and reducing the required insight into the solution space.

The second of the two enhancements covered in this chapter outlined a technique called Multi-Variable Return, for improving the performance of the algorithm when evolving programs with an imperative structure. MVR makes use of a simple method to evaluate multiple variations of each program without any additional executions, by calculating one fitness score per in-scope variable. It was experimentally shown that where multiple candidates for the return variable exist, required computational effort was reduced and there is a strong suggestion that the drop in computational effort is correlated with the number of available variables.

Chapter 6

An Analysis of Genetic Programming with Software Metrics

6.1 Introduction

Human programmers use many tools and techniques to help them produce better computer programs, such as unit tests, design patterns, program verification and formal specifications. As the program code that can be evolved with GP becomes more complex, the methods used by human programmers become more relevant to the evolutionary process of code production. Harnessing these techniques may help to overcome the same problems that human programmers use them for, such as increasing scalablity by producing more structured or modularised code. There are already examples in the GP literature of borrowing aspects of the software development process to enhance the GP search, such as the use of unit tests to evaluate the fitness of individuals [48,164] and the evolution of programs that satisfy formal specifications [9]. Another set of tools which we believe may have some useful applications in GP are *software metrics*. Software metrics are quantifiable measures of some property of software or the software engineering process. Some

very simple software metrics are already widely used within GP, for example, program length and depth, which are often used as part of a constraint to tackle bloat. But, there are many other measures which may have some application to the GP algorithm.

In this chapter, GP evolved code is analysed with various well established software metrics from the software engineering community. To the author's knowledge, a detailed analysis of this kind has not previously been conducted. The motivation of this analysis is to identify potential applications for software metrics to the GP algorithm, to improve the fitness of solutions or reduce the time spent on program evaluation. If a complexity metric correlates highly with evaluation time and correct solutions are still consistently found at lower complexities, then the metric could be used to focus the search effort on areas of lower complexity which are cheaper to evaluate. This would be particularly valuable on problems where fitness evaluation is very computationally expensive, such as image processing applications.

The rest of this chapter will be organised as follows. Section 6.2 will describe some of the related GP research that this analysis builds on. Then in section 6.3 the subject of software metrics is introduced, with a description of some well known software metrics for measuring code size and complexity. Seven of these metrics are described in more detail and are then applied to programs generated by the SFGP system. The results of this analysis are presented in section 6.4. Then conclusions are drawn in section 6.6 before the chapter is summarised in section 6.7.

6.2 Related Work

Some simple metrics for measuring program size are already widely used in GP. A constraint on program depth or program length is one of the common ways for controlling the well documented problem of bloat [89, 91, 147]. Without any restriction, the average size of programs tends to increase rapidly over generations,

which can have negative implications for the time required to evaluate a program and leads to programs that are difficult to understand. Crane et al. studied the effects of program size limits in tree GP [33] and linear GP [106]. They found that both length and depth were effective methods for controlling the average size of programs in tree GP, but that limiting program length has less impact on tree shapes than limiting depth. Beadle and Johnson [12] also considered the number of functions, the number of terminals and the number of unique terminals, in their analysis of program size. Other research has sought to measure program shape [12] or the time complexity of the algorithms that are evolved [3]. Fitness scores used to guide the selection mechanism are also a form of metric, which attempt to quantify the quality of a program with respect to a stated problem.

There are examples of software metrics being used with metaheuristic algorithms in a software engineering context. In the search-based software engineering community there have been a range of studies that looked to apply software metrics to the problem of program design [15, 73, 146]. For example, both Simons et al. [146] and Bowman [15] used multi-objective genetic algorithms to effectively assign methods and attributes to classes in a class diagram based on a measure of coupling and cohesion. Jensen and Cheng [73] also used the degree of coupling and cohesion to guide their use of GP for automatic refactoring of computer programs to use design patterns. The conclusion of their study was that their system could successfully refactor programs to introduce design patterns, but it could not be automated completely, as a human is required to review the range of alternative design patterns that are possible.

6.3 Introduction to Software Metrics

Software metrics is the subject of measuring properties of software and the software engineering process. They have been used for a diverse range of purposes including the prediction of software quality, performance optimisation and the management of project resources [46]. One of the main motivations for measuring

attributes of software is for quality control. Aspects of the code can be measured to indicate those modules that may be error-prone or more difficult to maintain, to highlight those areas most likely to benefit from further testing or refactoring. There are two broad categories of software metric: *direct* metrics, where the property itself is measurable and *derived* metrics, where there is no way to measure the property directly (or where the software does not exist yet), so the metric is a prediction based on a known relationship with other properties which are measurable. In this chapter, we are interested in either direct or derived metrics, but only those which are dependent on the actual program code. So, measures of the development process, such as programmer productivity, are not discussed here.

The simplest aspect of code to measure is that of program size. There are a number of different approaches for measuring size. The number of lines of code is commonly used, but requires a careful definition of what constitutes a line of code. For example, should comments and blank lines be included? There is some evidence that the number of lines of code correlates with the number of defects in a program [50]. Other attempts to measure size have concentrated on the amount of functionality that the code delivers [6, 38]. Halstead [59] proposed a range of code metrics which measured different properties including aspects of program size. These metrics are discussed in section 6.4.

Many software metrics that have been proposed are based on the control-flow graph of a program. These are often claimed to measure structural complexity. The control-flow graph of a program is a graph where each node represents a series of sequential instructions and edges are used at branching points to indicate the alternative execution paths that are possible. McCabe's cyclomatic complexity [103] is probably the most well known of these metrics, but Prather's μ measure [131] and NPATH [112] are other complexity measures calculated based on the control-flow. Each of these metrics are used in our analysis and will be described in detail in section 6.4. Oviedo [128] used a similar measure derived from the data flow characteristics of a program. A comparison of cyclomatic complexity, number of statements, Oviedo's measure and one of Halstead's metrics

has been conducted [165]. One conclusion from this study was that both the cyclomatic complexity and the number of statements consider the components of a program to have inherent complexity, while Oviedo's data-flow measure places emphasis on the context of components. Halstead's metric was determined to fit somewhere between these two situations. Other measures of logical complexity include a measure of the density of IF statements [52] and a metric proposed by Thayer et al. [155], which involves summing the number of logic statements, branches and the loops/IFs at each nesting level.

There are other metrics which have sought to measure system complexity – the structural design of systems comprised of multiple modules [157, 176, 178]. These cannot be applied to programs of the scale evolved in this thesis, because we are currently only concerned with the generation of code for individual subroutines. Similarly, a range of software metrics have been proposed specifically for the design of object-oriented programs [58, 95]. These may have some application for those GP systems that claim to be able to evolve object-oriented code, such as OOGP [2], but are currently out of the scope of this research.

6.4 Analysis of Genetic Programs

There are a vast number of software metrics that have been proposed, only some of which could be used in this analysis. Many are not applicable to the type of program that our system produces and many more are difficult to incorporate in a study for other reasons, such as producing a tuple as a score rather than a single metric value [61]. The metrics that are used were chosen because they are relatively widely known and are sufficiently simple to be comparable to the other metrics in the study. The metrics in this study are: cyclomatic complexity, Halstead's programming effort, NPATH, Prather's measure μ and the total number of statements. In addition, the commonly used GP program metrics of program depth and program length (number of nodes) are added to the comparison. In total this gives seven metrics.

Table 6.1: Pearson linear correlation coefficient between each metric and both the fitness and the evaluation time. The p-value in all cases is $< 2.2 \times 10^{-16}$, except for the Prather metric on the Parity problem, where $p = 4.3 \times 10^{-6}$ for the fitness property and $p = 0.8789$ for the time property.

Metric	*Property*	Factorial	Fibonacci	Parity	Reverse	Sort	Triangles
Length	Fitness	-0.0499	-0.1163	-0.1023	-0.1156	-0.1127	0.0731
	Time	0.1636	0.2044	0.3594	0.2112	-0.0840	0.0830
Depth	Fitness	-0.1035	-0.1360	-0.1266	-0.1750	-0.1510	0.0214
	Time	0.0645	0.0732	0.1198	0.1115	0.0609	0.0426
Statements	Fitness	-0.0737	-0.0702	-0.1098	-0.1134	-0.1000	0.0705
	Time	0.1504	0.2320	0.3590	0.2099	0.3157	0.0830
Cyclomatic	Fitness	-0.0737	-0.0702	-0.0848	-0.1134	-0.1000	0.0705
	Time	0.1504	0.2320	0.1388	0.2099	0.3157	0.0830
Effort	Fitness	0.0452	-0.1639	-0.0753	0.0226	-0.0075	0.2355
	Time	0.0818	0.1367	0.3556	0.2040	-0.0531	0.1736
Prather	Fitness	-0.0383	-0.0437	0.0020	-0.0922	-0.0840	0.0809
	Time	0.1541	0.2334	-6.7×10^5	0.2141	0.3261	0.0864
NPATH	Fitness	-0.0098	-0.0068	-0.0073	-0.0207	-0.0026	0.0820
	Time	0.0814	0.1406	0.1398	0.1076	0.2986	0.0726

Data was gathered from 500 runs of each of the six test problems used throughout this thesis. Each individual across all runs and all generations was recorded, along with its fitness, the time required to evaluate it, the generation it was found in and the value of each of the seven metrics when applied to it. The control parameters used for all problems were as for the SFGP experiments in chapter 4, shown in Tables 4.8–4.13, with the exception that a maximum depth of 10 was used for all problems. This was to allow fair comparison between the problems using the depth metric and to try to reduce the extent to which the complexity of the programs is restricted by a tight size bound.

6.4.1 Explanation of Metric Charts

Charts are used in this chapter to portray the relationship between a metric and the fitness and evaluation time properties. Each of these charts contain a lot of data and require some explanation. Figure 6.1 shows an example of one of these charts. Each chart displays two sets of data, which are described separately here.

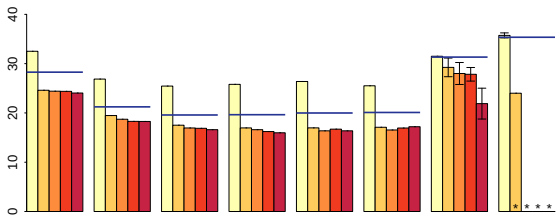


Figure 6.1: Example metric chart, which shows the relationship between a metric and either the fitness or evaluation time property of individuals on a problem. The metric values are grouped along the x axis and split into 5 bars representing 10 generations each, where the height of the bars is for the fitness or evaluation time property.

Average Data

The thick horizontal bars (shown in Figure 6.2 with the generational data removed), show the relationship between a metric and either the program fitness or evaluation time. Every individual in the study is sorted into one of 8 groups based on the value of the metric under investigation. Each of these groups covers an equal range of metric values, so if the minimum metric value in the sample was 1 and the maximum value was 80, then those individuals with a value of 1–10 would be inserted in the first group, 1–20 in the second group and so on. Grouping the individuals by metric value like this allows us to more easily compare the different metrics which would otherwise be on vastly different scales. Each horizontal bar on the chart is then the average fitness or evaluation time of the individuals in that group, with group 1 on the left through to group 8 on the right.

Generational Data

The horizontal bars described in the previous section include all individuals regardless of their generation. However, in GP it is often useful to consider how trends change throughout the course of a run. To give a more complete picture, the individuals in each group in our charts are separated into 5 bars which show 10 generations each, so the first bar in each group shows the average fitness or evaluation time for individuals in that group from generations 1–10, the second bar from generations 11–20 and so on. By looking just at the first bar in each

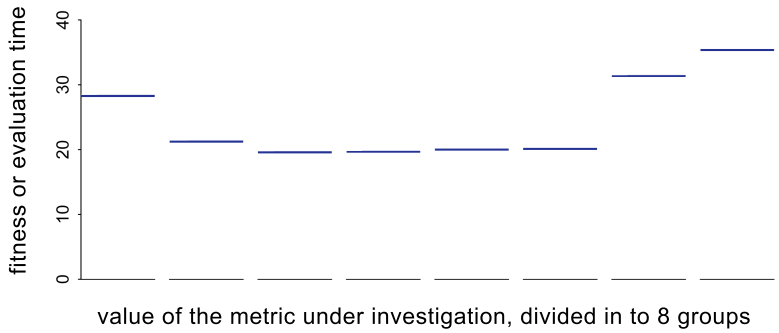


Figure 6.2: Example metric chart with generational data removed. Each horizontal bar indicates the mean fitness or evaluation time of the individuals in that group. The individuals are split in to groups based on their value for the metric being studied, with each group covering an equal range of metric values.

group it is possible to spot trends that occur only at the beginning of the runs and then this can be compared to the other bars to see how the situation develops. Figure 6.3 shows an example of these generational metric bars, with the horizontal bars removed.

Error bars are included on each of the generation metric bars, showing the standard error. Where no error bar is present, it indicates that the error was too small to accurately display. This occurs frequently, as many of the bars show the mean from a sample of 100,000s of values. However, in a few cases error bars are absent because the sample size is just 1 or less. An asterisk (*) is displayed at the base of the bar to indicate this. This is also important to distinguish the scenario where no individuals were identified in that group, from the case where the mean property value is zero or close to zero.

6.4.2 Program Tree Length

Program tree length is a measure of program size that is frequently used in genetic programming. It is a simple count of the number of nodes in a program tree. It is well established that during the evolutionary process, the average size of programs grows rapidly [89, 91, 147]. This is known as code bloat. Because of this, the size of programs are often constrained by properties such as program length and depth [78, 97]. The impact of these limits has previously been explored [33] and

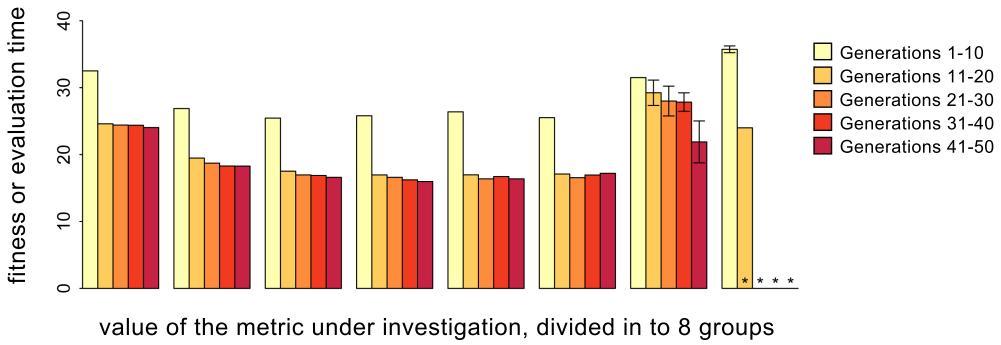
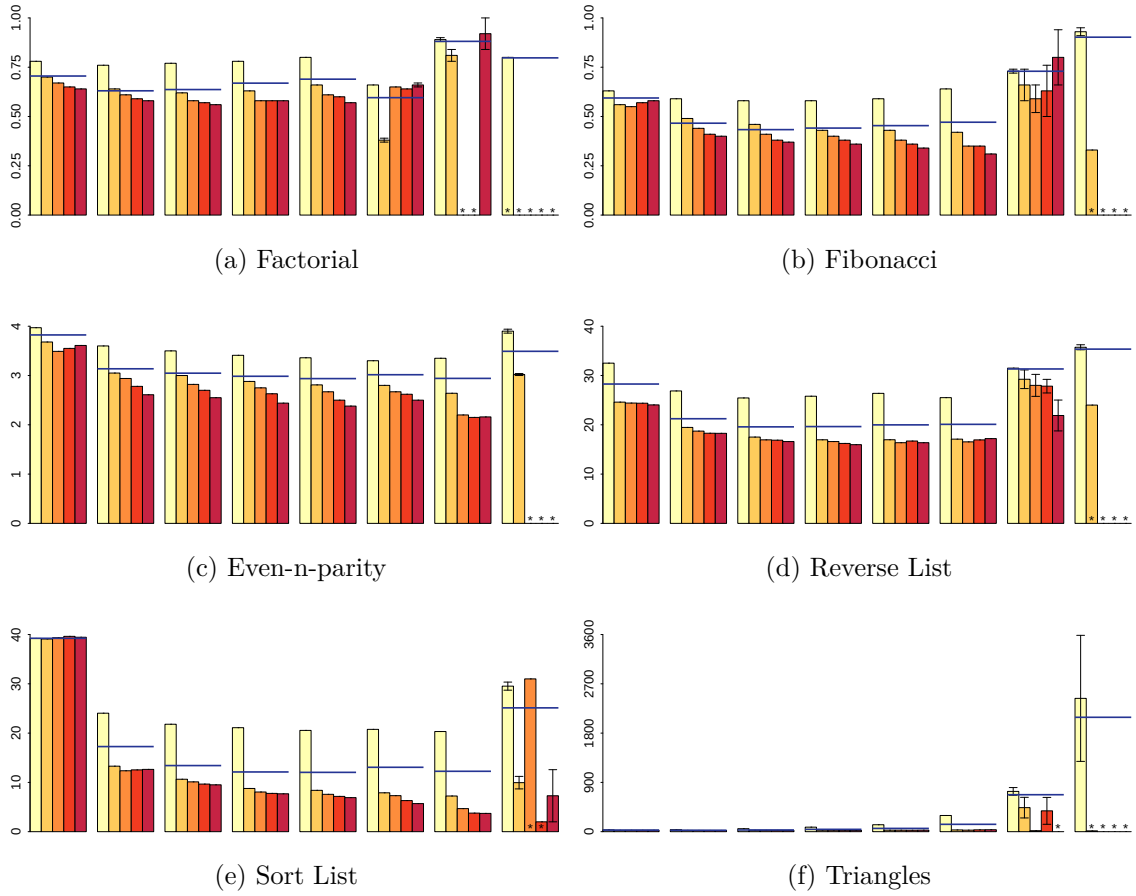


Figure 6.3: Example metric chart showing generational data with horizontal (average) bars removed. Each group is divided in to 5 bars based on the generation the individuals were discovered in. The first bar in each group shows the average metric value for the individuals in that group that were discovered in generations 1–10.

found to be equally effective at controlling the average program size. Rosca [135] examined the relationship between generality of programs and program length. He recognised that small solutions are associated with generalisation, but noted that it is difficult to produce small solutions with parsimony pressure without damaging the effectiveness of the algorithm. Rosca's novel suggestion was to use a measure of *effective* code size instead.

Table 6.1 shows the Pearson correlation coefficient between each metric and both the fitness and evaluation time of a program. It suggests that there is a very weak negative correlation between the length of a program and the program's fitness on all but one of the problems studied. Examining the chart in Figure 6.4 shows a more complete picture. There is a reasonably consistent pattern on most of the problems. The overall trend seems to be for fitness scores to drop as program length increases, but possibly with an increase in fitness in the longest programs, which could be the impact of the depth restrictions. The exception to this is on the triangles problem, where the problem fitness seems to increase with program length, although the trend is obscured by very small values. As would be expected, the generation bars show that in most groups of program lengths the fitness goes down as the generations progress. One notable exception to this is on the sort list problem, where the first group of program lengths display high



(a) Factorial (b) Fibonacci

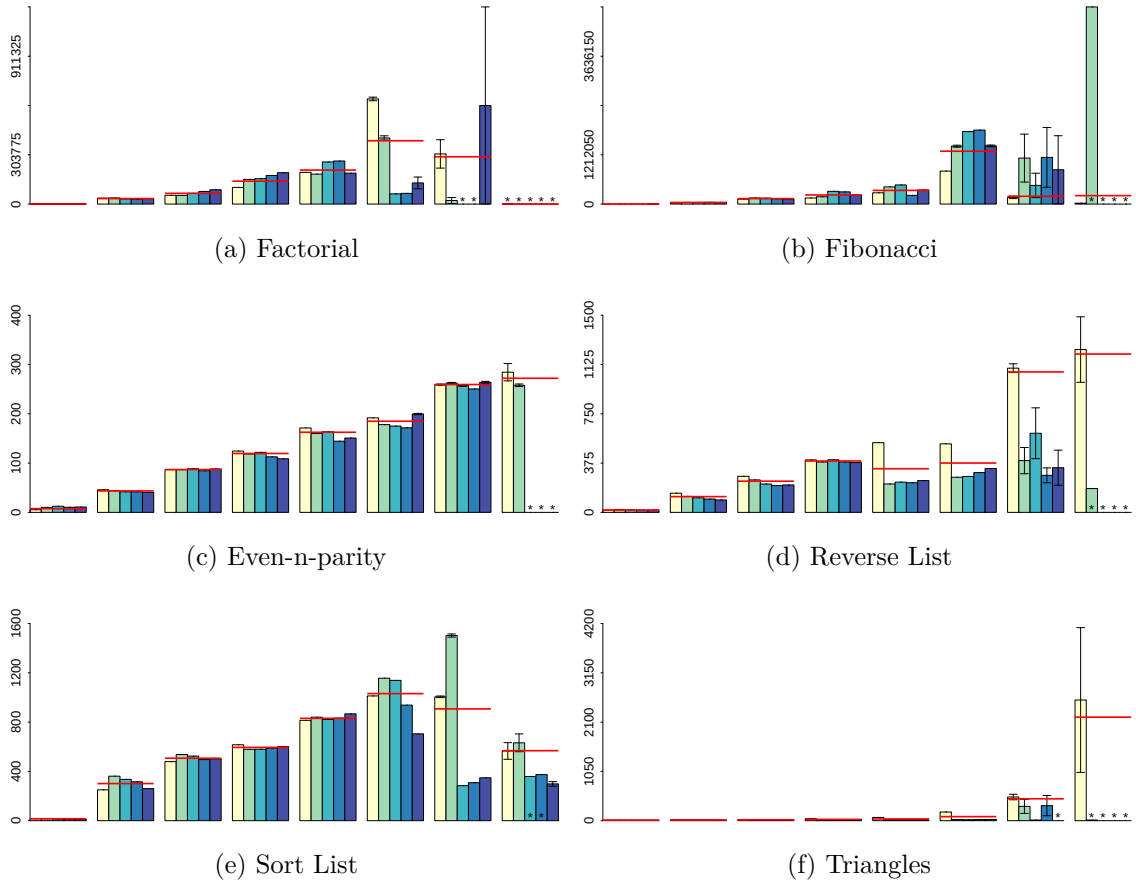
(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.4: Length × Fitness. Charts showing the relationship between the program length metric and the fitness of individuals

fitness throughout the generations. This is because the program is not solvable with programs of this length, given the available syntax.

Solutions to all problems are found with a large range of different program lengths. Figure 6.6 shows the distribution of program lengths of correct solutions that are found. Most notably solutions to the sort-list problem on average are found with almost twice as many nodes as for any of the other problems. This could be seen as an indicator of the difficulty of the problem, but it is also influenced by the syntax that is used for the problem, with some constructs naturally requiring more nodes because they have more inputs. These results do not illustrate the number of nodes that are actually effective, rather than introns that do



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.5: Length × Time. Charts showing the relationship between the program length metric and the time required to evaluate individuals

not contribute, so it is possible that the larger programs would be considerably shorter if subjected to some post-processing to remove ineffective statements.

As might be expected, a small positive correlation is seen between a program's length and its evaluation time. There are some interesting generational features to be noted in the evaluation time chart in Figure 6.5 though. On several of the problems, most notably sort-list, evaluation appears to be quicker in later generations than it is in earlier generations. One explanation for this, is that the more fit programs of later generations are more likely to set up loops with sensible upper bounds and to not nest more loops than necessary in comparison to the less fit individuals in the first generations.

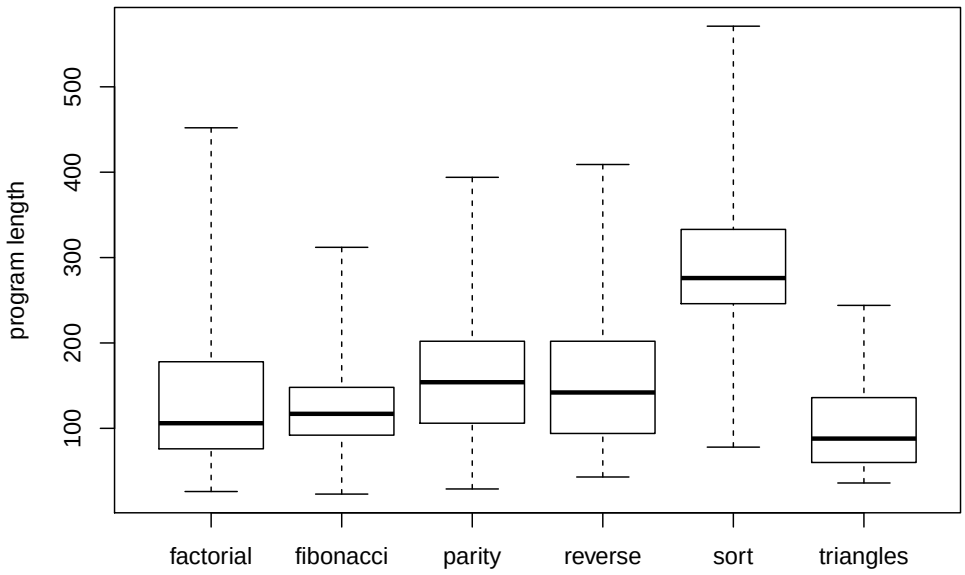
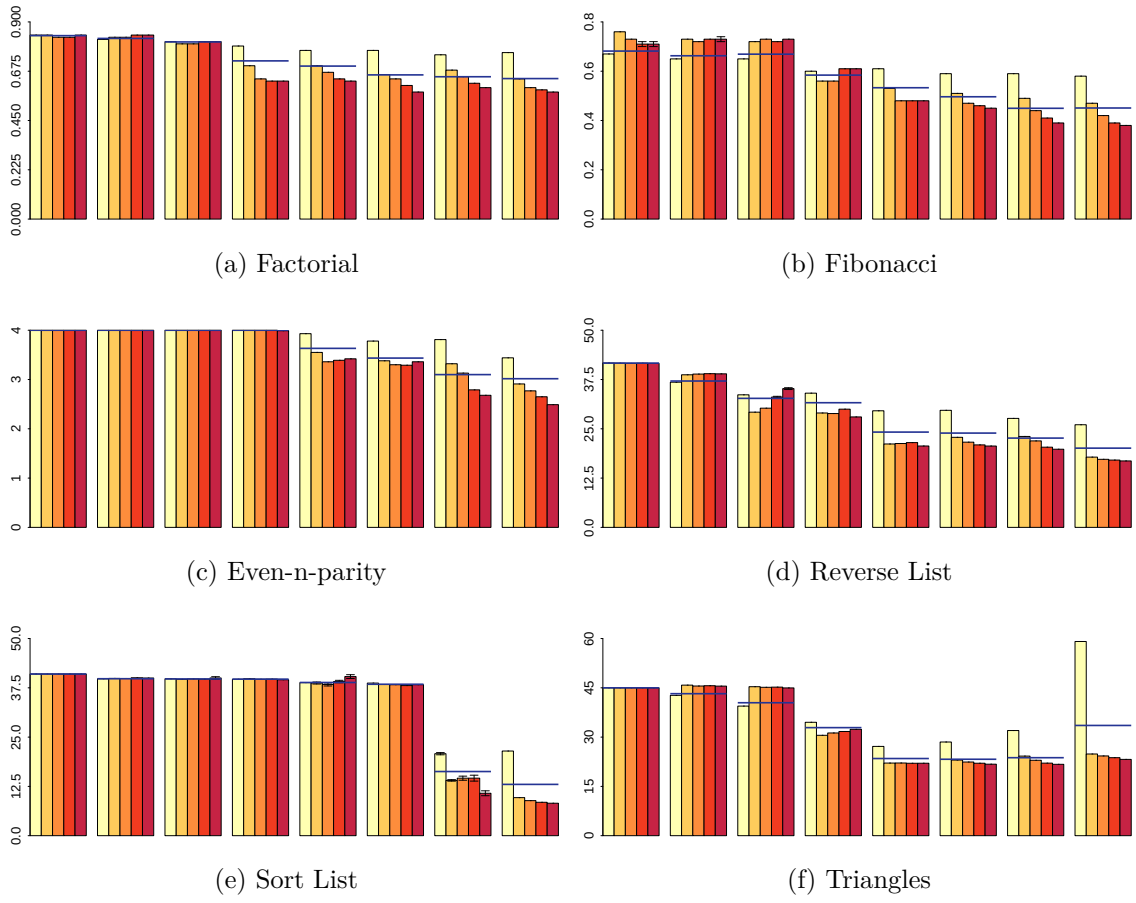


Figure 6.6: Boxplot showing the distribution of program lengths for programs that solved all the training cases on each of the test problems

6.4.3 Program Tree Depth

Program tree depth is an alternative measure of program size, which is also widely used in genetic programming as a size restraint. In all experiments in this thesis, depth constraints are used to ensure programs can be evaluated within a reasonable time. This is particularly necessary because loops are used, which can substantially increase evaluation time when they are nested multiple times. The depth constraint is likely to have an impact on the fitness and evaluation time of programs that are near to the limit, so ideally the depth constraints would be removed for our study. However, this would not be possible without adding an alternative constraint to limit the evaluation time and it is preferable to use the same constraint as has been used in practice. There is also value in seeing what the actual impact of the depth constraint is.

Table 6.1 shows that a weak negative correlation exists between program depth and program fitness on all but one problem. This trend is seen clearly in the charts for program depth shown in Figure 6.7. Each problem shows several program depth groups with very high fitness values, regardless of generation, followed by a downwards trend of fitness. This seems to suggest that, in general, using a greater



(a) Factorial (b) Fibonacci

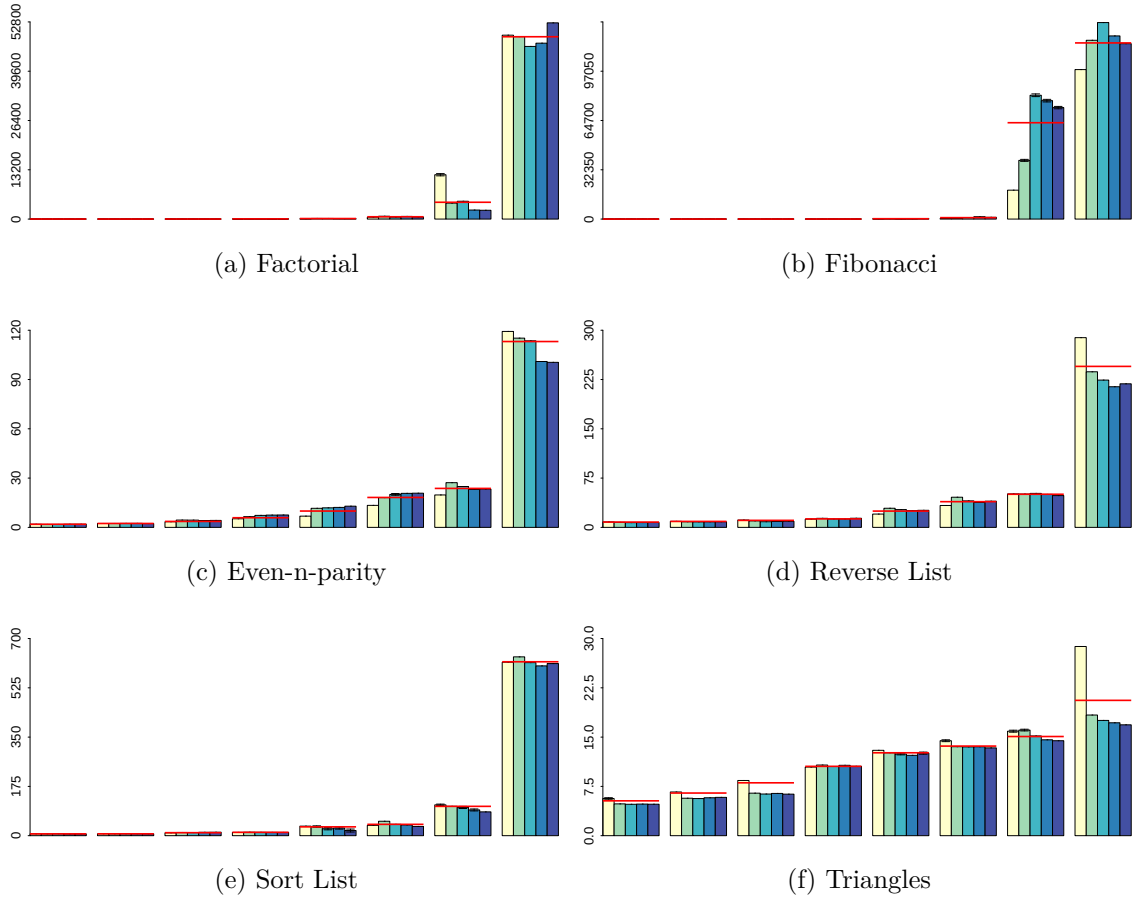
(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.7: Depth × Fitness. Charts showing the relationship between the program depth metric and the fitness of individuals

depth should be beneficial for performance. However, there is a positive correlation between depth and evaluation time which is seen both in the correlations table and the evaluation time charts in Figure 6.8. The influence of nested loops means that there is an exponential relationship between the depth and evaluation time at the greatest depths. In practice, this means that there is a point where the improved fitness of deeper programs is more than offset by the increased evaluation time and it might become more efficient to use a larger population or perform additional runs, than to increase program depth.

Figure 6.9 shows an overview of the distribution of depths for the programs that solved all training cases. The vast majority of solutions are found at the maximum allowable depth. The extent to which the lower whisker extends is a



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.8: Depth × Time. Charts showing the relationship between the program depth metric and the time required to evaluate individuals

reasonable indicator of the minimum depth that must be available to represent a correct solution. This suggests that the maximum depth settings that were used in the experiments in chapters 4 and 5 did not provide much surplus depth than what is required to solve the problem. Correct solutions are only discovered with lower depths than the maximum setting on the reverse list and sort list problems.

6.4.4 Number of Statements

The number of statements or number of lines of code, is another measure of program size. It has been used in the software development industry as a simple estimate of complexity and as a crude measure of programmer productivity. Programming guidelines often include recommendations for the maximum number

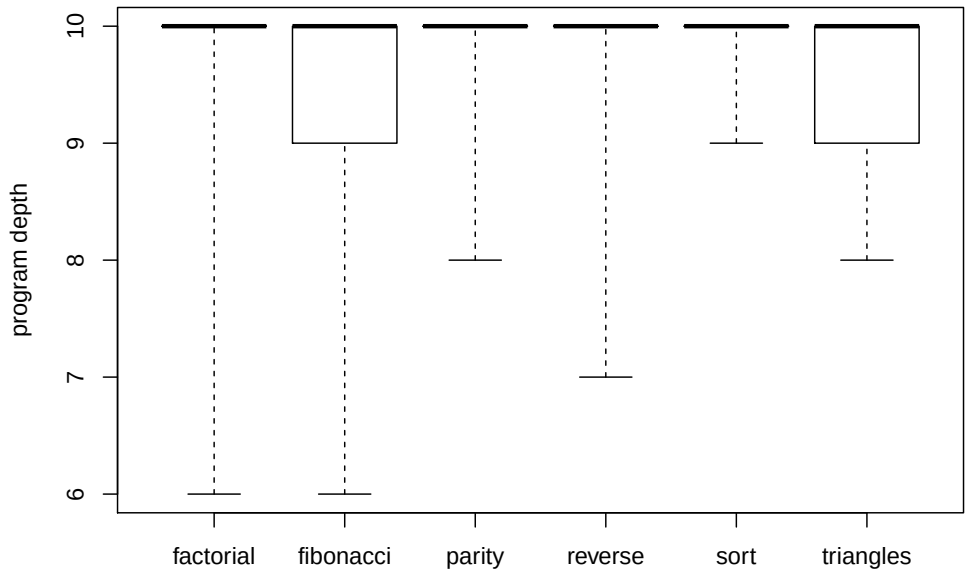
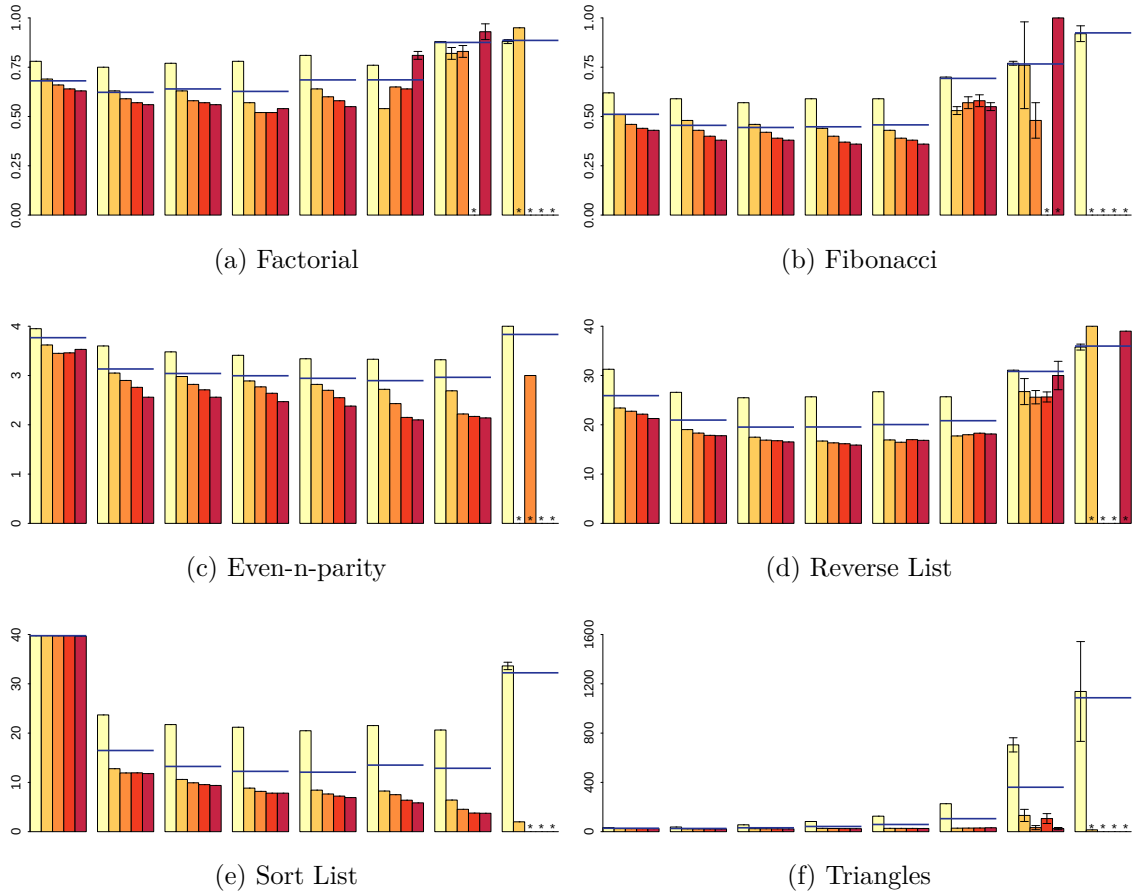


Figure 6.9: Boxplot showing the distribution of depth values for programs that solved all the training cases on each of the test problems

of lines of code to put in a module or subroutine. For example, the Java Code Conventions state that files longer than 2000 lines should be avoided [151]. The advantage of using number of lines of code as a simple complexity metric is that it is so easily calculated, but it has received heavy criticism [76,77]. As a method for measuring productivity, it is easily manipulated because code can be artificially bloated. Even when used honestly there are problems, as experienced programmers tend to emphasise code reuse and make more efficient use of language features and libraries, which leads to smaller programs. It also has the weakness of not being very useful for comparing code written in different languages, as different languages are naturally more verbose than others.

There are a number of subtle variations in the method of calculating the number of lines of code. Blank lines and comments may optionally be included in the count and for different languages different definitions of a statement are possible. In our calculation in SFGP, the metric value is a count of the number of nodes in a program tree that are a subtype of the abstract `Statement` node-type. Number of statements is an inherently imperative metric and may be considered to be the



(a) Factorial

(b) Fibonacci

(c) Even-n-parity

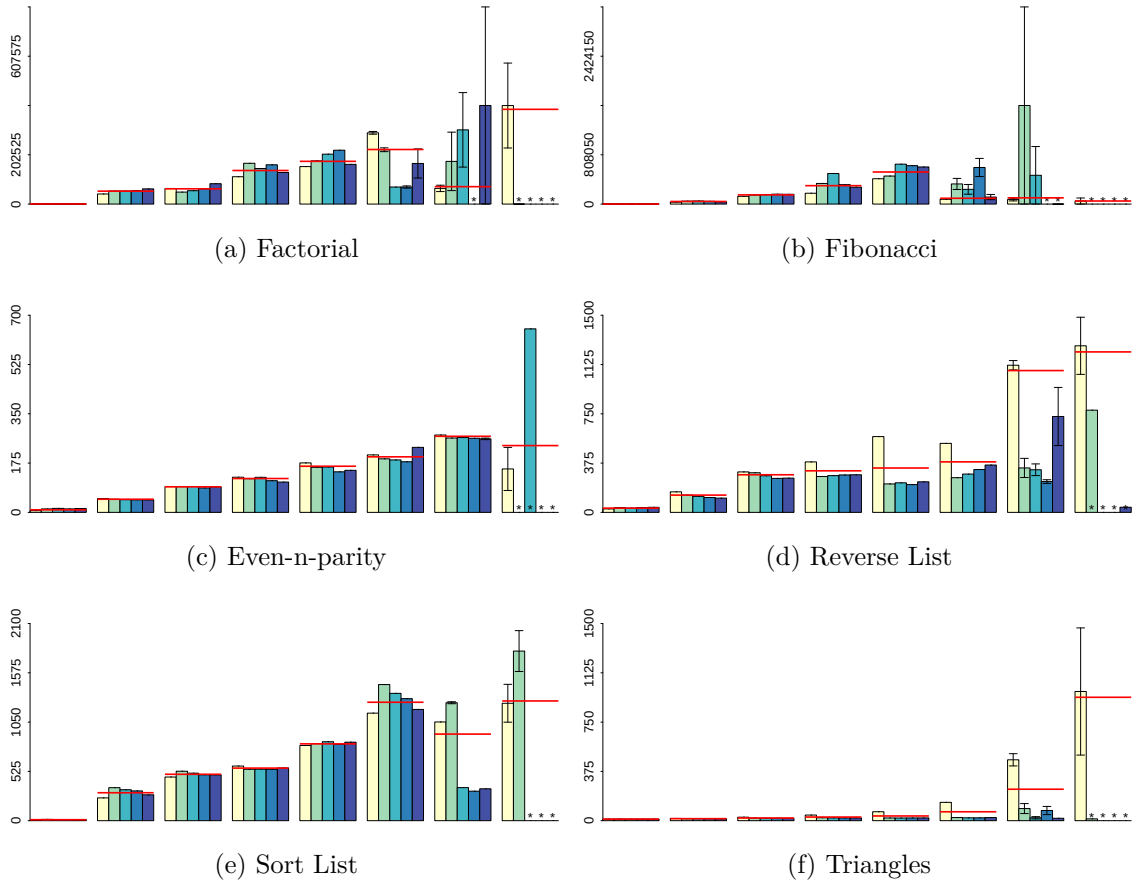
(d) Reverse List

(e) Sort List

(f) Triangles

Figure 6.10: Statements × Fitness. Charts showing the relationship between the number of statements metric and the fitness of the individuals

equivalent of program length or program depth which are not very intuitively applied to imperative programs. Our results reinforce this idea, with the fitness and evaluation time charts for the number of statements metric, shown in Figures 6.10 and 6.11, displaying a strong similarity to the program length charts in Figures 6.4 and 6.5. This suggests that for the evolution of imperative programs, a program length constraint could be replaced with a number of statements constraint with similar impact. For imperative programs, a number of statements parameter would be more intuitive to set than either a parameter for maximum program depth or maximum number of nodes. The distribution of number of statements for solutions, in Figure 6.12, is also very similar to the equivalent plot for program lengths.



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.11: Statements × Time. Charts showing the relationship between the number of statements metric and the time required to evaluate individuals

6.4.5 Cyclomatic Complexity

McCabe's Cyclomatic Complexity [103] is a well known software complexity metric. It is intended to provide a quantitive measurement of program complexity to be used as an indicator of how difficult code is to test or maintain. The cyclomatic complexity of a program is calculated as the number of execution paths through its flow graph. The flow graph for a program is a directed graph, where a vertex corresponds to a sequential block of code and edges correspond to branches in the control flow. The cyclomatic complexity $V(G)$ of a graph G, with n vertices, e edges and p connected components is given by the formula:

$$V(G) = e - n + 2p \tag{6.1}$$

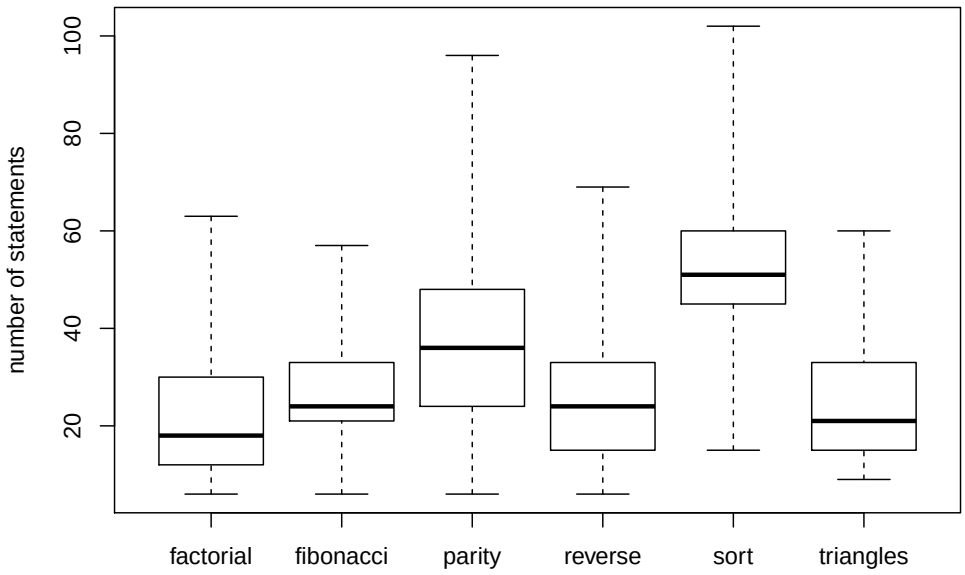
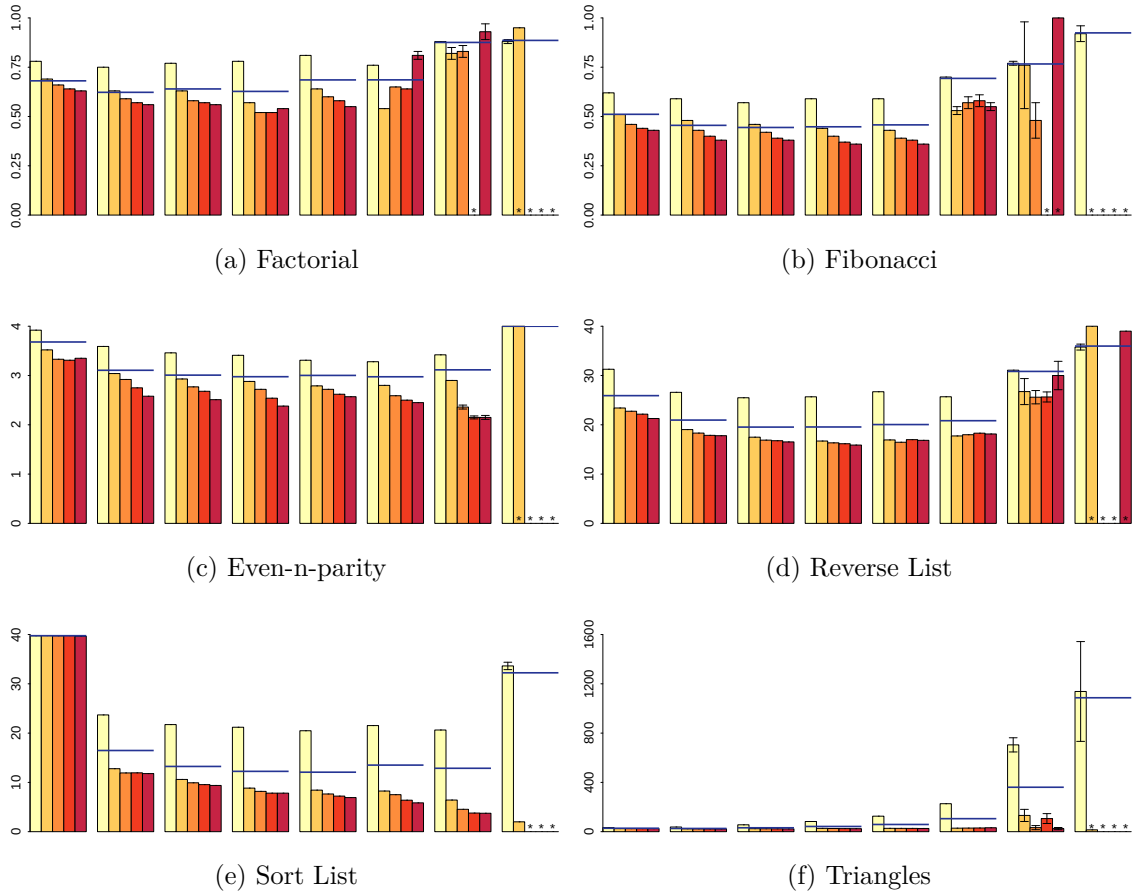


Figure 6.12: Boxplot showing the distribution of number of statement metric values for programs that solved all the training cases on each of the test problems

In practice, the cyclomatic complexity of a piece of code without multiple independent modules (connected components) can be easily calculated without reference to the control graph, as 1 plus the number of branching operations. A branching operation is any program construct that causes a deviation from the linear flow of the program. This includes if-statements and loops, but also predicates such as the boolean `AND` operation. The resulting score will always be $V(G) \geq 1$. McCabe recommended that programmers should limit the complexity of software modules to less than 10, with the intention that this would keep the number of independent paths manageable for testing.

A number of extensions and modifications to cyclomatic complexity have been proposed. Myers [111] identified that there are two ways of drawing flow diagrams, with each obtaining different cyclomatic numbers and suggested that the complexity of a program could be presented as an interval between these bounds. Hansen [61] introduced a lexicographically ordered 2-tuple score composed of the cyclomatic score and a simple count of the number of operators, which he suggested provided a more intuitive ordering of programs by complexity than either McCabe's or Myers's measures. Gong and Schmidt [55] identify another problem



(a) Factorial

(b) Fibonacci

(c) Even-n-parity

(d) Reverse List

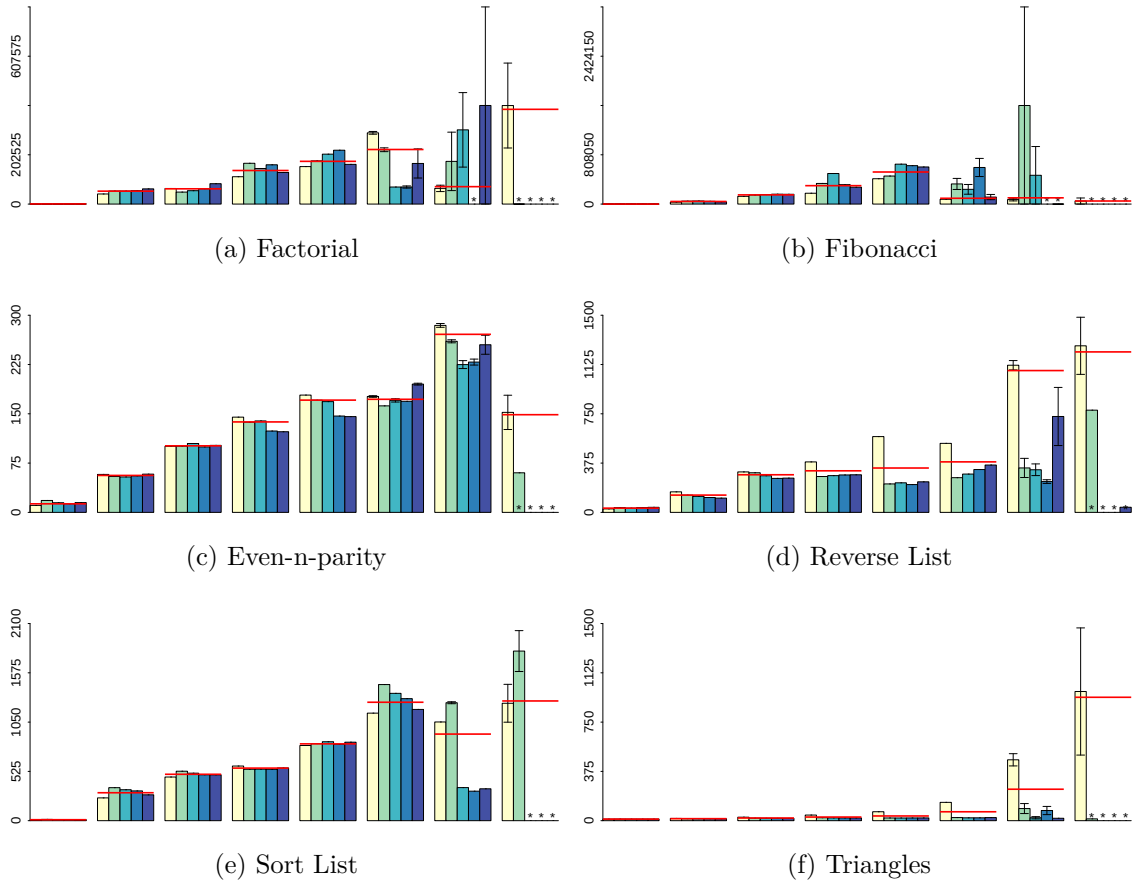
(e) Sort List

(f) Triangles

Figure 6.13: Cyclomatic × Fitness. Charts showing the relationship between the cyclomatic complexity metric and the fitness of the individuals

with McCabe's complexity metric. They criticise it for not considering the degree of nesting, which they argue contributes to the intuitive interpretation of complexity. To correct this, they proposed a method for assigning a value between 0 and 1 to the level of nesting of a program, which is then added to the cyclomatic number to give a decimal score of the complexity.

Although widely used, there has been some dispute as to the value of cyclomatic complexity as an indicator of faulty code. The main criticism is that it is no more accurate as a predictive measure than the number of lines of code [44,53,143]. Van der Meulen and Revilla [108] studied over 70,000 small C++ programs and found that cyclomatic complexity correlated highly with the number of lines of code, but did not correlate well with the number of defects. They do note that a



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.14: Cyclomatic × Time. Charts showing the relationship between the cyclomatic complexity metric and the time required to evaluate individuals

weakness in their research was that the programs studied were small ("dozens to several hundred lines"), but this is relevant to our study where the programs are of a similar scale. Our results, in Figures 6.13 and 6.14, corroborate the view that cyclomatic complexity measures little more than program length. The charts for both fitness and evaluation time are remarkably similar to the respective charts for the program length and number of statements metrics. Given the additional complexity of computing the cyclomatic complexity, in comparison to these simpler size metrics, it seems unlikely that there could be any practical application for cyclomatic complexity in improving the performance of the GP algorithm. The caveat to this conclusion is that the situation may be different for larger programs.

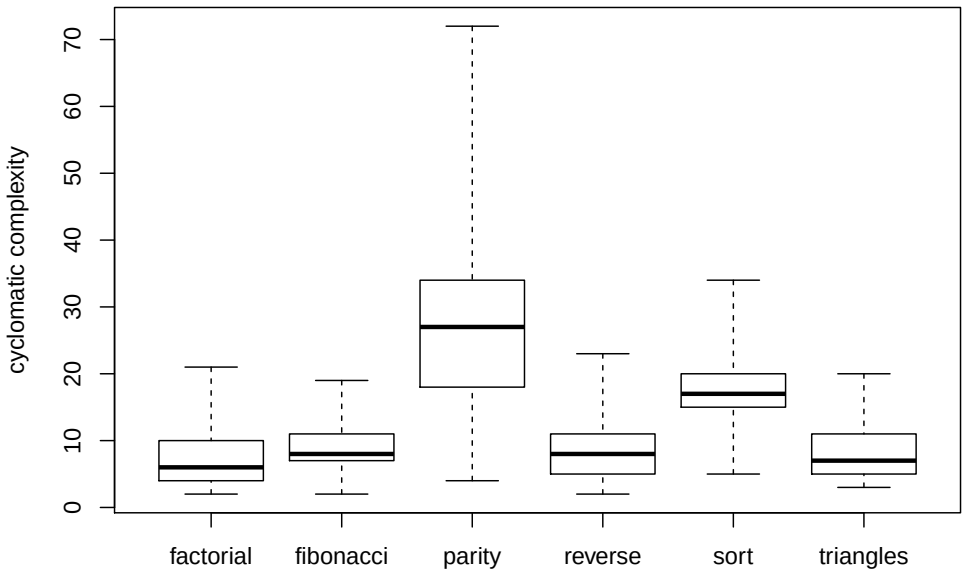
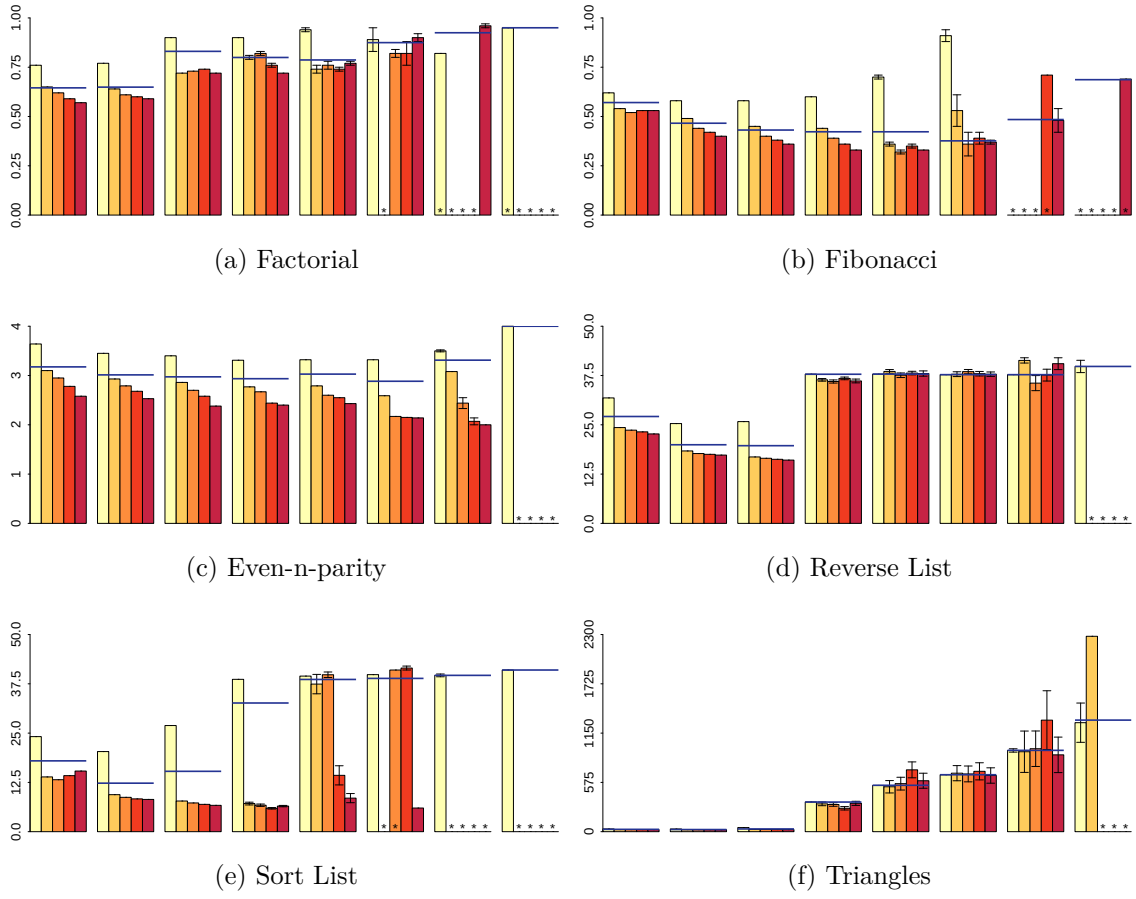


Figure 6.15: Boxplot showing the distribution of cyclomatic complexity values for programs that solved all the training cases on each of the test problems

Figure 6.15 shows the distribution of cyclomatic complexity values in the correct solutions that were found for each problem. Solutions for all six problems are identified below the recommended maximum cyclomatic complexity value of 10 and the mean cyclomatic complexity is also below the recommended limit on four of the problems. This is promising, but there is quite a range of complexities, with many solutions considerably higher than the recommended limit. Given two semantically equivalent programs, it would be preferable to have the one with lower complexity, so incorporating encouragement for less complex programs in to the fitness function may be useful. But given the apparent correlation between cyclomatic complexity and number of statements, minimising the number of statements might be just as effective. These box plots do not contain details of program generalisation, so it may be that programs which solve the test cases as well as the training cases have a different distribution of complexities.

6.4.6 Halstead's Effort

Halstead proposed a large range of different software metrics for measuring different aspects of software, which he collectively described as software science [59].



(a) Factorial (b) Fibonacci

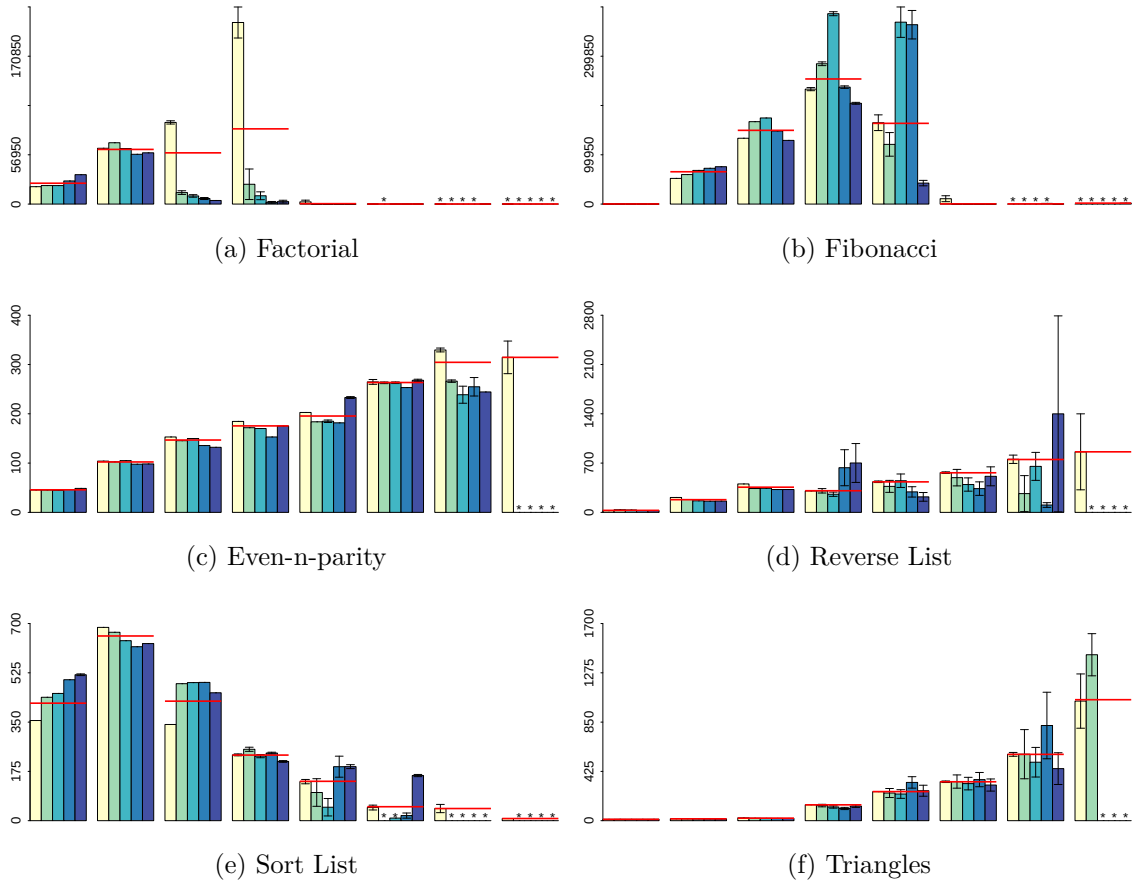
(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.16: Effort × Fitness. Charts showing the relationship between Halstead's program effort metric and the fitness of the individuals

These metrics include program difficulty, intended as a measure of the challenge involved in writing and understanding a program; program volume, to consider the density of the functionality in a program; and program effort which was designed to be used in estimating the time required to produce the code. Each of these metrics were based on calculations comprising four simple direct measures of the code.

- $n1$ = the number of unique operators
- $n2$ = the number of unique operands
- $N1$ = the total number of operators
- $N2$ = the total number of operands



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.17: Effort × Time. Charts showing the relationship between Halstead's program effort metric and the time required to evaluate individuals

Halstead laid out exactly which tokens constituted operators and which would be operands. Operators included tokens such as + - % = and most reserved words, while all non-reserved word identifiers and literal characters, numerics and string values were considered to be operands. The following list of derived software science metrics are based on the direct measures above:

- Program length: $N = N1 + N2$
- Vocabulary size: $n = n1 + n2$
- Program volume: $V = N * log2(n)$
- Difficulty level: $D = (n1/2) * (N2/n2)$
- Program effort: $E = V * D$

- Time to implement: $T = E/18$
- Number of bugs: $B = E^{2/3}/3000$

Our study looked at the program effort metric, E, which has been demonstrated to correlate with the number of errors in a module [49] and debugging time [35]. This suggests it may be a good alternative complexity metric to the cyclomatic complexity measure. However, the results of many of the experiments that suggested these correlations have since been demonstrated to be statistically flawed [60, 93], so the measure is somewhat controversial.

A comparison of our fitness and evaluation time charts for program effort (in Figures 6.16 and 6.17), to the equivalent charts for program length and number of statements, show far less similarity than was the case for cyclomatic complexity. However, there is little consistency across the problems studied, including similar problems such as factorial and Fibonacci. Table 6.1 shows that there is a positive correlation between program effort and program fitness on three of the problems studied, but a negative correlation on the remaining three. There is slightly more consistency seen between the program effort and evaluation time, with a positive correlation clearly identifiable on five of the problems. However, the sort list problem displays a weak negative correlation, which is suprisingly clear in the evaluation time charts. Also interesting are the differences between the phases of the runs; there seems to be no common trend between the metric groups. In general the results for Halstead's program effort metric seem to be slightly erratic and very problem dependent.

6.4.7 Prather's Measure μ

Prather presented an "axiomatic framework" for software complexity measures [131] and demonstrated that both McCabe's and Halstead's measures satisfy each of the axioms. A third metric, μ, was proposed and also determined to satisfy the requirements of the framework. This measure was designed to overcome some of the criticisms aimed at both of the other metrics, such as too close a correlation

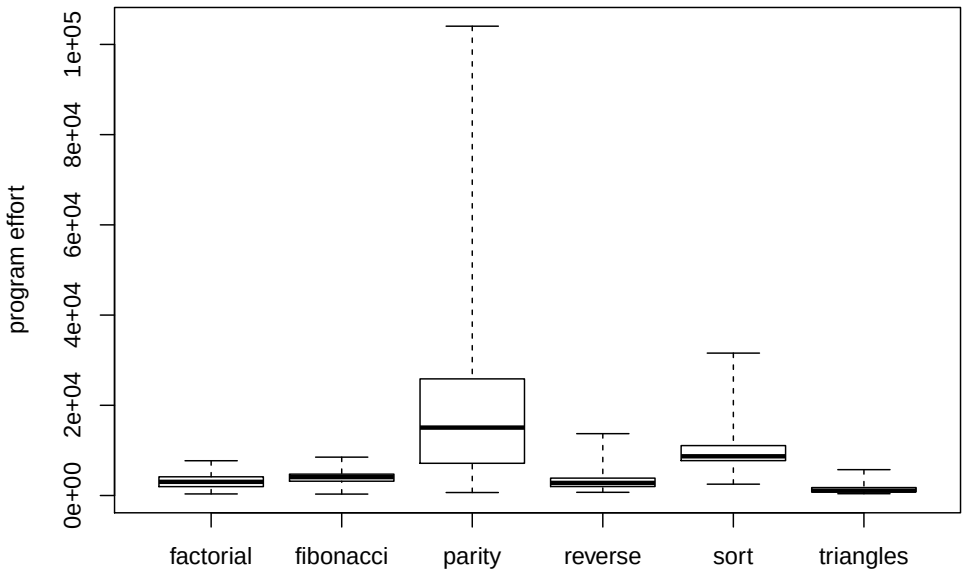


Figure 6.18: Boxplot showing the distribution of program effort metric values for programs that solved all the training cases on each of the test problems

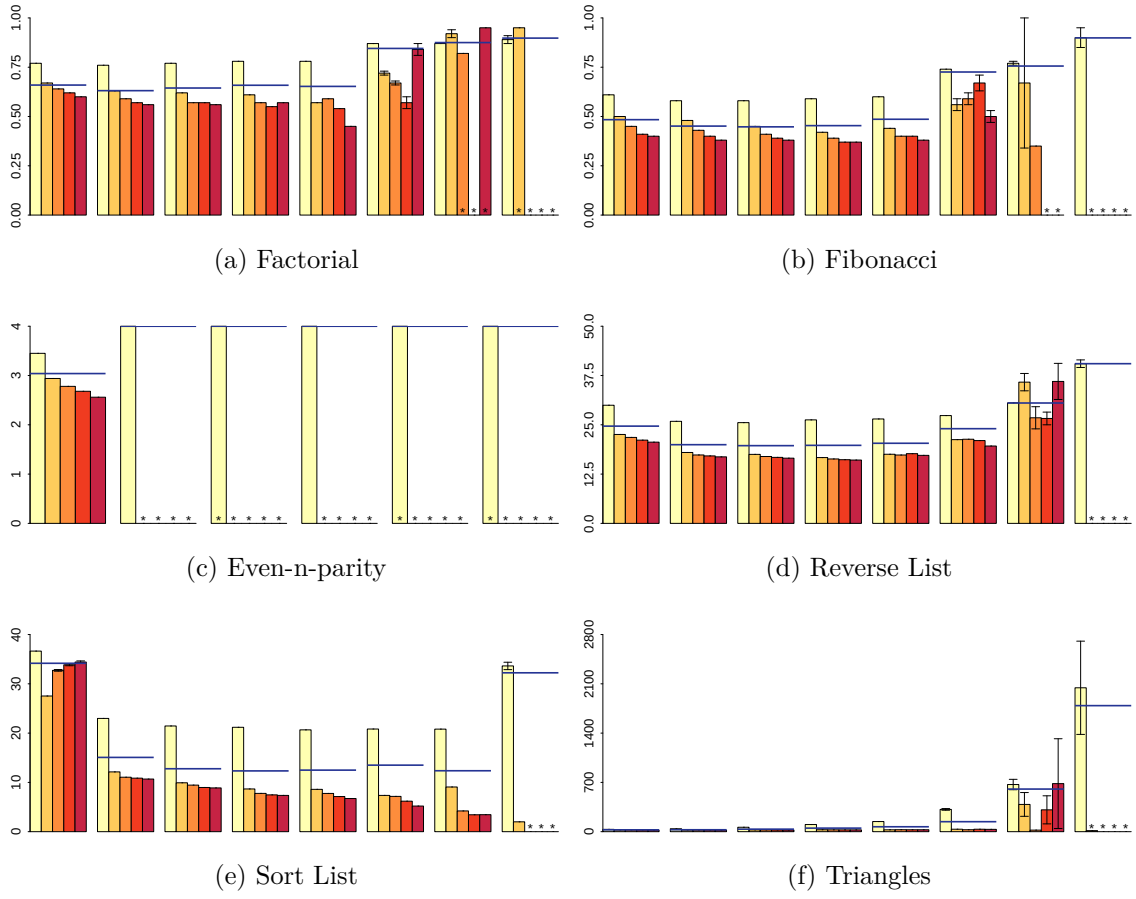
with the number of lines of code and no attention given to the degree of program nesting.

The calculation of μ starts by considering all simple statements as having a complexity of 1. The following three rules are then applied for each type of structured construct to produce a complexity score for a complete program.

- $\mu(S_1, S_2, S_n) = \sum \mu(S_i)$ - The complexity of a sequence of statements is the sum of its constituent statements' complexities.

- $\mu(\text{if } P \text{ then } S_1 \text{ else } S_2) = 2^{|P|} \times max(\mu(S_1), \mu(S_2))$ - The complexity of a conditional statement is twice the size of which ever has the largest complexity; the if block or the else block. If P is a complex condition composed of multiple boolean predicates then $2^{|P|}$ is used, where $|P|$ is the number of boolean relations.

- $\mu(\text{while } P \text{ do } S) = 2\mu(S)$ - The complexity of a loop is twice the complexity of the loop's body.



(a) Factorial (b) Fibonacci

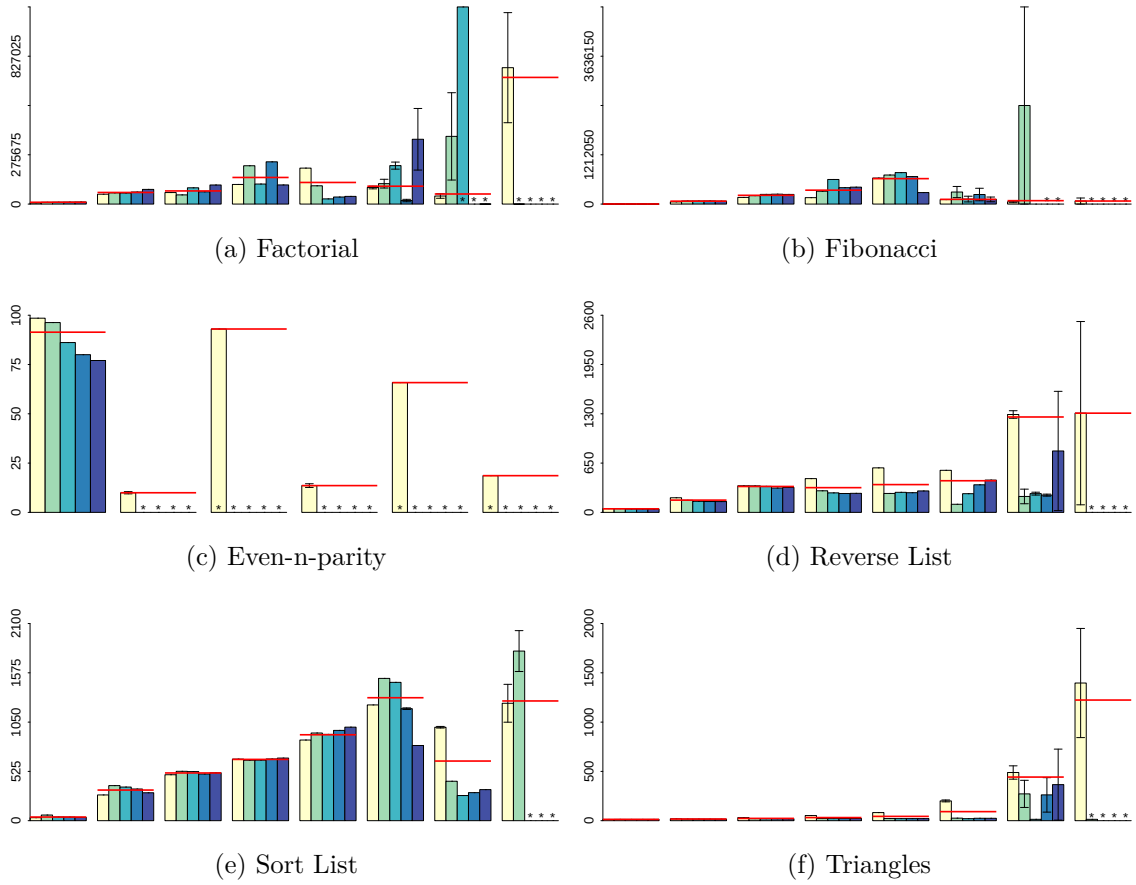
(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.19: Prather × Fitness. Charts showing the relationship between Prather's measure μ metric and the fitness of the individuals

Prather recommended a maximum module complexity of 100 and suggested that the measure be used to guide testing in a similar fashion to the cyclomatic complexity.

Table 6.1 suggests there is very little correlation identifiable between μ and program fitness. However, there is a positive correlation between μ and the evaluation time. This is also reflected in the charts in Figures 6.19 and 6.20. The results on the even-n-parity problem are being distorted by extreme values for μ, with some values as high as 4.14×10^{10}, because of the occurrence of if-statement conditions with a high number of boolean predicates. The second of the three rules for calculating μ puts a large emphasis on the contribution of multiple predicates to the complexity of a program. However, an example of these long condition clauses



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.20: Prather × Time. Charts showing the relationship between the μ metric and the time required to evaluate individuals

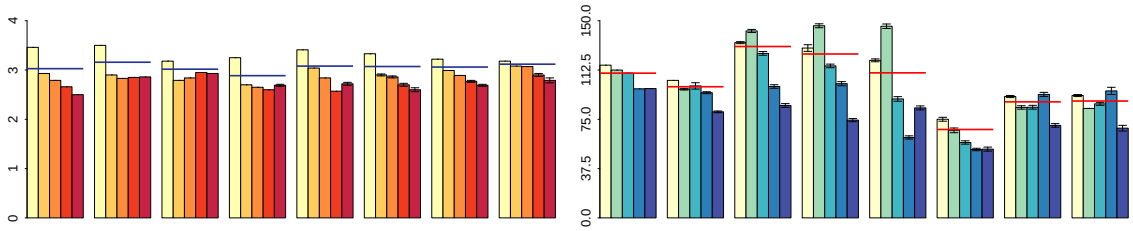
makes it clear that this highlights an unpleasant characteristic of the programs that can be produced with this system:

```
if ((((!((loopVar && resultVar) && resultVar) || !resultVar)
        && (true && ((false || !true) && ((loopVar
        || ((resultVar && false) || (resultVar && false)))
        && !((resultVar && loopVar) && !false)))))) {
    ...
}
```

Experienced human programmers do not write imperative code like this and any programs that relied on clauses like this would be frowned upon in a real software system. Furthermore, it seems that programs like this do not produce solutions,

as Table 6.4 shows that no solutions are found with these high μ values. This raises an argument for size constraints for imperative programs that are based on more sensible properties than program depth, or other encouragements to target less complex programs. Complexity metrics like μ may have some application in this area.

To enable any correlations for μ to be more easily seen, a second pair of charts have been produced for the even-n-parity problem with outliers removed. These charts are shown in Figure 6.21. The fitness charts across the problems are a little inconsistent, but there does seem to be a similar trend on at least three of the problems, with the lowest fitness scores found in the middle groups. The evaluation time charts suggest that higher μ values tend to be combined with a longer fitness evaluation. But, the picture is not completely clear and there are some anomalies. For example, the sort list problem shows a very clear trend through the metric groups, but the 7th metric group is much lower than the trend suggests it would be. Despite its exaggeration of the complexity of boolean predicates, the trends suggest that the Prather measure may have more potential for application than either the cyclomatic complexity or Halstead's effort metric.



(a) Fitness (b) Evaluation time

Figure 6.21: Charts for the even-n-parity problem, showing the relationship between Prather's measure μ metric and both the fitness and time required to evaluate individuals, where outliers are removed

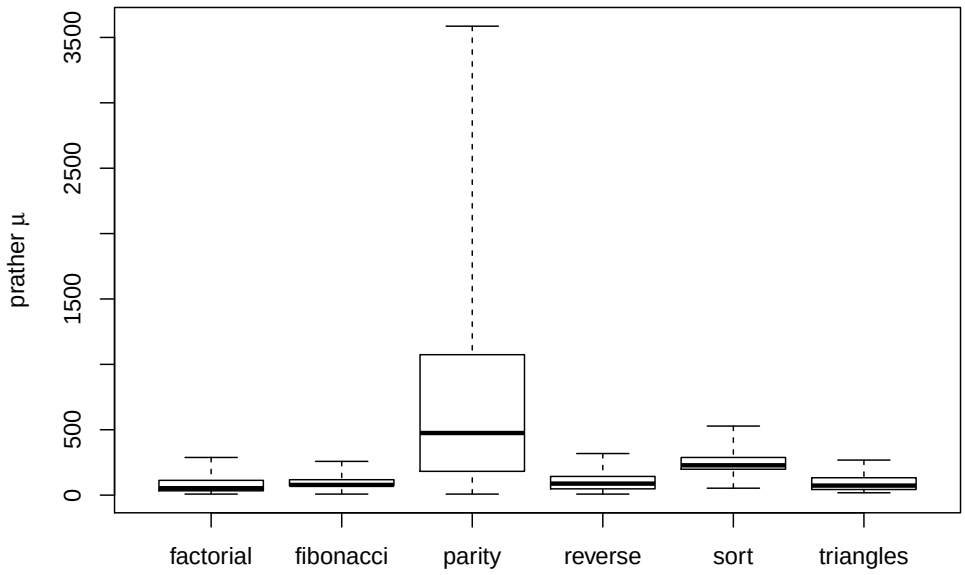
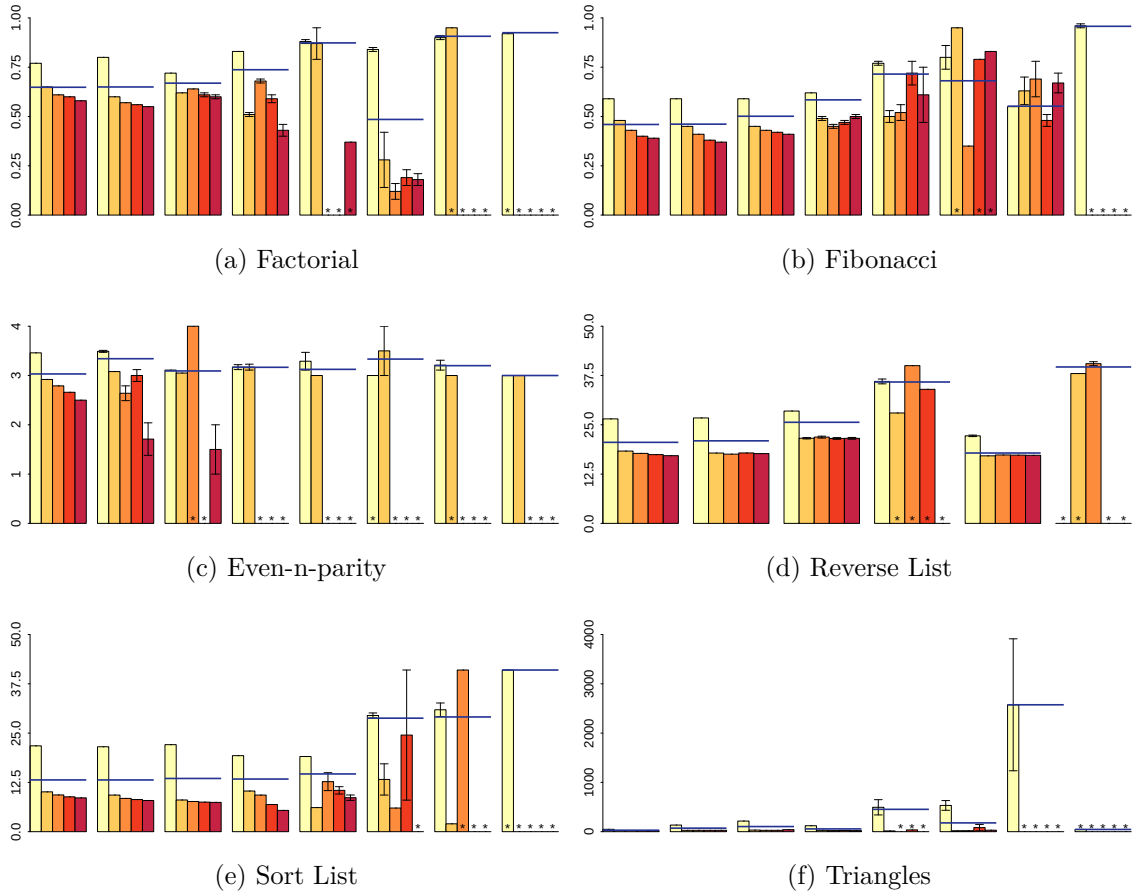


Figure 6.22: Boxplot showing the distribution of μ values for programs that solved all the training cases on each of the test problems. As explained in the text, the prather metric produces some extreme values for μ on the even-n-parity problem, so on this problem only, outliers have been removed.

6.4.8 NPATH Complexity

NPATH [112] was proposed by Nejmeh as a finite measure of the potentially infinite number of execution paths through a program, for the purpose of test coverage. The author claims NPATH overcomes several shortcomings with McCabe's metric, namely that nesting levels are not considered, that different constructs are not distinguished between and that a poor relationship exists between the cyclomatic complexity and the required testing effort of a program.

As with the cyclomatic complexity, the NPATH metric is based on the control flow graph of a program. It is defined as a count of the number of acyclic execution paths through a function. To apply NPATH in practice, Nejmeh provided a series of *execution path expressions* for common high-level programming constructs. These can be easily applied to produce a complexity score from the source code (or syntax tree) of any program. Table 6.2 lists a summary of some of the relevant execution path expressions, where $NP(x)$ is an application of NPATH on the component x. As an example, the NPATH complexity of an if statement is a sum of the NPATH complexities of its constituent parts (a boolean expression and



(a) Factorial (b) Fibonacci

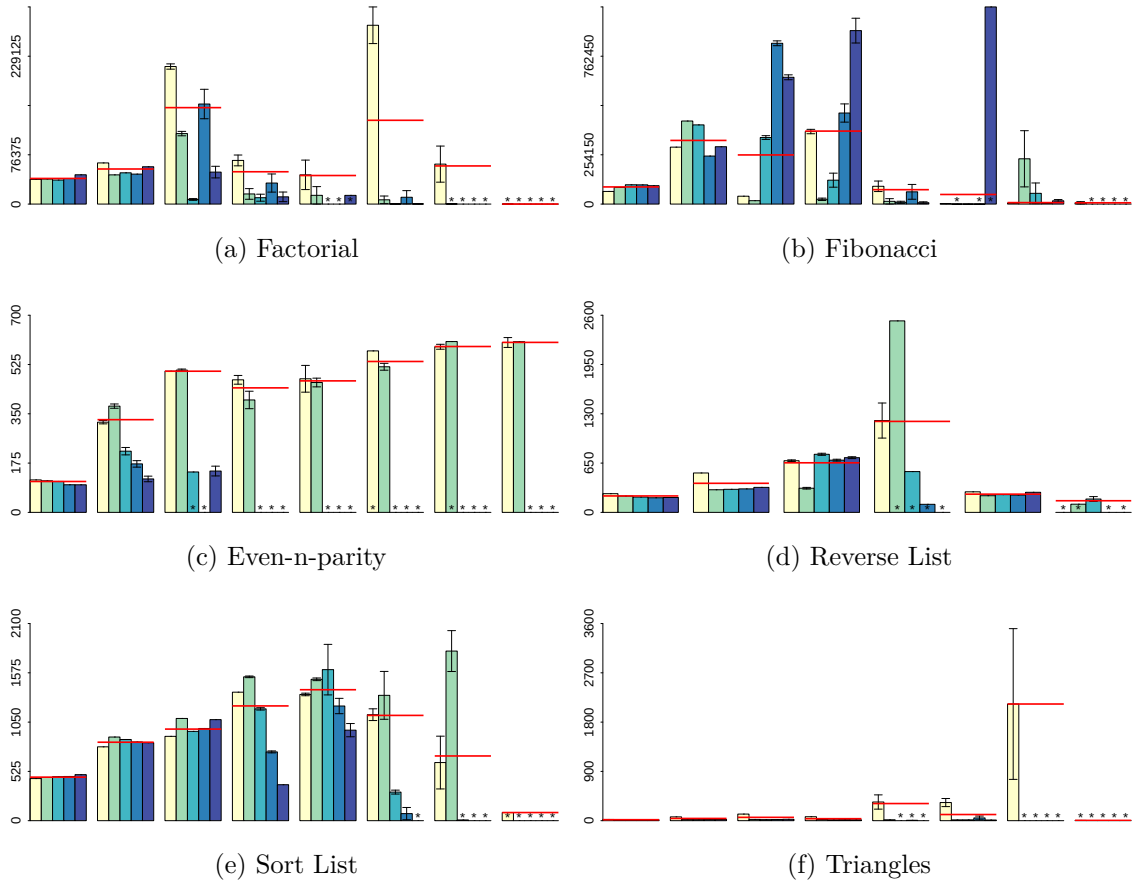
(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.23: NPATH × Fitness. Charts showing the relationship between the NPATH metric and the fitness of the individuals

a conditionally executed sequential code block), plus 1. Further explanation and examples are given in [112]. Based on practical studies, the author recommended an NPATH threshold of 200 for a function and suggested methods for reducing complexity below this level.

Nejmeh performed a comparative study of NPATH against McCabe's cyclomatic complexity, number of lines of code and the number of lexical tokens, where each was computed for 821 different UNIX C functions. A strong correlation was reported between cyclomatic complexity and the two lexical measures, which it was suggested leads to the conclusion that they are measuring the same thing. But NPATH had little correlation, so it was concluded that NPATH is measuring different factors of complexity to these other metrics. Our study seems to



(a) Factorial (b) Fibonacci

(c) Even-n-parity (d) Reverse List

(e) Sort List (f) Triangles

Figure 6.24: NPATH × Time. Charts showing the relationship between the NPATH metric and the time required to evaluate individuals

confirm these findings, with little similarity between the fitness and evaluation time charts in Figures 6.23 and 6.24 and the related charts for the cyclomatic complexity, number of statements and program length metrics.

Table 6.1 shows that there is very little correlation between the NPATH metric and program fitness on these problems. Examining the charts in Figures 6.23 and 6.24 highlights one reason for this. Many of the NPATH groupings are empty or have only very few individuals in them, because the metric does not produce sufficiently distinct values to distinguish individuals. For example, on the reverse-list problem, all programs were assigned one of only six different NPATH scores. This lack of diversity is not a desirable quality, but would likely be less of a

Table 6.2: Summary of NPATH execution path expressions, where $NP(x)$ is an application of NPATH on the component x.

`if`	$NP(<if\text{-}range>) + NP(<expr>) + 1$
`if-else`	$NP(<if\text{-}range>)+NP(<else\text{-}range>)+NP(<expr>)$
`while`	$NP(<while\text{-}range>) + NP(<expr>) + 1$
`for`	$NP(<for\text{-}range>) + NP(<expr1>) +$ $NP(<expr2>) + NP(<expr3>) + 1$
`return`	1
`sequential`	1
Expressions	Number of `AND` and `OR` operators in expression
Function call	1
Function	$\prod_{i=1}^{i=N} NP(\texttt{Statement}_i)$

problem with larger programs. There is a weak positive correlation between the NPATH metric and the time taken to evaluate an individual.

6.4.9 Summary of Analysis

This section gives a quick summary of the relationships discovered between each metric and the fitness and evaluation time properties of programs.

Program Length

- Weak negative correlation between length and fitness
- Positive correlation with evaluation time

Program Depth

- Negative correlation with fitness, so deeper programs have better fitness
- Positive exponential correlation with evaluation time

Number of Statements

- Very close similarity to program length and cyclomatic complexity metrics

Cyclomatic Complexity

- Very close similarity to program length and number of statements metrics

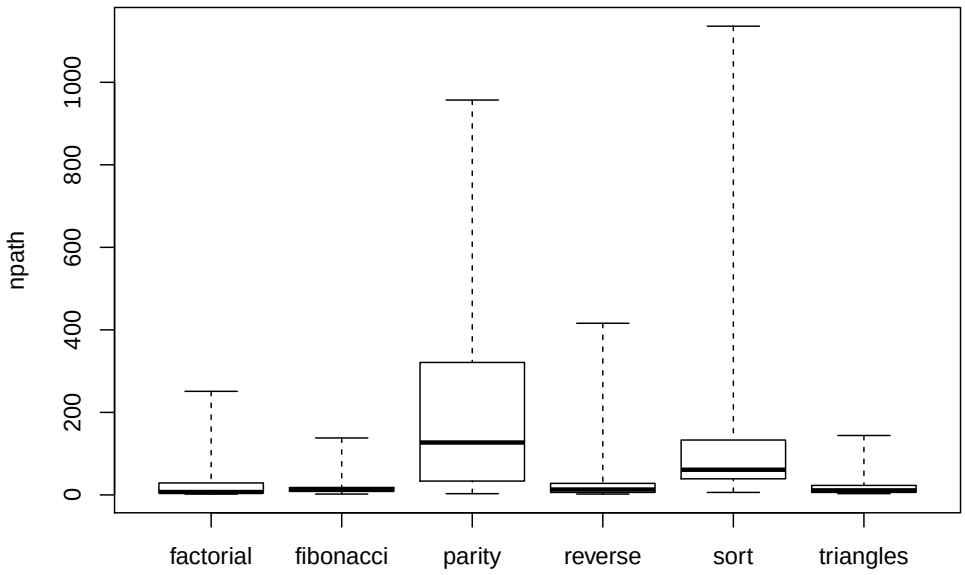


Figure 6.25: Boxplot showing the distribution of NPATH metric values for programs that solved all the training cases on each of the test problems

Halstead's Effort

– Problem dependent - no consistency across problems

Prather's μ

– Some consistency on factorial, Fibonacci and reverse-list problems

– A weak positive correlation between μ and evaluation time

– Exaggeration of the contribution of nested predicates to complexity

NPATH

– Problem dependent - little consistency across problems

– Insufficient variety in metric values, which makes it difficult to distinguish programs

– Weak positive correlation with evaluation time

6.5 Comparing the Metrics

It has already been mentioned that some of the metrics appear to have a high correlation with each other. This was noticeable from comparisons of their fitness

Table 6.3: Correlation between software metrics, as calculated over all programs using Spearman's rank correlation. The p-value in all cases is $< 2.2 \times 10^{-16}$

	Length	*Depth*	*Statements*	*Cyclomatic*	*Effort*	*Prather*	*NPATH*
Length	-	0.5553	0.9373	0.8940	0.7918	0.8961	0.8855
Depth	0.5553	-	0.5054	0.4995	0.5103	0.5196	0.4961
Statements	0.9373	0.5054	-	0.9743	0.6992	0.9703	0.9568
Cyclomatic	0.8940	0.4995	0.9743	-	0.7266	0.9857	0.9662
Effort	0.7918	0.5103	0.6992	0.7266	-	0.7273	0.7118
Prather	0.8961	0.5196	0.9703	0.9857	0.7273	-	0.9690
NPATH	0.8855	0.4961	0.9568	0.9662	0.7118	0.9690	-

and evaluation time charts. However, Table 6.3 shows the statistical correlation between the metrics over all programs that were found on all problems. The correlations were calculated using Spearman's rank correlation coefficient. There is some degree of correlation between all of the metrics. This should be expected since they are all influenced to some degree by program size. The cyclomatic complexity is often criticised for its high correlation with program size, but interestingly both Prather's μ measure and NPATH have comparable correlations with program size, as measured by either program length or the number of statements. This is a little surprising as both μ and NPATH take account of the degree of nesting and it was expected that this would lead to a more useful metric. It seems to be the case that the use of a fixed code-block size of 3 results in a close association between the degree of nesting and the program size, because each if-statement or loop will always add at least an additional 3 statements. If a range of different smaller code-block sizes are used, it may help to break this association and reduce the correlation between the metrics that consider nesting and the program size.

Although the charts for each metric show the trends in fitness values, the metric values associated with low fitness do not necessarily correspond to the areas where solutions are identified. Table 6.4 illustrates this point. It shows the number of programs in each metric group that solve all training cases, per 10,000 individuals. For example, the fitness charts for the even-n-parity problem show that the point of lowest fitness is with a higher number of statements. Yet, Table 6.4 suggests that a higher density of solutions are found with a lower number of statements. In general, there is no clear picture that more solutions are found

at higher complexities or larger program sizes. This is significant as it raises the possibility of focusing the search efforts to areas of lower complexity which may be cheaper to evaluate.

Also important is whether the solutions that are found are able to generalise. Table 6.5 lists the proportion of the solutions that solved all the training cases which also solved all the test cases. It may be expected that smaller or less complex programs would be more likely to generalise, but these results suggest that this may not always be the case on all problems. Whilst the sort list problem does seem to show the expected trend using any of the metrics, other problems had little or no trend, or in some cases the opposite of what was expected. On the even-n-parity problem, a far higher proportion of solutions generalised if they had a cyclomatic complexity score in group 6 than a lower score in groups 1, 2 or 3. However, the values in this table should be used with caution. Error values are omitted to avoid cluttering an already saturated table, but all the values in the table should be considered to be suspicious at best, as the sample size in most cases is smaller than would be desirable. The number of generalising solutions is severely limited by the number of runs that are performed, so it would be preferable for data to be gathered from 1000s of runs rather than just 500. Although of course this would come at great computational expense.

The authors of many code metrics make suggestions about what a reasonable value for their metric is, when applied to a set unit of code, such as a sub-routine. The advice is generally that code which exceeds this threshold should be considered overly complex and should ideally be refactored. Human programmers are taught to follow 'good practice', such as naming variables appropriately, breaking up complex expressions and using standard idioms where possible [79, chapter 1]. This helps to produce clear code which is easier to understand and maintain. But the GP algorithm is unaware of any such guidelines and as a result the programs it produces are unlikely to conform to the code metric authors' expectations of good code. Table 6.6 shows the mean metric value of all the programs on each problem. This table shows that the even-n-parity has the highest average metric

Table 6.4: Number of individuals in each group of metric values that solve all training cases per 10,000 individuals

Metric	*Problem*	1	2	3	4	5	6	7	8
Length	Factorial	1.82	3.63	7.75	0.84	0.51	2.17	-	-
	Fibonacci	1.12	3.03	4.04	4.03	1.29	0.65	-	-
	Parity	24.63	43.27	30.41	20.17	14.82	4.27	0.37	-
	Reverse	1.29	1.20	0.87	1.02	0.83	0.76	-	-
	Sort	-	1.34	1.85	4.46	2.58	2.35	13.51	-
	Triangles	0.41	0.61	0.55	0.38	0.51	-	-	-
Depth	Factorial	-	-	-	1.72	0.95	8.21	2.16	3.99
	Fibonacci	-	-	-	1.68	1.91	1.77	10.95	2.93
	Parity	-	-	-	-	-	53.56	43.70	24.23
	Reverse	-	-	-	-	0.27	4.89	0.41	0.97
	Sort	-	-	-	-	-	-	4.92	2.96
	Triangles	-	-	-	-	-	0.79	0.75	0.55
Statements	Factorial	0.95	5.11	8.07	1.41	0.54	0.82	-	-
	Fibonacci	0.56	3.00	5.45	1.84	0.08	-	-	-
	Parity	26.21	45.36	26.76	20.20	13.88	7.20	0.47	-
	Reverse	1.31	1.21	0.81	1.06	1.03	0.71	-	-
	Sort	-	1.20	1.86	4.59	2.80	2.18	10.84	-
	Triangles	0.29	0.63	0.57	0.43	0.54	-	-	-
Cyclomatic	Factorial	0.95	5.11	8.07	1.41	0.54	0.82	-	-
	Fibonacci	0.56	3.00	5.45	1.84	0.08	-	-	-
	Parity	20.56	39.28	28.43	17.91	6.35	1.63	-	-
	Reverse	1.31	1.21	0.81	1.06	1.03	0.71	-	-
	Sort	-	1.20	1.86	4.59	2.80	2.18	10.84	-
	Triangles	0.29	0.63	0.57	0.43	0.54	-	-	-
Effort	Factorial	4.21	3.52	-	-	-	-	-	-
	Fibonacci	0.98	3.39	3.78	3.67	-	-	-	-
	Parity	42.15	25.09	15.32	9.01	2.85	0.44	-	-
	Reverse	1.40	1.10	0.78	-	-	3.05	-	-
	Sort	1.41	3.07	3.49	0.88	-	-	-	-
	Triangles	0.50	0.53	0.52	1.65	3.12	3.23	-	-
Prather μ	Factorial	3.02	3.83	8.98	0.52	0.42	-	-	-
	Fibonacci	0.73	4.99	3.59	1.61	0.22	-	-	-
	Parity	24.45	-	-	-	-	-		
	Reverse	1.53	0.96	0.98	0.73	0.80	-	-	-
	Sort	-	1.54	2.71	4.53	1.65	4.55	3.09	-
	Triangles	0.41	0.62	0.57	0.58	0.08	-	-	-
NPATH	Factorial	3.92	3.14	2.02	-	-	-	-	-
	Fibonacci	3.01	10.56	-	-	-	-	-	-
	Parity	24.47	-	3.08	-	-	-	-	-
	Reverse	1.04	0.61	1.23	-	-	-		
	Sort	3.45	1.77	3.03	1.27	20.35	-	-	-
	Triangles	0.55	0.08	-	-	-	-	-	-

Table 6.5: Proportion of programs which solve all training cases that also solve all test cases, in each group of metric values

Metric	*Problem*	1	2	3	4	5	6	7	8
Length	Factorial	32.88	3.82	1.13	19.51	0.00	0.00	-	-
	Fibonacci	77.38	3.22	0.77	1.01	0.00	0.00	-	-
	Parity	2.19	3.86	3.38	7.77	9.87	17.19	100.00	-
	Reverse	11.11	81.89	77.35	36.72	80.00	100.00	-	-
	Sort	-	55.43	43.77	14.26	19.19	18.56	3.17	-
	Triangles	100.00	100.00	100.00	100.00	100.00	-	-	-
Depth	Factorial	-	-	-	100.00	91.30	8.82	17.90	3.23
	Fibonacci	-	-	-	97.06	34.88	24.04	1.07	1.79
	Parity	-	-	-	-	-	3.45	3.49	5.57
	Reverse	-	-	-	-	100.00	25.32	88.89	72.76
	Sort	-	-	-	-	-	-	50.00	22.07
	Triangles	-	-	-	-	-	100.00	100.00	100.00
Statements	Factorial	37.47	2.81	1.05	22.58	0.00	0.00	-	-
	Fibonacci	74.00	2.90	0.60	2.92	0.00	-	-	-
	Parity	2.17	3.58	3.45	8.49	9.34	8.05	100.00	-
	Reverse	30.38	81.35	75.33	39.22	80.56	100.00	-	-
	Sort	-	58.59	42.74	15.42	16.88	19.77	2.82	-
	Triangles	100.00	100.00	100.00	100.00	100.00	-	-	-
Cyclomatic	Factorial	37.47	2.81	1.05	22.58	0.00	0.00	-	-
	Fibonacci	74.00	2.90	0.60	2.92	0.00	-	-	-
	Parity	4.39	3.70	4.49	7.97	20.83	41.18	-	-
	Reverse	30.38	81.35	75.33	39.22	80.56	100.00	-	-
	Sort	-	58.59	42.74	15.42	16.88	19.77	2.82	-
	Triangles	100.00	100.00	100.00	100.00	100.00	-	-	-
Effort	Factorial	7.45	2.56	-	-	-	-	-	-
	Fibonacci	74.00	2.90	0.60	2.92	0.00	-	-	-
	Parity	3.51	5.02	9.07	16.34	25.71	100.00	-	-
	Reverse	34.85	74.04	62.20	-	-	100.00	-	-
	Sort	52.87	21.46	13.73	100.00	-	-	-	-
	Triangles	100.00	100.00	100.00	100.00	100.00	100.00	-	-
Prather μ	Factorial	10.38	3.07	1.45	11.76	0.00	-	-	-
	Fibonacci	39.47	1.15	1.33	0.00	0.00	-	-	-
	Parity	5.49	-	-	-	-	-		
	Reverse	52.94	85.93	53.17	71.43	75.00	-	-	-
	Sort	-	62.72	26.56	13.06	25.95	8.47	0.00	-
	Triangles	100.00	100.00	100.00	100.00	100.00	-	-	-
NPATH	Factorial	5.73	0.00	12.50	-	-	-	-	-
	Fibonacci	4.04	0.00	-	-	-	-	-	-
	Parity	5.48	-	100.00	-	-	-	-	-
	Reverse	66.81	95.24	100.00	-	-	-		
	Sort	22.14	26.15	14.29	25.00	4.17	-	-	-
	Triangles	100.00	100.00	-	-	-	-	-	-

value for all of the complexity metrics (but not the size metrics). Part of the reason for this is that the programs are larger, with only the sort-list problem having higher values for the size metrics, but more significant is the impact of the deeply nested logical expressions that have already been described. For the metrics that have recommended maximum values (cyclomatic complexity, Prather's measure and NPATH), the mean metric values on most problems is actually below this threshold.

Table 6.6: Mean software metric value for all programs on each problem with the standard deviation. The standard deviation for Prather μ on the even-n-parity problem was 4.77×10^9.

Problem (Target)	*Factorial*	*Fibonacci*	*Parity*	*Reverse*	*Sort*	*Triangles*
Length	113.1±66.4	109.6±50.2	192.1±76.3	163.0±74.3	271.6±95.5	101.4±51.3
Depth	9.6±1.1	9.6±1.0	9.9±0.6	9.8±0.8	10.0±0.3	9.4±1.2
Statements	16.1±11.3	21.8±11.1	46.5±19.0	27.6±13.4	49.4±17.5	23.3±12.4
Cyclomatic (<10)	5.4±3.8	7.3±3.7	32.5±13.8	9.2±4.5	16.5±5.8	7.8±4.1
Effort	3468.0±1934.6	3794.0±1678.7	27730.0±20173.0	3145.0±1354.0	9089.0±3694.0	1233±616.4
Prather μ (<100)	55.4±55.0	81.1±56.8	8.2×10^6	111.3±67.4	231.4±93.6	85.1±63.0
NPATH (<200)	13.1±26.5	19.4±30.2	533.4±1791.7	32.8±56.8	139.7±151.0	20.0±31.9

6.6 Conclusions

The aim of this analysis was to identify metrics that could be used to focus the GP search in a way that would improve fitness or reduce evaluation time. However, none of the set of complexity metrics that have been studied here seem to be very well suited for these applications. It is a disappointing conclusion, but each of the metrics suffer from flaws that make them unsuitable. In particular, they have high correlation with program size and show little consistency across problems. This is at least in part caused by the underlying fact that these metrics were designed for human written code. As a result they are distorted by introns and as seen with the Prather metric, can be overwhelmed by deeply nested expressions which are typical in evolved code. This highlights a need for a metric that is targeted specifically at GP programs. Such a metric could ignore introns and take account of the typical traits of GP evolved programs.

There are other possible applications for complexity metrics that can be addressed in future work. It has already been mentioned in this chapter, that program depth is not a very intuitive size constraint to set for imperative programs. Number of statements is a more appropriate measure of program size and this could be used in place of a maximum depth parameter. However, it would probably be necessary to combine this with a constraint on the nesting of loops, because nested loops can cause an exponential increase in evaluation time. Alternatively, a single complexity metric that incorporates nesting information, such as the Prather μ measure or NPATH, could be used. The maximum setting for such a parameter could be set based on recommended maximum values for that metric.

Another possible application of software metrics is to improve the readability of solutions by reducing complexity. A simple method to encourage less complex solutions would be to incorporate the complexity into the fitness function. This approach has previously been used with size metrics to successfully reduce the size of programs and tackle bloat [36, 41]. However, these studies report that this can result in program size being minimised at the expense of fitness, so must be combined with measures to increase population diversity. A related application is the refactoring of existing programs to increase readability. Previous studies have sought to evolve programs that are functionally equivalent to an existing solution while improving some property. For example, Ryan and Ivan [140] evolved parallel versions of existing sequential programs. Complexity metrics could be used to guide such automated refactorings to produce programs of lower complexity.

6.7 Summary

Software metrics were proposed as a method for improving the fitness and evaluation time of programs in GP. A detailed analysis was performed of seven software metrics as applied to high-level imperative programs evolved with GP and how these metrics correlated with fitness and evaluation time. The metrics analysed included cyclomatic complexity, number of statements and NPATH. Our results

confirmed previous findings that many complexity metrics correlate highly with program size and there was also high correlation with each other, suggesting that they are measuring similar properties. Little consistency was seen in the trends between the complexity metrics and the fitness. It was concluded that the complexity metrics used in this study do not have qualities that would make them suitable for applications to improve fitness or evaluation time in GP. Other applications suggested for future work included replacing maximum depth parameters with a complexity based constraint and using complexity metrics to increase the readability of solutions without changing their fitness.

Chapter 7

Conclusions

The research presented in this thesis has sought to tackle the problems associated with evolving high-level imperative programs using genetic programming with a tree representation. Historically, the use of GP algorithms has focused on producing functional code that more closely resembles mathematical formulae than the more typical well structured imperative code produced by modern human programmers. In the course of this thesis, previous work related to the task of evolving high-level imperative programs has been explored and some novel techniques have been presented to further this goal.

There are complex structural rules that high-level imperative programs must abide by. For example, code blocks must be made up of a sequential list of statements and many programming constructs operate on specific types of input. This makes them difficult to evolve because of the random nature of an evolutionary algorithm in constructing the programs. In this thesis, an original approach to adding structural constraints to a tree representation was presented. These constraints were shown to be sufficient to impose an imperative structure made up of code-blocks and statements that represent common program language constructs. Some standard programming constructs have more complex requirements. The use of iteration in GP has been researched in some detail, but is still largely avoided because of the complexities involved with using them productively and avoiding infinite loops. The mechanism for adding structural constraints was shown to

also be adept at supporting several forms of loop, which make use of variables to supply context information, such as the current iteration index. It was found that the use of a tightly constrained high-level imperative structure along with sensible program constructs was able to easily produce solutions to benchmark problems such as even-n-parity, which have previously been considered to be difficult for GP.

Very little previous work has supported the evolution of variable declarations in the programs produced by a genetic programming system. However, variable declarations are a fundamental component of imperative computer programs. Allowing the set of available variables to be modified through the evolution removes some of the burden of foresight required and allows loop constructs to supply their own variables, rather than relying on variables supplied as inputs. It was demonstrated that with some small modifications, a dynamic syntax could be supported in GP, which allows constructs to declare and make use of new limited-scope variables. Results from experimental comparisons suggest that the use of loops with variable declarations can also have performance benefits.

Software metrics are widely used in software engineering to guide the software development process. However, they have not been explicitly applied to GP algorithms to any great extent. In this thesis, we considered whether software metrics may have some useful applications to the evolution of high-level imperative programs. Possible applications included guiding the search towards areas of the fitness landscape with less complex solutions. The imperative programs generated with our GP system were analysed with a range of seven software metrics designed to measure program size or complexity. The results from this analysis confirmed previous findings from the software metrics literature that some popular complexity metrics are of little value because they are highly correlated with simpler size metrics. It was hoped that some of the metrics would have clear and consistent correlations with fitness and the time required to evaluate programs. However, this was not the case and any trends were unclear or varied widely across the problems.

7.1 Contributions

Chapters 4, 5 and 6 describe the original research of this thesis. In this section, the contributions this research makes to extend the genetic programming literature will be summarised.

Montana's Strongly Typed Genetic Programming system provides a mechanism for constraining data-types within program trees that are evolved. In chapter 4, Strongly Formed Genetic Programming was introduced to extend STGP to add structural constraints. This was achieved by requiring all non-terminals to additionally specify a node-type for each of their inputs that would restrict which nodes could be attached as a child to the non-terminal. These structural constraints are powerful enough to satisfy our main motivation of evolving high-level imperative programs, but also provide a general mechanism to impose structural constraints on program trees for other purposes.

It was demonstrated that the structural constraints of the SFGP system can be used to enforce a high-level imperative structure on program trees and to model common imperative programming constructs, including various forms of loop. An interesting trait of this approach is that the program trees that are generated can be directly converted to equivalent source code using the syntax of any imperative programming language that can support the constructs used. This is possible because each program tree only represents the syntactic structure of the program not concrete syntax; it is an abstract syntax tree.

The work presented in chapter 5 makes two main contributions. Firstly, some modifications to the SFGP system were presented that allow a dynamic syntax to be supported. A dynamic syntax allows the available syntax to be modified for a position in the tree, by the nodes that preceed it in the tree. This could have many applications, but the initial motivation in this thesis was to support contructs that can declare limited-scope variables. Node-types that model variable declaration statements and loops that supply their own index/element variables were defined.

The results of testing these new declarative constructs showed that on some problems they could significantly improve evolutionary performance. Potentially of more importance, however, is that programs make use of loop constructs which more closely resemble those found in common programming languages, which generally involve variable declarations. Also, the burden of knowing which auxilary variables to supply in addition to the inputs is removed.

The second contribution of chapter 5 is a simple enhancement to the fitness evaluation procedure which in certain situations can bring a substantial boost to success rates. The basic idea of this technique, which we call Multi-Variable Return (MVR), is to consider each possible variable in a program as if it was the return variable, rather than just a single variable. It was shown that where multiple variables of the correct data-type were available, the MVR technique leads to a statistically significant reduction in the required computational effort to find a correct solution. This same idea can be extended to apply to any scenario where one of multiple possible values is normally designated as the return value, for example, Linear GP representations that designate one memory register as a program's result.

A detailed analysis of the high-level imperative programs evolved with SFGP using software metrics is presented in chapter 6. Seven software metrics are used which are designed to measure program size or complexity. The analysis attempts to identify correlations between the metrics and both the fitness of individuals and the time required to evaluate them. It also looks for trends in how these correlations change through the generations of a run and what areas of the search-space, as measured by these metrics, that correct solutions are found in. A study of this kind has not been conducted previously. The particular metrics studied were found to be unsuitable for the intended applications of increasing the evolutionary performance of GP through improvements in fitness or evaluation time. However, it provides a basis on which other research can build to apply software metrics to enhance the GP algorithm in other ways or with other metrics. A number of potential applications were discussed in chapter 6.

7.2 Further Work

The research presented in this thesis takes a few steps in the direction of evolving high-level imperative code that more closely resembles the programs written by human programmers. However, there is further work to be done.

Unlike STGP, SFGP is currently unable to fully support generic functions, where operators are defined to support inputs with different data-types differently. There is no doubt that this is a limitation. With support for generic functions in SFGP, constructs could be implemented to work differently for different sets of inputs. This is not easy to implement in the same way as in STGP, because any lookup tables would be substantially complicated by the additional node-type property. Also, a dynamic syntax, as used to support variable declarations, is incompatible with the idea of lookup tables that are produced from just the initial syntax. One possible solution is to use a combination of lookup tables for the data-type and a backtracking algorithm for the node-type.

One major challenge that should be confronted is to attempt to solve harder problems. In order to tackle more challenging problems, the range of constructs that can be supported will need to be expanded. It should be easy to define node-types that model other common types of bounded loops, but what about unbounded loops? Another key issue that will need to be addressed, to solve harder problems, is the topic of scalability. The size of the search-space increases exponentially with the size of the programs required to solve the problem. A popular answer to the question of scalability is modularisation. Can the structural constraints of SFGP be used to produce modular programs? One way may be to allow multiple sub-routines, where the declaration of a sub-routine is handled in much the same way as the declaration of a variable.

The depth based size constraints that have been used so far with SFGP are unwieldy and unintuitive to set. Also, the unbalanced shape of trees that are constrained within structural bounds means depth constraints have an irregular impact across a program tree. For high-level imperative programs, it would be

more appropriate to use a size constraint based on the number of statements and the degree of statement nesting. One possibility raised in chapter 6 is that complexity metrics could be used to provide more useful constraints. Another possible application of software metrics that could be explored is to improve the readability of solutions by encouraging lower complexity. With such a vast number of software metrics available, the analysis that has been performed in this thesis could be extended by looking at other metrics. However, these metrics were all designed for human written programs, so better results may be obtained by designing metrics specifically for application to GP evolved programs. Such a metric could be based only on executable code, ignoring introns and placing emphasis on key properties, such as the nesting of loops.

Appendix A

Java Code Templates

Section 4.4.3 describes how code templates can be used to convert the program trees generated by SFGP into the source code of a specific programming language. Table A.1 provides a complete listing of the code templates for all node-types for the Java programming language. There are a few additional complexities that must be handled in practice:

- The identifier used for all variables must be selected to be unique. This is an implementation detail, but is important in order to maintain the semantics of the language.
- The first line of the `ForEachLoop` template assumes that arrays of length zero are not possible. Otherwise this assignment will need to be protected from this scenario to avoid a Java exception.
- The templates for the `Loop`, `ForLoop` and `ForLoopDecl` constructs do not include the maximum iteration bound. This can easily be added as a second loop condition, but it would be preferable that a solution would be checked that it is not dependent upon the constraint and so it could be removed.

Table A.1: Complete listing of source code templates for the Java programming language, where ⟨*child-n*⟩ is replaced by the source code for the nth child, ⟨*data-type*⟩ is replaced by the data-type of the node and ⟨*data-type-n*⟩ is replaced by the data-type of the nth child.

SubRoutine	public ⟨*data-type*⟩ methodName() ⟨*child-1*⟩
ReturnBlock	{ ⟨*child-1*⟩ ⟨*child-2*⟩ ⟨*child-(n-1)*⟩ return ⟨*child-n*⟩; }
CodeBlock	{ ⟨*child-1*⟩ ⟨*child-2*⟩ ⟨*child-n*⟩ }
Loop	for(int i=0; i<⟨*child-1*⟩ i++) ⟨*child-2*⟩
ForLoop	⟨*child-1*⟩ = 1; for(int i=1; i<=⟨*child-2*⟩; i++,⟨*child-1*⟩=i) ⟨*child-3*⟩
ForEachLoop	⟨*child-2*⟩ = ⟨*child-1*⟩[0]; for(int i=0; i<⟨*child-1*⟩.length; i++,⟨*child-2*⟩=⟨*child-1*⟩[i]) ⟨*child-3*⟩
ForLoopDecl	int varName = 1; for(int i=1; i<⟨*child-1*⟩; i++,varName=i) ⟨*child-2*⟩
ForEachLoopDecl	for(⟨*data-type-1*⟩ varName: ⟨*child-1*⟩) ⟨*child-2*⟩
IfStatement	if(⟨*child-1*⟩) ⟨*child-2*⟩
Declaration	⟨*data-type-1*⟩ varName = ⟨*child-1*⟩;
Assignment	⟨*child-1*⟩ = ⟨*child-2*⟩;
ElementAssignment	⟨*child-1*⟩[⟨*child-2*⟩] = ⟨*child-3*⟩;
SwapElements	Utilities.swap(⟨*child-1*⟩, ⟨*child-2*⟩);
Add	⟨*child-1*⟩ + ⟨*child-2*⟩
Subtract	⟨*child-1*⟩ - ⟨*child-2*⟩
Multiply	⟨*child-1*⟩ * ⟨*child-2*⟩
Divide	Utilities.divide(⟨*child-1*⟩, ⟨*child-2*⟩)
And	⟨*child-1*⟩ && ⟨*child-2*⟩
Or	⟨*child-1*⟩ || ⟨*child-2*⟩
Not	!⟨*child-1*⟩
GreaterThan	⟨*child-1*⟩ > ⟨*child-2*⟩
LessThan	⟨*child-1*⟩ < ⟨*child-2*⟩
ArrayLength	⟨*child-1*⟩.length
ArrayElement	⟨*child-1*⟩[⟨*child-2*⟩]
Concat	⟨*child-1*⟩ + ⟨*child-2*⟩

Bibliography

[1] Russell J. Abbott. Object-oriented genetic programming, an initial implementation. In *Proceedings of the Sixth International Conference on Computational Intelligence and Natural Computing*, Cary, September 2003.

[2] Russell J. Abbott, Jiang Guo, and Behzad Parviz. Guided genetic programming. In *The 2003 International Conference on Machine Learning; Models, Technologies and Applications (MLMTA 2003)*, Las Vegas, June 2003. CSREA Press.

[3] Alexandros Agapitos and Simon M. Lucas. Evolving efficient recursive sorting algorithms. In Gary G. Yen et al., editor, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006)*, pages 2677–2684, Vancouver, July 2006. IEEE Press.

[4] Alexandros Agapitos and Simon M. Lucas. Learning recursive functions with object oriented genetic programming. In Pierre Collet at al., editor, *Proceedings of the 9th European Conference on Genetic Programming (EuroGP 2006)*, volume 3905 of *LNCS*, pages 166–177, Budapest, April 2006. Springer.

[5] Alexandros Agapitos and Simon M. Lucas. Evolving modular recursive sorting algorithms. In Marc Ebner et al, editor, *Proceedings of the 10th European Conference on Genetic Programming (EuroGP 2007)*, volume 4445 of *LNCS*, pages 301–310, Valencia, April 2007. Springer.

[6] Allan J. Albrecht. Measuring application development productivity. In *Proceedings of IBM Application Development Symposium*, pages 83–92. IBM Press, October 1979.

[7] Peter J. Angeline. Subtree crossover: Building block engine or macromutation? In John R. Koza et al., editor, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 9–17, Stanford, July 1997. Morgan Kaufmann.

[8] Andrea Arcuri. On the automation of fixing software bugs. In *ICSE Companion 2008: Companion of the 30th International Conference on Software Engineering*, pages 1003–1006, Leipzig, 2008. ACM Press.

[9] Andrea Arcuri and Xin Yao. Coevolving programs and unit tests from their specification. In *IEEE International Conference on Automated Software Engineering*, Atlanta, November 2007. IEEE Press.

[10] Andrea Arcuri and Xin Yao. A novel co-evolutionary approach to automatic software bug fixing. In Jun Wang et al., editor, *Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC 2008)*, pages 162–168, Hong Kong, June 2008. IEEE Press.

[11] Wolfgang Banzhaf, Simon Harding, William B. Langdon, and Garnett Wilson. Accelerating genetic programming through graphics processing units. In Rick L. Riolo at al., editor, *Genetic Programming Theory and Practice VI*, Genetic and Evolutionary Computation, chapter 15, pages 229–249. Springer, May 2008.

[12] Lawrence Beadle and Colin G. Johnson. Semantic analysis of program initialisation in genetic programming. *Genetic Programming and Evolvable Machines*, 10(3):307–337, September 2009.

[13] Franck Binard and Amy Felty. An abstraction-based genetic programming system. In Peter A. N. Bosman, editor, *Late Breaking Papers at the Genetic*

and Evolutionary Computation Conference (GECCO 2007), pages 2415–2422, London, July 2007. ACM Press.

[14] Walter Bohm and Andreas Geyer-Schulz. Exact uniform initialization for genetic programming. In Richard K. Belew and Michael Vose, editors, *Foundations of Genetic Algorithms IV*, pages 379–407, San Diego, August 1996. Morgan Kaufmann.

[15] Michael Bowman, Lionel C. Briand, and Yvan Labiche. Solving the class responsibility assignment problem in object-oriented analysis with multi-objective genetic algorithms. *IEEE Transactions on Software Engineering*, 36(6):817–837, November 2010.

[16] Markus Brameier and Wolfgang Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, February 2001.

[17] Markus Brameier and Wolfgang Banzhaf. *Linear Genetic Programming*. Number XVI in Genetic and Evolutionary Computation. Springer, 2007.

[18] Scott Brave. Evolving recursive programs for tree search. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 10, pages 203–220. MIT Press, Cambridge, MA, USA, 1996.

[19] Colin J. Burgess and Martin Lefley. Can genetic programming improve software effort estimation? A comparative evaluation. *Information and Software Technology*, 43(14):863–873, December 2001.

[20] Edmund K. Burke, Steven Gustafson, Graham Kendall, and Natalio Krasnogor. Is increased diversity in genetic programming beneficial? an analysis of the effects on performance. In Ruhul Sarker et al., editor, *Proceedings of the 2003 Congress on Evolutionary Computation (CEC 2003)*, pages 1398–1405, Canberra, December 2003. IEEE Press.

[21] Jonathan Byrne, James McDermott, Michael O'Neill, and Anthony Brabazon. An analysis of the behaviour of mutation in grammatical evolution. In Anna I. Esparcia-Alcazar et al., editor, *Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010)*, volume 6021 of *LNCS*, Istanbul, April 2010. Springer.

[22] Tom Castle, Lawrence Beadle, and Fernando E. B. Otero. Epochx: genetic programming software for research. http://www.epochx.org, 2007.

[23] Tom Castle and Colin G. Johnson. Positional effect of crossover and mutation in grammatical evolution. In Anna I. Esparcia-Alcazar et al., editor, *Proceedings of the 13th European Conference on Genetic Programming (EuroGP 2010)*, volume 6021 of *LNCS*, Istanbul, April 2010. Springer.

[24] Phil T. Cattani and Colin G. Johnson. ME-CGP: Multi expression cartesian genetic programming. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010)*, Barcelona, July 2010. IEEE Press.

[25] Darren M. Chitty. A data parallel approach to genetic programming using programmable graphics hardware. In Dirk Thierens et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, volume 2, pages 1566–1573, London, July 2007. ACM Press.

[26] Fuey Sian Chong and William B. Langdon. Java based distributed genetic programming on the internet. In Wolfgang Banzhaf et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 163–166, Orlando, July 1999.

[27] Steffen Christensen and Franz Oppacher. An analysis of Koza's computational effort statistic for genetic programming. In James A. Foster et al., editor, *Proceedings of the 5th European Conference on Genetic Programming (EuroGP 2002)*, volume 2278 of *LNCS*, pages 182–191, Kinsale, April 2002. Springer.

[28] Vic Ciesielski and Xiang Li. Analysis of genetic programming runs. In R I Mckay and Sung-Bae Cho, editors, *Proceedings of The Second Asian-Pacific Workshop on Genetic Programming*, Cairns, December 2004.

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

[30] Chris Clack and Tina Yu. Performance enhanced genetic programming. In Peter J. Angeline et al., editor, *Proceedings of the Sixth Conference on Evolutionary Programming*, volume 1213 of *LNCS*, Indianapolis, 1997. Springer.

[31] Robert Cleary and Michael O'Neill. An attribute grammar decoder for the 01 multiconstrained knapsack problem. In Günther R. Raidl and Jens Gottlieb, editors, *Proceedings of the 5th European Conference on Evolutionary Computation in Combinatorial Optimization (EvoCOP 2005)*, volume 3448 of *LNCS*, pages 34–45, Lausanne, March 2005. Springer.

[32] Nichael L. Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, editor, *Proceedings of the International Conference on Genetic Algorithms and the Applications*, pages 183–187, Pittsburgh, July 1985.

[33] Ellery F. Crane and Nicholas F. McPhee. The effects of size and depth limits on tree based genetic programming. In Tina Yu, Rick L. Riolo, and Bill Worzel, editors, *Genetic Programming Theory and Practice III*, volume 9 of *Genetic Programming*, chapter 15, pages 223–240. Springer, May 2005.

[34] Ronald L. Crepeau. Genetic evolution of machine language software. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 121–134, Tahoe City, July 1995.

[35] Bill Curtis, Sylvia B. Sheppard, Phil Milliman, M. A. Borst, and Tom Love. Measuring the psychological complexity of software maintenance tasks with the halstead and mccabe metrics. *IEEE Transactions on Software Engineering*, 5(2):96–104, March 1979.

[36] Edwin D. de Jong, Richard A. Watson, and Jordan B. Pollack. Reducing bloat and promoting diversity using multi-objective methods. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 11–18, San Francisco, July 2001. Morgan Kaufmann.

[37] Marina de la Cruz, Alfonso Ortega, and Manuel Alfonseca. Attribute grammar evolution. In José Mira and José R. Álvarez, editors, *Artificial Intelligence and Knowledge Engineering Applications: A Bioinspired Approach, Part II*, volume 3562 of *LNCS*, pages 182–191, Las Palmas, June 2005. Springer.

[38] Tom DeMarco. *Controlling Software Projects: Management, Measurement and Estimates*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1986.

[39] Jose J. Dolado and Luis Fernandez. Genetic programming, neural networks and linear regression in software project estimation. In C. Hawkins et al., editor, *International Conference on Software Process Improvement, Research, Education and Training*, pages 157–171, London, September 1998. British Computer Society.

[40] Gilles Dowek. Imperactive core. In *Principles of Programming Languages*, Undergraduate Topics in Computer Science, pages 1–17. Springer, London, 2009. 10.1007/978-1-84882-032-6_1.

[41] Anikó Ekárt and Sandor Z. Németh. Selection based on the pareto nondomination criterion for controlling code growth in genetic programming. *Genetic Programming and Evolvable Machines*, 2(1):61–73, March 2001.

[42] Maria Claudia F. P. Emer and Silvia R. Vergilio. GPTesT: A testing tool based on genetic programming. In W. B. Langdon et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1343–1350, New York, July 2002. Morgan Kaufmann.

[43] Maria Claudia F. P. Emer and Silvia R. Vergilio. Selection and evaluation of test data based on genetic programming. *Software Quality Journal*, 11(2):167–186, June 2003.

[44] Michael Evangelist. Software complexity metric sensitivity to program structuring rules. *Journal of Systems and Software*, 3(3):231–243, September 1983.

[45] Matthew Evett, Taghi Khoshgoftaar, Pei der Chien, and Ed Allen. Using genetic programming to determine software quality. In *Proceedings of the Twelfth International FLAIRS Conference*, pages 113–117. AAAI, 1999.

[46] Norman E. Fenton. *Software Metrics: A Rigorous Approach*. Chapman & Hall, London, 1991.

[47] Jenny R. Finkel. Using genetic programming to evolve an algorithm for factoring numbers. In John R. Koza, editor, *Genetic Algorithms and Genetic Programming*, pages 52–60. Stanford Bookstore, Stanford, December 2003.

[48] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. A genetic programming approach to automated software repair. In Guenther Raidl et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2009)*, pages 947–954, Montreal, July 2009. ACM Press.

[49] Yasao Funami and Maurice H. Halstead. A software physics analysis of Akiyama's debugging data. In *Proceedings of the Symposium on Computer Software Engineering*, pages 133–138, 1976.

[50] John E. Gaffney. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, 10(4):459–464, July 1984.

[51] Chris Gathercole and Peter Ross. Tackling the boolean even-n-parity problem with genetic programming and limited-error fitness. In John R. Koza et al., editor, *Proceedings of the Second Annual Conference on Genetic Programming*, pages 119–127, Stanford, July 1997. Morgan Kaufmann.

[52] Tom Gilb. *Software Metrics*. Winthrop Publishers, 1976.

[53] Geoffrey K. Gill and Chris F. Kemerer. Cyclomatic complexity density and software maintenance productivity. *IEEE Transactions on Software Engineering*, 17(12):1284–1288, December 1991.

[54] D E Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[55] Huisheng Gong and Monika Schmidt. A complexity measure based on selection and nesting. *SIGMETRICS Performance Evaluation Review*, 13(1):14–19, June 1985.

[56] Neill Graham. *Introduction to Pascal*. West Publishing Co., St. Paul, MN, USA, 3rd edition, 1988.

[57] Steven Gustafson. *An Analysis of Diversity in Genetic Programming*. PhD thesis, School of Computer Science and Information Technology, University of Nottingham, Nottingham, February 2004.

[58] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, October 2005.

[59] Maurice H. Halstead. *Elements of Software Science*. Operating and Programming Systems Series. Elsevier Science Inc., New York, 1977.

[60] Peter G. Hamer and Gillian D. Frewin. M.H. Halstead's software science - a critical examination. In *Proceedings of the 6th International Conference on Software Engineering*, pages 197–206, Los Alamitos, 1982. IEEE Press.

[61] Wilfred J. Hansen. Measurement of program complexity by the pair: (cyclomatic number, operator count). *SIGPLAN Notices*, 13(3):29–33, March 1978.

[62] Simon Harding and Wolfgang Banzhaf. Fast genetic programming on GPUs. In Marc Ebner et al., editor, *Proceedings of the 10th European Conference on Genetic Programming*, volume 4445 of *LNCS*, pages 90–101, Valencia, April 2007. Springer.

[63] Simon Harding, Julian Miller, and Wolfgang Banzhaf. Self modifying cartesian genetic programming: Fibonacci, squares, regression and summing. In Leonardo Vanneschi et al., editor, *Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009)*, volume 5481 of *LNCS*, pages 133–144, Tübingen, April 2009. Springer.

[64] Simon Harding, Julian Miller, and Wolfgang Banzhaf. Self modifying cartesian genetic programming: Parity. In Andy Tyrrell, editor, *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2009)*, pages 285–292, Trondheim, May 2009. IEEE Press.

[65] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. Search based software engineering: A comprehensive analysis and review of trends, techniques and applications. Technical report, King's College London, 2009.

[66] Christopher Harris. Enforcing hierarchy on solutions with strongly typed genetic programming. In John R. Koza, editor, *Late Breaking Papers at the 1997 Genetic Programming Conference*, Stanford, July 1997. Stanford Bookstore.

[67] Brad Harvey, James A. Foster, and Deborah Frincke. Byte code genetic programming. In John R. Koza, editor, *Late Breaking Papers at the 1998 Genetic Programming Conference*, Wisconsin, July 1998. Stanford University Bookstore.

[68] Thomas D. Haynes, Dale A. Schoenefeld, and Roger L. Wainwright. Type inheritance in strongly typed genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 18, pages 359–376. MIT Press, Cambridge, MA, USA, 1996.

[69] John H. Holland. *Adaptation in natural and artificial systems.* MIT Press, Cambridge, MA, USA, 1975.

[70] Lorenz Huelsbergen. Toward simulated evolution of machine language iteration. In John R. Koza et al., editor, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 315–320, Stanford, July 1996. MIT Press.

[71] Lorenz Huelsbergen. Learning recursive sequences via evolution of machine-language programs. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 186–194, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[72] Hitoshi Iba. Random tree generation for genetic programming. In Hans-Michael Voigt et al., editor, *Parallel Problem Solving from Nature IV, Proceedings of the International Conference on Evolutionary Computation*, volume 1141 of *LNCS*, pages 144–153, Berlin, September 1996. Springer.

[73] Adam C. Jensen and Betty H. C. Cheng. On the use of genetic programming for automated refactoring and the introduction of design patterns. In Juergen Branke et al., editor, *Proceedings of the Genetic and Evolutionary*

Computation Conference (GECCO 2010), pages 1341–1348, Portland, July 2010. ACM Press.

[74] Yaochu Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9(1):3–12, October 2005.

[75] Colin G. Johnson. Genetic programming crossover: Does it cross over? In Leonardo Vanneschi et al., editor, *Proceedings of the 12th European Conference on Genetic Programming (EuroGP 2009)*, volume 5481 of *LNCS*, pages 97–108, Tübingen, April 2009. Springer.

[76] Capers Jones. Software metrics: Good, bad and missing. *Computer*, 27(9):98–100, 1994.

[77] Thaddeus C. Jones. Measuring programming quality and productivity. *IBM Systems Journal*, 17(1):39–63, March 1978.

[78] Claire J. Kennedy. *Strongly Typed Evolutionary Programming*. PhD thesis, Computer Science, University of Bristol, Bristol, December 1999.

[79] Brian W. Kernighan and Rob Pike. *The practice of programming*. Addison-Wesley Longman, Boston, MA, USA, 1999.

[80] Taghi M. Khoshgoftaar and Yi Liu. A multi-objective software quality classification model using genetic programming. *IEEE Transactions on Reliability*, 56(2):237–245, June 2007.

[81] Taghi M. Khoshgoftaar, Yi Liu, and Naeem Seliya. Genetic programming based decision trees for software quality classification. In *Proceedings of the Fifteenth International Conference on Tools with Artificial Intelligence*, pages 374–383, Los Alamitos, November 2003. IEEE Computer Society Press.

[82] Kenneth E. Kinnear, Jr. Evolving a sort: Lessons in genetic programming. In *Proceedings of the 1993 International Conference on Neural Networks*, volume 2, pages 881–888, San Francisco, March 1993. IEEE Press.

[83] Kenneth E. Kinnear, Jr. Generality and difficulty in genetic programming: Evolving a sort. In Stephanie Forrest, editor, *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 287–294, Urbana-Champaign, July 1993. Morgan Kaufmann.

[84] Evan Kirshenbaum. Genetic programming with statically scoped local variables. In Darrell Whitley et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2000)*, pages 459–468, Las Vegas, July 2000. Morgan Kaufmann.

[85] John R. Koza. Hierarchical genetic algorithms operating on populations of computer programs. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 768–774, Detroit, August 1989. Morgan Kaufmann.

[86] John R. Koza. Genetic evolution and co-evolution of computer programs. In Christopher Taylor Charles Langton, J. Doyne Farmer, and Steen Rasmussen, editors, *Artificial Life II*, volume X of *SFI Studies in the Sciences of Complexity*, pages 603–629. Addison-Wesley, New Mexico, February 1991.

[87] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[88] Willian B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20, pages 395–414. MIT Press, Cambridge, MA, USA, 1996.

[89] Willian B. Langdon. The evolution of size in variable length representations. In *Proceedings of the 1998 IEEE International Conference on Evolutionary Computation*, pages 633–638, Anchorage, May 1998. IEEE Press.

[90] Willian B. Langdon. Size fair and homologous tree genetic programming crossovers. In Wolfgang Banzhaf et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, volume 2, pages 1092–1097, Orlando, July 1999. Morgan Kaufmann.

[91] Willian B. Langdon and Riccardo Poli. Fitness causes bloat. In P. K. Chawdhry, R. Roy, and R. K. Pant, editors, *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer, June 1997.

[92] Vladimir I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966.

[93] A. M. Lister. Software science — the emperor's new clothes? *Australian Computer Journal*, 14(2):66–70, May 1982.

[94] Yi Liu and Taghi M. Khoshgoftaar. Genetic programming model for software quality classification. In *Sixth IEEE International Symposium on High Assurance Systems Engineering*, pages 127–136, Boco Raton, October 2001. IEEE Press.

[95] Mark Lorenz and Jeff Kidd. Object-oriented software metrics. *Journal of Systems and Software*, 44(2):147–154, 1994.

[96] Sean Luke and Liviu Panait. A survey and comparison of tree generation algorithms. In Lee Spector et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001)*, pages 81–88, San Francisco, July 2001. Morgan Kaufmann.

[97] Sean Luke and Liviu Panait. A comparison of bloat control methods for genetic programming. *Evolutionary Computation*, 14(3):309–344, Fall 2006.

[98] Sean Luke and Lee Spector. A revised comparison of crossover and mutation in genetic programming. In John R. Koza et al., editor, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 208–213, Wisconsin, July 1998. Morgan Kaufmann.

[99] Eduard Lukschandl, Henrik Borgvall, Lars Nohle, Mats Nordahl, and Peter Nordin. Distributed java bytecode genetic programming. In Riccardo Poli et al., editor, *Proceedings of the 3rd European Conference on Genetic Programming (EuroGP 2000)*, volume 1802 of *LNCS*, pages 316–325, Edinburgh, April 2000. Springer.

[100] Penousal Machado and Amilcar Cardoso. Speeding up genetic programming. In *Proceedings of the Second International Symposium on Artificial Intelligence, Adaptive Systems*, Havana, March 1999.

[101] Hammad Majeed and Conor Ryan. A less destructive, context-aware crossover operator for GP. In Pierre Collet et al., editor, *Proceedings of the 9th European Conference on Genetic Programming (EuroGP 2006)*, volume 3905 of *LNCS*, pages 36–48, Budapest, Hungary, April 2006. Springer.

[102] Sidney R. Maxwell III. Experiments with a coroutine execution model for genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 413–417, Orlando, June 1994. IEEE Press.

[103] Thomas J. McCabe. A complexity measure. In *Proceedings of the 2nd International Conference on Software Engineering*, Los Alamitos, 1976. IEEE Computer Society Press.

[104] Daniel McGaughran and Mengjie Zhang. Evolving more representative programs with genetic programming. *International Journal of Software Engineering and Knowledge Engineering*, 19(1):1–22, 2009.

[105] Robert I. McKay, Nguyen Xuan Hoai, Peter A. Whigham, Yin Shan, and Michael O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3/4):365–396, September 2010. Tenth Anniversary Issue: Progress in Genetic Programming and Evolvable Machines.

[106] Nicholas F. McPhee, Alex Jarvis, and Ellery F. Crane. On the strength of size limits in linear genetic programming. In Kalyanmoy Deb et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, volume 3103 of *LNCS*, pages 593–604, Seattle, June 2004. Springer.

[107] Nicholas F. McPhee, Brian Ohs, and Tyler Hutchison. Semantic building blocks in genetic programming. In Michael O'Neill et al., editor, *Proceedings of the 11th European Conference on Genetic Programming (EuroGP 2008)*, volume 4971 of *LNCS*, pages 134–145, Naples, March 2008. Springer.

[108] Meine J. P. van der Meulen and Miguel A. Revilla. Correlations between internal software metrics and software dependability in a large population of small c/c++ programs. In *Proceedings of the The 18th IEEE International Symposium on Software Reliability*, pages 203–208, Washington, 2007. IEEE Computer Society Press.

[109] David J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[110] Alberto Moraglio, Fernando E. B. Otero, Colin G. Johnson, Simon Thompson, and Alex A. Freitas. Evolving recursive programs using non-recursive scaffolding. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2012)*. IEEE Press, June 2012.

[111] Glenford J. Myers. An extension to the cyclomatic measure of program complexity. *SIGPLAN Notices*, 12:61–64, October 1977.

[112] Brian A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188–200, February 1988.

[113] Peter Nordin. A compiling genetic programming system that directly manipulates the machine code. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 14, pages 311–331. MIT Press, 1994.

[114] Peter Nordin and Wolfgang Banzhaf. Evolving Turing-complete programs for a register machine with self-modifying code. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 318–325, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[115] Peter Nordin, Wolfgang Banzhaf, and Frank D. Francone. Efficient evolution of machine code for CISC architectures using instruction blocks and homologous crossover. In Lee Spector et al., editor, *Advances in Genetic Programming 3*, chapter 12, pages 275–299. MIT Press, Cambridge, MA, USA, June 1999.

[116] Peter Nordin, Frank D. Francone, and Wolfgang Banzhaf. Explicitly defined introns and destructive crossover in genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 6–22, Tahoe City, July 1995.

[117] Mihai Oltean and Crina Grosan. Evolving evolutionary algorithms using multi expression programming. In Wolfgang Banzhaf, Thomas Christaller, Peter Dittrich, Jan T. Kim, and Jens Ziegler, editors, *Advances in Artificial Life. 7th European Conference on Artificial Life*, number 2801 in Lecture Notes in Artificial Intelligence, pages 651–658, Dortmund, September 2003. Springer.

[118] Mihai Oltean and Crina Grosan. A comparison of several linear genetic programming techniques. *Complex Systems*, 14(4):282–311, 2004.

[119] Michael O'Neill and Conor Ryan. Automatic generation of caching algorithms. In Kaisa Miettinen, Marko M. Mäkelä, Pekka Neittaanmäki, and Jacques Periaux, editors, *Evolutionary Algorithms in Engineering and Computer Science*, pages 127–134, Jyväskylä, Finland, 30 May - 3 June 1999. John Wiley & Sons.

[120] Michael O'Neill and Conor Ryan. Evolving multi-line compilable C programs. In Riccardo Poli et al., editor, *Proceedings of the 2nd European Conference on Genetic Programming (EuroGP 1999)*, volume 1598 of *LNCS*, pages 83–92, Göteborg, May 1999. Springer.

[121] Michael O'Neill and Conor Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, August 2001.

[122] Michael O'Neill and Conor Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

[123] Una-May O'Reilly and Franz Oppacher. An experimental perspective on genetic programming. In R. Manner and B. Manderick, editors, *Parallel Problem Solving from Nature 2*, pages 331–340, Brussels, September 1992. Elsevier Science.

[124] Una-May O'Reilly and Franz Oppacher. A comparative analysis of GP. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 2, pages 23–44. MIT Press, Cambridge, MA, USA, 1996.

[125] Alfonso Ortega, Marina de la Cruz, and Manuel Alfonseca. Christiansen grammar evolution: Grammatical evolution with semantics. *IEEE Transactions on Evolutionary Computation*, 11(1):77–90, February 2007.

[126] Fernando E. B. Otero, Tom Castle, and Colin G. Johnson. Epochx: Genetic programming in Java with statistics and event monitoring. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion (GECCO 2012)*, Philadelphia, July 2012. ACM Press.

[127] Mouloud Oussaidene, Bastien Chopard, Olivier V. Pictet, and Marco Tomassini. Parallel genetic programming: An application to trading models evolution. In John R. Koza et al., editor, *Genetic Programming 1996:*

Proceedings of the First Annual Conference, pages 357–380, Stanford, July 1996. MIT Press.

[128] Enrique I. Oviedo. Control flow, data flow and program complexity. In Martin Shepperd, editor, *Software Engineering Metrics I*, pages 52–65. McGraw-Hill, Inc., New York, US, 1993.

[129] Tim Perkis. Stack-based genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 148–153, Orlando, June 1994. IEEE Press.

[130] Riccardo Poli, William B. Langdon, and Nicholas F. McPhee. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[131] Ronald E. Prather. An axiomatic theory of software complexity measure. *The Computer Journal*, 27(4):340–347, 1984.

[132] Jose C. Ribeiro, Mario Zenha-Rela, and Francisco Fernandez de Vega. An evolutionary approach for performing structural unit-testing on third-party object-oriented Java software. In Natalio Krasnogor et al., editor, *Proceedings of the International Workshop on Nature Inspired Cooperative Strategies for Optimization*, volume 129 of *Studies in Computational Intelligence*, pages 379–388, Acireale, November 2007. Springer.

[133] Jose C. Ribeiro, Mario Alberto Zenha-Rela, and Francisco Fernandez de Vega. A strategy for evaluating feasible and unfeasible test cases for the evolutionary testing of object-oriented software. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, pages 85–92, Leipzig, 2008. ACM Press.

[134] Denis Robilliard, Virginie Marion-Poty, and Cyril Fonlupt. Genetic programming on graphics processing units. *Genetic Programming and Evolvable Machines*, 10(4):447–471, December 2009.

[135] Justinian Rosca. Generality versus size in genetic programming. In John R. Koza et al., editor, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 381–387, Stanford, July 1996. MIT Press.

[136] Brian J. Ross. Logic-based genetic programming with definite clause translation grammars. *New Generation Computing*, 19(4):313–337, 2001.

[137] Franz Rothlauf and David E. Goldberg. Tree network design with genetic algorithms - an investigation in the locality of the pruefernumber encoding. In Scott Brave and Annie S. Wu, editors, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 238–244, Orlando, July 1999.

[138] Franz Rothlauf and David E. Goldberg. Pruefer numbers and genetic algorithms: A lesson on how the low locality of an encoding can harm the performance of gas. In *Proceedings of the 6th International Conference on Parallel Problem Solving from Nature*, pages 395–404, London, 2000. Springer.

[139] Franz Rothlauf and Marie Oetzel. On the locality of grammatical evolution. In Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 320–330, Budapest, Hungary, 10 - 12 April 2006. Springer.

[140] Conor Ryan. *Automatic Re-engineering of Software Using Genetic Programming*. Kluwer Academic Publishers, November 1999.

[141] Arjan Seesing and Hans-Gerhard Gross. A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134, 2006.

[142] Yin Shan, Robert I. McKay, Chris J. Lokan, and Daryl L. Essam. Software project effort estimation using genetic programming. In *Proceedings*

of the International Conference on Communications Circuits and Systems, volume 2, pages 1108–1112, Canberra, June 2002.

[143] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, March 1988.

[144] Shinichi Shirakawa and Tomoharu Nagao. Evolution of sorting algorithm using graph structured program evolution. In *Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics*, pages 1256–1261, Montreal, October 2007. IEEE Press.

[145] Shinichi Shirakawa, Shintaro Ogino, and Tomoharu Nagao. Graph structured program evolution. In Dirk Thierens et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, volume 2, pages 1686–1693, London, July 2007. ACM Press.

[146] Christopher L. Simons and Ian C. Parmee. Single and multi-objective genetic operators in object-oriented conceptual software design. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, pages 1957–1958, New York, 2006. ACM Press.

[147] Terence Soule, James A. Foster, and John Dickinson. Code growth in genetic programming. In John R. Koza et al., editor, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 215–223, Stanford, July 1996. MIT Press.

[148] Lee Spector, Jon Klein, and Maarten Keijzer. The push3 execution stack and the evolution of control. In Hans-Georg Beyer et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, volume 2, pages 1689–1696, Washington DC, June 2005. ACM Press.

[149] Lee Spector and Alan Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, March 2002.

[150] Thomas Sterling. Beowulf-class clustered computing: Harnessing the power of parallelism in a pile of PCs. In John R. Koza et al., editor, *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Wisconsin, July 1998. Morgan Kaufmann.

[151] Sun Microsystems Inc. *Code Conventions for the Java Programming Language*, 1997.

[152] Astro Teller and David Andre. Automatically choosing the number of fitness cases: The rational allocation of trials. In John R. Koza, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max Garzon, Hitoshi Iba, and Rick L. Riolo, editors, *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 321–328, Stanford University, CA, USA, 13-16 July 1997. Morgan Kaufmann.

[153] Astro Teller and Manuela Veloso. Program evolution for data mining. *The International Journal of Expert Systems*, 8(3):216–236, 1995.

[154] M. David Terrio and M. I. Heywood. Directing crossover for reduction of bloat in GP. In W. Kinsner, A. Seback, and K. Ferens, editors, *IEEE CCECE 2003: IEEE Canadian Conference on Electrical and Computer Engineering*, pages 1111–1115. IEEE Press, May 2002.

[155] Thomas A. Thayer, Myron Lipow, and Eldred C. Nelson. *Software reliability: a study of large project reality.* North-Holland, 1978.

[156] TIOBE Software BV. TIOBE programming community index. http://www.tiobe.com/tiobe_index, March 2012.

[157] Douglas A. Troy and Stuart H. Zweben. Measuring the quality of structured designs. In Martin Shepperd, editor, *Software engineering metrics I*, pages 214–226. McGraw-Hill, Inc., New York, NY, USA, 1993.

[158] Nguyen Q. Uy, Nguyen X. Hoai, Michael O'Neill, and Robert I. McKay. The role of syntactic and semantic locality of crossover in genetic programming.

In Robert Schaefer, Carlos Cotta, Joanna Kolodziej, and Guenter Rudolph, editors, *PPSN 2010 11th International Conference on Parallel Problem Solving From Nature*, volume 6239 of *Lecture Notes in Computer Science*, pages 533–542, Krakow, Poland, 11-15 September 2010. Springer.

[159] Matthew Walker, Howard Edwards, and Chris Messom. Confidence intervals for computational effort comparisons. In Marc Ebner et al., editor, *Proceedings of the 10th European Conference on Genetic Programming (EuroGP 2007)*, volume 4445 of *LNCS*, pages 23–32, Valencia, April 2007. Springer.

[160] Mingxu Wan, Thomas Weise, and Ke Tang. Novel loop structures and the evolution of mathematical algorithms. In Sara Silva et al., editor, *Proceedings of the 14th European Conference on Genetic Programming (EuroGP 2011)*, volume 6621 of *LNCS*, pages 49–60, Turin, April 2011. Springer.

[161] Shixian Wang, Yuehui Chen, and Peng Wu. Function sequence genetic programming. In De-Shuang Huang, Kang-Hyun Jo, Hong-Hee Lee, Hee-Jun Kang, and Vitoantonio Bevilacqua, editors, *Proceedings of the 5th International Conference on Intelligent Computing*, volume 5755 of *LNCS*, pages 984–992, Ulsan, September 2009. Springer.

[162] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using a hybrid evolutionary algorithm. In Gary G. Yen et al., editor, *Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC 2006)*, pages 3193–3200, Vancouver, July 2006. IEEE Press.

[163] Stefan Wappler and Joachim Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In Maarten Keijzer et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2006)*, volume 2, pages 1925–1932, Seattle, July 2006. ACM Press.

[164] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In Stephen

Fickas, editor, *Proceedings of the 31st International Conference on Software Engineering*, pages 364–374, Vancouver, May 2009.

[165] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Softw. Eng.*, 14:1357–1365, September 1988.

[166] Peter A. Whigham. Grammatically-based genetic programming. In Justinian P. Rosca, editor, *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, pages 33–41, Tahoe City, July 1995.

[167] Peter A. Whigham and Robert I. McKay. Genetic approaches to learning recursive relations. In Xin Yao, editor, *Progress in Evolutionary Computation*, volume 956 of *Lecture Notes in Artificial Intelligence*, pages 17–27. Springer, 1995.

[168] David R. White, John Clark, Jeremy Jacob, and Simon M. Poulding. Searching for resource-efficient programs: low-power pseudorandom number generators. In Maarten Keijzer et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2008)*, pages 1775–1782, Atlanta, July 2008. ACM Press.

[169] Gayan Wijesinghe and Vic Ciesielski. Evolving programs with parameters and loops. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC 2010)*, Barcelona, July 2010. IEEE Press.

[170] Garnett Wilson and Malcolm Heywood. Learning recursive programs with cooperative coevolution of genetic code mapping and genotype. In Dirk Thierens et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, volume 1, pages 1053–1061, London, July 2007. ACM Press.

[171] Mark S. Withall, C. J. Hinde, and R. G. Stone. An improved representation for evolving programs. *Genetic Programming and Evolvable Machines*, 10(1):37–70, March 2009.

[172] Man L. Wong and Kwong S. Leung. Combining genetic programming and inductive logic programming using logic grammars. In *Proceedings of the 1995 IEEE Conference on Evolutionary Computation*, volume 2, pages 733–736, Perth, November 1995. IEEE Press.

[173] Man L. Wong and Kwong S. Leung. Learning programs in different paradigms using genetic programming. In *Proceedings of the Fourth Congress of the Italian Association for Artificial Intelligence*. Springer, 1995.

[174] Man L. Wong and Kwong S. Leung. Evolving recursive functions for the even-parity problem using genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 11, pages 221–240. MIT Press, Cambridge, MA, USA, 1996.

[175] W. Wulf and Mary Shaw. Global variable considered harmful. *SIGPLAN Notices*, 8(2):28–34, February 1973.

[176] B. H. Yin and J. W. Winchester. The establishment and use of measures to evaluate the quality of software designs. In *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues*, pages 45–52, New York, 1978. ACM Press.

[177] Tina Yu and Chris Clack. PolyGP: A polymorphic genetic programming system in haskell. In John Koza, editor, *Late Breaking Papers at the GP-97 Conference*, pages 264–273, Stanford, CA, USA, 13-16 July 1997. Stanford Bookstore.

[178] Jianjun Zhao. On assessing the complexity of software architectures. In *Proceedings of the Third International Workshop on Software Architecture*, pages 163–166, New York, 1998. ACM Press.

127 107 142 80 9 110 143 139

150 154 22 51 160 160 70 44 73 101

125 134 127 55 27 4 105 99 117

142 160 123 172 142 149 41 44