A Study of Genetic Programming and Grammatical Evolution for Automatic Object-Oriented Programming

by

Kevin Chizoba Igwe

Submitted in fulfilment of the academic
requirements for the degree of
Master of Science in the
School of Mathematics, Statistics, and Computer Science,
University of KwaZulu-Natal,
Pietermaritzburg

April 2016

As the candidate's supervisor I have/have not approved this thesis/dissertation for submission

Signed: _____

Name: Prof. Nelishia Pillay

Date: _____

PREFACE

The experimental work described in this dissertation was carried out in the School of Mathematics, Statistics, and Computer Science, University of KwaZulu-Natal, Pietermaritzburg, from February 2014 to April 2016, under the supervision of Professor Nelishia Pillay.

These studies represent original work by the author and have not otherwise been submitted in any form for any degree or diploma to any tertiary institution. Where use has been made of the work of others it is duly acknowledged in the text.

_____ _____

Supervisor: Prof. Nelishia Pillay *Candidate*: Kevin Chizoba Igwe

DECLARATION 1: PLAGIARISM

I, Kevin Chizoba Igwe (student number: 212553209) declare that:

1. The research reported in this dissertation, except where otherwise indicated or acknowledged, is my original work.
2. This dissertation has not been submitted in full or in part for any degree or examination to any other university.
3. this dissertation does not contain other persons' data, pictures, graphs or other information, unless specifically acknowledged as being sourced from other persons;
4. This dissertation does not contain other persons' writing, unless specifically acknowledged as being sourced from other researchers. Where other written sources have been quoted, then:
    a. their words have been re-written but the general information attributed to them has been referenced
    b. where their exact words have been used, their writing has been placed inside quotation marks, and referenced
5. This dissertation does not contain text, graphics or tables copied and pasted from the internet, unless specifically acknowledged, and the source being detailed in the dissertation and in the References sections.

Signed: _____

Candidate: Kevin Chizoba Igwe

Date: _____

DECLARATION 2: PUBLICATIONS

DETAILS OF CONTRIBUTION TO PUBLICATIONS that form part and/or include research presented in this thesis

Publication 1:

Igwe, K., Pillay, N.: A Comparative Study of Genetic Programming and Grammatical Evolution for Evolving Data Structures. In: Proceedings of the 2014 Pattern Recognition Association of South Africa, pp. 115 – 121, Cape Town, South Africa (2014).

Publication 2:

Igwe, K., Pillay, N.: A Study of Genetic Programming and Grammatical Evolution for Automatic Object Oriented Programming: A Focus on the List Data Structure. In: Proceedings of the 7th World Conference on Nature and Biologically Inspired Computing (NaBIC 2015), pp. 151 – 163, Pietermaritzburg, South Africa (2015).

_____ _____

Supervisor: Prof. Nelishia Pillay *Candidate*: Kevin Chizoba Igwe

ABSTRACT

Manual programming is time consuming and challenging for a complex problem. For efficiency of the manual programming process, human programmers adopt the object-oriented approach to programming. Yet, manual programming is still a tedious task. Recently, interest in automatic software production has grown rapidly due to global software demands and technological advancements. This study forms part of a larger initiative on automatic programming to aid manual programming in order to meet these demands.

In artificial intelligence, Genetic Programming (GP) is an evolutionary algorithm which searches a program space for a solution program. A program generated by GP is executed to yield a solution to the problem at hand. Grammatical Evolution (GE) is a variation of genetic programming. GE adopts a genotype-phenotype distinction and maps from a genotypic space to a phenotypic (program) space to produce a program. Whereas the previous work on object-oriented programming and GP has involved taking an analogy from object-oriented programming to improve the scalability of genetic programming, this dissertation aims at evaluating GP and a variation thereof, namely, GE, for automatic object-oriented programming. The first objective is to implement and test the abilities of GP to automatically generate code for object-oriented programming problems. The second objective is to implement and test the abilities of GE to automatically generate code for object-oriented programming problems. The third objective is to compare the performance of GP and GE for automatic object-oriented programming.

Object-Oriented Genetic Programming (OOGP), a variation of OOGP, namely, Greedy OOGP (GOOGP), and GE approaches to automatic object-oriented programming were implemented. The approaches were tested to produce code for three object-oriented programming problems. Each of the object-oriented programming problems involves two classes, one with the driver program and the Abstract Data Type (ADT) class. The results show that both GP and GE can be used for automatic object-oriented programming. However, it was found that the ability of each of the approaches to automatically generate code for object-oriented programming problems decreases with an increase in the problem complexity. The performance of the approaches were compared and statistically tested to determine the effectiveness of each approach. The results show that GE performs better than GOOGP and OOGP.

ACKNOWLEDGMENTS

The author gratefully acknowledges the financial support of the National Research Foundation (NRF). The NRF shall not be held responsible or be given attributes for opinions expressed and conclusions arrived at. Those are of the author.

My gratitude goes to almighty God, my rock and my strength, who has sustained me till this very moment.

I would like to thank my supervisor, Professor Nelishia Pillay for her guidance on every step I took to produce this research work. A million thanks to my family and friends who have encouraged and supported me. Special thanks to my brother Mr D.O Igwe for providing a shelter for me during this research period. Also, to Ms Chiedza Mlingwa for proofreading this dissertation and to Ms Jane C. Nnam for her words of encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

CHAPTER 1: INTRODUCTION

1.1 Purpose of the Study

Manual programming is time consuming and challenging for a complex problem. In recent times, interest in automatic software production has grown rapidly due to global software demands and technological advancements. Human programmers work tirelessly to meet the global market software demands. For the efficiency of the manual programming process, programmers adopt the object-oriented approach to programming. Yet, it can take days, if not months, for a human programmer to write a program.

Automatic programming has been proposed to aid the manual programming process. The term automatic programming changes over time [1, 2]. In this study, automatic programming is a method for producing code that could have been written by a programmer.

Genetic programming (GP) is an evolutionary algorithm which searches a program space to generate a program. A program generated by GP is executed to produce a solution to the problem at hand. GP has been applied to many problem domains. It has been primarily used to solve optimization problems and has been successful in finding optimal or near optimal solutions to the problems. Grammatical Evolution (GE) is a variation of GP which adopts a genotype-phenotype distinction. A grammar is defined for GE and the genotypic space is mapped to the production rules of the grammar to produce a program.

Previous work on automatic programming using genetic programming has focused on the procedural programming paradigm [3] and not on the object-oriented paradigm. One of the advantages of the object-oriented paradigm is code reuse. Previous work on object-oriented genetic programming has involved taking an analogy from object-oriented programming to improve the scalability of genetic programming. Despite the successful application of genetic programming to many problem domains, automatic object-oriented genetic programming has not been fully explored. Also, the use of GE for automatic programming has not been previously investigated. This dissertation hypothesizes that both GP and GE can be used for automatic object-oriented programming. As such, this study of using GP and GE for automatic programming is the first step towards a long term goal of automated software production.

1.2 Objectives

The primary aim of this dissertation is to evaluate the use of genetic programming and a variation thereof, namely, grammatical evolution, for automatic object-oriented programming. This primary aim does not mean that software will be produced for industrial use, instead, it means that genetic programming and grammatical evolution will be investigated to determine their abilities in producing code for classes and programs that use the classes. To realize the primary aim defined above, this dissertation will conduct a survey of the related literature on genetic programming, grammatical evolution and automatic object-oriented programing. The objectives of the research presented in this dissertation are as follows:

- Objective 1: Evaluate genetic programming for automatic object-oriented programming.

  Genetic programming will be developed and evaluated to produce code for classes and programs that use the produced classes.

- Objective 2: Evaluate grammatical evolution for automatic object-oriented programming.

  Grammatical evolution will be developed and evaluated to produce code for classes and programs that use the produced classes.

- Objective 3: Compare the performance of GP and GE for automatic object-oriented programming.

1.3 Contributions

This dissertation makes the following contributions:

- This dissertation provides a thorough survey of automatic Object-Oriented Genetic Programming (OOGP). It also extends and improves the previous work by introducing Greedy Object-Oriented Genetic Programming (GOOGP). It was found that GOOGP is able to increase the success rate of OOGP.
- This is the first study investigating the use of GE for automatic object-oriented programming.

- The performance of genetic programming and grammatical evolution were compared for automatic object-oriented programming. It was found that grammatical evolution scales better than genetic programming when evolving code for classes.

1.4 Dissertation Layout

The rest of the dissertation is structured as below:

Chapter 2: Genetic Programming

The chapter provides a thorough survey of genetic programing. Each step in the GP algorithm is described and analyzed. Some advanced features of genetic programming are also discussed.

Chapter 3: Grammatical Evolution

Chapter 3 provides detailed descriptions of grammatical evolution, and discusses important aspects such as the genotype representation and the mapping process.

Chapter 4: GP and Automatic Object-Oriented Programming

The concept of automatic programming and the aspects of automatic programming relevant to this dissertation are discussed in the chapter. The related studies on object-oriented genetic programming and grammatical evolution are also discussed and analyzed.

Chapter 5: Methodology

Chapter 5 discusses the methodology, the experimental setup and the test data used to evaluate the developed approaches.

Chapter 6: Genetic Programming Approach for Automatic Object-Oriented Programming

The Object-Oriented Genetic Programming (OOGP) approach for automatic programming is presented in the chapter. A greedy OOGP approach, a variation of OOGP which uses a greedy method to generate the individuals in the initial population is also presented. The representation used, the methods of initial population generation, and the operators used for regeneration are discussed.

Chapter 7: Grammatical Evolution Approach for Automatic Object-Oriented Programming

The Object-Oriented Grammatical Evolution (OOGE) approach for automatic programming is presented in the chapter. Each process in OOGE is described.

Chapter 8: Fitness Evaluation and Parameters for the Stack ADT and Problem1

The chapter presents the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the stack Abstract Data Type (ADT). It also presents the functions for fitness evaluation and parameters for a programming problem, namely, Problem1 that uses the stack ADT to determine whether or not a word, a phrase or a sentence is a palindrome.

Chapter 9: Fitness Evaluation and Parameters for the Queue ADT and Problem2

The chapter presents the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the queue Abstract Data Type (ADT). It also presents the functions for fitness evaluation and parameters for a programming problem, namely, Problem2 that uses the queue ADT to perform a breadth-first traversal of any given parse tree.

Chapter 10: Fitness Evaluation and Parameters for the List ADT and Problem3

The chapter presents the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the list Abstract Data Type (ADT). It also presents the functions for fitness evaluation and parameters for a programming problem, namely, Problem3 that populates an empty list with integers and sorts the list.

Chapter 11: Result and Discussion

The chapter presents and discusses the results of applying the approaches to generate code for classes and programs that use the generated classes. The performance of the OOGP, GOOGP and OOGE approaches are compared.

Chapter 12: Conclusion and Future work

Finally, chapter 12 presents the findings and the summary of how each objective was achieved. Possible extensions of the work presented in this dissertation are outlined.

CHAPTER 2: GENETIC PROGRAMMING

2.1 Introduction

This chapter presents a detailed description of genetic programming and an analysis of each step in the genetic programming algorithm.

Section 2.2 provides an introduction to the genetic programming algorithm. Various control models that can be used by genetic programming are detailed in section 2.3 while section 2.4 presents the program representation in genetic programming. A program in genetic programming is expressed in terms of the elements of the function and terminal set. Function and terminal sets are described in section 2.5 while section 2.6 presents sufficiency and closure properties. Section 2.7 describes different ways genetic programming generates the individuals in the initial population. Each individual in the population must be evaluated to measure how good the individual is at solving the problem at hand; evaluation is described in section 2.8. Regeneration is a process of creating new individuals. This requires selecting individuals from the current population and applying genetic operators to them. Selection methods and genetic operators are described in section 2.9 and 2.10 respectively. The genetic programming algorithm is executed until a termination criterion is met; this is discussed in section 2.11. There have been advancements in genetic programming that aim at improving the performance of the approach; this is discussed in section 2.12. Bloat is discussed in section 2.13. Section 2.14 discusses the requirements for setting up a genetic programming system while Section 2.15 discusses genetic programming strengths and weaknesses. Section 2.16 presents the chapter summary.

2.2 Introduction to Genetic Programming Algorithm

Genetic programming was introduced by Koza [4] and was initially used to evolve programs in Lisp. GP is an Evolutionary Algorithm (EA) [4] because it emulates Darwin's theory of evolution. It is one of the machine learning techniques that has been successfully applied to various problem domains. These domains include data classification, image processing, natural language processing and electronic circuit design [5, 6].

GP generates programs by mimicking the natural evolutionary processes of selection, reproduction, mutation and crossover. Unlike a GA which searches the solution space, GP searches a program space. Generally, a GP run starts with the creation of the individuals in the initial population [4]. These individuals are randomly created from the function and

terminal sets. The process continues with the evaluation of the individuals to determine their fitness. The next generation is formed by applying genetic operators to the fitter individuals; thus breeding the population of the next generation. These fitter individuals are selected using a selection method. During each generation, evaluation is done to check if a solution has been found. The GP run terminates if a solution is found. Other termination criteria may include a specified runtime limit being exceeded and/or a specified number of generations being reached. At the end of a GP run, the best individual is returned as the result of the run. Koza [7] presents the GP algorithm shown in Figure 2.1.

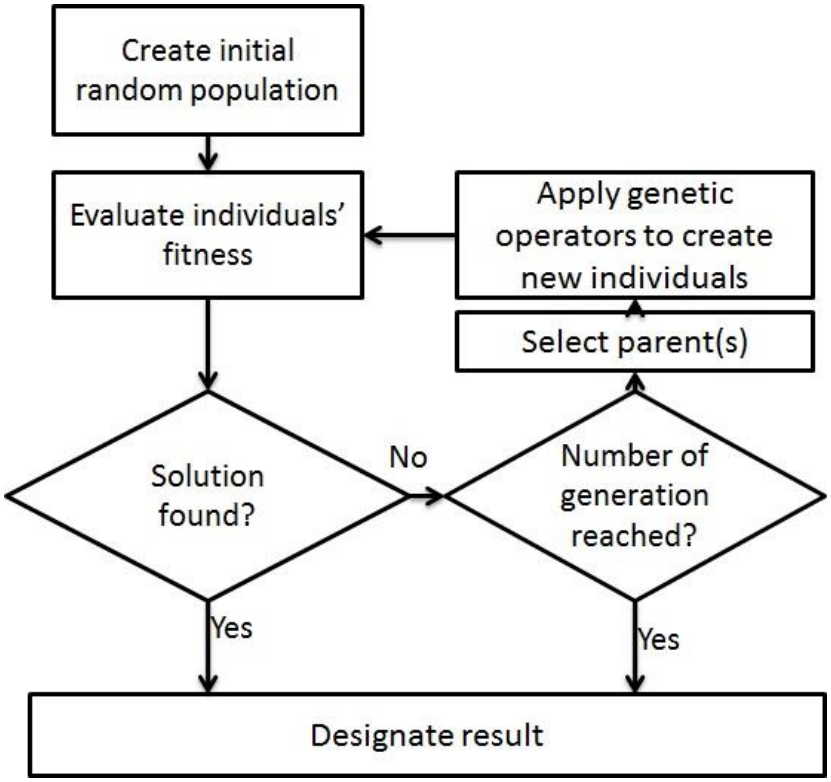


Figure 2.1. The standard GP algorithm

A control model, as the name implies, specifies how the regeneration process is regulated. The two control models widely used by GP are the generational control model and the steady state control model. Section 2.3 provides more details about control models. A GP algorithm implements one of these control models.

A GP run involves the processes from the initial population generation to the designation of the result as shown in Figure 2.1. GP is stochastic in nature and due to the possibility of random noise more than one run is performed. Each run is performed with a different random

number generator seed. Seeding each run ensures that a result can be reproduced whenever the same seed is used.

2.3 Control Model

The three control models that can be implemented by a GP algorithm as described by Bruce [8] include the generational [4, 7], the steady state [7, 9, 10] and varying population size. The varying population size control model is scarcely used. The generational and steady state are explained in more detail in subsections 2.3.1 and 2.3.2 respectively.

2.3.1 Generational Control Model

The generational control model is traditionally used in genetic programming. In a generational control model, one generation is distinct from another during the GP run. A complete population of individuals is maintained for each generation. The initial population generation is randomly created from the function and terminal sets[1]. With the exception of the initial population generation, the individuals in each generation are created by applying genetic operators to the selected individuals in the previous generation. The newly created individuals called offspring become the current generation and replace those in the previous generation. The population size remains constant throughout the GP run.

1. Randomly create n individuals.
2. Repeat
    a. Evaluate the individuals in the current population.
    b. Repeat
        i. Use a selection method to select an individual or individuals from the current population.
        ii. Apply a genetic operator to the selected individual(s) to create new individuals.
        iii. Add the created individuals to a new population.
    Until n individuals have been created
Until a termination criterion has been met

Figure 2.2. The generational control model algorithm

According to Pillay [11], two population arrays are maintained, one for the current population and the other for the new population. The algorithm is presented in Figure 2.2. The major advantage of this control model is that it is easy to implement.

[1] Section 2.5 explains the function and terminal sets.

2.3.2 Steady state control Model

The steady state control model [12] is adopted from GAs by GP researchers [6]. Unlike the generational control model, this control model does not have distinct generations. A constant population size is maintained throughout the GP run. In the steady state control model, parents are selected using one of the selection methods. Genetic operators are applied to the parents to create offspring. A number of individuals with poor fitness are selected using an inverse selection method. These selected individuals are replaced with the created offspring. Once an individual has been introduced into the population, it can be selected for replacement or as a parent for the purpose of creating another individual. A single population array is therefore maintained throughout the GP run [11]. The algorithm is presented in Figure 2.3.

1. Randomly create n individuals.
2. Repeat
    a. Evaluate the individuals in the current population.
    b. Use a selection method to select individuals from the current population.
    c. Apply genetic operators to the selected individuals to create new individuals.
    d. Use an inverse selection method to select individuals (from the current population) to replace.
    e. Replace the selected individuals in d. above with the newly created individuals.

   Until a termination criterion is met

Figure 2.3. A steady-state Control Model

Since there is no distinct generation, a generation is modelled as those steps performed from one Repeat to another as specified in the algorithm in Figure 2.3. Given that two population arrays are maintained for the generational control model while a single population array is maintained for the steady state control model, the former uses more memory than the latter.

2.4 Program Representation

A parse tree, also known as a syntax tree, is traditionally used to represent an individual in GP [4]. This is referred to as tree based genetic programming. In tree based GP, a parse tree, as shown in Figure 2.4 is a computer program which can be viewed as a data structure where the nodes are hierarchical. The nodes are the elements of the function and terminal sets[2]. The root node is usually an element of a function set. In Figure 2.4, the leaf nodes are at the

[2]Function and terminal set are discussed in section 2.5

maximum depth. The depth of a node is the number of edges via the longest path between the root and the node[3]. Labels for the root nodes are usually the elements of a terminal set. This representation assumes the form of an inverted tree. The number of child nodes any node may have is called the arity of that node. The arity of a node is the number of arguments of the node. For instance, in Figure 2.4, the "+" node has an arity of 2 while the "n" node has an arity of 0.

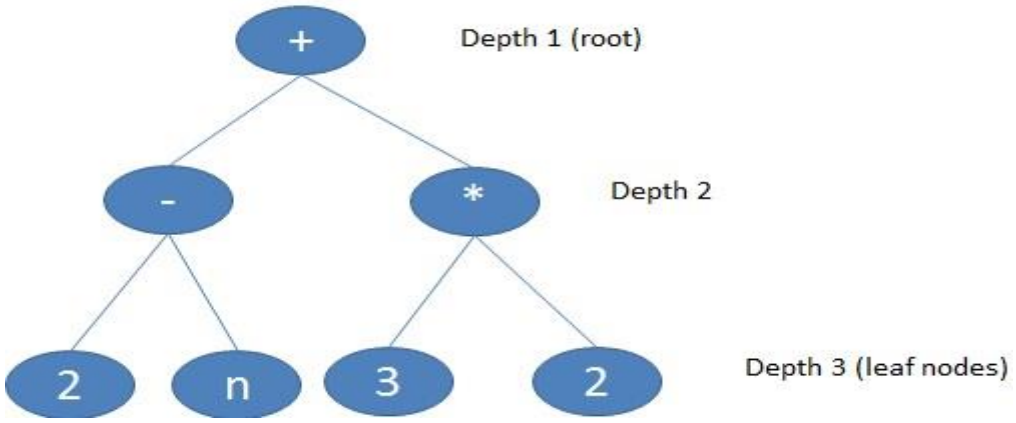


Figure 2.4. A tree based GP representation

Each parse tree (a program), is evaluated by performing a postfix or prefix traversal of the tree. Pillay [11] explained that the prefix traversal has a shorter runtime compared to the postfix traversal when executing conditional statements such as the *if-then-else*. This is an advantage of the prefix traversal over the postfix traversal.

2.5 Function and Terminal Set

According to Koza [7], one of the preparatory steps in solving a problem using GP is to choose the terminals and functions for the problem. Both these terminals and functions are the components of the program to be evolved [6]. Each node in a parse tree is an element of either the function or terminal set. Thus, each individual consists of nodes from the function set $F = \{f_1, f_2, f_3, \ldots f_n\}$ and the terminal set $T = \{t_1, t_2, t_3, \ldots t_m\}$ (where n and m are the numbers of functions and terminals respectively). Elements of the function and terminal sets are collectively referred to as primitives [11]. They form the internal representation language in which programs are expressed. Langdon [6] pointed out that the choice of primitives is a decisive factor for the success of the genetic programming algorithm.

[3] If the root is assumed as at depth 1, then the depth of a node is calculated as the number of edges via the longest path minus 1.

2.5.1 Function Set

The elements included in the function set are problem dependent. A function set may include standard mathematical operators, standard logical operators, and/or programmer defined operators. Examples of functions described in [4] and [11] include:

- Arithmetic functions: +, -, *.
- Conditional operators: If-then-else.
- Variable assignment functions: ASSIGN.
- Loop statements: WHILE...DO; REPEAT...UNTIL, FOR...DO.
- Programmer defined operators such as prog2 which combines two programming statements.

Any node with an arity greater than zero is considered a function node.

2.5.2 Terminal Set

Any element with an arity of zero is regarded as an element of the terminal set. The terminal set is the set of elements that form the leaf nodes of a parse tree. They do not take arguments. Functions which do not take arguments are included in the terminal set. According to Banzhaf *et al.* [5], a random ephemeral constant, $\Re$, may be used as an element of a terminal set to represent a range of numbers, say (0.0, 0.5]. If the ephemeral constant is chosen as a node when creating a parse tree, it is replaced by a value randomly chosen in the specified range.

2.6 Sufficiency and Closure

Koza [4] defines the sufficiency property as being satisfied if the required program can be expressed in terms of the elements of the function and terminal sets. If the sufficiency property is not satisfied, the GP algorithm may converge prematurely. Premature convergence occurs when the algorithm converges to a local optimum. A local optimum is a candidate solution from which a solution cannot be evolved. On the other hand, extraneous functions may increase the search space and prevent the GP algorithm from converging to a solution [11].

The closure property is satisfied if each function in the function set is able to accept as its argument a value that may be returned by any other function in that set as well as a value each terminal may take as input [4]. Consider the function set $F1 = \{+, -, *\}$ and the terminal set $T1 = \{x, 1\}$ for a symbolic regression problem. The closure property is satisfied because any program (i.e., a parse) expressed in terms of the elements of $F1$ and $T1$ is syntactically correct. Two examples are shown in Figure 2.5.

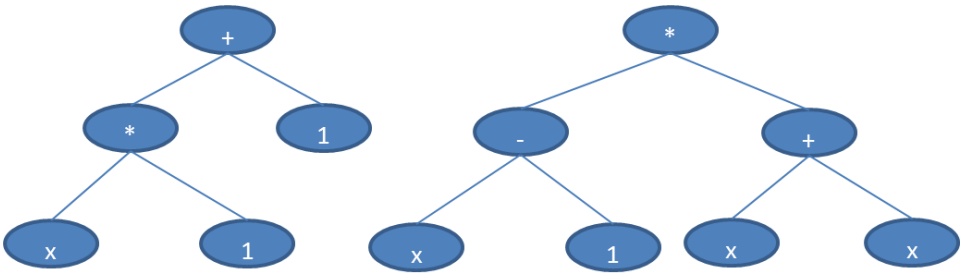


Figure 2.5. Two trees created when the closure property is satisfied

If $F1$ and $T1$ are replaced with the function set $F2 = \{+, -, *, OR\}$ and the terminal set $T2 = \{x, 1, true\}$ respectively, the closure property is not satisfied. This is because not all the parse trees expressed in terms of the elements of $F2$ and $T2$ are syntactically correct. Two examples are shown in Figure 2.6. These parse trees are syntactically incorrect because arithmetic operations cannot be performed on a Boolean value i.e., $true$ or $false$. Also the Boolean operator, namely, OR, cannot take an integer value as an argument. However, the closure property can be satisfied if the Boolean functions return 1 and 0 instead of $true$ or $false$.

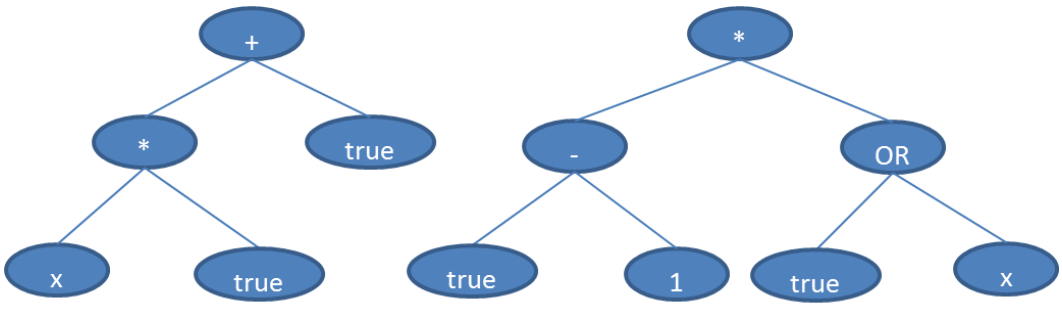


Figure 2.6. Two trees created when the closure property is not satisfied

2.7 Initial Population Generation

The individuals in the initial population are parse trees created using one of the initial population generation methods described in [4], namely, the full, grow, or ramped half-and-half method. Nodes in an individual are randomly chosen from either the function or terminal set. In Koza's implementation [4], the root node is randomly chosen from the function set. This ensures that a trivial program is not created. A tree size must be specified as one of the GP parameters[4]. The size of a tree is the maximum depth or the number of nodes in that tree [11]. Each of the initial population generation methods is explained in the following sections.

2.7.1 The Full Method

The full method creates individuals of the same size. The depth of each leaf node must be equal to the specified maximum depth. Nodes at a level less than the specified maximum depth are chosen from the function set. Nodes at a level equal to the specified maximum depth are chosen from the terminal set. Figure 2.7 shows three trees created using the full method. Notice that the number of nodes in each of the trees is not equal to the other but all the trees are of the same depth.

Diversity in the population is usually less when this method is used. This may lead the GP algorithm to converge prematurely. Variety is the measure of diversity in the population. It calculates the percentage of the number of individuals that are different in structure. High variety guarantees a high genetic diversity in the population which helps prevent premature convergence.

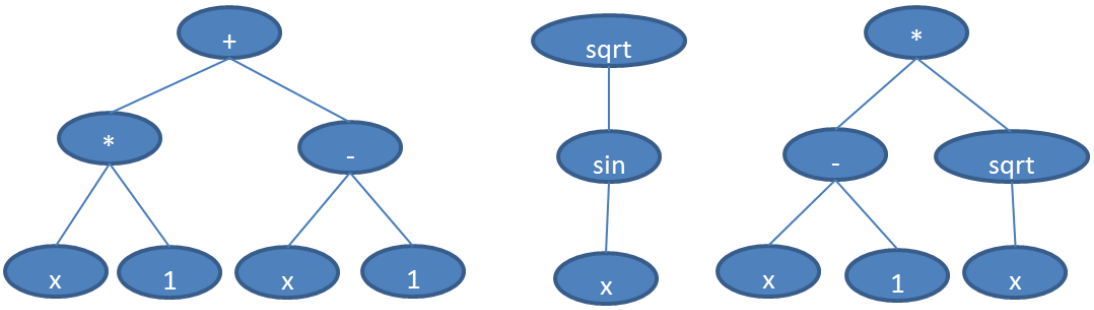


Figure 2.7. Three trees created using the full method

[4]GP parameters are discussed in Section 2.15

2.7.2 The Grow Method

The grow method creates individuals of various shapes and sizes. As pointed out by Pillay [11], the depth of each leaf node must be less or equal to the specified maximum depth. Nodes at a level less than the maximum depth are chosen from either the function or terminal set, while those at the level equal to the maximum depth are chosen from the terminal set. Figure 2.8 shows a tree created using the grow method.

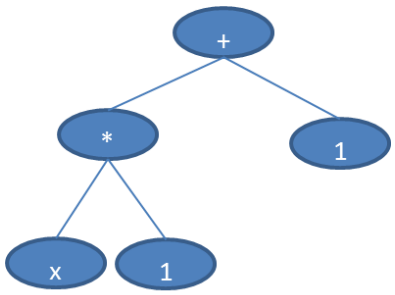


Figure 2.8. A tree created using the grow method.

This method provides more diversity in the population compared to the full method.

2.7.3 Ramped Half-and-Half

The ramped half-and-half method creates individuals of different shapes and sizes. It creates an equal number of individuals at each depth in the range of 2 to the maximum depth. Half of these individuals are created using the grow method while the other half are created using the full method.

For example, assume a population size of 12 with a specified maximum depth of 4. Using ramped half-and-half will create 4 individuals at depth 2, 3 and 4 as shown in Figure 2.9. Out of the 4 individuals at each depth, 2 individuals will be created using the full method (left) while the remaining two will be created using the grow method (right).

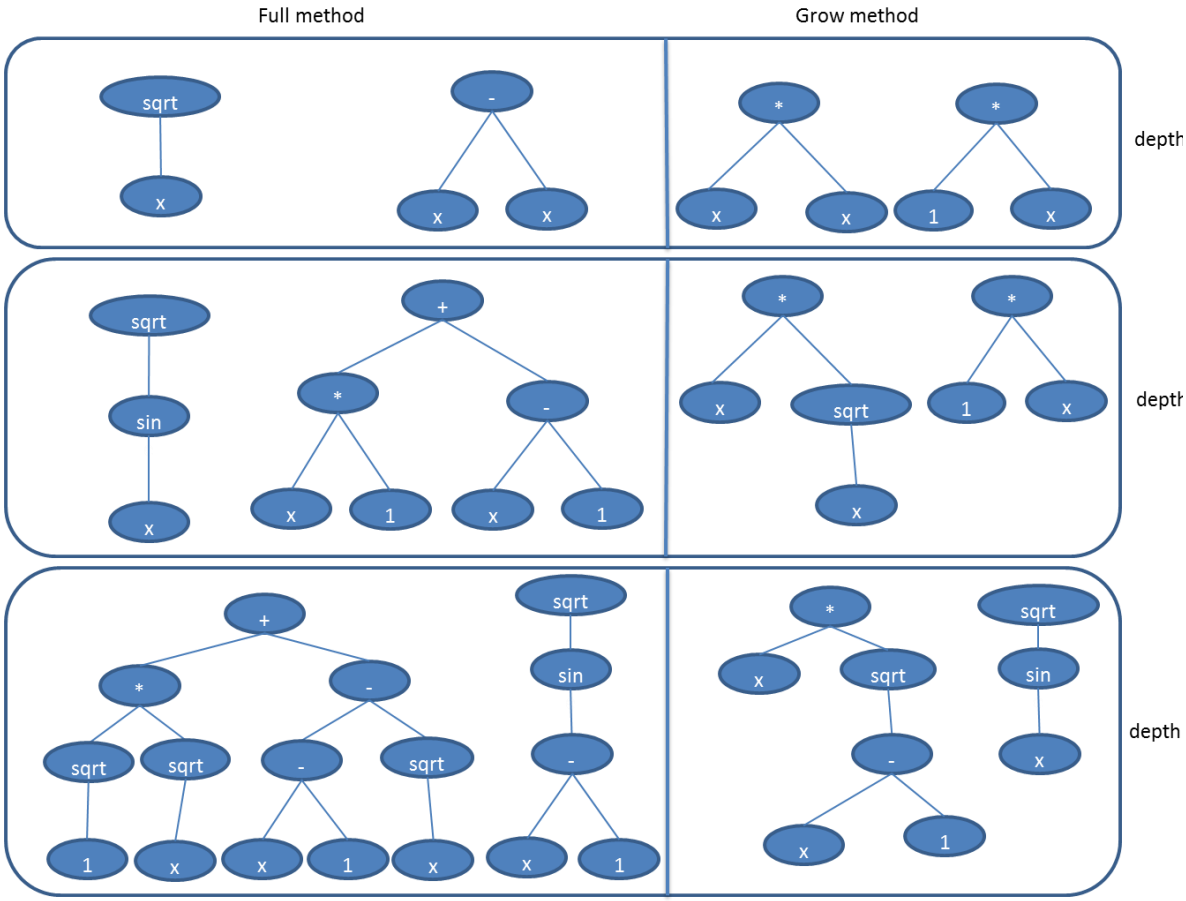


Figure 2.9. An initial population (of size 12) created using the ramped half-and-half

The major advantage of the ramped half-and-half method over the other methods is that it promotes diversity in the initial population.

2.8 Evaluation

At each generation of the evolution, each individual in the current population is evaluated and assigned a scalar value representing the individual's fitness. Fitness is the measure of how good an individual is at solving the problem at hand. In order to evaluate an individual, fitness cases and a fitness function are required.

2.8.1 Fitness Cases

A fitness case comprises an input or a set of input variables and their corresponding output values describing the target behaviour of the required program [4]. Fitness cases are problem dependent. For a symbolic regression problem [4], assume a researcher wants to use GP to evolve an unknown mathematical function of the form $y = f(x)$. The fitness cases are the values of x and the corresponding values of y. For example, the input values 0, 1, 2 and 3

correspond to the output values 1, 2, 5 and 10 respectively, for the expression $x^2 + 1$ to be evolved. For the artificial ant problem [4], assume a researcher wants to use GP to devise a program which navigates an artificial ant to pick up the pieces of food in a 32 X 32 grid. The fitness case for this problem is the starting position of the ant, the initial facing direction of the ant and the pieces of food in the grid. Potentially, there is more than one configuration for the fitness case but only one fitness case is used per run. Table 2.1 illustrates the fitness cases for the induction of a Boolean 3-symmetry function. The Boolean 3-symmetry function takes a binary string of length 3 and returns a value of true (i.e., 1) if the bits in the string are arranged symmetrically, otherwise it returns the value of false (i.e., 0).

Table 2.1. Fitness case for the Boolean 3-symmetry function

Input String			**Target Output**
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

In some domains, fitness cases are divided into training and testing sets. GP attempts to learn from the training set after which the evolved program is tested using the testing set [5]. Banzhaf *et al.* [5] and Pillay [11] argued that a small training set is likely to produce unreliable solutions that do not generalize. Generalization is considered a very significant performance evaluation criterion. Hence the fitness cases must represent a sufficient amount of the problem domain and the areas of special interest to the researcher. For example, assume GP is being used to evolve a program for calculating the factorial of a positive integer. The factorial of both 1 and 0 is 1. This is a special case which needs to be included in the fitness cases. Langdon [6] suggests that the size of the fitness cases be somewhat restricted to ensure that GP runtime remains feasible. Thus, a balance is required between GP runtime and generalization.

2.8.2 Fitness function

A number of fitness functions that can be implemented by GP are given in Koza [4]. According to Banzhaf *et al.* [5], various objectives can be tested by fitness functions. These include the correctness of a program, a minimization of the runtime of the programs and a minimization of the program size. A fitness function can be a measure of multiple criteria defined for the GP algorithm [6]. As mentioned earlier, each individual in the population is assigned a fitness value. A fitness function is used to determine the fitness value that will be assigned to an individual. GP uses the fitness values to determine the areas of the program space to search for a global optimum. Therefore the fitness function needs to be properly defined so that that GP clearly distinguishes solutions from near-solutions.

The simplest and commonly used fitness measure is the raw fitness. The function for calculating the raw fitness is domain dependent. For problems like symbolic regression, Equation 2-1 defines the raw fitness [11]. This is defined as the sum of the absolute value of the difference between the output produced by the individual and the target value [4, 13] over the set of fitness cases.

Equation 2-1. The raw fitness calculation

$$r(i,t) = \sum_{j=1}^{Ne} s(i,j) - c(j)$$

Where: $r(i,t)$ = the raw fitness of the ith individual at generation t.

$s(i,j)$ = the value returned by the ith individual for the jth fitness case.

$c(j)$ = the target value for the jth fitness case.

Ne = the number of fitness cases.

The hits criterion is used to determine whether a solution has been found. A hit is made if the output returned by an individual is equal to the target output.

In certain domains, a lower fitness means a better fitness. In others, a higher fitness means a better fitness. In solving the Boolean symmetry problem, the number of hits is equal to the raw fitness. This implies that a high fitness value means a better fitness. In solving the artificial ant problem [4], the fitness function can be defined as the number of pieces of food picked up. Again, a higher fitness means a better fitness.

Other fitness measures described by Koza [4] include standardized fitness, adjusted fitness and normalized fitness. The standardized fitness denoted by $s(i,t)$ redefines the raw fitness such that a lower numerical value is an indication of a better fitness [4, 11]. The best standardized fitness is zero. The standardized fitness is equal to the raw fitness if a lower fitness value indicates a better fitness. If a higher fitness value indicates a better fitness, the standardized fitness is calculated by subtracting the raw fitness from the maximum possible raw fitness. Assume r_{max} as the maximum possible raw fitness, the standardized fitness is calculated from the raw fitness as defined by Equation 2-2.

Equation 2-2. The standardized fitness calculation

$$s(i,t) = \begin{cases} r(i,t), & if\ a\ lower\ fitness\ value\ means\ a\ better\ fitness \\ r_{max} - r(i,t), & otherwise \end{cases}$$

The adjusted fitness and normalized fitness are calculated if the GP algorithm uses fitness proportionate section. In Koza [4] and Pillay [11], the adjusted fitness, denoted by $a(i,t)$, is described as being adequate for distinguishing a good individual from a very good individual. The value of the adjusted fitness is in the range 0 to 1. A higher value is an indication of a better fitness. According to Koza [4] and Pillay [11] the adjusted fitness is calculated from the standardized fitness as given by Equation 2-3.

Equation 2-3. The adjusted fitness calculation

$$a(i,t) = \frac{1}{1 + s(i,t)}$$

The normalized fitness measure is used when the fitness-proportionate selection method (discussed in section 2.9.2) is implemented. The normalized fitness, denoted by $n(i,t)$, is calculated from the adjusted fitness as given by Equation 2-4. The letter M denotes the population size.

Equation 2-4. The normalized fitness calculation

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^{M} a(k,t)}$$

The normalized fitness lies in the range 0 to 1. Like the adjusted fitness, a higher value is an indication of a better fitness. The sum of the normalized fitness values for each individual in the population must be equal to 1.

According to Langdon [6], the fitness of a function should be carefully designed to avoid a deceptive fitness function which may drive the GP algorithm towards local optima.

2.9 Selection Methods

Parents, which are used to create offspring, are selected using a selection method. The selection methods described by Koza [4] include tournament selection, fitness-proportionate selection and rank selection. Banzhaf *et al.* [5] state that the selection method employed by a GP algorithm has an effect on the runtime of the algorithm as well as the success of the GP algorithm. Bruce [8] explains the effect of selection methods using a concept called selection pressure. Selection pressure is the extent to which the selection method is biased towards the highly fit individuals in the population being selected [8]. Tournament and fitness proportionate selection methods are the most commonly used selection methods. Both selection methods are discussed further in section 2.9.1 and 2.9.2 respectively.

2.9.1 Tournament Selection

Tournament selection requires that a tournament size be specified. A number of individuals equal to the tournament size are randomly chosen. The individual with the best fitness becomes the winner of the tournament. The winner is used as a parent to create an offspring for the next generation.

The tournament size is used to increase or decrease the selection pressure. The selection is done with replacement. A high tournament size implies more competition. Thus, an increase in the size of the tournament increases the selection pressure. A small tournament size leaves the low fitness individuals with a fair chance of being selected. A small tournament size implies less competition. Therefore, a small tournament decreases the selection pressure [14]. A high selection pressure may lead to premature convergence while a very small tournament size may slow down the convergence rate.

For a steady state GP algorithm, an inverse selection method is needed to choose the individuals to replace. This must be an individual of the population with poor fitness. Inverse tournament selection could be used for this purpose. Inverse tournament selection is applied as follows: a number of individuals equal to the specified tournament size are randomly

chosen; the individual with the worst fitness is replaced in the population by the newly created offspring.

2.9.2 Fitness-Proportionate Selection

In fitness-proportionate selection, the fitness of a particular individual is calculated proportionate to the entire population. This selection method is also known as the roulette-wheel selection [15]. Fitness-proportionate selection requires more calculations and uses more memory compared to tournament selection. This is because the normalized and adjusted fitness of each individual in the population need to be calculated. Also, a mating pool needs to be created. A mating pool contains the individuals from which a parent must be selected to create the next generation. Each individual occurs in the mating pool a number of times determined by the fitness of the individual. The individuals of the population with poor fitness have a less chance of being added to the mating pool. Fitness-proportionate selection is applied as follows:

a. Calculate the standardized fitness as given in Equation 2-2.
b. Calculate the adjusted fitness as given in Equation 2-3.
c. Calculate the normalized fitness as given in Equation 2-4.
d. Create a mating pool as follows: for each individual,
    i. Calculate the product of the value obtained in c. and the population size.
    ii. Round up the value obtained in i. above to determine the number of times the individual will occur in the matting pool.
    iii. Add the individual in the mating pool a number of times specified in ii. above.

An individual is randomly selected from the mating pool. The genetic operator is applied to the parent to create an offspring for the next generation. In the example in Table 2.2, the fitness-proportionate selection method is applied with a population size **P** = 5.

Table 2.2. Illustrating fitness-proportionate selection method

Individual	A Standardized Fitness	B Adjusted Fitness	C Normalized Fitness	C x P	Number of Occurrences in Mating Pool
Individual1	24	0.040	0.061	0.31	0
Individual2	9	0.100	0.151	0.76	1
Individual3	15	0.063	0.095	0.48	0
Individual4	2	0.333	0.504	2.52	3
Individual5	7	0.125	0.189	0.95	1
Total		0.661	1.000		

The mating pool for Table 2.2 is {Individual2, Individual4, Individual4, Individual4, Individual5}. The number of occurrences of Individual1 and Individual2 is 0. Hence both these individuals are not added to the mating pool. Individual4 has 60% chance of being selected.

As illustrated in Table 2.2, there is a high chance that an individual with a better fitness will be selected several times while those with poor fitness may not be selected at all. Selecting the same individual several times could lead to a loss of diversity early in the population. Diversity is important to represent more of the program space. Lack of diversity could lead to a premature convergence.

As mentioned in section 2.9.1, if a steady state GP algorithm is implemented, an inverse selection method is needed to choose the individual to replace. Inverse fitness-proportionate selection could be used to select a weaker individual in the population. To use the inverse fitness-proportionate selection method, the normalized fitness is defined as given in Equation 2-5.

Equation 2-5. The inverse fitness proportionate selection probability calculation

$$n_{inv}(i,t) = 1.0 - \frac{a(i,t)}{\sum_{i=1}^{M} a(i,t)}$$

Where $n_{inv}(i,t)$ is the normalized fitness, $a(i,t)$ is the adjusted fitness and M is the population size. Table 2.3 shows the values obtained from Equation 2-5. The value recorded

in the first row (for Individual1) is the largest and indicates that Individual1 will be to be replaced.

Table 2.3. Illustrating inverse fitness-proportionate selection

Individual	**Normalized Fitness** $n_{inv}(i,t)$
Individual1	0.94
Individual2	0.85
Individual3	0.91
Individual4	0.50
Individual5	0.81

2.10 Genetic Operators

At each generation of the evolutionary process, genetic operators are applied to the selected parents to create offspring. The newly created individuals form a new generation. A number of genetic operators have been reported in the literature [4]. The three commonly used genetic operators are the reproduction, crossover and mutation operators. GP uses a genetic operator application rate to determine the number of offspring that will be created using the operator. For example, assume a population size of 20 and the reproduction, crossover and mutation rates of 20, 50 and 30 respectively. Four offspring will be created by applying the reproduction operator, 10 will be created by applying the crossover operator while 6 offspring will be created by applying the mutation operator. To understand the effect of these operators on the search, the terms neighbourhood, exploitation and exploration are defined. The neighbourhood N(*s*) of a solution $s \in S$, an area of a search space, is the subset of solutions which can be obtained by applying genetic operators to *s*. Whereas exploitation involves refining the search in the neighbourhood of the current solution with the hope of improving the current solution, exploration involves searching a larger portion of the search space with the hope finding another solution area that could be refined.

2.10.1 Reproduction

The reproduction operator is the simplest of the operators. It is applied to one individual as follows: a selection method is used to select an individual; the operator copies the selected individual into the population of the next generation.

The reproduction operator promotes convergence. A high application rate of this operator may cause an individual to be repeatedly copied into the next generation. This reduces diversity in the population. Diversity is required at the early stage of the GP algorithm.

2.10.2 Crossover

The crossover operator mimics the gene recombination process in nature. It is traditionally applied to two individuals. Banzhaf *et al*. [5] and Pillay [11] describe the crossover operator as an operator that retains the information passed by the parents to the offspring. The crossover operator is applied as follows. Two parents are selected using one of the selection methods. A crossover point is randomly chosen in each of the parents. Figure 2.10 labels these points as P1 and P2. The sub-trees rooted at the chosen points are swapped to form two offspring.

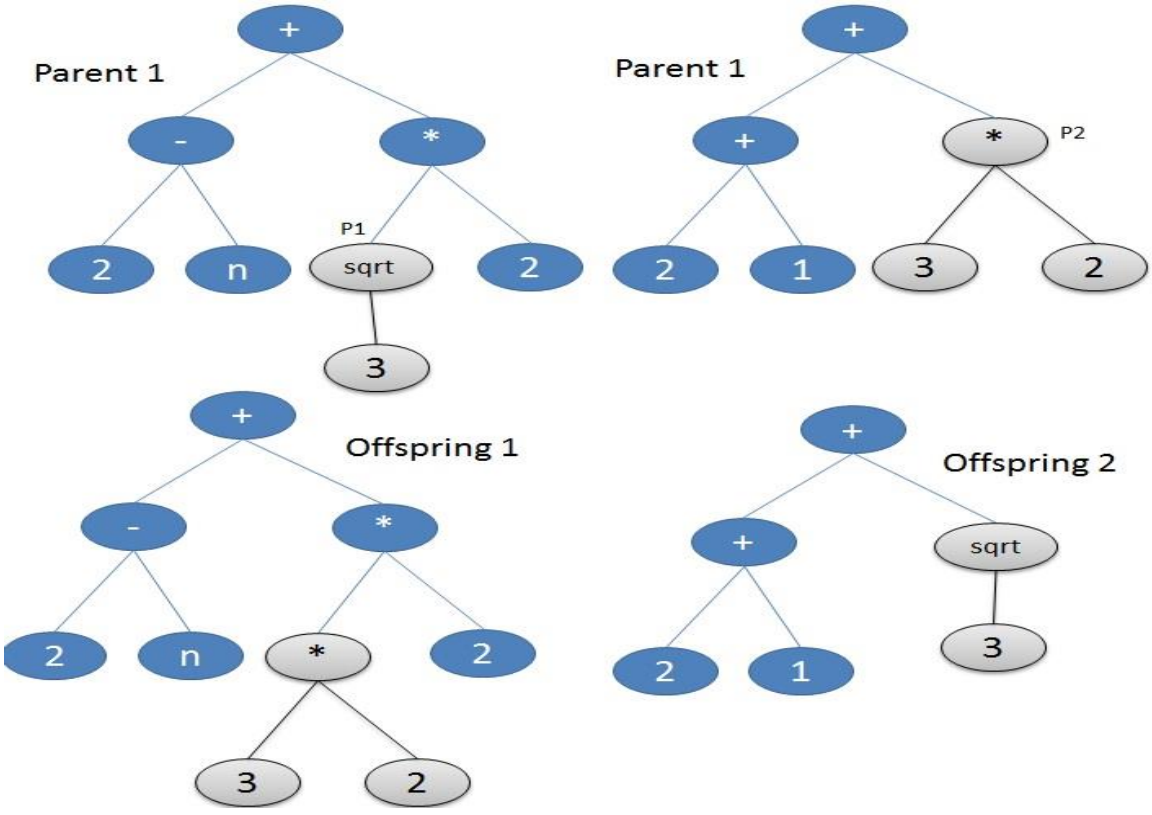


Figure 2.10. The crossover operator

In order to control the size of the offspring created, a limit is usually placed on the tree size. The tree size is measured in terms of the tree depth or number of nodes in the tree. If the depth of the offspring created exceeds the specified maximum depth, each of the function nodes at the specified maximum depth is replaced by a randomly selected terminal. Placing a

limit on the tree size reduces bloat. Bloat is an excessive increase in the size of the tree without a change in the fitness of the individual.

The crossover is a local search operator. The aim of a local search operator is to explore the neighbourhoods of the candidate solutions. The research conducted in [16, 17] suggest that the GP algorithm that applies a high rate of the crossover operator is susceptible to premature convergence. Crossover has also been criticized for its destructive effects [5, 11]. It breaks up good building blocks. The good building blocks could form part of a solution.

Koza [4] and Banzhaf *et al*. [5] implemented the crossover operator with a bias towards choosing a function node as a crossover point. The choice of the crossover point is biased to avoid mere swapping of the terminals. Beadle and Johnson [18] state that preventing the crossover from merely swapping a terminal is likely to cause a bigger jump in the search space. They point out that for certain problems, the disadvantage is that it does not allow small refinements in the search space [18].

At the early stage of the evolution process, exploration is needed more than exploitation. As the generation progresses, there is a need for the algorithm to converge. Hence exploitation will be needed more than exploration. One could apply a measure that prevents the crossover operator from swapping terminals at the early stage of the evolution process but allows terminals to be swapped at a later stage. By this, small refinements are allowed while ensuring a balance between exploration and exploitation.

2.10.3 Mutation

The mutation operator is applied to one individual. Banzhaf *et al*. [5] and Pillay [11] describe the mutation operator as an operator that increases the diversity of the population by directing the search towards a new area of the search space. Thus, the mutation operator increases the diversity in the population. Pillay [11] states that the main purpose of mutation is to maintain diversity in the population. The mutation operator performs exploration more than exploitation and is a global search operator.

The mutation operator chooses a parent using a selection method. A mutation point, e.g., P1 in Figure 2.11 is randomly chosen from the parent. The subtree rooted at the chosen point is deleted. A new subtree is created using the method used to create the initial population, e.g. grow, and inserted at the mutation point.

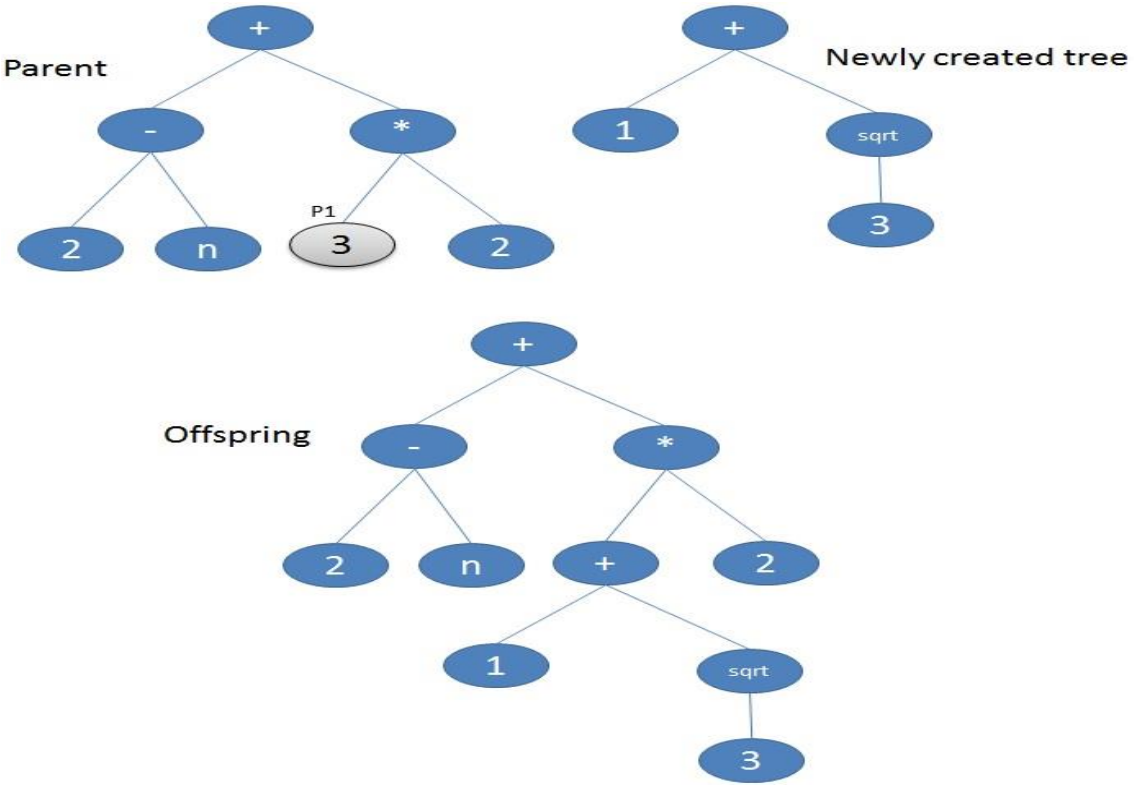


Figure 2.11. The mutation operator

Similar to the crossover operator, a limit is usually placed on the size of the offspring created by the mutation operator. A high application rate of the mutation operator may slow down the algorithm convergence because the operator has the tendency of directing the search to a different area of the search space.

2.11 Termination Criteria

As specified in Figure 2.1 the GP algorithm stops when a solution has been found or when the specified number of generations has been completed. However, in certain problem domains, a solution for the problem at hand is not known. In others, a perfect solution to a problem is not possible to find. In these cases the GP algorithm can terminate if a certain runtime is exceeded or a near-solution is found. Hence the termination criteria is domain dependent [19].

According to Koza [4] the result of a run is the best individual over all the generations. Pillay [11] states that the best individual of the last generation can be returned as the result of a run. In certain problems such as the problem to compute the factorial of a given integer [11], both these have been found to return the same result.

2.12 Advancements in Genetic Programming

This section describes some of the extensions to genetic programming that are relevant to this study.

Montana [20] introduced the use of strongly-typed genetic programming to ensure that syntactically correct programs are created and to reduce the search space. Section 2.12.1 discusses Strongly-typed GP.

GP was initially used to evolve functions in Lisp without any need for memory [4]. In order for GP to induce algorithms, memory and iteration was introduced. Also, the use of memory and iteration improves the GP problem solving ability. Section 2.12.2 and 2.12.3 discuss the use of memory and iteration respectively.

Programmers break a complex program into modules. In GP, Koza [7] introduced a concept of modularization called Automatically Defined Functions (ADFs). This has been found to be beneficial when applying genetic programming to complex problems. Section 2.12.4 describes ADFs as a means of modularization in genetic programming.

2.12.1 Strongly Typed GP

Strongly-typed GP [20] is one of the techniques GP systems have used to ensure that each function takes one or more valid arguments and to ensure that the programs generated are syntactically correct. Also, instead of allowing all the possible combinations of functions and terminals, typing can be used to specify certain combinations thereby reducing the search space.

Typing is done by assigning types to nodes and the arguments of the nodes. This is called point typing. For instance, the logical operator, greater than or equal to (>=), checks if the first argument is greater than or equal to the second argument. It cannot take a Boolean operator such as the OR operator as an argument because the OR operator returns a true or false value, which is not a number. However, the >= operator must return a true or false value as a result of the comparison made. ADFs introduced by Koza [31] uses branch typing. In this typing, each ADF branch as well as the main program branch is assigned a type. Genetic operators, e.g. crossover, are allowed between two branches of the same type to avoid creating an invalid offspring. According to Pillay [11], strong typing facilitates translation of GP generated algorithms into a programming language.

2.12.2 The Use of Memory

The solution to many problems requires that the values computed earlier in the program be used later in the program. To facilitate this, memory is required to store these values which can be located via variables representing the memory location of the values. In GP, the values stored in the memory are represented using variable names and the operators that operate on the values are called memory manipulation operators. The type of memory manipulation operators a GP system should employ is dependent on the problem domain. This study discusses the use of memory under two broad categories, namely, named and indexed memory.

2.12.2.1 Named Memory

Koza [21] described the use of named memory. A memory location is represented using a variable name so that the content of the memory can be accessed via the variable name. The SV i.e., settable variable, proposed by Koza [4] is a variable name which is set early in an individual and used later in the individual.

The counter and iteration variables introduced by Pillay [22] function as a local variables. The variables were maintained for each instance of the *for* iterative operator (see 2.12.3).

2.12.2.2 Indexed Memory

Most programming languages provide one or more ways of manipulating a vector and a multidimensional array. Some of the operators GP uses to manipulate the vectors are the *read* and *write* operators [23] and the *aread* operator [11]. Both the *read* and the *aread* operators take a single integer argument. The argument is an index of a vector. Both the *read* and the *aread* operators return the element in the memory location indexed by the integer argument. If the integer argument is not a valid index of a memory location, the default value of the data type is returned. This is done to ensure that closure property is satisfied. The *write* operator takes two arguments. The first argument represents a value to be written to a memory location indexed by the second argument which must evaluate to an integer value. The operator returns the value previously stored in the memory location indexed by its second argument as shown in Figure 2.12. If the *write* operator attempts to write to an invalid index, the default value of the data type is returned.

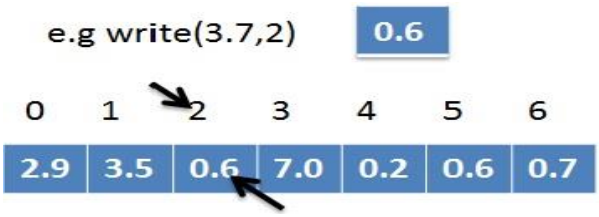


Figure 2.12. The operation of write memory manipulation operator

The operation in Figure 2.12 replaces the memory location indexed by 2 with a value of 3.7 and returns the value originally in the memory location (i.e. 0.6).

2.12.3 Iteration

Iteration is used in high level programming languages such as C, FORTRAN, PASCAL and Java to achieve efficiency. The major benefit of iteration is that it reduces code repetition and code length by providing mechanisms for repeated execution of a code segment. The solution to many problems is facilitated by the use of iteration. Iteration allows GP to enjoy the benefit mentioned above. A number of iterative operators have been used in GP. Examples of these operators include the DU (Do Until) operator [4], *for* operator [22], *while* and *dowhile* operators [24, 25], *for_loop1* and *for_loop2* operators [26].

The DU operator takes two arguments. The first argument is the code which must be evaluated at least once. The second is a condition which specifies when the iteration must stop. The second argument returns a value of true or false. The first argument is evaluated again if the second argument returns a value of false; otherwise the evaluation of the operator stops. The possibility of infinite iteration or an iteration that takes too long has been one of the problems associated with using iterative operators in GP. To avoid this problem, there must be a preset maximum number of iterations that can be performed by an instance of the DU operator. The execution of the operator terminates if the number is exceeded. In the Blocks World Problem [4], 25 iterations were allowed for an instance of the DU operator and 100 iterations for an individual.

The *for* operator takes three arguments. The first and second arguments are integers. The third argument is iteratively executed a number of times indicated by the difference between its first two arguments plus one. Thus, if the difference between the first and second argument is n, then the third argument will be executed $n + 1$ times. Two variables, namely, the counter variable and the iteration variable, are maintained for each instance of the *for* operator as shown in Figure 2.13. The counter variable is instantiated to the value of its first argument while the iteration variable is instantiated to the default value of the type of the third argument. If the third argument is a "*" or "/" node, the iteration variable is instantiated to 1 instead of 0. This is done to prevent division by 0 or multiplication by 0. Both the counter and iteration variables are added to the terminal set when creating the parse tree representing the third argument of the *for* operator. The operator returns the value of its last iteration i.e., the value held by the iteration variable at the last execution of the third

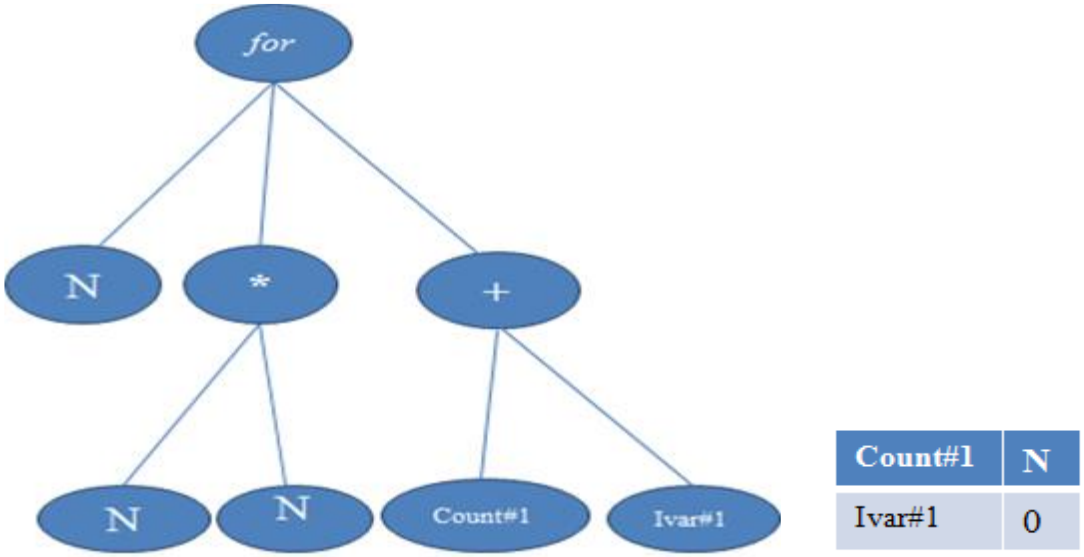


Figure 2.13. An individual program using the *for* operator (left) with the iteration and counter variables (right)

argument. The individual in Figure 2.13 is a program that sums all the digits from a given number to its square. The counter variable is initialised to the value of the first child N, while the iteration variable is initialised to zero.

Pillay [25] implemented the *while* and *dowhile* operators similar to that implemented in [24]. Both these operators take two arguments each and use the counter and iteration variables in the same way as the *for* operator. The first argument of the *while* operator is a condition which specifies when the execution of the second argument must stop. The condition must be checked first. The *dowhile* operator executes its first argument at least once before checking the condition, i.e., the second argument. The counter variable of the *while*

and *dowhile* operator is added to the terminal set when creating the parse tree representing the first argument of the operators. Both the counter and iteration variables are added to the terminal set when creating the parse tree representing the second argument.

The first and the second argument of the *for* operator are used as bounds that specify the number of iterations and avoid infinite iterations. However, the difference between the first and second argument may be a large number which may result in an iteration that takes too long. The *while* and *dowhile* operators could also result in infinite iterations or a time-consuming iterations. Hence a maximum number of iterations must also be specified when using these operators. To satisfy the closure property when the iteration body is not executed at all, the *for* operator returns the default value of it third argument, the *while* operator returns the default value of its second argument and the *dowhile* operator returns the default value of its first argument.

Li [26] introduced the use of the *for-loop1* and *for-loop2* iterative operators. The major difference between the *for-loop1* and *for-loop2* operators is that the former takes two arguments while the later takes three arguments. The first argument of the *for-loop1* must evaluate to an integer and specifies the number of times the second argument will be executed. The first and second arguments of the *for-loop2* must evaluate to integers. The third argument must be evaluated a number of times specified by subtracting the value of the first argument from the value of the second argument. The third argument is not executed if the first argument is greater than the second argument. Li [26] describes the *for-loop1* and *for-loop2* operators into two forms, namely, simple and unrestricted. The operators are in simple form if the integer arguments are randomly generated within the range of 1 and the specified maximum number of iteration. They are in unrestricted form if the integer arguments are expressions that must be evaluated to integer values.

2.12.4 Modularization

Modularization as described by Koza [7] and Banzhaf *et al*. [5] is a technique used by human programmers in problem-solving. Modularization emphasizes separating the functionality of a program into a number of functional modules such that each module solves a different task and can be reused. The GP algorithm is faced with the challenge of solving complex problems. As a result, Automatically Defined Functions (ADFs) [7] was introduced into the genetic programming algorithm to increase the problem solving ability of genetic

programming. The use of ADFs allows for simultaneous induction of the main program and one or more subroutines.

In Koza's implementation of ADFs, each individual has one or more branches. One of the branches must represent the main program while the others represent functions. The branch that represents the main program is referred to as the results producing branch while other branches are referred to as the function defining branches. Figure 2.14 illustrates an individual consisting of one ADF. The individual has one function defining branch and a results producing branch. Each node in the individual as shown by Figure 2.14 is described as follows:

1. The *Progn* is a connective node that combines the function defining branch and the results producing branch.
2. The *defun* node mimics a keyword in Lisp programming. It marks the beginning of each function in Lisp.
3. Name of the ADF
4. The argument list of the ADF
5. A place-holder for the result produced by the body of the ADF
6. The body of the ADF
7. A place-holder for the result produced by the result producing branch (main program), and
8. The body of the result producing branch

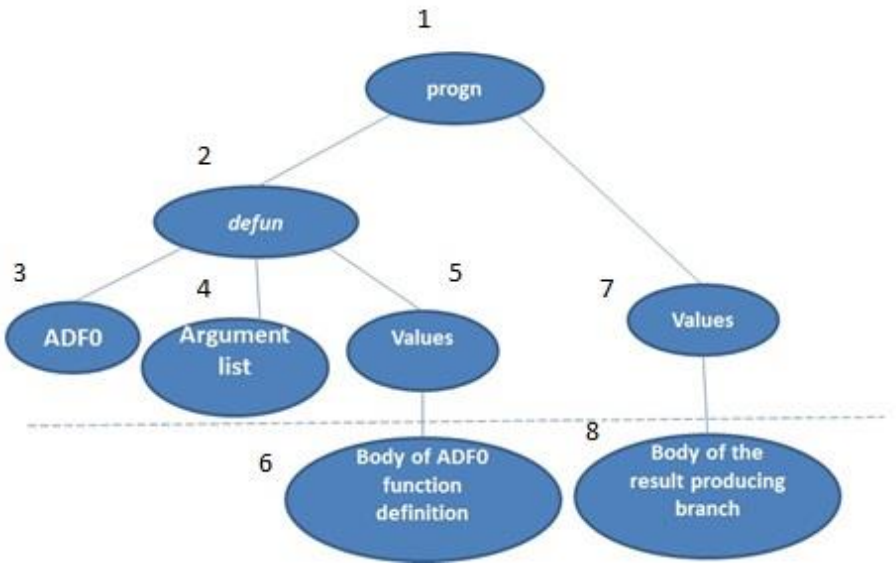


Figure 2.14. An individual consisting of one ADF

An individual program with an ADF can be divided into two parts, namely, variant and non-variant. The non-variants, part above the dotted line (Figure 2.14) is fixed while the variants, part below, can change during the GP run. Branch typing is commonly used for ADFs. Genetic operators are allowed between two branches if both the branches are of the same type.

Using ADFs requires that the user specify the architecture of the overall program in addition to the basic preparatory steps in using GP [5, 7]. The number of function defining branches and the number of arguments for each ADF need to be specified. The function and terminal sets for both the results producing branch and function defining branch must be specified. Also, function calls between the ADFs must be specified. From the study presented in [7], the use of ADF is beneficial when dealing with a complex problem. However, ADFs are not effective when applied to a simple problem that can be solved without using ADFs [11].

Bruce [8] introduced a representation format which allows each ADF to be a separate tree. In Bruce's representation shown in Figure 2.15, each individual consists of n trees stored in a fixed size array. One of the n trees is an individual representing the main program while the rest represent ADFs. Thus, there are n-1 ADFs and 1 main program in an individual with n trees. A number of advantages of Bruce's representation over Koza's representation of individuals with ADF as specified in [8, 25] include:

- It simplifies program representation by allowing a clear separation of ADFs from the main program.
- Since the parse trees are clearly separated, genetic operators do not need to be restricted to a certain branch of the parse trees.
- It presents more general means of representation since an individual without ADFs is simply a tree stored in a single cell array.

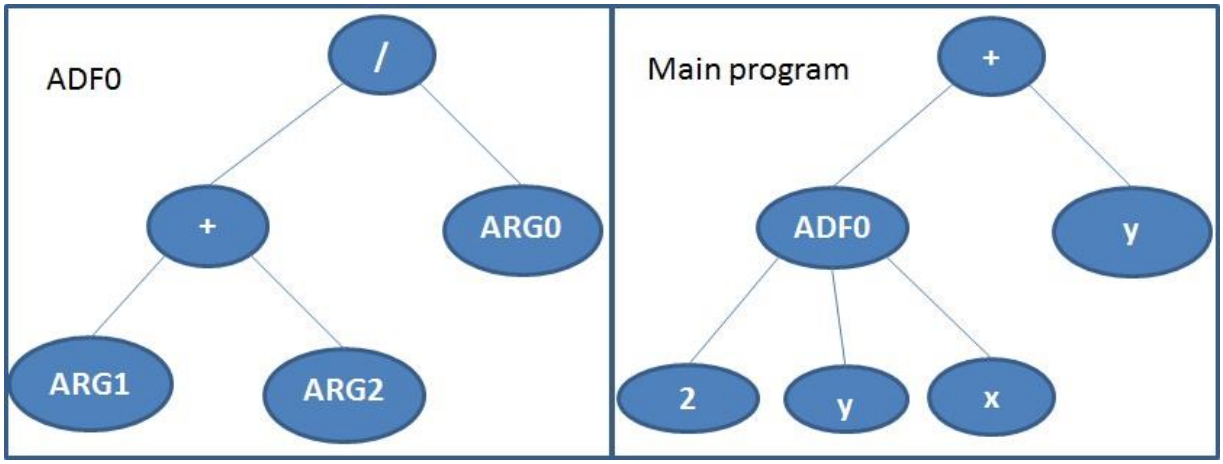


Figure 2.15. Alternative ADF representation introduced by Bruce

The representation introduced by Bruce provides easier implementation and easier understanding of GP with ADFs. This is because the ADFs are separated the way human programmers separate modules in high level programming languages such as Java. Adding to the advantages of Bruce's representation, Langdon [6] introduced pass-by-reference. If a variable passed to the ADF is changed during execution of the ADF, the value changes in the calling program.

2.13 Bloat in GP

During the GP run, functions and terminals are combined in a way that might not be anticipated by a human programmer. Some of these combinations are not necessary because the fitness of the program that contains them is not affected if they are removed. Such combinations or parts of code are redundant code and are known as introns [19]. Over the years, introns have been studied and have been categorized as syntactic and semantic introns [27, 28]. Syntactic introns are code that does not contribute to the fitness of the program because the code is not executed. For example, (if (2 < 1) X + 2 else X – 3). The conditional test will always result in false. Hence X + 2 will never be executed. Semantic introns are code that are executed but have no effect on the fitness of the program. Examples of such code are (MOVE-LEFT MOVE-RIGHT), (0 AND 0). A program size increases as the GP generations progress. This increases the introns in the program and also introduces bloat. Bloat is an increase in introns [5, 29]. In Koza [4], the edit operator is used to reduce the amount of introns a program contains. Researchers [30, 31] have argued that the presence of bloat is a mechanism that GP uses to reduce the destructive effect of the crossover operator. The crossover operator is less destructive if it removes an intron from a program but more destructive if it removes a building block from the program. Thus, bloat can be considered

beneficial. However, as the GP generations progress during a run, the individuals become fitter and the variation in the population reduces. If bloat occurs, the fitness of the best individuals might be difficult to improve and, as such, the algorithm may converge prematurely. Research in the area of bloat in GP is ongoing.

2.14 Strengths and Weaknesses of GP

Since GP was proposed by Koza [4], it has gained popularity and has been used in many applications. Despite much research in the field and the benefits provided by GP, there are number of problems faced by GP researchers. These benefits and problems are highlighted as the strengths and weaknesses of GP in the following sections.

2.14.1 Strengths

Ease of understanding: The programs generated by GP make use of terminals and functions that are well known by a human programmer and thus, are very easy to understand.

Solution variants: Due to the stochastic nature of GP, a different seed is used for each run. As a result, various variants of the required solution can be produced by GP.

2.14.2 Weakness

Premature convergence: GP is susceptible to premature convergence which is caused by factors such as the lack of diversity in the initial population and the seed used for the run [11].

2.15 Setting up a Genetic Programming System

Before implementing a genetic programming system, there are certain parameters that must be set. A decision must also be made about the process e.g., the control model to be used. In [6, 7, 11], the authors list the basic preparatory steps that must be performed before implementing a genetic programming system. These steps include the following:

- Choose the terminal and function sets.
- Define the fitness function.
- Choose a selection method.
- Choose genetic operators.
- Specify the termination criteria and other control parameters such as the population size, the maximum tree size and the application rate for each of the genetic operators.

The study conducted by Pillay [11] points out that the genetic programming parameters are problem-dependent. Thus, trial runs need to be performed and the parameters that give the best result must to be chosen.

2.16 Chapter Summary

This chapter described the genetic programming algorithm. Firstly, it provided the description of two control models, namely, generational and steady state. The control model commonly used by GP is the generational control model. The GP algorithm starts by creating the initial population. Each individual in the population is represented as a parse tree. The population is evaluated to check how good each individual in the population is at solving the problem at hand. If a solution has not been found or the number of generations specified has not been reached, a selection method is used to select parents. Genetic operators are applied to the selected parents to create offspring that form the next generation. The chapter highlighted what is required to set up a GP system. It described some advancements in GP. These include strongly typed GP which facilitates the generation of syntactically correct programs, the use of memory, iteration and modularization in GP. Bloat in GP was briefly discussed. Finally, the chapter stated some GP strengths and weakness. The next chapter will discuss GE, a variation of GP.

CHAPTER 3: GRAMMATICAL EVOLUTION

3.1 Introduction

The previous chapter introduced genetic programming which searches a program space for a program which when executed will produce a solution to a problem at hand. This chapter introduces grammatical evolution, a variation of genetic programming. Grammatical evolution uses a genotype-phenotype distinction and maps from a genotypic space to a phenotypic space to produce a program.

Section 3.2 introduces grammatical evolution, followed by section 3.3 which describes the use of grammars in grammatical evolution. GE represents a chromosome i.e., a genotype as a binary string this is discussed in section 3.4. Section 3.5 discusses the initial population generation. The genotype needs to be expressed as a program; the mapping process is detailed in section 3.6 while wrapping is discussed in section 3.7. Section 3.8 and section 3.9 describe the evaluation and selection methods respectively while section 3.10 discusses genetic operators which are used to produce the offspring. Section 3.11 discusses the termination criteria. Bloat is discussed in section 3.12. Section 3.13 discuses modularity, a concept used to improve the scalability of GE. Each evolutionary algorithm has its strengths and weaknesses, section 3.14 highlights the strengths and weaknesses of grammatical evolution. Lastly, section 3.15 presents a summary of the important aspects of grammatical evolution.

3.2 Introduction to Grammatical Evolution

Genetic Programming (GP) was proposed by Koza [4] and uses a parse (syntax) tree to represent a solution. Since the inception of GP, variants of GP have been proposed. GE and Gene Expression Programming (GEP) [32] are variations of genetic programming which separate a genotypic space from a program space (i.e., phenotypic space). This work however, focuses on GE rather than GEP which is not directly applicable to object-oriented programs evolution. GE uses a grammar and maps the genotype to the production rules of the grammar to produce a program. GE represents a chromosome as a group of binary strings called codons, thus the genotype is a binary string. A program is executed by firstly converting each codon to a denary value. Each denary value is then mapped to a production rule of the grammar to form a program. The program is then executed [1].

Grammatical evolution is known to obtain good results for complex problems. It has been successfully applied to solve problems in many domains. These domains include engineering

[33, 34], biology [35] and forecasting [36]. Some features are common to both GP and GE. Such features include the control models, evaluation, selection methods and termination criteria. These features have been explained in chapter 2 and are not explained again in this chapter.

3.3 A Grammar in Grammatical Evolution

Grammars provide a means of building up a complex structure from small building blocks [1]. A grammar consists of a set of non-terminals, a set of terminals and a list of production rules. One of the non-terminals must be designated as the start symbol. For example, $S \rightarrow aSb, \quad S \rightarrow \varepsilon$ is a grammar for the language $a^n b^n$ where n is an integer and ε is an empty string. The language defines the combinations of a's and b's such that (1) the number of a's is equal to the number of b's and (2) no b must precede an a. There is one non-terminal, namely, S. Also, there are two terminals, namely, a and b, and two production rules in the example. A production rule is made up of terminals and non-terminals. It allows the non-terminal at the left hand side to be replaced by the right hand side of the production rule.

In the context of GE, terminals are the elements that can appear in the program produced by GE. Thus GE terminals include the operators and the values they operate on. Examples are, *, /, +, 1, x and y. Non-terminals are the elements expressed as a composition of terminals. Each non-terminal can be expanded into one or more terminals. GE uses a Context-Free Grammar (CFG) and the Backus-Naur Form[5] (BNF) notation for the CFG. In the BNF notation, the non-terminals are enclosed inside the symbol < > while the production rules that have the same non-terminal at the left hand side are separated by the symbol "|".

A symbolic regression problem is used to explain how GE represents a program. The problem involves evolving a function that fits a given set of points. Let the start symbol be $< expr >$. The non-terminals is the set $N = \{expr, op, pre\_op, var, digit\}$ and the terminals, set $T = \{\text{X}, *, -, +, /, 1.0, (,), cos\}$. Then, the production rules P are represented as:

$$< expr > ::= < expr >< op >< expr > \quad (0)$$
$$\mid \; < pre\_op > (< expr >) \quad (1)$$
$$\mid \; < var > \quad (2)$$

$$< op > \; ::= \; * \quad (0)$$

[5] Also known as Backus Normal Form

$$| - \quad (1)$$
$$| + \quad (2)$$
$$| / \quad (3)$$

$$< pre\_op > ::= cos \quad (0)$$

$$< var > ::= X \quad (0)$$
$$| < digit > \quad (1)$$

$$< digit > ::= 0 \quad (0)$$
$$| 1 \quad (1)$$
$$| 2 \quad (2)$$

The number of production rules, say p_n, for each of the non-terminals is 3, 4, 1, 2 and 3 respectively. The production rules are numbered from 0 to $p_n - 1$.

Like the closure properties [4] and typing [20] introduced in GP, a grammar is used to ensure that the programs generated by GE are syntactically correct. It specifies only the possible combinations of the elements of the language [1]. Aside from ensuring that syntactically correct programs are generated, GE uses grammar to reduce the search space. By structuring the grammar such that certain functions are not allowed to take certain terminals as arguments, the grammar can be used to obtain a better solution for the problem at hand. It provides an easy platform to incorporate domain knowledge of the problem. The production rules can be structured in a way that facilitates the combinations of terminals in prefix or postfix notation. For example, $< expr > ::= < op >< expr >< expr >$ is structured in prefix notation while $< expr > ::= < expr >< expr >< op >$ is structured in postfix notation. Depending on the problem domain, the terminals could be the constants, variables, operators and control statements typical of a particular programming language. For example, for, {, }, int, $boolean$, $else$, and if have valid meanings in Java and as such, programs generated by GE as a valid combination of these terminals can be compiled using Java compiler.

3.4 Genotype Representation

As mentioned earlier, the GE represents a genotype as a binary string. A binary string is made up of one or more bits. Examples of binary strings include 001, 101 and 0000101 having the denary equivalent of 1, 5 and 5 respectively. In GE, a bit is called an allele while 8

alleles form a codon. Several codons are grouped to form a chromosome which is the genotype. An example of a chromosome is given in Figure 3.1.

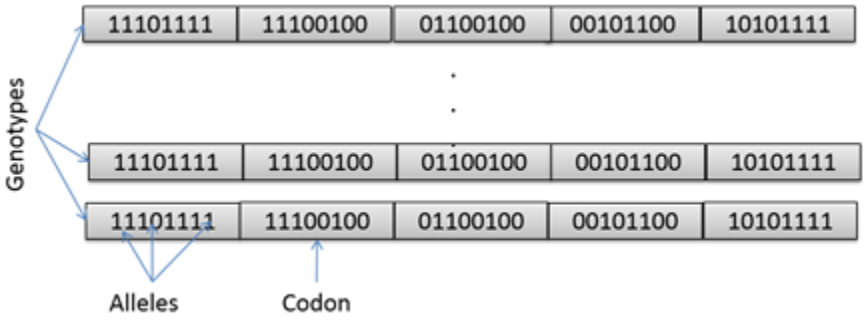


Figure 3.1. A binary representation of chromosomes

One problem with a binary representation is low locality [37, 38]. Locality is a term used to describe how a change in the genotype affects the phenotype. A small change in the genotype that results in a small change in the phenotype is known as high locality. High locality enables the search to effectively explore the neighbourhood of the current solution. Conversely, if a small change in the genotype results in a big change in the phenotype, the algorithm is said to have a low locality and, as such, likely to perform a random search.

Figure 3.2. An integer representation of a chromosome

An alternative representation is the use of integers rather than a binary string [39]. Unlike the binary representation that groups alleles to form a codon, each integer is a codon. Figure 3.2 illustrates an integer representation of a chromosome. One advantage of the integer representation is that it eliminates time spent in converting the codons from binary to integer.

3.5 Initial Population Generation

The chromosomes in the initial population are randomly created. Parameters which must be set before creating the initial population include the initial codon length which specifies the number of codons in a chromosome, the allele length which specifies the number of bits in each codon and the population size. As with GP, the choice of parameters has an effect on the search algorithm. Thus there is a need to perform trial runs and choose the parameters that provide the best result.

3.6 Mapping from the Genotype to the Phenotype

In the standard GE [1, 40], the mapping process starts by firstly converting the binary strings to denary values. Figure 3.3 shows a comparison between the mapping process in GE and a mapping process in natural evolution.

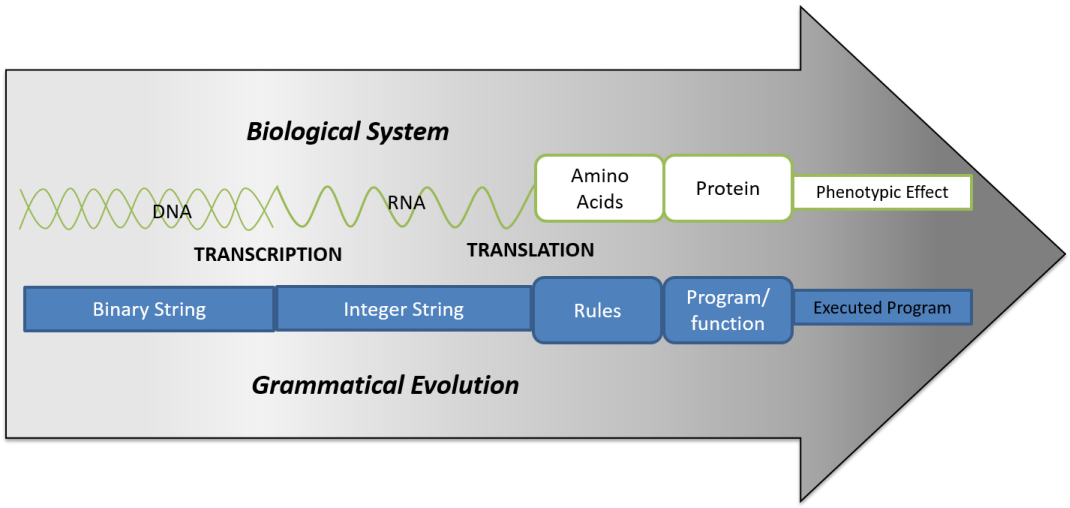


Figure 3.3 A comparison of the mapping process between the grammatical evolution and a biological system.

The chromosome is interpreted codon by codon. Each codon is converted to a denary value. An example of a chromosome with each codon converted to an integer equivalent is given in Figure 3.4. The mapping process starts from the first denary value, counting from the left. The modulus of this value and the number of production rules for the start symbol is taken. The result is an integer which corresponds to one of the production rules for the start symbol. If the production rule contains at least one non-terminal on the right hand side, the leftmost non-terminal is expanded first. The next available denary value is used. Again the modulus of the value and the number of production rules for the non-terminal is used to determine the production rule which may contain another non-terminal to be expanded. This process continues until all the non-terminals have been expanded.

234	98	2	13	74	36	56	...	77	9	3

Figure 3.4 An integer equivalent of a chromosome

A derivation tree is used to visualize the mapping process. It shows all the non-terminals that have been expanded and the result of their expansion. The GE mapping process corresponds

to a depth first expansion. This implies that the leftmost non-terminal in a derivation tree is expanded first.

To illustrate the mapping process, the chromosome shown in Figure 3.4 is mapped to the BNF grammar defined in section 3.4.1 as follows. The first codon value is 234 and the number of production rules for the start symbol is 3. The modulus of 234 and 3 is 0. This indicates that the right hand side of the first production rule, i.e., $< expr >< op >< expr >$ will be expanded next. The next step is to expand the leftmost non-terminal, namely, $< expr >$.

Again, there are 3 production rules for $< expr >$. The next codon value is 98. A production rule is therefore chosen using 98 mod 3 which results in 2. This is production rule number 2 for the non-terminal $< expr >$. Thus the expression so far becomes $< var >< op >< expr >$.

Continuing in the same manner, a production rule is chosen using 2 mod 2 which results in 0. This is production rule number 0 for the non-terminal $< var >$. The expression is transformed to $\mathrm{X} < op >< expr >$. Next, $< op >$ is expanded to obtain $\mathrm{X} - < expr >$. There is one non-terminal, $< expr >$, in the expression. The non-terminal is expanded and the expression transformed to $\mathrm{X} - < var >$. The expression is finally transformed to X – X after expanding the non-terminal $< var >$. Figure 3.5 illustrates the mapping process described above. It shows how the mapping process corresponds to a depth first expansion.

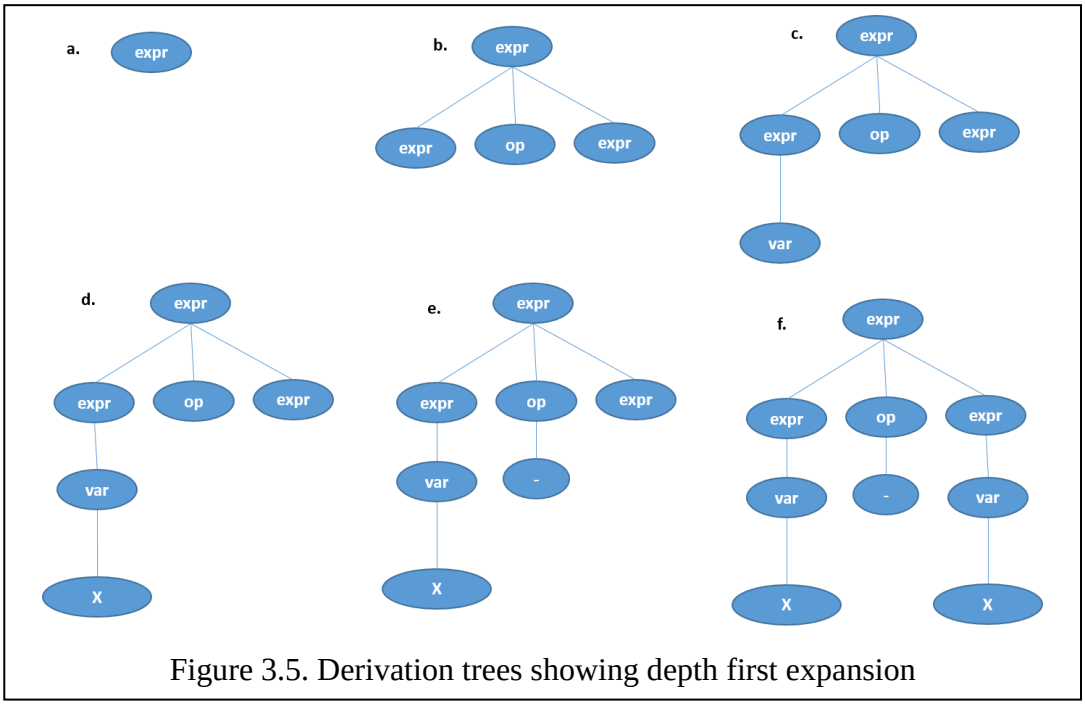


Figure 3.5. Derivation trees showing depth first expansion

As in natural evolution, GE exhibits a many-to-one mapping i.e., more than one genotype can map to the same program. For example, if the first six denary values of the chromosome in Figure 3.4 are changed to 57, 140, 90, 33, 197 and 244, the mapping process will still result in the same program obtained. Thus the variety in the genotypic space does not map to the same variety in the phenotypic space.

3.7 Wrapping

There is a possibility that the last codon in the chromosome will be used while there are still non-terminals to be expanded. GE uses wrapping to cater for such a problem. Wrapping is the reuse of the codons in the chromosome, from the left to the right, during the mapping process. A maximum number of wraps must be set. If the maximum number of wraps is reached and the mapping process is not completed, the non-terminals will be replaced with terminals. The individual is assigned the worst possible fitness [1]. The worst possible fitness is dependent on the problem domain. In the symbolic regression problem, the worst possible fitness is the highest integer.

Each codon could recursively map to the same production rule i.e., recursive mapping. For example, assume that all the codon values of a chromosome are even numbers and the production rules for the start symbol are as given below:

$$< expr > ::= < expr > < op > < expr > \quad (0)$$

$$| \quad < var >, \quad (1)$$

The modulus of any even number and 2 is 0. Thus, each codon in the chromosome maps to the production rule number 0. If recursive mapping occurs, the non-terminals will be replaced with terminals. Also, the individual is assigned the worst possible fitness [1]. One of the disadvantages of this is that the population might contain many individuals with worst possible fitness as a result of incomplete mapping.

3.8 Evaluation

As in GP, each individual is evaluated to know how close they are to the required program. Evaluation has been explained in chapter 2 section 2.8.

3.9 Selection

Selection methods are used to choose individuals that will participate in creating the next generation. The selection methods that can be used by GE have been explained in chapter 2, section 2.9.

3.10 Genetic Operators

As with GP, genetic operators are applied for the purpose of regeneration. These operators are applied in the hope that the individuals produced will be fitter than their parents. Whereas GP applies genetic operators to the individuals represented as parse trees, GE applies genetic operators to the chromosomes represented as binary strings. The crossover and mutation operators [1] are the commonly used genetic operators in GE. Unlike GP which uses a genetic operator application rate to determine the number of individuals that will be created using the operator, GE uses a probability to determine if a genetic operator will be applied to the chromosome. For example, assume the crossover probability to be 70. For each chromosome in the population, if a randomly generated number in the range 0 and 100 is less than 70, the operator is applied. The operators are explained in the following sections.

3.10.1 Crossover

A number of crossover operators have been investigated by O'Neill and Ryan [1, 41]. These include one-point, two-point and homologous crossover operators. GE uses the one-point crossover [1] which is explained first in this section.

As with GP, the crossover operator is applied to two parents. One-point crossover operates as follows. Two parents are selected using one of the selection methods. Two crossover points, one from each of the parents, are randomly selected. These points are denoted as p1 and p2 as shown in Figure 3.6. The binary string from the beginning of a chromosome to a crossover point is denoted as the head while the rest is denoted as the tail. The tail of parent1 is appended to the head of parent2 and the tail of parent2 to the head of parent1 to produce two offspring.

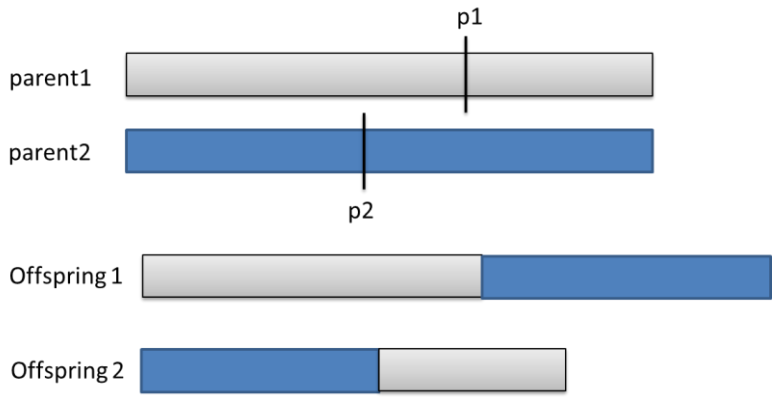


Figure 3.6. A one point variable length crossover

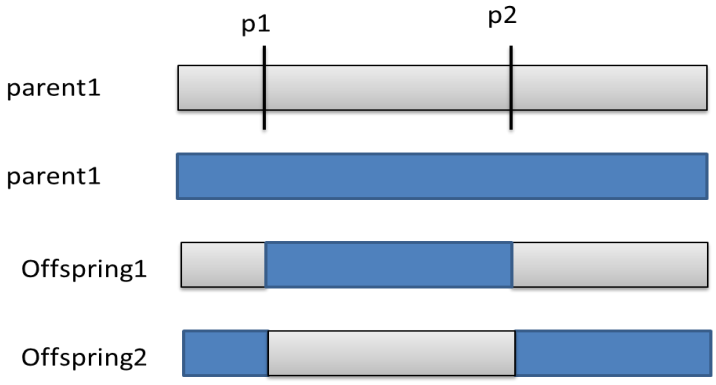


Figure 3.7. A Two point crossover

Instead of swapping the binary string located from a selected point to the end of a chromosome, the two-point crossover randomly selects two points, say point p1 and p2 as shown in Figure 3.7. The binary string located from p1 to p2 is swapped between the two selected parents.

Homologous crossover stores the number of each production rule expanded during the mapping process and uses the number to determine the crossover point. To illustrate homologous crossover, consider the two parents shown in Figure 3.8, the production rules

chosen during the mapping process are given as H1 and H2 for parent1 and parent2 respectively. Recall that if the mapping process uses parent1 and the grammar in section 3.3.1, the start symbol $< expr >$ is expanded by taking the modulus of 234 and 3. The result is 0 which is the number of the production rule stored at the first index of H1. Other values of H1 and H2 are obtained in a similar way. A region of similarity is where the number of rules selected is the same in H1 and H2. Where these numbers are different is called a region of

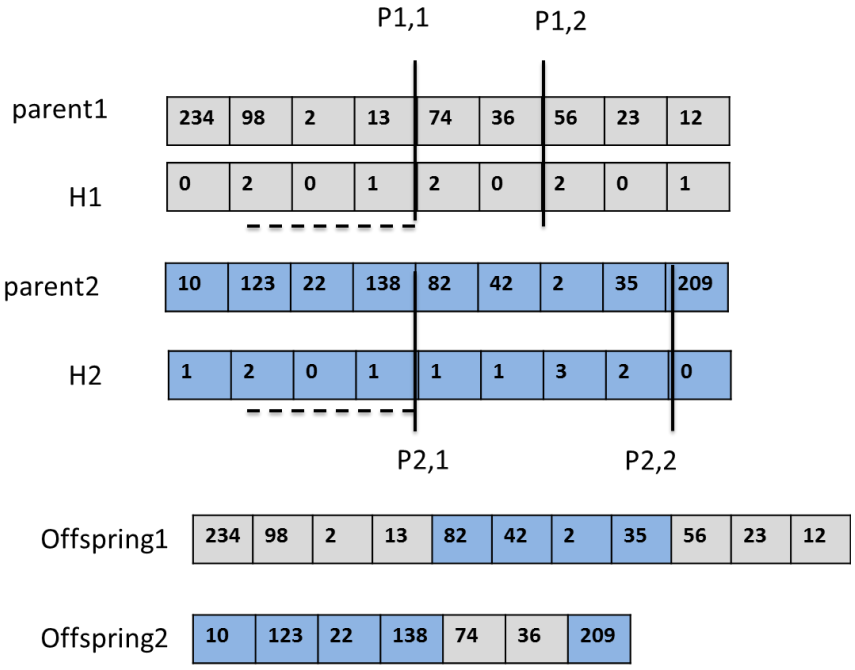


Figure 3.8. A homologous crossover

dissimilarity. The region of similarity is denoted by a dotted line in Figure 3.8. The homologous crossover is applied to parent1 and parent2 as follows: Two points, one from each parent, are randomly selected from a region of similarity. These points are the same in both chromosomes and are denoted in Figure 3.8 as p1,1 and p2,1. Two more points, one from each parent, say p1,2 and p2,2 are randomly chosen from a region of dissimilarity. For each parent, the codon values between the two points are copied to the other parent. The homologous crossover uses more memory and increases the runtime because it requires the history of the rules chosen be stored. Also, it is not clear what will happen if there is no region of similarity between the two selected chromosomes.

Generally, a crossover operator aims at searching the neighbourhood of the current solution in the hope for a better solution. It is a local search operator which promotes convergence. GE researchers have also criticized the crossover operator for its destructive effects [41, 42]. In the context of GE, O'Neill and Ryan [41, 42] have defended the one point crossover as

being less destructive and more efficient than the two point and homologous crossover. They state that a few crossover points decreases the destructive effect of the crossover operator [41, 42].

3.10.2 Mutation

Whereas GP uses a selection method to select the individual the mutation operator will be applied to, in GE, the mutation operator is applied to the offspring produced by the crossover operator. GE uses a bit mutation operator [1]. The operator checks each bit locus and flips the bit if a randomly generated probability is less than a preset mutation probability. A bit flip implies that the bit is changed to 1 if it is 0 and vice versa. The operator aims at increasing the diversity of the population by taking the search to a new area of the search space[6].

The derivation tree structure has been used to examine the level of locality introduced by the mutation operator [43, 44]. Byrne *et al*. [43, 44] categorize the mutation operator as a nodal mutation or a structural mutation. Nodal mutation changes a single node in a derivation tree while structural mutation changes the structure of the derivation tree. An example of nodal mutation is replacing a node with a node of the same arity. An example of structural mutation is replacing a node with a node of different arity. Whereas nodal mutation searches more of the neighbourhood of the candidate solution, structural mutation searches new areas of the search space. Hence nodal mutation is a high locality mutation while structural mutation is a low locality mutation. Ensuring a balance between nodal mutation and structural mutation is beneficial to the search. This could be achieved by specifying the percentage of the mutation operator that must result in nodal or structural mutation. However, this is a time consuming process because the offspring needs to be mapped to production rules to determine if the mutation is nodal or structural.

Like the crossover operator, the mutation operator could be destructive to a good building block. Castle and Johnson [45] studied the effect of the mutation operator in terms the mutation point. In order to determine the effect of mutation, the fitness of each individual was calculated before and after the mutation is performed. Also, the mutation points were recorded. Castle and Johnson [45] report that the mutation operator that occurs at the beginning of a chromosome is more destructive than those that occur at another place in the chromosome. The mutation operator performs more exploration if the mutation point is at the

[6] This is not always the case as there is no guarantee that the mutated string will be different from the newly created string.

beginning of the chromosome [45]. Whereas the crossover operator aims at exploiting the neighbourhood of the candidate solution, the mutation operator aims at exploring the search space. As with GP, a high probability of the mutation operator may slow down convergence. Striking a balance between the probabilities of both the mutation and crossover operators will balance the exploration and exploitation ability of the search algorithm.

3.11 Termination Criteria

As with GP, the GE algorithm terminates when a solution has been found or a specified number of generations has been reached. The termination criteria has been explained in chapter 2, section 2.11.

3.12 Bloat in GE

Whereas in GP, bloat occurs in the parse trees, in GE bloat occurs in the chromosome. Bloat is the excessive increase in the size of the chromosome. Like in GP, introns lead to bloat. Introns are part of the chromosome that are not used during the mapping process. During this process, each gene in the chromosome is converted to a denary value which is mapped to a production rule of the grammar. Some genes are not used if all the non-terminals have been expanded before the last gene in the chromosome. These genes that are not used are introns. Whereas the role of introns in the prevention of the destructive effects of the crossover operator has been highly recognised in GP, an intron in GE plays a lesser role in the prevention of the destructive effect of the crossover operator. This is because introns can only occur at the end of a chromosome.

Harper and Blair [46] use parsimony pressure to control bloat. Parsimony pressure is the penalization of the chromosomes that are large in size. During selection, if the size of a chromosome, say chromo-A, exceeds 3000 bits and another chromosome, say chromo-B has the same fitness as chromo-A, chromo-B is selected. The study applies this rule to 5% of the population and hence controls bloat.

3.13 Modularization in GE

As with GP, modularization has been introduced in GE [47]. While the use of ADFs and other forms of modularization are becoming popular in GP, they are rarely used in GE. The most simple ADF used in GE is presented in Ryan [47]. In the study, there is only one production rule from the start symbol. The production rule has two non-terminals. The first must be expanded to obtain the main program while the second must be expanded to obtain an ADF. The ADF is called from the main program if a terminal representing the ADF is

obtained during the expansion of the non-terminal for the main program. For a symbolic regression problem, an example of a production rule for the start symbol is $< expr > :: = < code >< adfcode >$. The non-terminal $< code >$ is expanded to obtain the main program while the non-terminal $< adfcode >$ is expanded to obtain the ADF.

Harper and Blair [46] proposed Dynamically Defined Functions (DDFs) as a concept of modularity. Whereas the number of ADFs a program must contain is preset in Ryan [47], Harper and Blair [46] use production rules to decide the number of DDFs a program must contain. A non-terminal is introduced which must firstly be expanded. The result specifies whether or not DDFs should be used. If the latter is the case, the expansion terminates otherwise the result is further expanded to decide the number of DDFs that must be included and the number of parameters each of the DDFs must take. Also, Hemberg *et al*. [48] proposed a concept of modularity called $(GE)^2$. While Harper and Blair [46] use one chromosome during the mapping process to obtain a program that has a main program and one or more DDFs, Hemberg *et al*. [48] use two separate chromosomes during mapping process. The first is used to expand the terminals that determine the number of ADFs a program must have and the number of parameters each of the ADFs must take. The second is used to generate the following: The main program, the body of the functions and a call to one or more functions. DDF and $(GE)^2$ are more difficult to implement compared to ADFs implemented by Ryan. $(GE)^2$ uses more memory than DDF because it makes use of two separate chromosomes. Whereas the use of ADFs degrades the GP performance on a simple problem [7], Hemberg *et al*. [48] state that the performance of GE with ADFs is dependent on the size and type of problems. Thus the implications of ADFs in GE remain open for more research.

3.14 Benefits of GE

Programs can be generated in a particular language: GE uses a genotype–phenotype distinction that allows for generalized encoding. A grammar defines a particular language which allows programs to be generated in that language.

Maintaining of genetic diversity: In GE which uses 8-bit codon, there is a many-to-one mapping. This implies that different genotypes can represent the same phenotype thereby maintaining genetic diversity within the population. If one of the genotypes is eliminated from the population, the phenotype can still be represented.

Generated programs are easy to understand: Because bloat does not occur in the programs generated by GE, they are easy to understand.

The search can be easily biased to obtain a better success rate: By restricting the syntax of the grammar, GE is focused on certain areas of the search space. This can increase the success rate of the GE algorithm.

Challenges posed by typing and closure in GP are avoided: The valid syntactical structures are specified in the grammar. Thus, the use of a grammar overcomes the challenges posed by typing and closure in GP.

3.15 Chapter Summary

This chapter described GE using the generational control model. The variable-length binary representation used by GE was described. Each chromosome is composed of 8-bit codons and is randomly generated during initial population generation. Each codon in the genotype has to be mapped to an integer which is consequently mapped to a grammar defined by production rules to obtain a program. Having reviewed both GP in chapter 2 and GE in this chapter, the next chapter will examine how GP and GE have been used for the purpose of automatic object-oriented programming.

CHAPTER 4: GP AND AUTOMATIC OOP

4.1 Introduction

The studies on genetic programming and variations of genetic programming have taken an analogy from computer programming to perform optimization. In some instances, the scalability of the approach is improved in the process. Scalability in the context of this dissertation is the ability to increase the success rate of GP when applied to a complex problem. Automatic programming, also in the context of this dissertation, is automating the process of writing code. Programming languages adhere to different programming paradigms, namely, functional, declarative, procedural and object-oriented. GP has been used to automatically evolve programs that conform to these paradigms. Research evaluating GP for automatic object-oriented programming has evolved classes but did not evolve programs that use the evolved classes. Also, there has been no work done that investigates GE for automatic object-oriented programming. Hence, the aim of this dissertation is to evaluate GP and GE for automatic object-oriented programming. This chapter, firstly reviews studies on GP and GE which are relevant to automatic object-oriented programing. It then provides an analysis of the reviewed studies and justifications for the GP and GE processes that will be used in this study.

Firstly, section 4.2 provides an overview of programming paradigms and how GP has benefited from these paradigms. Section 4.3 looks at studies using genetic programming for automatic procedural programming. Section 4.4 provides an overview of research using genetic programming for automatic object-oriented programming. Research that takes an analogy from object-oriented programming to improve GP scalability is presented in section 4.5. Section 4.6 presents an analysis of the studies reviewed and justifications for what will be done in this dissertation. Section 4.7 presents the chapter summary.

4.2 GP and Programming Paradigm

Automatic programming commenced as an effort towards aiding the manual programming process which is time consuming and challenging for a complex problem. GP has been used to automatically evolve programs that conform to different programming paradigms. According to Reinfelds [49], programming paradigms include functional, declarative, procedural and object-oriented paradigms.

Functional programs consist of functions which are called recursively. A value obtained from a function forms an input to another function until the desired value is obtained [50]. As specified in section 2.2 of chapter 2, the first implementation of GP evolved functional programs in Lisp. Declarative programming declares knowledge by providing rules for program execution [51]. These rules include conditions and stopping criteria. A procedural program is a sequence of instructions that are executed from top to bottom. Procedural programming determines the data values that will be used, represents the values by associating them with storage and specifies the step-by-step sequence of transforming the data stored to a desired output. As GP advances, researchers started looking at evolving procedural programs. This allows for iteration and the use of memory in GP which have been explained in chapter 2, section 2.12. However, procedural programs still have some limitations which are mostly the reusability and scalability of programs [52]. These limitations were significantly improved with the introduction of object-oriented programming, thereby allowing for creation of programs as objects that can be reused. The benefits of object-oriented programming lead to two categories of research on GP and object-oriented programing. The first takes an analogy from object-oriented programming to improve GP scalability while the second looks at using GP for automatic object-oriented programming.

4.3 GP for Automatic Procedural Programming

With the introduction of different programming paradigms, researchers started looking at using GP for automatic procedural programming [11]. The study conducted by Pillay [11] uses GP to induce solution algorithms to novice procedural programming problems. The GP system generates each program, represented as a parse tree, in an internal representation language which was defined to facilitate the evolution of language independent programs. Also, Igwe and Pillay [3] use GP for automatic generation of solution algorithms for problems involving memory, iteration, conditional statements and modularization. The study uses the same representation used by Pillay [11].

Pillay and Chalmers [53] implement a GP system that takes as input a problem specification together with an Object-Oriented Design (OOD) generated by a rule-based expert system. The rule-based expert system determines the possible classes, subclass, superclass and the methods required. The GP algorithm then uses this information to evolve the required methods sequentially. The GP algorithm implemented in the study uses a generational control model and represents each individual as a parse tree. The grow method was used to generate

the individuals in the initial population while crossover and mutation were used for regeneration. The evolved methods are added to the function set when evolving subsequent methods. This can be seen as a method call from another method.

The studies presented in this section are an indication that evolutionary algorithms, specifically GP, can be used to generate code which includes functions and algorithms. However, none of the studies shows how GP can be used to generate and reuse a class. This is the main aim of object-oriented programming.

4.4 Automatic Object-Oriented Genetic Programming

In order to take advantages of object-oriented paradigm, Bruce investigates the sequential and simultaneous induction of methods in OOGP [8, 54]. The studies conducted by Bruce are among the first investigation done in this domain. Each individual is a chromosome consisting of one or more genes. Each gene is a parse tree representing a method. Whereas the structure of the chromosome is the same as that used by Bruce to implement ADFs, the functioning of the chromosome differs. The fitness of each method in the chromosome is evaluated to obtain a fitness value for the method. This is not so for the ADFs. The fitness of the ADFs are evaluated by evaluating the fitness of the main program. Bruce uses tournament selection, a steady-state control model and a population size of 1000 throughout the studies. The studies [8, 54] tested GP for the induction of methods for the stack, queue and priority queue. For each data structure, five methods were evolved. Strongly typed genetic programming proposed by Montana [20] was used as a means of enforcing syntactically correct programs. Typing has been explained in chapter 2, section 2.12.1.

The studies conducted by Bruce show that GP can be used to evolve classes but did not investigate evolving programs that use the classes.

4.5 Object-Oriented Genetic Programming for GP Scalability

In order to demonstrate that object-oriented programming can be used to improve GP scalability, Langdon [6] investigates the induction of methods for three Abstract Data Types (ADTs), namely, the stack, queue and list. Langdon uses a similar representation as Bruce. Again, each individual is a chromosome consisting of one or more genes. Each gene is a parse tree which represents a method. Breaking down the programs into methods increases the GP scalability. Like Bruce's work, tournament selection and a steady-state control model were used. Both Bruce [8, 54] and Langdon [6] evolve five methods each for the stack ADT.

Crossover was defined to operate on one method in an individual each time the operator is called. A method to be operated on is chosen with a probability of 1/5.

In the study, the GP success rate is higher for the stack data structure than it is for the queue and list data structures. High-level functions and terminals were defined to improve the GP success rate for the queue and list data structures. The high-level functions include the *set_aux* which takes a single integer argument and sets a named memory, namely, *aux,* to the value this argument evaluates to. The high-level terminals include the *dec_aux* and *inc_aux.* Both these terminals are regarded as high-level terminals because they are defined and implemented closely to the way a human can understand. They perform functions on another terminal which is *aux*. the *dec_aux* decrements the value of *aux* while *inc_aux* increments the value. The study also implemented ADFs with a new concept called pass-by-reference explained in chapter 2, section 2.12. If *aux* is incremented within the body of an ADF, this increment is retained in the main program.

By using the evolved ADTs to generate solutions to the Dyck language and reverse polish notation problems, the study shows that the evolved ADTs can be beneficial to GP. The success rates of GP with and without the use of ADTs were compared. Langdon shows that the use of ADTs improves GP scalability.

4.6 Analysis and Justifications

GE has a number of features which automatic programming can benefit from. One of these features is the ability to generate programs in a particular language defined by the grammar [1]. Despite these features, GP has attracted more interest in automatic programming. As mentioned earlier, there has been no work done that investigates GE for automatic object-oriented programming. This is one of the objectives this dissertation addresses.

With the exception of the studies conducted by Bruce [8, 54] which focused on the object-oriented programming problems, previous work aimed at using GP for automatic programming focused on procedural programming problems. Bruce tested GP for the automatic induction of ADTs, namely, the stack, queue and priority queue, but did not show how the data structures can be used by another program. This defeats one of the main aims of object-oriented programming, to use an evolved class. Also, the generated programs consist of high level functions which can be difficult to understand. It will be beneficial if the programs can be converted to a programming language. The study conducted by Langdon [6] has focused on GP scalability rather than testing GP with the aim of automatic programming.

The study, however, introduced the pass-by-reference which can be beneficial to automatic object-oriented programming.

Like the study conducted by Bruce [8], this dissertation focuses on the object-oriented programming paradigm. An object-oriented program is made up of different classes. One or more of the classes will have a main method to run the application. In object-oriented programming, an instance of a class is an object which can be used by another program. Whereas Bruce did not investigate evolving the programs that use the evolved classes, this will be investigated in this study. This study will also show that the evolved programs can be converted to a programming language.

The object-oriented programming problem that will be used to test GP and GE involves two classes, one with the driver program and the Abstract Data Type (ADT) class. There are three rationales behind the use of ADTs as an application domain. The first is that ADTs have been used in computer science as classical programming problems to demonstrate the object-oriented programming concepts [55]. The second is to allow comparison with other work since ADTs have been used as an application domain to evaluate the scalability of GP [6]. The third is that ADTs allow for classification of the object-oriented programming problems into different categories based on their difficulty level. This study considers three possible means of using a produced class. They are:

i. Represent each individual as a chromosome containing genes. One gene represents the driver program and each of the others represents a method of the class to be used by the driver program. Run the approach. As the generations progress, add each evolved class method in the function or terminal set of the driver program.
ii. Run the approach for evolving the class and write the solution to a file. Run the approach for evolving programs that use the evolved class. During this process, read the evolved class methods and assign each as a separate ADF which can be referred to by the method's name. Call the ADFs in the driver program as required.
iii. Run the approach for evolving the class. If a solution for the class is found, add each evolved method to the function set of the driver program if the method takes one or more arguments. If the method takes no argument, add it to the terminal set of the driver program. Run the approach for evolving the driver program.

The advantage of (i) is that it reduces coding. The major disadvantage is that there is no guarantee that a solution for the class would be evolved for the particular seed provided.

Hence the program that uses the class may never be evolved. The advantage of (ii) is a clear separation of the program for evolving the class and the driver program. However, reading each ADF from a file makes the program more complicated. (iii) has the same advantage as (ii). Adding to this advantage, the evolved methods are simply added to the function or terminal set of the driver program.

Considering the advantages and disadvantages of (i), (ii) and (iii), the GP and GE approaches will make use of (iii).

4.6.1 Analysis of GP for Automatic OOP

4.6.1.1 Program representation

Intuitively, it is easier to induce methods sequentially than simultaneously [8]. This is because it is easier for an algorithm to concentrate on one task at a time rather than trying to achieve different tasks at a time. Nevertheless, there are a number of reasons why one would want an automatic programming system to simultaneously induce methods of a class. In sequential evolution, the behaviour of a method has less influence or benefit to another method. In the study conducted by Langdon, function calls are allowed between methods in order to aid the evolution of methods requiring the functions provided by another method.

This study will simultaneously induce methods of a class. Each individual will be represented as a chromosome consisting of parse trees. Each parse tree will represent a method.

4.6.1.2 Control Model

Unless otherwise stated, GP that will be implemented will make use of the generational control model discussed in chapter 2. The generational control model is easy to implement, has a distinct generations and generally works well with GP.

4.6.1.3 Initial Population Generation

The initial population of a GP algorithm is generated using one of the three methods, namely, the full, grow and ramped half-and-half, described in chapter 2, section 2.7. In the study conducted by Pillay [11], the choice of the best initial population generation method is problem dependent. Therefore, the three choices for generating the initial population will be provided for the GP algorithm. Trial runs will be performed to determine the choice that works best for each problem. For each problem at hand, the best choice will be used for the final run of the GP algorithm.

The initial population of a GP algorithm is randomly generated. If the search space is large, randomly generating the initial population can be ineffective in producing programs that can be improved. In order to direct the search in the initial population, a greedy approach, namely, Greedy OOGP (GOOGP), will be introduced. This approach will be tested as an alternative to randomly generating the initial population. This dissertation hypothesizes that OOGP approach to automatic programming can be improved using the GOOGP approach. This hypothesis will be tested by comparing the performance of GOOGP and OOGP for each problem on which the approaches are tested.

4.6.1.4 Selection Method

Tournament selection will be used for the OOGP algorithm. This selection method is easy to implement and uses less memory than the fitness proportionate selection. Also, GP runtime is reduced when using tournament selection than when using fitness proportionate selection. This is because tournament selection requires evaluating only the individuals participating in the tournament while fitness proportionate selection requires evaluating the entire population.

4.6.1.5 Genetic Operators

Like in Langdon [6], genetic operators will be adapted to the representation that will be used. Whereas in Langdon [6], crossover are applied to the two selected parse trees representing the methods , in this study, crossover will be applied to the two selected parse trees as well as the chromosomes. This will allow methods to be exchanged between two parents. Methods exchange between two parents can improve the search. For instance, assume the first parent consist of five methods in which three are correctly induced. If the remaining two methods are correctly induced in the second parent, exchanging these two methods between the parents will result in a solution being found. This study will use a probability for each method position to determine whether or not the method will be exchanged.

4.6.1.6 Advanced features

Advanced features of GP which include iteration, typing, use of memory and modularization will be used.

The *for* operator is apt for looping over a given range while the *for_loop1* operator is apt for looping n a number of times (see section 2.12.3 in chapter 2). The *while* operator is apt for testing a condition before entering a loop. Hence, depending on the problem, one of the iterative operators, namely, the *while, for and for_loop1,* will be used. For each instance of the *for_loop1* operator, counter and iteration variables will be introduced. Both these

variables will be introduced into the terminal set when creating the second argument of the *for_loop1* operator.

Like in Langdon [6], the indexed memory operators, namely, the *read* and *write* will be used for problems requiring indexed memory.

Given the advantages of the ADFs implemented by Bruce [8], this dissertation will implement ADFs as a separate parse tree in the chromosome. Function calls between ADFs will be allowed but recursive calls will not be allowed. This will be done to prevent infinite recursion. Also, the ADFs will use the concept of pass-by-reference introduced by Langdon [6]. This has been shown to be beneficial to OOGP [6].

4.6.2 Analysis of GE for Automatic OOP

Since this is the first study using GE for object-oriented programming, the study will draw from OOGP and GE. The approach will be referred to as the Object-Oriented Grammatical Evolution (OOGE).

4.6.2.1 Program representation

As with OOGP, the methods of a class will be simultaneously induced. Each individual in the population will be represented as a chromosome consisting of genes. Each gene will be binary strings representing a method.

4.6.2.2 Control Model

OOGE that will be implemented will make use of the generational control model discussed in chapter 2.

4.6.2.3 Initial Population Generation

Like in GE, the individuals in the initial population will be randomly generated as discussed in section 3.5 of chapter 3.

4.6.2.4 Selection Method

The OOGE approach will make use of the tournament selection for the same reason as specified in section 4.6.1.4.

4.6.2.5 Genetic Operators

As with OOGP, genetic operators will be adapted to the representation that will be used. Crossover will be performed on the genes as well as on the chromosomes.

4.6.2.6 Advanced features

As with OOGP, iteration, use of memory and modularization will be used. The memory manipulation operators and the iterative operators will be the same as those used for OOGP. Depending on the problem domain, one or more of these operators will be added as terminals in the grammar.

OOGE will make use of ADFs introduced by Ryan [47] because it is simple to implement. It requires specifying how many ADFs and the number of arguments each ADF will take. This study, however, will make use of a separate chromosome for each ADF. This will allow genetic operators to change the ADFs without any change in the main program. As with OOGP, function calls between ADFs will be allowed but recursive calls will not be allowed. Also, the ADFs will use the concept of pass-by-reference introduced by Langdon [6].

4.7 Chapter Summary

Rather than evaluating GP with the aim of automatic object-oriented programming, most researchers aim at improving scalability of GP. Also, the studies that evaluate GP with the aim of automatic programming evolved classes but did not investigate the induction of programs using these classes. No work has been done that evaluates GE for automatic object-oriented programming. From the studies reviewed, more work needs to be done to measure the ability of both GP and GE to be used for automatic object-oriented programing. The next chapter will discuss the methodology that will be used to achieve this.

CHAPTER 5: METHODOLOGY

5.1 Introduction

This chapter presents the methodology which will be used to achieve the objectives defined in chapter 1.

Section 5.2 describes four types of research methodologies and identifies the methodology that is most appropriate for this study. Section 5.3 restates the objectives and explains how the identified methodology will be used to achieve the objectives. The performance evaluation and statistical testing are discussed in section 5.4. Section 5.5 describes the problem domain used to evaluate genetic programming and grammatical evolution while the technical specifications are described in section 5.6. Lastly, section 5.7 summarizes the chapter.

5.2 Research Methodologies

According to Johnson [56], the four types of research methodologies commonly used in computer science are proof by demonstration, empiricism, mathematical proof techniques, and hermeneutics and observational studies.

Proof by demonstration involves developing a system, testing the performance of the system and iteratively refining the system until a desired result is obtained or no further improvement can be made. At each stage of the refinement, the reason for the failure of the system is identified and corrected. Empiricism is used to test the truthfulness of a specific hypothesis. A particular method must be devised and followed in order to test the hypothesis. Statistical analysis of the result of the test is conducted in order to prove or disprove the hypothesis. A mathematical proof involves formal reasoning to validate or disprove a hypothesis. It sometimes involves making some abstract mathematical assumptions. In observational studies, a prototype system is developed and used by trained personnel while observing and evaluating the working system.

The main aim of this study is to evaluate genetic programming and grammatical evolution for automatic object-oriented programming. The process of evaluating GP and GE for automatic object-oriented programming involves identifying the GP primitives and parameters as well as identifying the GE parameters and a grammar for a particular object-oriented programming problem. This is done by implementing a system with initial sets of primitives and parameters, and a grammar for the GE approach. The parameters are then tuned in an attempt

to obtain a system capable of producing code for a class and a program that uses the produced class. Thus, proof by demonstration is the most suitable methodology for this study. Once the initial system is developed, the rest of the proof by demonstration process involves two alternating phases, namely, the testing phase and the refinement phase. The testing phase involves performing runs. If no solution is found, the possible reason for not finding a solution must be identified. The refinement phase involves varying the parameters. The GP primitives and the grammar for the GE approach may also be changed. In addition to the primitives and parameters, it may also be necessary to change the features of the GP and GE algorithms such as the control model, the fitness function, selection method, and genetic operators. The following section describes how the proof by demonstration methodology is applied to achieve the objectives formulated for this study.

5.3 Achieving the Objectives using the Proof by Demonstration Methodology

The three objectives formulated for this study are as follows. The first is to evaluate genetic programming for automatic object-oriented programming while the second is to evaluate grammatical evolution for automatic object-oriented programming. As mentioned in section 4.6 of chapter 4, each object-oriented programming problem that will be used to evaluate GP and GE involves two classes, one with the driver program and the Abstract Data Type (ADT) class. The third is to compare the performance of GP and GE for automatic object-oriented programming. To achieve these objectives, the following will be done:

1. Develop and implement Object-Oriented Genetic Programming (OOGP) and Greedy Object-Oriented Genetic Programming (GOOGP).
2. Develop and implement Object-Oriented Grammatical Evolution (OOGE).
3. Test the ability of OOGP, GOOGP and OOGE to produce code for the stack, queue and list data structures described in section 5.5.1.
4. Using different random number generator seeds, perform thirty runs of OOGP, GOOGP and OOGE for the stack, queue, and list data structures. This is done because of the randomness associated with genetic programming. A solution may not be found in one run due to the random choices made.
5. For each of the OOGP , GOOGP and OOGE approaches, if at least one solution is not found for each of the stack, queue and list data structures, make one or more changes to the following:
    - The primitives (i.e., the elements of the internal representation language)
    - Fitness cases

- Fitness function
- The standard features in the algorithm such as genetic operators and selection methods.
- For OOGP and GOOGP:
  - The GP parameters such as the maximum initial tree depth, the tournament size, the application rate of the genetic operators and the maximum offspring depth.
- For OOGE:
  - The GE parameters such as the length of the allele and the application rate of the genetic operators. The grammar for the problem may as well be changed (if necessary).

6. For each OOGP, GOOGP and OOGE that finds at least one solution for the tested data structures, using different random number generator seeds, perform thirty runs to evolve the driver program. The programming problems are described in section 5.5.2.
7. If at least one driver program is not evolved in 6, make one or more changes as described in 5.
8. Compare and report on the performance of OOGP, GOOGP and OOGE.

5.4 Performance Evaluation and Statistical Testing

The performance evaluation of the approaches is based on the success rates, the runtimes and the average fitness of the approaches. For each approach, 30 runs will be performed. This will be done because a normal distribution is required to perform a statistical test and at least 30 samples are required to obtain a normal distribution. The success rate of an approach is the number of solutions found in the 30 runs of the approach. Thus, a 100% success rate means that 30 solutions were found in 30 runs, while a 60% success rate means that 18 solutions were found in 30 runs. The runtimes are important for two reasons. The first reason is to know how long it may take an automatic system to produce the required result. The second reason is to compare the amount of search needed by OOGP, GOOGP and OOGE to get to an optimum.

Statistical tests will be used to test whether or not the difference between the mean of the success rate, average runtime and average fitness of the approaches is significant. Table 5.1 shows the level of significance, critical values and decision rules used for the Z-tests.

Table 5.1. Z-test table showing level of significance, critical values and decision rules

P	Critical Value	Decision Rule
0.01	2.33	Reject H_o if Z > 2.33
0.05	1.64	Reject H_o if Z > 1.64
0.1	1.28	Reject H_o if Z > 1.28

5.5 Description of the Object-Oriented Programming Problems

This section describes the application domain used for testing the approaches. The ADTs are described in sections 5.5.1 while the programming problems that use the ADTs are described in section 5.5.2.

5.5.1 The Abstract Data Types (ADTs)

As mentioned in chapter 4, the three ADTs that will be used to evaluate GP and GE are the stack, queue and list. Like the study conducted by Bruce, the ADTs are array based.

The stack is the simplest and is also among the most important ADTs [55]. It is widely used in many applications. It is used by applications such as the Java compiler in evaluating arithmetic expressions. Other applications of the stack include storing the page-visited history in a web browser and keeping text changes such that the undo sequence in a text editor is possible [55].

Objects are inserted and removed from the stack according to the Last-In-First-Out (LIFO) principle [55]. Langdon [6] defines five basic stack operations as *MakeNull*, *Push*, *Top* (i.e. *Peek*), *Pop* and *Empty*. Their formal descriptions are listed in Table 5.2.

Table 5.2 Methods for the stack ADT

Methods	Function
makeNull()	Sets the pointer to the stack to -1. The return value is ignored.
push()	Push an element onto the stack. The return value is ignored.
peek()	Returns the topmost element on the stack.
pop()	Returns the topmost element on the stack, removes the element from the stack and decrements the pointer by 1.
empty()	Returns an integer less than zero if the stack is empty, otherwise it returns an integer greater or equal to zero.

Like the stack ADT, the queue is among the simplest and is also among the most important ADTs [55]. Also, queues are widely used in many applications. A typical example is to hold data that requires to be processed by a service provider [6]. The first element stored is processed first while the last element stored is processed last. Queues are useful in many applications. For example, an operating system holds a process in a queue. A process is an instance of a computer program that will be executed. Also customers queue for services in a bank.

Elements are inserted and removed from the queue according to the First-In-First-Out (FIFO) principle [55]. Langdon [6] defines five basic queue operations as *MakeNull*, *Enqueue*, *Front*, *Dequeue*, *Push* and *Empty*. Their formal descriptions are listed in Table 5.3.

Table 5.3 Methods for the queue ADT

Methods	Function
makeNull()	Sets the pointer to the queue to -1. The return value is ignored.
enqueue()	Enqueues an element. The return value is ignored.
front()	Returns the element in the front of the queue.
dequeue()	Returns the element in the front of the queue, removes the element from the queue and decrements the pointer by 1.
empty()	Returns an integer less than zero if the queue is empty, otherwise it returns an integer greater or equal to zero

The last data structure considered in this study is the list. A list represents a collection of linearly arranged elements [55]. It provides methods for accessing, inserting, and removing arbitrary elements. Arbitrary elements can be inserted at any valid positions in the list. The stack and queue can be seen as restricted forms of the list. If access to a list is restricted to just one end, it is called a stack. Also, if access to a list is restricted to both ends such that one end is used for adding elements and the other for removing elements, it is called a queue. Based on the list operation defined in Goodrich and Tamassia [55], the five methods considered sufficient for testing the abilities of GP and GE to produce code for the list ADT are given in Table 5.4.

Table 5.4 Methods for the list ADT

Methods	Function
makeNull()	Sets the pointer to the list to -1. The return value is ignored.
insertAt (p, x)	Shifts all the elements at the position indexed by q ($q >= p$) one position to the right. Inserts an element x at the position indexed by p. Increments the pointer by 1. Return value is ignored.
getElement(p)	Returns the element at the position indexed by p.
removeElement(p)	Removes the element at the position indexed by p. Shifts all the elements at the position indexed by q ($q > p$) one position to the left. Decrements the pointer by 1. Returns the removed element.
empty()	Returns an integer less than zero if the list is empty, otherwise it returns an integer greater or equal to zero.

5.5.2 Problems Solved Using the Evolved ADTs

This application domain comprises three programming problems. Each of the problems can be solved using one of the ADTs described in section 5.5.1. These problems are as follows:

- **Problem1**: Write a program that uses the stack ADT to determine if a given word or sentence is a palindrome. Inenaga *et al*. [57] define a palindrome as a symmetric string that reads the same from left to right and right to left. If space and special characters occur in the string, these are ignored.
- **Problem2**: Write a program that uses the queue ADT to perform a breadth-first traversal of any given parse tree.
- **Problem3**: Write a program to populate a list with integers and sort the list. The program, given the list ADT should be able to populate and sort the list. Much work [58–61] has been published that uses GP to generate the code that sorts a given list of integers. Whereas these studies provide the GP algorithm with one or more list/s of integers, this study provides the GP algorithm with an empty list and a set of integers. Thus the task is to populate and sort the list as well.

5.5.3 The Object-Oriented Programming Problems Classification Based on Difficulty Levels

The ADTs are categorised into 3 levels of difficulty based on their functionalities and the number of access points to the data structure. Intuitively, it is easy for a programmer to know how to get an element (pop) off a stack if the programmer knows that there is one "entry" point which is the same as the "exit" point in a stack. The knowledge of the position of the element is not necessary. On the other hand, if a programmer is required to retrieve an element from a list, the knowledge of the position of the element in the list would be required in order to carry out the task. Thus, the operation of the list is more difficult than that of the stack. This has also been mentioned in Langdon [6] where the operations of the list ADT have been implemented. Table 5.5 specifies the levels of difficulties of the problems. These levels are easy (level 1), medium (level 2) and hard (level 3).

Table 5.5 The difficulty levels of the test data

	Level 1	Level 2	Level 3
ADT Class	Stack	Queue	List
Problems	Problem 1	Problem 2	Problem 3

5.6 Technical Specifications

The algorithms, written in Java 1.7 using Netbeans 8.0.2, were developed on an Intel core i7, 3.1GHz machine with 8192 MB of RAM. Simulations were run on the same. An instance of the Java Random class is used to generate a pseudorandom number which modifies the initial seed and produces a new random number where necessary.

5.7 Chapter Summary

This chapter described the methodology used to achieve the objectives described in chapter 1. It described the application domains used to evaluate GP and GE. Finally, the chapter provided the technical specifications for the study. The next chapter will discuss the GP approach to automatic programming.

CHAPTER 6: GENETIC PROGRAMMING APPROACH FOR AUTOMATIC OBJECT-ORIENTED PROGRAMMING

6.1 Introduction

This chapter presents the genetic programming approach to automatic object oriented programming. This is referred to as the Object-Oriented Genetic Programming (OOGP) approach.

Section 6.2 defines the programming problem specification which forms input to OOGP. An overview of the OOGP algorithm is presented in section 6.3. Section 6.4 describes the program representation while section 6.5 describes the initial population generation. The fitness evaluation and selection are described in section 6.6 and section 6.7 respectively. The genetic operators are described in section 6.8 while section 6.9 describes the termination criteria. A new approach to OOGP, namely, greedy object-oriented genetic programming is described in section 6.10. Finally, section 6.11 summarizes the chapter.

6.2 Programming Problem Specification

A programming problem specification specifies the appropriate subset of the internal representation language, constants and assumptions that should be made in order for OOGP to generate a solution to a specific problem. A problem specification contains the following:

- The number of methods in the class required to be evolved.
- A function set that is used generally by OOGP to evolve the required methods. This must be a subset of the internal representation language described in section 6.5.1 and ADFs that have one or more arguments.
- A terminal set that is used generally by OOGP to evolve the required methods. This includes constants and ADFs that have no argument.
- A set of fitness cases comprising input values and their corresponding target output values. Each input is represented as a variable which is added to the terminal set when evolving the method. The type of the variable must be specified. Examples of types are float, Boolean, and string.

6.3 An Overview of the OOGP Algorithm

Like the GP algorithm described in section 2.2 of chapter 2, the OOGP approach makes use of the generational control model. Whereas GP represents each individual as a parse tree, each individual in OOGP is a class represented as a chromosome consisting of genes. Each

gene is a parse tree representing a method of the class. The individuals in the initial population are created using one of the three initial population methods, namely, the grow, full or ramped half-and-half. OOGP uses one or more fitness functions during evaluation. Tournament selection is used for selecting parents and two genetic operators, namely, crossover and mutation, are used for regeneration. A solution evolved by OOGP is a class containing one or more methods (parse tress). The next section describes the program representation in more detail.

6.4 Program Representation

The study adopts the representation used by Bruce and Langdon [6]. Each individual in the population, i.e. a chromosome, represents a class. Each chromosome consists of n parse trees, where n is the number of methods of the class. Each parse tree, i.e. a gene, represents a method. One of the methods must be the main method. Figure 6.1 is an example of an individual with n = 3 parse trees. The nodes of each parse tree are generated by combining the elements of the function and terminal sets. If the problem uses ADFs, the ADFs will correspond to private methods in the class. Eeach individual with ADFs will contain $n + m$ parse trees where m is the number of ADFs in the class.

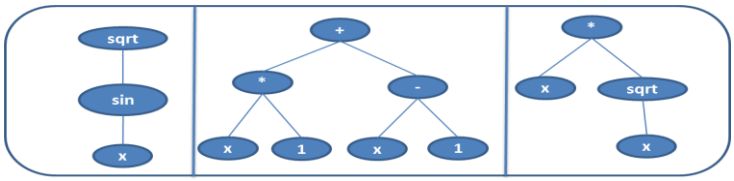


Figure 6.1. An example of individual in the population

6.5 The Initial Population

OOGP caters for three initial population methods, namely, grow, full and ramped half-and-half. These initial population methods have been described in section 2.7 of chapter 2. Each parse tree i.e., a gene in an individual is created using one of the three initial population methods. The process of creating genes is repeated until n genes are created for the individual.

As mentioned in section 6.2, the functions are elements of the internal representation language. Typing is imposed on the internal representation language to ensure that programs are syntactically correct. It also ensures that the genetic operators produce syntactically correct offspring. Typing has been explained in section 2.12.1 of chapter 2. OOGP uses point

typing. Each node is assigned a type. The arguments of each node are also assigned a type referred to as the argument type. A node, say node N1, takes node N2 as an argument if the type of N2 is the same as the argument type of N1. The type of the root of a parse tree must be the same as the type of the method the parse tree represents. For instance, an arithmetic operator such as + cannot be the root of a parse tree representing a method that returns a string. The types are given in Table 6.1 while the internal representation language is described in the following sections.

Table 6.1. An overview of the typing for OOGP

S/N	Type	Description
1	MEMORY	Used for iteration and counter variables of the *for* iterative operator (see section 6.5.1.6)
2	FLOAT	Used for all the nodes of type float. Depending on the problem, functions such as +, -, and * can be of type float or int.
3	INT	Used for all the nodes of type int, e.g. +, -, *. Depending on the problem, these functions can be of type float.
4	BOOLEAN	Used for all the nodes of type Boolean, e.g., >=, <.
5	STRING	Used for all the nodes of type string, e.g., *append*
6	GENERIC	Used to first define a node which will assume a type during the initial population generation or later during the run of the GP algorithm. Thus they are replaced later with anyone of the other types (more explanation in section 6.5.1.6)

6.5.1 The Internal Representation Language

6.5.1.1 Arithmetic Operators

The arithmetic operators catered for by the system and their corresponding arity and types are listed in

Table 6.2. The operators perform the standard arithmetic operations of addition, subtraction, multiplication, division, square root of a non-negative number and the absolute value of a number.

Table 6.2. The arithmetic operators

Operator	Arity	Argument Types	Type
+	2	INT/FLOAT, INT/FLOAT	INT/FLOAT
-	2	INT/FLOAT, INT/FLOAT	INT/FLOAT
*	2	INT/FLOAT, INT/FLOAT	INT/FLOAT
/	2	INT/FLOAT, INT/FLOAT	INT/FLOAT
sqrt	1	INT/FLOAT	INT/FLOAT
abs	1	INT/FLOAT	INT/FLOAT

Each of the operators except the *sqrt* and *abs* operators has an arity of 2. The *sqrt* operator and the *abs* operator, each has an arity of 1. If the type of the / operator is treated as integer, divisions such as 2/5 returns 0. The / and *sqrt* operators are protected. Thus:

- If an attempt is made to divide a number by zero, a value of 1 is returned.
- If the argument of the *sqrt* operator is negative, a value of 1 is returned.

6.5.1.2 Logical Operators

The system makes use of arithmetic logical operators and string logical operators. These operators are used for conditional checks and are listed in Table 6.3.

Table 6.3. The logical operators

Operator	Arity	Argument Types	Type
==	2	FLOAT, FLOAT	BOOLEAN
!=	2	FLOAT, FLOAT	BOOLEAN
<=	2	FLOAT, FLOAT	BOOLEAN
>=	2	FLOAT, FLOAT	BOOLEAN
<	2	FLOAT, FLOAT	BOOLEAN
>	2	FLOAT, FLOAT	BOOLEAN
strequal	2	STRING, STRING	BOOLEAN
strnoteq	2	STRING, STRING	BOOLEAN
Not	1	BOOLEAN	BOOLEAN

Each of the operators has an arity of 2 and is of type Boolean. With the exception of the *Not, strequal* and *strnoteq* operators, the argument type of each of the operators is float. The *strqual* and the *strnoteq* operators check if two strings are identical. These two operators are not case sensitive. Thus, the word "you" is identical to "You" and also to "YOU". The *strqual* operator returns a value of true if both its arguments are identical otherwise it returns false. The *strnoteq* operator returns a value of true if its arguments are not identical otherwise it returns a value of false. The argument types of both these operators are string. The internal representation language includes a single unary operator, namely, *Not*. The operator functions as a logical complement operator. It inverts the value of its argument. Thus, *Not* true implies false.

6.5.1.3 String Operators

The *append* and *char_At* are string operators. The former takes two string arguments. It returns the string formed by appending the second argument to the right of the first argument. The latter takes two arguments. The first argument must evaluate to a string while the second argument must evaluate to an integer. It returns the character at a position indexed by its second argument in its first string argument.

6.5.1.4 Conditional Operators

The internal representation language includes the *if* operator implemented by Pillay [11]. The operator functions like the if-then-else statement in a programming language such as Java. It takes three arguments. The first argument is of type Boolean and thus its function is to determine whether the second or third argument will be executed. The second and the third arguments are generic. If the first argument of the *if* operator evaluates to true, it returns the value of executing its second argument, otherwise it returns the value of executing its third argument.

6.5.1.5 Memory Manipulation Operators

These include the index memory operators and the named memory operators that are used by OOGP. These operators are explained below.

6.5.1.5.1 Indexed memory operators

The indexed memory manipulation operators that form part of the internal representation language are the *read* and *write* operators. These operators are used to access a linear

memory structure with each program having its own indexed memory. Each of the operators can be generated as a valid node in the program provided there is a linear structure to be accessed. The *read* and *write* operators have been explained in chapter 2, section 2.12.

The single argument of the *read* operator must be an integer. Thus any node that returns an integer can be a valid argument of the *read* operator. The *write* operator takes two arguments. The first argument represents a value to be written to a memory location indexed by the second argument which must be an integer. If an argument representing an index of the *write* or *read* operator evaluates to a value less than zero or greater than the size of the memory structure of the program, (a) the operator does nothing and (b) a default value for the type is returned. For example, the default value for float is 0.0 while the default value for String is an empty string.

6.5.1.5.2 Named Memory Operators

The system maintains a single named memory location, *aux*, which a program can use as a temporary memory location. The *set_aux*, *dec_aux* and *inc_aux* operators defined in [6] are used to provide access to this single memory location. These operators have been explained in chapter 4, section 4.4.

6.5.1.6 Iterative Operators

The *for*, the *while* and the *for_loop* iterative operators form part of the internal representation language. These operators have been explained in chapter 2, section 2.12.3.

The *for* operator takes three arguments. The first and second arguments must evaluate to integer values. The third argument is generic i.e., it can be of any type which will be the type of that instance of the *for* operator. The third argument is iteratively executed a number of times indicated by the difference between its first two arguments plus one. Thus, if the difference between the first and second argument is n, then the third argument will be executed $n + 1$ times. Two variables are maintained for each instance of the *for* operator. First is the counter variable. This is used to track the number of iterations performed by the operator. The counter variable is assigned the value of the argument of the *for* operator. It is incremented on each iteration if the first argument of the *for* operator is less or equal to the second argument, otherwise it is decremented on each iteration. For example, assuming the first argument of an instance of the *for* operator evaluates to an integer n and the second argument evaluates to another integer m. The counter variable of that instance of the *for* operator is incremented on each iteration if n <= m, otherwise it is decremented on each

iteration. Second is the iteration variable which stores the value the third argument of an instance of the *for* operator evaluates to. The iteration variable is initialized to the default value for the type of the third argument of the *for* operator. If the third argument of the *for* operator is of type float and is an arithmetic operators ***** or /, the iteration variable is initialized to 1 instead of the default value which is 0. This prevents multiplication or division by zero. Both the counter and iteration variables are added to the terminal set when creating the subtree representing the third argument of the *for* operator.

The *while* operator takes two arguments. The first argument is a condition which specifies when the execution of the second argument must stop. The condition must be checked first. Also, this operator is generic and uses counter and iteration variables like the *for* operator. The counter variable of the *while* operator is added to the terminal set when creating the subtree representing the first argument of the operators. Both the counter and iteration variables are added to the terminal set when creating the subtree representing the second argument.

The *for_loop* operator takes two arguments. The first argument must evaluate to an integer and the second argument is executed a number of times specified by the value the first argument evaluates to. In this study, the counter and iteration variables are provided for the *for_loop* operator. The counter variable is initialized to 0. The iteration variable is initialized to the default value for the type of the second argument of the *for-loop* operator. Both the counter and iteration variables are added to the terminal set when creating the subtree representing the second argument of the *for-loop* operator.

Each instance of the *for*, the *while* and the *for_loop* operator has an integer appended to the end of the operator. The integer is used as a unique identify of the counter and iteration variables of the operator. For instance, an instance of the *for* operator, namely, *for*2 has an iteration and counter variable represented as *Ivar2* and *Cvar2* respectively. The number of iterations allowed for each instance of an iterative operator is problem dependent.

6.5.1.7 Multiple Statements: Blocks

A program may consist of more than one statement. Such statements are almost always executed sequentially in a top down manner. Koza [7] uses *progn* in an individual containing an ADF to combine two branches that execute from left to right. Pillay [11] uses *blockn,* where *n* is a positive integer, for the same purpose. The node takes *n* arguments of any type and returns the result of evaluating its last argument. In this study, the internal representation

language includes the multiple block statements *blockn* and *Fblockn* where n takes a value in the range [2, 3]. Both these operators takes n arguments. The type and the argument types of the operators are generic. Hence they are instantiated during initial population generation. The *blockn* operator returns the result of evaluating its nth argument while the *Fblockn* operator returns the result of evaluating its 1st argument. However, any side effect caused by evaluating the other arguments of the operators is retained by the program.

6.6 Fitness Evaluation

The fitness of an individual is the sum of the fitness of the methods in the individual. Because different methods are expected to perform unique functions when executed, each method is evaluated to check how well it performs the expected function. Each method in the individual is assigned a scalar value representing the method's fitness. These scalar values are then summed to obtain a fitness value for the individual. The fitness of a private method, i.e., an ADF is not evaluated. A private method can only be accessed by evaluating the method that called the private method.

6.7 Selection

As discussed in chapter 4 section 4.6.1.4, OOGP uses the tournament selection method to select the parents that will be used to create the next generation. Tournament selection has been described in chapter 2 section 2.9.1.

6.8 Genetic Operators

OOGP uses two genetic operators, namely, crossover and mutation. The reproduction operator simply copies an individual to the next generation. Both the crossover and mutation operators can produce offspring which is the same as copying an individual to the next generation. Thus the reproduction operator is not used. The parents selected for any of the operations are firstly duplicated to avoid modifying the current population. As with GP, OOGP uses the application rate of a genetic operator to determine the number of individuals that will be created using the operator. The crossover and mutation operators are explained below.

6.8.1 The Crossover Operator

Two phases of crossover are performed. These are named the external and internal crossover. This is done to ensure a proper mixing of the genetic materials from the two parents selected for the crossover operation, i.e., two parents can exchange genes as well as subtrees within a

gene in one crossover performed. Other than the crossover rate, a probability is introduced for both the external and the internal crossover. As mentioned in chapter 4, section 4.6, this is done to allow a parent to exchange more than one gene with another parent. The external crossover is illustrated in Figure 6.2. Two parents are selected using tournament selection. A preset probability called the external crossover probability is used to decide whether or not to perform external crossover on the selected parents. For each gene position, the genes are swapped between the parents if the randomly generated probability in the range 1 to 100 is less than the preset probability

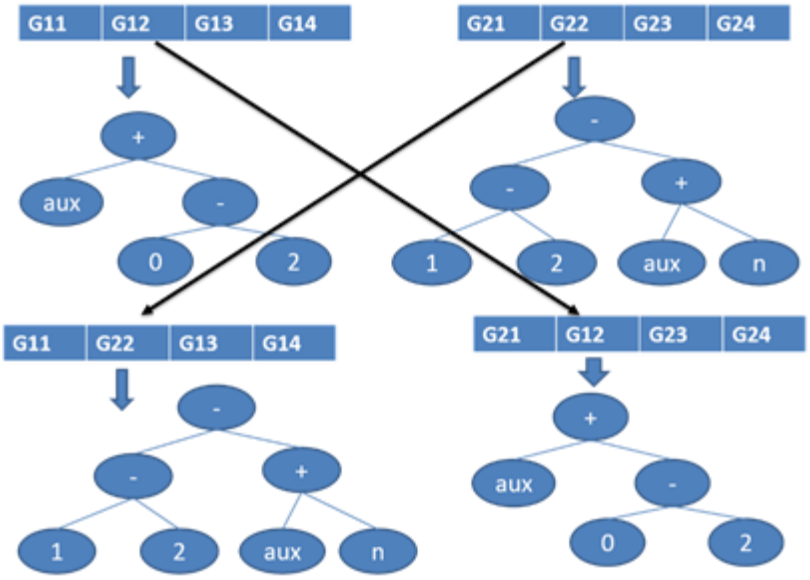


Figure 6.2. An illustration of the external crossover operator

Assume that the selected parents are $G_{11}G_{12}G_{13}G_{14}$ and $G_{21}G_{22}G_{23}G_{24}$ as shown is Figure 6.2. Each chromosome is comprised of four genes with each gene *Gij* representing a parse tree. Given that the preset probability is 40% and the randomly generated probabilities for each gene are 64%, 35%, 80%, 45% respectively, the resulting offspring are $G_{11}G_{22}G_{13}G_{14}$ and $G_{21}G_{12}G_{23}G_{24}$.

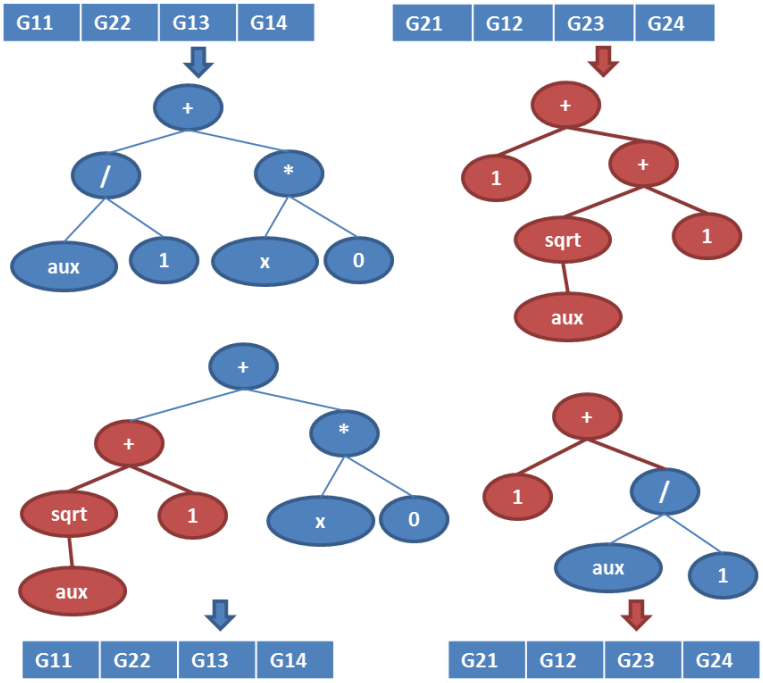


Figure 6.3. An example of internal crossover

Internal crossover is applied once the external crossover is completed. Again, a number is randomly generated in the range 1 to 100. If this number is less than the preset internal crossover probability, the following is done: An index is randomly chosen, crossover is applied to the parse trees at the selected index in both parents. Figure 6.3 illustrates internal crossover. The first internal crossover point is randomly chosen from the first parent. The second internal crossover point is randomly chosen from the nodes with the same type as the first internal crossover point. The subtrees rooted at the nodes are swapped. Thus, the operator produces two offspring.

Whereas the maximum depth limits the size of the tree created in the initial population, a maximum offspring depth limits the size of the offspring created during the regeneration. Each function node at a depth level equal to the maximum offspring depth is replaced by a randomly generated terminal node. This is called pruning. The terminal node must be of the same type as the function node which it replaces. Pruning ensures that the preset maximum offspring depth is not exceeded.

The GP algorithm implemented by Koza [4] allowed both the offspring created by the crossover operation to be added to the new population. However, the study conducted by Pillay [25] has shown that choosing the fitter offspring improves the performance of the GP system. Hence, the crossover operator employed in this study returns the fitter of the two

offspring. The returned offspring is added to the new population. The overall crossover algorithm for OOGP is illustrated in Figure 6.4.

Input: External crossover probability P_{ex}

Input: Internal crossover probability P_{in}

Output: An offspring

1. Select the first parent Parent1 = $G_{11}G_{12}$ ... G_{1n} using a selection method
2. Select the second parent Parent2 = $G_{21}G_{22}$ ... G_{2n} using a selection method
   (where a gene, G_{ij} is a parse tree and n is the number of genes in an individual).
3. For each j ∈ {1, 2, ... , n} {
   a. Randomly generate an integer pj between 1 and 100
   b. If $pj < P_{ex}$, swap G_{1j} and G_{2j}
   }
4. Randomly generate an integer p between 1 and 100
5. If $p < P_{in}$
   a. Randomly select a position i in the chromosome
   b. perform crossover between genes G_{1i} and G_{2i} at position i in both parents as follows
      i. Randomly choose a crossover point, $pt1$, from G_{1i}
      ii. Randomly choose another crossover point, $pt2$, from G_{1i}
      iii. Swap the subtree rooted at $pt2$ and $pt1$ between G_{1i} and G_{2i}
6. Return the fitter of the two offspring

Figure 6.4. The crossover algorithm for OOGP showing both the external (1 to 3) and the internal crossover (4 to 6)

Note that the crossover produces offspring which is the same as copying an individual to the next generation if $pj \geq P_{ex}$ for all j and $p \geq P_{in}$. Steps 1 to 3 shows the external crossover

6.8.2 The Mutation Operator

A parent is selected using the tournament selection and an index in the parent is randomly chosen. A copy of the parse tree at this index is made. A mutation point is randomly chosen in the parse tree. The subtree rooted at this point is deleted. The grow method is used to create a new subtree that replaces the deleted subtree. During this process, a specified

mutation depth is used as the maximum depth. The root of the new subtree must be of the same type as the root of the deleted subtree. This operator produces one offspring. The offspring is pruned and added to the new population.

6.9 Termination Criteria

The termination criteria for the OOGP approach are the same as the termination criteria for the GP algorithm described in chapter 2, section 2.2. An OOGP run stops if (1) a solution has been found or (2) the specified number of generations has been reached. The best individual in the last generation is returned.

6.10 The Greedy Object-Oriented Genetic Programming (GOOGP) Approach to Automatic programming

As discussed in chapter 4, section 4.6.1.3, this study introduces a greedy OOGP approach. Whereas OOGP randomly creates a tree for each gene in a chromosome, GOOGP creates a pool of trees and the fittest tree becomes the gene for the chromosome. The reason for this approach has been given in section 4.6.1.3. The GOOGP approach is employed as follows:

1. Randomly create a population of *m* parse trees for each gene.
2. Evaluate the population using the process described in section 6.6 to determine the fitness of each tree.
3. Store the fittest tree in the population as the gene for the chromosome.

The genetic programming algorithm employing the greedy approach is referred to as GOOGP. Other processes in the GOOGP approach are the same as the OOGP approach. The greedy population size, i.e., the value of m in 1 above, must be included as a parameter for any problem that uses the GOOGP approach.

6.11 Chapter summary

The chapter presented the OOGP approach for automatic object-oriented programming and a variation of OOGP, namely, GOOGP which uses a greedy method for initial population generation. The next chapter will present the GE approach for automatic object-oriented programming.

CHAPTER 7: GRAMMATICAL EVOLUTION APPROACH FOR AUTOMATIC OBJECT-ORIENTED PROGRAMMING

7.1 Introduction

This chapter presents the grammatical evolution approach for automatic object oriented programming. This is referred to as Object-Oriented Grammatical Evolution (OOGE) approach. Section 7.2 defines the format of the programming problem specification which forms input to OOGE. The OOGE algorithm is described in section 7.3 while section 7.4 summarizes the chapter.

7.2 Programming Problem Specification

The programming problem specification, discussed in section 6.2 of chapter 6, for OOGP is the same for OOGE except for one change, namely, rather than specifying a function and terminal set, a grammar is defined for OOGE. Functions in the program generated by OOGE are the subset of the internal representation language.

7.3 The OOGE Algorithm

The OOGE approach uses the generational control model. It also uses the same fitness evaluation, selection and termination criteria described for OOGP.

Section 7.3.1 describes the grammar. Section 7.3.2 describes the program representation while section 7.3.3 describes the initial population generation. The fitness evaluation and selection are described in section 7.3.4 and section 7.3.5 respectively. The genetic operators are described in section 7.3.6 while section 7.3.7 describes the termination criteria.

7.3.1 The Grammar

As described in chapter 3, section 3.3, a grammar consists of a set of non-terminals, a set of terminals and production rules. In order not to confuse the terminals with the terminal set in GP, this study refers to the terminals of a grammar as GE-terminals and the non-terminals as GE-non-terminals. The GE-terminals for the OOGE are the same elements of the internal representation language described for OOGP. In OOGP, *blockn* and *Fblockn* are used to combine two or more program statements. In order to achieve the same in OOGE, the start symbol can be replaced with a single statement or more than one statements. The production rules for the start symbol are $< Stmts > ::= < Stmt > \mid < Stmt >; < Stmts >$. Other production rules that can be included in the grammar are problem specific.

7.3.2 Program Representation

Like the OOGP, each individual in the population is a chromosome consisting of genes. Each chromosome represents a class. Whereas in OOGP, each gene in the chromosome is a parse tree; each gene is a binary string in OOGE. Again, each gene in a chromosome corresponds to a method of the class and is made up of codons. Each codon is composed of *n* alleles

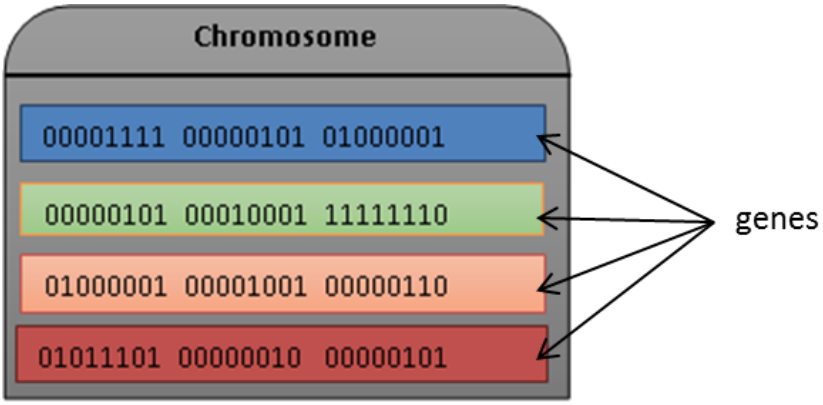


Figure 7.1. An Example illustration of a chromosome

specifying the length of the codon. Figure 7.1 illustrates an example of a chromosome in OOGE.

The chromosome contains four genes each representing one of the four methods for the class. Each gene has 3 codons of length $n = 8$, i.e., each codon is made up of 8 alleles. During the mapping process, a gene can produce one or more parse tress.

7.3.3 Initial Population Generation

Each gene in a chromosome is randomly created. In order to produce and execute a program, a binary to denary mapping is performed for each codon in the gene. This implies converting each codon to a denary equivalent. Figure 7.2 shows the chromosome in Figure 7.1 after the binary to denary mapping. Each denary value is then mapped onto a production rule of the grammar.

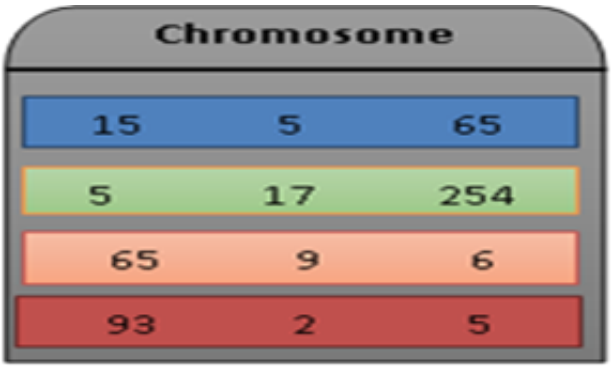


Figure 7.2. The chromosome in Figure 7.1 after the binary to denary mapping

First, the mapping process starts from the first denary value in the first gene. Using the GE mapping process described in section 3.6 of chapter 3, the value is mapped to a production rule of the grammar. Consider the production rules, $< Stmts > ::= < Stmt > | < Stmt > ; < Stmts >$, for the start symbol. The result of 15 mod 2 is 1 which maps to the second production rule, namely, $< Stmt >; < Stmts >$, and hence indicates that the method will have more than one statement. The mapping of the denary values to production rules continues until one or more parse trees representing the method is created. Again, the mapping starts from the first denary value in the second gene and creates one or more parse trees representing the method. The process continues until each method corresponding to a gene is created.

7.3.4 Fitness Evaluation

Once the mapping process is completed, the fitness of each chromosome is calculated by evaluating the methods created. The fitness evaluation for the OOGE is the same as described for OOGP and has been described in chapter 6, section 6.6.

7.3.5 Selection

Like the OOGP, OOGE uses the tournament selection method. This has been described in chapter 2, section 2.9.1.

7.3.6 Genetic Operators

As in the case of OOGP, the algorithm implements two genetic operators, namely, the crossover and mutation operators. These operators are applied to the binary string in OOGE. Whereas in OOGP, both these operators use application rates to determine whether they

would occur or not, each of the operators use a probability in OOGE. This is consistent with GE. The offspring created by crossover are mutated. The crossover and mutation operators are explained below.

7.3.6.1 The Crossover Operator

Both the external and internal crossover are applied for OOGE. Again, this is done to ensure a proper mixing of the genetic material from the two chromosomes selected for crossover operation. Two chromosomes are selected using the tournament selection method. The external crossover is applied as follows. For each chromosome index, a random probability is generated in the range 1 to 100. If the probability is less than the preset external crossover probability, the genes at the index are swapped between the chromosomes. This is illustrated in Figure 7.3 and Figure 7.4. Assuming the preset probability is 58% and the probability generated for each chromosome index (from top to bottom) are 60, 53, 21, and 97. Only the genes at the 2nd and 3rd index are swapped.

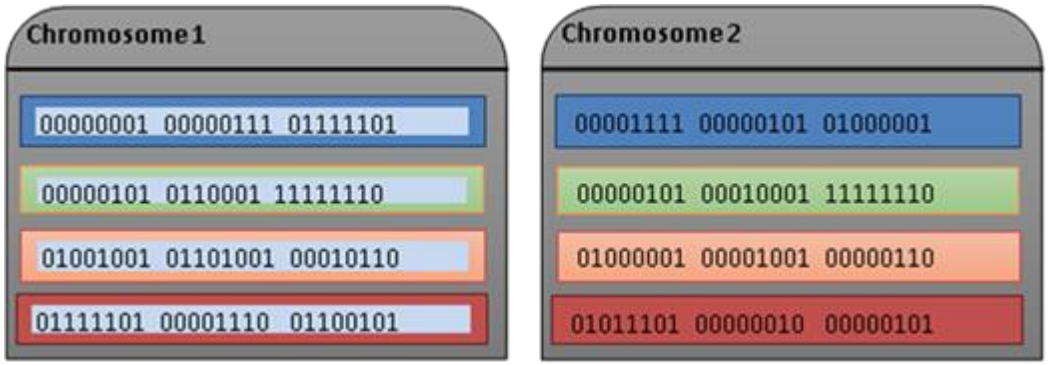


Figure 7.3. Example of two chromosomes before the external crossover operation

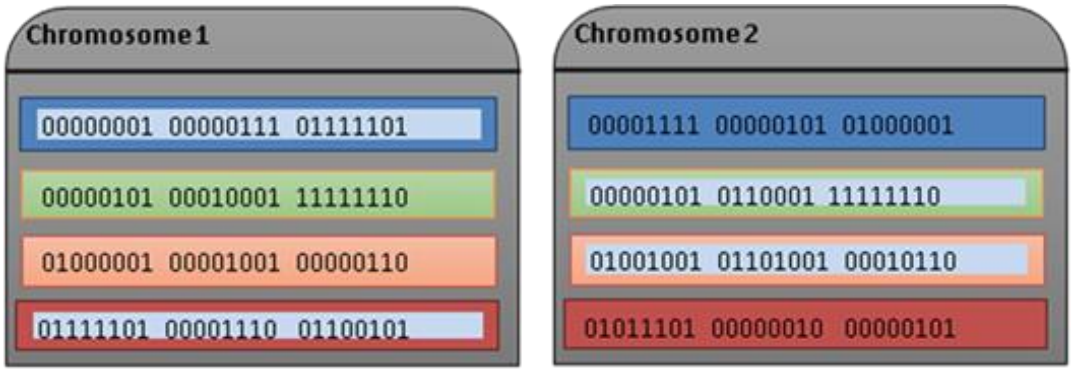


Figure 7.4. The chromosomes after the external crossover operation

Once external crossover is done, a random probability is then generated again in the range 1 to 100. If the probability is less than the preset internal crossover probability, the internal crossover is applied as follows: A number is randomly chosen in the range 0 to the number of genes minus 1. This ensures that any number chosen is a valid chromosome index. A typical two-point crossover [15] explained in chapter 3, section 3.10.1 is then applied to the genes at the chosen index. Figure 7.5 illustrates an example where the number 2 is chosen (assume the start index as 0). To be consistent with GP, the fitter of the two offspring is returned as the result of the crossover operation which will be mutated. The parameter values are problem dependent.

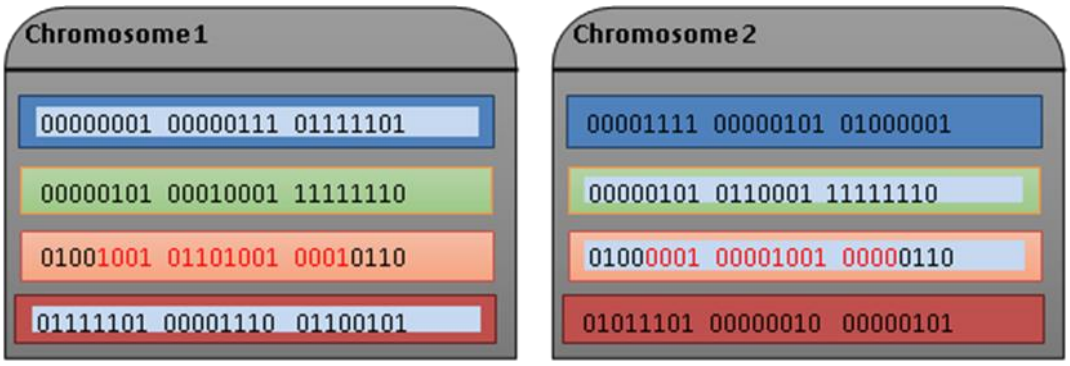


Figure 7.5. The example operation of the internal crossover

7.3.6.2 The Mutation Operator

In OOGE, the offspring produced from the crossover operation is mutated. This is typical of GE. As with the crossover operator, two preset probabilities are used for the mutation operator. These probabilities are the mutation probability and the bit flips probability. A

random probability in the range 1 to 100 is generated. If this probability is less than the mutation probability, bit mutation is performed as follows. For each bit in the codon, a random number between 1 and 100 is generated. If this generated number is less than the bit flip probability the bit is flipped, i.e. if the bit is 1 it becomes 0 and vice versa.

7.3.7 Termination Criteria

OOGE uses the same termination criteria as OOGP. Each run stops if (1) a solution has been found or (2) the specified number of generations has been reached. The best individual in the population is returned.

7.4 Chapter Summary

The chapter presented the OOGE approach for automatic object-oriented programming. Each individual is a chromosome consisting of genes. The individuals in the population are randomly generated. Like in GE, each individual in the OOGE population are mapped to the grammar defined for the system to create a program. The next chapter will provide the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to evolve solutions for the stack ADT and Problem1 described in chapter 5, section 5.5.2.

CHAPTER 8: FITNESS EVALUATION AND PARAMETERS FOR THE STACK ADT AND PROBLEM1

8.1 Introduction

This chapter provides the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the stack Abstract Data Type (ADT) and Problem1 described in section 5.5.2 of chapter 5. Whereas GE searches a space of binary strings, GP search is performed on parse tress. Thus parameters required are not exactly the same. For instance, GE requires the length of the allele to be set as a parameter while GP requires a tree depth. Again, GP uses application rates while GE uses application probabilities. Hence GE parameters will be set differently from the GP parameters.

Section 8.2 describes the programming problem specification for the stack ADT while section 8.3 presents the OOGP, GOOGP and OOGE parameters for the stack ADT. Section 8.4 and section 8.5 describe the programming problem specification, and the OOGP, GOOGP and OOGE parameters respective for Problem1. Section 8.6 summarizes the chapter.

8.2 Programming Problem Specification for the Stack ADT

The stack ADT has been described in chapter 5, section 5.5.1. Section 8.2.1 describes the fitness evaluation and fitness cases used. Section 8.2.2 provides the OOGP and GOOGP primitives while section 8.2.3 presents the OOGE grammar.

8.2.1 Fitness Evaluation

The fitness is maximized. A fitter individual is one whose fitness value is higher. Fifteen cases are randomly generated for each run. Each fitness case is a stack. The length of the stack is randomly generated between 1 and 15. The elements of the stack are randomly generated integer values in the range 1 to 999. An individual is evaluated by applying each of its methods to the 15 fitness cases to check how good the method is at performing the required operation on the stack. A set of problem dependent criteria that must be met by each method is defined. The method is scored based on the number of criteria it meets. Table 8.1 shows the criteria that must be met by the stack ADT methods.

Table 8.1 Problem specific criteria for the stack ADT fitness evaluation

Method	**Criteria**
makeNull()	• Stack pointer must be set to -1.
peek()	• No change in pointer value. • Elements on stack should not be altered. • The value returned must be the topmost element of the stack. • Only one value must be returned.
push(i)	• Pointer must be updated correctly. • Elements on the stack should not be altered. • The pushed element, *i*, must be at the top of the stack. • The element must be pushed onto the stack only once.
pop()	• Pointer must be updated correctly. • Elements on the stack should not be altered. • The correct value must be returned.
empty()	• No change in pointer value. • Elements on the stack should not be altered. • The correct position of the pointer must be returned.

The stack ADT methods, namely, *makeNull(), peek(), push(), pop()* and *empty()* can attain a maximum score of 1, 4, 4, 3 and 3 respectively. Thus, each individual can attain a maximum fitness of 15 per fitness case and 225 over all 15 fitness cases.

8.2.2 OOGP and GOOGP Primitives

The functions for the stack ADT are a subset of the internal representation language and are given in Table 8.2. The constants are 1 and 0.

Table 8.2 OOGP functions and terminals (primitives) for the queue ADT

Stack ADT methods	**Primitives**
makeNull()	+, -, *, /, set_aux, read, write, 0, 1, aux
peek()	+, -, *, /, set_aux, read, write, 0, 1, aux
push(i)	+, -, *, /, read, write, set_aux, block2, *i* (integer to be push onto the stack), 0, 1, aux
pop()	+, -, *, /, block2, read, write, set_aux, 0, 1, aux
empty()	+, -, *, /, set_aux, read, write, block2, 0, 1, aux

8.2.3 OOGE Grammar for the Stack ADT

The grammar for the stack ADT consists of the GE-terminals and GE-Non-terminals given in Table 8.3. The grammar is listed in **Appendix A.1.**

Table 8.3 GE-Non-terminals and GE-terminals for the stack ADT

Start symbol	Non-terminals	Terminals
Stmts	stmts, stmt, expr, var	+, -, *, /, write, read, set_aux, , 0, 1, aux, (,), ;, *i* (integer to be pushed onto the stack).

8.3 Parameters for the Stack ADT

The OOGP, GOOGP and OOGE parameters for the stack ADT are given in Table 8.4. The parameter values were obtained empirically by performing trial runs. For example, in the trial runs for OOGP, population sizes of 50, 100 and 500 were tested. Tournament sizes in the range 2 to 5 were also tested. A population size less than 100 is not enough to represent the search space while a population size greater than 100 makes no improvement on the success rate. Different combinations of genetic operators were tested. Starting from 50% crossover and 50% mutation rates; an increase in the crossover rate and decrease in the mutation rate produced a better result than an increase in the mutation rate and decrease in the crossover rate. Finally, 0% mutation and 100% crossover rates produced the best result. Thus, 0% mutation is appropriate for the algorithm and was used for the final run. This corresponds to Koza's work [4, 7] which demonstrated that GP was not performing a simple random search and as such, mutation was not always necessary.

Table 8.4 Parameters for the stack ADT

Approach	Parameter	Parameter value
OOGP and GOOGP	Population size	100
	Maximum tree depth	5
	Tournament size	2
	Crossover rate	100%
	Mutation rate	0%
	External crossover probability	50%
	Internal crossover probability	50%
	Maximum offspring depth	10
	Number of generations	100
GOOGP	Greedy population size	500
OOGE	Population size	500
	Codon length	10
	Allele length	8
	Tournament size	4
	Mutation probability	30%
	Bit flip probability	70%
	External crossover probability	50%
	Internal crossover probability	70%
	Number of Generations	100

8.4 Programming Problem Specification for Problem1

Problem1 has been defined in section 5.5.2 of chapter 5 and is listed as one of the problems the stack ADT can be used to solve. Section 8.4.1 describes the fitness evaluation and fitness cases used. Section 8.4.2 provides the GOOGP primitives while section 8.4.3 presents the OOGE grammar.

8.4.1 Fitness Evaluation

The fitness is maximized for the problem. Twenty five fitness cases were provided. Twenty of the fitness cases are palindromes while 5 of the cases are not palindromes. Each individual

in the population is evaluated on each fitness case. Each output value is of type Boolean and is written to memory. This value and the expected output are compared. A solution must be able to classify 20 fitness cases as palindromes and the others as not. The fitness of an individual is incremented for each correctly classified case. An individual is a solution if the individual correctly classifies all 25 fitness cases. The possible maximum fitness is 25. The fitness cases are shown in Table 8.5.

Table 8.5. The fitness cases for Problem1 *[62]*

Fitness case	**Expected outcome**
Noel sees Leon	TRUE
Sore was I ere I saw Eros	TRUE
Too hot to hoot	TRUE
push	FALSE
Sex at noon taxes	TRUE
Euston saw I was not Sue	TRUE
Dior Droid	TRUE
back	FALSE
Able was I ere I saw Elba	TRUE
Step on no pets	TRUE
Was it a rat I saw	TRUE
neet net	FALSE
No evil Shahs live on	TRUE
Harass selfless Sarah	TRUE
adda	TRUE
deep	FALSE
degged	TRUE
murdrum	TRUE
eke	TRUE
mill	FALSE
acca	TRUE
gig	TRUE
madam	TRUE
Lepers repel	TRUE
Ten animals I slam in a net	TRUE

8.4.2 GOOGP Primitives

The function and terminal sets for Problem1 are a subset of the internal representation language given in Table 8.6. The input variable *arg* represents an input value of the fitness case. The variable *len* is the length of the input value represented by *arg.*

Table 8.6 GOOGP functions and terminals for Problem 1

Primitives
+, -, block2, strequal, append, char_At, for_loop, one, zero, len, Cvar, Ivar, arg, the five methods of the stack ADT.

8.4.3 OOGE Grammar

The grammar for Problem1 consists of the GE-terminals and GE-Non-terminals given in Table 8.7. The grammar is given in **Appendix A.2.**

Table 8.7 GE-Non-terminals and GE-terminals for Problem 1

Start symbol	**Non-terminals**	**Terminals**
stmts	stmts, stmt, expr, var, loopStmt, loopExpr, loopVar.	+, -, for_loop, strequal, append, the five methods of the stack ADT, arg , zero, one, len , Cvar, Ivar, one, zero, }, {, ;,), (

8.5 Parameters for Problem1

The GOOGP and OOGE parameters for Problem1 are given in Table 8.8. Like the parameter values for the stack ADT, these values were obtained empirically by trial runs.

Table 8.8 Parameters for Problem 1

Approaches	Parameter	The Parameter Range
GOOGP	Population size	100
	Maximum depth	5
	Tournament Size	4
	Crossover rate	50%
	External crossover probability	50%
	Internal crossover probability	50%
	Mutation rate	50%
	Mutation Depth	8
	Maximum offspring depth	10
	Number of Generations	100
GOOGP	Greedy Population size	500
OOGE	Population size	500
	Codon length	10
	Allele length	8
	Tournament size	4
	Mutation probability	40%
	Bit flip probability	70%
	External crossover probability	50%
	Internal crossover probability	80%
	Number of Generations	100

8.6 Chapter Summary

This chapter presents the functions for fitness evaluation and parameters used by the OOGP, GOOGP and OOGE approaches to evolve code for the stack ADT and Problem1. The next chapter will provide the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE for the queue abstract data type and a programming problem that uses the queue.

CHAPTER 9: FITNESS EVALUATION AND PARAMETERS FOR THE QUEUE ADT AND PROBLEM2

9.1 Introduction

This chapter provides the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the queue Abstract Data Type (ADT) and Problem2 described in section 5.5.2 of chapter 5. Section 9.2 describes the programming problem specification for the queue ADT while section 9.3 presents the OOGP, GOOGP and OOGE parameters for the queue ADT. Section 9.4 and section 9.5 describe the programming problem specification and the GOOGP and OOGE parameters respectively for Problem2. Section 9.6 summarizes the chapter.

9.2 Programming Problem Specification for the Queue ADT

The queue ADT has been described in chapter 5, section 5.5.1. Section 9.2.1 describes the fitness evaluation and fitness cases used. Section 9.2.2 provides the OOGP and GOOGP primitives while section 9.2.3 presents the OOGE grammar.

9.2.1 Fitness Evaluation

The fitness is maximized. Again, 15 cases are randomly generated for each run. Each fitness case is a queue. The length of the queue is randomly generated between 1 and 15. The elements of the queue are randomly generated integer values in the range 1 to 999. An individual is evaluated by applying each of its methods to the 15 fitness cases to check how good the method is at performing the required operation on the queue. A set of problem dependent criteria that must be met by each method is defined. The method is scored based on the number of criteria it met. As with the stack, each individual can attain a maximum fitness of 15 per fitness case and 225 over all 15 fitness cases. Table 9.1 shows the criteria that must be met by the queue ADT methods.

Table 9.1 Problem specific criteria for the queue ADT fitness evaluation

Method	**Criteria**
makeNull()	• Queue pointer must be set to -1.
front()	• No change in pointer value. • Elements in the queue should not be altered. • The value returned must be the element in the front of the queue. • Only one value must be returned.
enqueue()	• Pointer must be updated correctly. • Elements in the queue should not be altered. • The enqueued value must be at the back of the queue. • The value must be enqueued only once.
dequeue()	• Pointer must be updated correctly • The index (position) of each element in the queue should be correctly updated. • The correct value must be returned.
empty()	• No change in pointer value. • Elements in the queue should not be altered. • The correct position of the pointer must be returned.

9.2.2 OOGP and GOOGP Primitives

The functions and terminals for the queue ADT are given in Table 9.2. The constants are 1 and 0. The type of the variable is integer. The element to be enqueued is represented by i.

Table 9.2 OOGP functions and terminals (primitives) for the queue ADT

Queue ADT methods	**Primitives**
makeNull()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar
front()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar
enqueue(i)	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar, *i*
dequeue()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar
empty()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar

9.2.3 OOGE Grammar

The grammar for the queue ADT consists of GE-terminals and GE-Non-terminals given in Table 9.3. The grammar is listed in **Appendix A.3.**

Table 9.3 GE-Non-terminals and GE-terminals for the queue ADT

Start symbol	**Non-terminals**	**Terminals**
Stmts	stmts, stmt, expr, var, loopStmt, loopExpr, loopVar.	+, -, *, /, for, write, read, set_aux, , 0, 1, aux, dec_aux, Cvar, Ivar, (,), { , }, ;, *i*

Table 9.4 Parameters for the queue ADT

Approach	**Parameter**	**Parameter value**
OOGP and GOOGP	Population size	500
	Maximum tree depth	5
	Tournament size	2
	Crossover rate	100%
	Mutation rate	0%
	External crossover probability	50%
	Internal crossover probability	50%
	Maximum offspring depth	10
	Number of generations	100
GOOGP	Greedy population size	500
OOGE	Population size	500
	Codon length	10
	Allele length	8
	Tournament size	4
	Mutation probability	40%
	Bit flip probability	70%
	External crossover probability	50%
	Internal crossover probability	80%
	Number of Generations	100

9.3 Parameters for the Queue ADT

The OOGP, GOOGP and OOGE parameters for the queue ADT are given in Table 9.4. These values were obtained empirically by trial runs as described for the stack ADT in chapter 9, section 8.3.

9.4 Programming Problem Specification for Problem2

Problem2 has been defined in section 5.5.2 of chapter 5 and is listed as one of the problems the queue ADT can be used to solve. Section 9.4.1 describes the fitness evaluation. Section 9.4.2 provides the OOGP and GOOGP primitives while section 9.4.3 presents the OOGE grammar.

9.4.1 Fitness Evaluation

Problem2 is a maximization problem. Five fitness cases are used. Each fitness case is a parse tree and the corresponding expected output, i.e., the nodes arranged in a breadth-first order. The nodes of each parse tree are numbered and each node has a unique number. Each node is represented by its number in the tree as illustrated in Figure 9.1. Figure 9.2 illustrates an example of a fitness case in the format used for the system. For each node in the fitness case, the child nodes are supplied as a list of nodes. The five fitness cases are given in **Appendix F**. Each fitness case provides the following information:

i. The number representing the root of the tree.
ii. The arity of each node
iii. For each node, a list of the child nodes. If the arity of the node is zero, an empty list is provided.

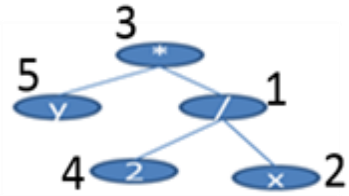


Figure 9.1. An example of a tree and the random numbering of the

input case
3/
4 /
1 / 4 , 2
5 /
3 / 5 , 1
2 /

Expected output
3, 5, 1, 4, 2

Figure 9.2. Input and the expected output for the example case in Figure 9.1.

Each individual is executed on the five fitness cases. For a given case, a solution, when executed, is expected to correctly arrange all the nodes in a breadth-first order. When executing an individual, the dequeued nodes are stored in an array in the order from the first dequeued node to the last dequeued. The fitness of an individual is incremented for each dequeued node in its correct position when compared to the expected output. For each fitness case, an individual can attain a maximum fitness equal to the number of nodes in the given case plus a bonus score of 2. The bonus score is awarded to an individual that dequeued exactly the expected number of nodes as supplied in the fitness case. This bonus score gives advantage to an individual that may dequeue all the nodes but not in the expected order.

9.4.2 OOGP and GOOGP Primitives

The function and terminal set for Problem2 are given in Table 9.5. Specific to Problem2 is the need to access the arity of a node. This is done via a function, namely, *getArity*. The *getArity* function takes a single argument which must be a node and returns the arity of the node. If the single argument of the *getArity* function evaluates to a value which does not represent any valid node, the root of the tree being evaluated is returned. This ensures that the closure property described in chapter 2, section 2.6 is met. Since the child nodes (if any) of each node are supplied as a list of nodes, there is a need for a function that would get a child node from the list. The *read* operator described section 6.5.1 of chapter 6 is used for this purpose.

Table 9.5 GOOGP functions and terminals for Problem 2

Primitives
for_loop, block2, block3, getArity, read, while, Not, the five methods of the queue ADT, root (the root of the tree), Cvar, Ivar

9.4.3 OOGE Grammar

The grammar for Problem2 consists of the GE-terminals and GE-Non-terminals given in Table 9.6. The grammar is given in **Appendix 4**.

Table 9.6 GE-Non-terminals and GE-terminals for Problem 1

Start symbol	**Non-terminals**	**Terminals**
stmts	stmts, stmt, expr, var, loopStmt, loopExpr, loopVar, cond.	for_loop, Arity, while, Not, root (the root of the tree), getElement, ADF0, Cvar, Ivar, the five methods of the queue ADT

9.5 Parameters for Problem2

The GOOGP and OOGE parameters for Problem2 are given in Table 9.7. These parameters were obtained empirically by trial runs.

Table 9.7 Parameters for Problem 2

Approaches	Parameter	The parameter range
GOOGP	Population size	100
	Maximum depth	5
	Tournament Size	4
	Crossover rate	50%
	External crossover probability	50%
	Internal crossover probability	50%
	Mutation rate	50%
	Mutation Depth	4
	Maximum offspring depth	10
	Number of Generations	100
GOOGP	Greedy Population size	500
OOGE	Population size	10
	Codon length	8
	Allele length	4
	Tournament size	40%
	Mutation probability	70%
	Bit flip probability	50%
	External crossover probability	80%
	Internal crossover probability	100
	Number of Generations	500

9.6 Chapter Summary

This chapter presents the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to evolve code for the queue ADT and Problem2. The next chapter will present the functions for fitness evaluation and parameters for the list abstract data type and a programming problem that uses the list.

CHAPTER 10: FITNESS EVALUATION AND PARAMETERS FOR THE LIST ADT AND PROBLEM3

10.1 Introduction

This chapter provides the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to produce code for the list Abstract Data Type (ADT) and Problem3 described in section 5.5.2 of chapter 5. Section 10.2 describes the programming problem specification for the list ADT while section 10.3 presents the OOGP, GOOGP and OOGE parameters for the list ADT. Section 10.4 and section 10.5 describe the programming problem specification and GOOGP and OOGE parameters respective for Problem1. Section 10.6 summarizes the chapter.

10.2 Programming Problem Specification for the List ADT

The list ADT has been described in chapter 5, section 5.5.1. Section 10.2.2 provides the OOGP and GOOGP primitives while section 10.2.3 presents the OOGE grammar. The following section describes the fitness cases and fitness evaluation.

10.2.1 Fitness Evaluation

The methods to be evolved for the list ADT are *makeNull*, *insertAt*, *getElement*, *removeElement*, and *empty*. As with the stack and queue, 15 cases are randomly generated for each run. Each fitness case is a list. Each individual can attain a maximum fitness of 15 per fitness case and 225 over all 15 fitness cases. The fitness cases are generated as described for the stack ADT. Table 10.1 shows the criteria that must be met by the list ADT methods.

Table 10.1 Problem specific criteria for the list ADT fitness evaluation

Methods	**Function**
makeNull()	• List pointer must be set to -1.
getElement(p)	• No change in pointer value. • Elements in the list should not be altered. • The value returned must be the value in the position indexed by p in the list. • Only one value must be returned.
insertAt (p, x)	• Pointer must be updated correctly. • The position of the elements in the list should be correctly updated. • The inserted value must be at the position indexed by p in the list. • The value must be inserted only once.
removeElement(p)	• Pointer must be updated correctly. • The position of the elements in the list should be correctly updated. Thus, the removed element must be that previously at the position indexed by p in the list before the execution of the method. • The removed value is returned
empty()	• No change in pointer value. • Elements in the list should not be altered. • The correct position of the pointer must be returned.

10.2.2 OOGP and GOOGP Primitives

The functions and terminals for the list ADT are given in Table 10.2. The constants are 1 and 0. The type of the variable is integer. The integer to be added to the list is represented by i while p represents an index in the list.

Table 10.2 OOGP and GOOGP functions and terminals for the list ADT

List ADT methods	**Primitives**
makeNull()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar
insertAt (p, x)	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, p, i, Cvar, Ivar
getElement(p)	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, p, Cvar, Ivar
removeElement(p)	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux , Cvar, Ivar
empty()	+, -, block2, for, write, read, inc_aux, dec_aux, set_aux, 0, 1, aux, Cvar, Ivar

10.2.3 OOGE Grammar

The grammar for the list ADT consists of the GE-terminals and GE-Non-terminals given in Table 10.3. The grammar is listed in **Appendix A.5.i.**

Table 10.3 GE-Non-terminals and GE-terminals for the list ADT

Start symbol	**Non-terminals**	**Terminals**
Stmts	stmts, stmt, expr, var, loopStmt, loopExpr, loopVar	+, -, write, read, set_aux, i, 0, 1, aux, dec_aux, inc_aux, Cvar, Ivar,), (, ;, }, {

10.3 Parameters for the List ADT

The OOGP, GOOGP and OOGE parameters for the list ADT are given in Table 10.4. Like the parameters for the stack and queue, these parameters were obtained by trial runs.

Table 10.4 Parameters for the list ADT

Approach	Parameter	Parameter value
OOGP and GOOGP	Population size	100
	Maximum tree depth	5
	Tournament size	2
	Crossover rate	100%
	Mutation rate	0%
	External crossover probability	50%
	Internal crossover probability	50%
	Maximum offspring depth	10
	Number of generations	100
GOOGP	Greedy population size	500
OOGE	Population size	1000
	Codon length	10
	Allele length	8
	Tournament size	4
	Mutation probability	40%
	Bit flip probability	70%
	External crossover probability	50%
	Internal crossover probability	80%
	Number of Generations	100

10.4 Programming Problem Specification for Problem3

Problem3 has been defined in section 5.5.2 of chapter 5 and is listed as one of the problems the list ADT can be used to solve. Section 10.2.1 describes the fitness evaluation. Section 10.2.2 provides the GOOGP primitives while section 10.2.3 presents the OOGE grammar.

10.4.1 Fitness Evaluation

An error function is used to determine the fitness of an individual in solving Problem3. The fitness is proportional to how small the error is. The set of fitness cases for this problem comprises of 10 integer arrays given in Table 10.5. The length of each case is between 4 and

10. The elements of each array are randomly chosen in the range 1 to 100. When executed, a solution is expected to adequately read the integer values for a given case; populate an empty list and sort the list. Each individual is executed on each of the 10 cases. The fitness of an individual in the population is calculated as follows:

1. For each fitness case, the number of inversions in the list is added to the fitness of the individual. The number of inversions in a list is a measure of the disorder in the list. If an array is sorted, the number of inversions is 0.
2. The fitness of an individual is increased by 2 for every integer in the given case that is not added to the list.
3. The fitness of the individual is increased by 2 if the number of elements in the list exceeds the number of elements in the given case.

The values obtained from 1 to 3 above are summed over all the fitness cases to obtain a fitness value. An individual with a fitness of zero is a solution.

Table 10.5. Fitness cases for Problem3

Fitness case
2, 3, 5, 8, 1, 89, 6, 40
9, 3, 5, 87, 2
20, 100, 68, 50, 9, 84
26, 4, 7, 21, 6, 8, 5, 1, 3, 12
100, 200, 71, 93
2, 3, 5, 4, 1, 89, 6, 40
9, 3, 1,0, 2, 87, 2
20, 1, 62, 50, 9, 84
5, 6, 8, 90, 1, 4, 3, 9
10, 20, 71, 93

10.4.2 OOGP and GOOGP Primitives

The function and terminal set for **Problem 3** are given in Table 10.6.

.

Table 10.6 GOOGP functions and terminals for Problem 3

Primitives
+, -, for_loop, block2, read, order, one, len, Cvar, Ivar, the five methods of the list ADT

10.4.3 OOGE Grammar

The grammar for Problem3 consists of the GE-terminals and GE-Non-terminals given in Table 10.7. The grammar is listed in **Appendix A.6.**

Table 10.7 GE-Non-terminals and GE-terminals for Problem3

Start symbol	**Non-terminals**	**Terminals**
stmts	stmts, stmt, expr, var, loopStmt, loopExpr, loopVar, cond, listMethods	+, -, for, block2, read, order, the five methods of the list ADT, len (the number of integers to be sorted), ADF0, one, len, Cvar, Ivar,}, {, ;,), (

10.5 Parameters for Problem3

The GOOGP and OOGE parameters for Problem3 are given in Table 10.8. These parameters were obtained empirically by trial runs.

Table 10.8 Parameters for Problem2

Approaches	Parameter	The Parameter Range
GOOGP	Population size	100
	Maximum depth	5
	Tournament Size	4
	Crossover rate	50%
	External crossover probability	50%
	Internal crossover probability	50%
	Mutation rate	50%
	Mutation Depth	4
	Maximum offspring depth	10
	Number of Generations	100
GOOGP	Greedy Population size	500
OOGE	Population size	500
	Codon length	10
	Allele length	8
	Tournament size	4
	Mutation probability	45%
	Bit flip probability	30%
	External crossover probability	75%
	Internal crossover probability	30%
	Number of Generations	100

10.6 Chapter Summary

This chapter presented the functions for fitness evaluation and parameters used by OOGP, GOOGP and OOGE to evolve code for the list ADT and Problem3. The next chapter will present the results of applying OOGP, GOOGP and OOGE to produce code for the object-oriented programming problems tested.

CHAPTER 11: RESULTS AND DISCUSSION

11.1 Introduction

This chapter reports on the results of applying OOGP, GOOGP and OOGE to produce code for three ADTs and the problems that use the ADTs. The ADTs are the stack, queue and list while the problems that use the ADTs are Problem1, Problem2 and Problem3 described in chapter 5, section 5.5.2. Whereas OOGP failed to produce at least one solution for the ADTs, GOOGP and OOGE were able to produce code for the classes and the programs that use the classes. The produced code are represented in tree format. Were the code could not be easily read in tree format these are presented in pseudo code.

Section 11.2 discusses the results obtained when applying OOGP, GOOGP and OOGE to produce code for the stack ADT and Problem1. Section 11.3 discusses the results obtained when applying OOGP, GOOGP and OOGE to produce code for the queue ADT and Problem2. The results of applying the approaches to produce code for the list ADT and Problem3 is discussed in section 11.4. Section 11.5 discusses the conversion of solution from an internal representation language to a programming language while section 11.6 compares OOGP, GOOGP and OOGE performances with other studies. The chapter summary is presented in section 11.7.

11.2 The Stack Abstract Data Types (ADT) and Problem1

This section reports on the results obtained when applying OOGP, GOOGP and OOGE to produce code for the stack ADT and Problem1 which uses the stack ADT. Section 11.2.1 presents and compares the performance of OOGP, GOOGP and OOGE when applied to produce code for the stack ADT. Section 11.2.2 reports on the results obtained for Problem1.

11.2.1 Comparison of OOGP, GOOGP and OOGE Performance for the Stack ADT

Each of the OOGP, GOOGP and OOGE approaches was tested for producing code for the stack ADT. Thirty runs were performed for each approach. The parameters have been listed in chapter 8, section 8.3. These parameters for OOGP and GOOGP are the same except that GOOGP uses an additional parameter, namely, the greedy population size. OOGE uses different parameters listed in chapter 8, section 8.3.

In the 30 runs of OOGP, no solution was found. The best individual was evolved in run 30. The fitness of the individual is 198. The individual did not correctly implement the *push*()

and the *pop*() methods of the stack ADT but satisfies all the specified criteria for the *makenull*(), *peak*(), and *empty*() methods.

In the 30 runs of GOOGP, 18 solutions were found. A solution was found in the initial generation of run 26. The solutions were found by GOOGP between the initial generation and generation 5. GOOGP found solutions early in the generations because the initial population is informed i.e., it uses the greedy method.

Fifteen solutions were found in the 30 runs of OOGE. The solutions were found between generation 12 and 97.

Table 11.1 shows the success rates, average fitness and average runtimes over 30 runs of the OOGP, GOOGP and OOGE. The results show that GOOGP performed competitively with OOGE. Based on the success rate, both GOOGP and OOGE outperform OOGP. From Table 11.1, GOOGP found three more solutions than OOGE. One cannot conclude that GOOGP outperforms OOGE or that OOGE outperforms GOOGP.

Table 11.1. Performance comparison of the approaches

Approach	Success Rate	Average Fitness	Average Runtime (ms)
OOGP	0	185.40	2146.23
GOOGP	60%	218.03	14998.4
OOGE	50%	217.96	15412.67

Figure 11.1 shows the average fitness of each generation for one run of each approach. The runs were chosen because the aim is to study the convergence over the entire run. Therefore all the runs that found solutions could not be examined. Hence, in the 30 runs of each approach, the run that produced the best individual in the last generation is examined. OOGP started with an initial population of individuals with a low fitness but converged prematurely at generation 5. The initial population of GOOGP started with a highly fit individual because the initial population was informed. It also converged prematurely at generation 5 but with a higher fitness than OOGP. OOGE started with an initial population of individuals with a low fitness. The fitness of the population improves as the generations progress. The slow

convergence is as a result of diversity maintained in the population. This has been described as one of the strengths of GE.

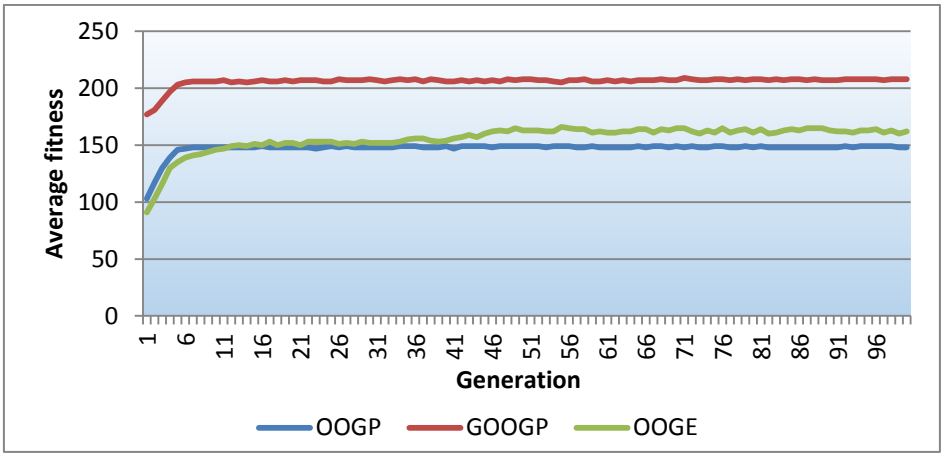


Figure 11.1 Convergence graph of the OOGP, GOOGP and OOGE approach showing the average fitness of each generation

Makenull()
set_aux − zero one

Push()
block2 + set_aux + aux one write i aux + aux one

Peek()
read aux

Pop()
− read aux block2 block2 zero zero set_aux − aux one

Empty()
+ aux zero

Figure 11.2 A stack ADT generated by GOOGP

```
makeNull()
set_aux ( − zero one ) ;

push()
write ( i , + one + zero aux ) ;
set_aux ( + one + zero aux ) ;

peek()
read ( aux ) ;

pop()
read ( aux ) ;
set_aux ( + − + + − + aux + one zero one zero zero one zero ) ;

empty()
aux ;
```

Figure 11.3 A stack ADT generated by OOGE

Figure 11.2 and Figure 11.3 show the solutions generated by GOOGP and OOGE respectively. The GOOGP solution was generated in the 3rd generation of run 5 while the OOGE solution was generated in the 87th generation of run 5. These solutions were chosen because they contain less redundant code and are easy to understand compared to other solutions generated. The *makenull()* method sets the pointer (aux) to -1 such that when an element is pushed onto the stack, it will be at the first position (i.e., index 0). Whereas the *push*() method generated by GOOGP increments the pointer before it pushes the value i onto the stack, the *push*() method generated by OOGE pushes a value onto the stack before it increments the pointer. Both the GOOGP and OOGE generated the same type of methods for the *peek(), pop()* and *empty()* methods. The *peek*() method reads the value of the pointer while the *pop*() method reads this value and consequently decrements the pointer. The *empty*() method returns an integer which if positive indicates that the stack is not empty. It indicates that the stack is empty if the value is negative. In both Figure 11.2 and Figure 11.3, the *empty*() method returns the pointer. This indicates whether or not the stack is empty provided that other methods work correctly.

11.2.2 Comparison of GOOGP and OOGE Performance for Problem1

From section 11.2.1, OOGP was not able to produce code for the stack ADT and hence is not evaluated for Problem1.

In 30 runs of GOOGP, no solution was found that correctly classifies all the 25 fitness cases provided. In run 2, a brittle solution was generated. A brittle solution does not generalize. Figure 11.4 shows the brittle solution. The brittle solution compares the first and last characters of the given case and returns the value of true if they are the same but false if they are not. The first character in a palindrome is the same as the last character in the palindrome. However, the first character in a string that is not a palindrome is not always different to the last character in the string. For example, the word "deepened" is not a palindrome but it has 'd' as the first and last character. Thus the solution does not generalize because the fitness cases provided do not represent a sufficient amount of the problem domain.

In Figure 11.4, the subtree that forms the first argument of the *strequal* node returns the last character in a given string represented by *arg*. This is done as follows. The second argument of the *for_loop1* operator executes a number of times determined by its first argument. In this case it executes a number of times equal to the length of *arg*, i.e. *len* + 1 + 0 – 1 times. At the beginning of the first iteration, the counter variable of the *for_loop1* is set to 0, the default value. At each of the iterations, the *Char_At* operator returns the character in *arg* specified by *Cvar1*, i.e., the counter. This character is stored in the iteration variable of the *for_loop1* operator. At the end of the last iteration, the operator returns the value of the iteration variable which in this case is the last character in *arg*.

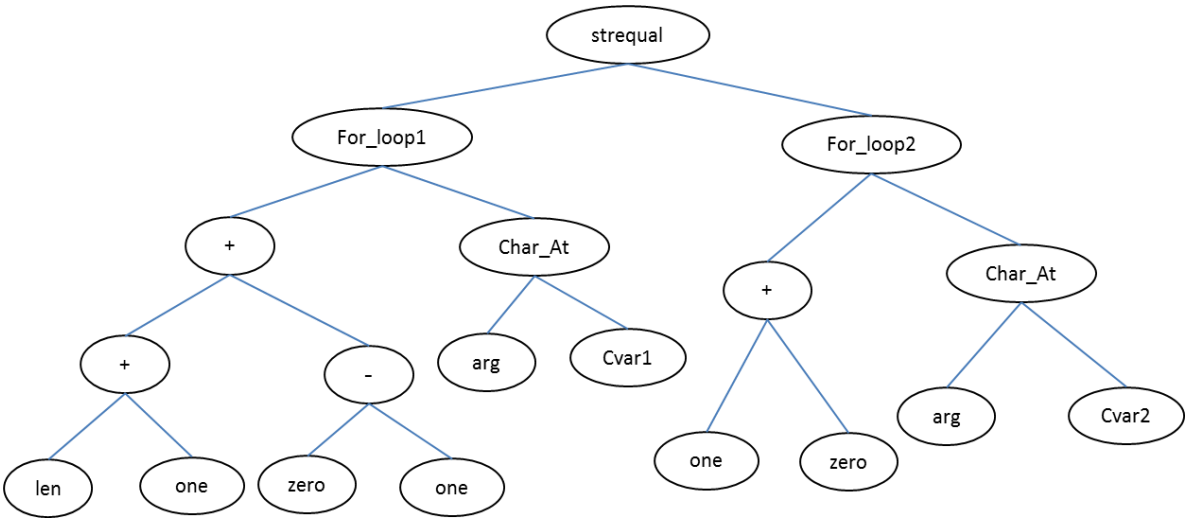


Figure 11.4 A brittle solution to the palindrome problem

The *for_loop2* operates just as the *for_loop1* but iterates only once. Thus it returns the first character in *arg*. The *strequal* operator compares the character returned by executing its first and second arguments and returns the value of true if they are the same but false if they are not.

It can be observed from Figure 11.4 that the individual did not make use of any stack method. This defeats the main aim of this implementation which is to use the stack ADT for solving the palindrome problem. In order to avoid generating brittle solutions, the changes specified in the next section were made.

11.2.2.1 Improvements and Changes

Two major changes were made to the programing problem specification. The first is that the set of fitness case was changed to contain 10 cases. Five of the cases are palindromes while 5 are strings that are not palindromes. Also, the given strings that are not palindromes were changed such that each string starts and stops with the same character. The fitness cases are given in **Appendix F**. The second is that the fitness evaluation is adjusted to reward the individuals that make use of the stack methods. With the changes, the possible maximum fitness is 7 multiplied by the number of fitness cases. Thus the new possible maximum fitness is 7*10 which is 70 and is obtained as follows:

- For each case, the fitness of an individual is increased by 1 if the number of elements pushed onto the stack is equal to the length of the given string. Also, the fitness of the individual is increased by 1 if each element pushed onto the stack is a character in the given case. (2 points)
- For each case, the fitness of an individual is increased by 3 if the *pop*() method is called a number of times equal to the length of the given string. (3 points)
- For each case, the fitness of an individual is increased by 2 if the individual correctly classifies the case. (2 points)

In 30 runs of GOOGP, no solution was found. The best individual of the 30 runs was generated in run 14. It has a fitness of 62 out of 70. The individual makes use of the stack methods. Figure 11.5 shows the fitness convergence of the individual.

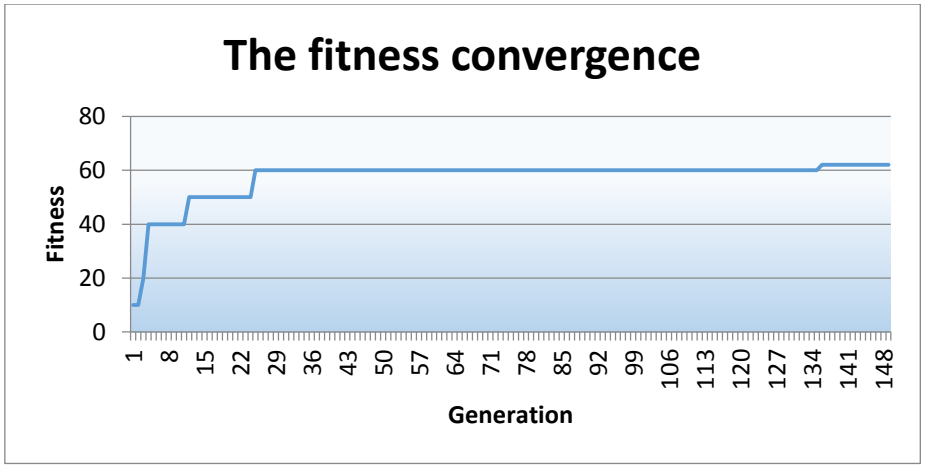


Figure 11.5 The fitness convergence of the best individual generated in run 14 for Problem1

The algorithm converged faster between generations 1 to 25. It took about 120 more generations before a better fitness of 62 was found at generation 135. The fitness remained the same up until generation 500.

Since GOOGP failed to find a solution, it is anticipated that the problem is complex. As specified in chapter 2, section 2.12.4, the aim of ADFs is to reduce the complexity of solving a problem by breaking it down into sub-problems. Langdon [6] used ADFs in solving difficult problems. Koza [7] also stated that ADFs help when used for complex problems. For these reasons, the programing problem specification is changed to include ADFs. This study uses two ADFs, namely, ADF1 and ADF2, for Problem1. ADF1 returns no value while ADF2 returns a string. This gives GOOGP more choice to decompose and solve the problem. To avoid recursive calls, only the main program is allowed to call the ADFs. Table 11.2 shows the primitives for the new changes made to the programming problem specification for the GOOGP approach. This is the final change. Hence OOGE uses two ADFs to allow a fair comparison of the two approaches. The modified grammar for the OOGE is given in **Appendix B.1.**

Table 11.2 The final primitives for the GOOGP approach

	Primitives	Stack Methods
Main program	+, -, block2, strequal, append, char_At, for, ADF1, ADF2, one, arg, zero, len, Cvar, Ivar, arg	All the five evolved methods
ADF1	+, -, block2, char_At, for_loop, arg, one, zero, len, Cvar, Ivar,	All the five evolved methods
ADF2	+, -, block2, append, char_At, for_loop, one, zero, len, Cvar, Ivar	All the five evolved methods

11.2.2.2 Performance comparison of GOOGP and OOGE

GOOGP found 4 solutions in the 30 runs performed. A solution was found early in generation 5. The 4 solutions were found between generation 5 and 18. OOGE found 3 solutions in 30 runs. These solutions were found in generations 29, 13 and 18. Table 11.3 shows the performance comparison of GOOGP and OOGE with ADFs. The average fitness of GOOGP is higher than the average fitness of OOGE. The GOOGP average fitness is 59.33 and the highest obtainable fitness is 70. The OOGE average fitness is 55.33. The average runtime of the GOOGP approach is less than the average runtime of the OOGE approach. On inspection of the solutions, it was found that the solutions generated by OOGE use a lot more iterations than those generated by GOOGP. Thus the lower average runtime is as a result of GOOGP not making much use of the iterative operator in the main program but relies on the ADFs to perform iterations. Iteration increases the OOGE runtime.

Statistical testing was conducted to determine the significance of the results. The hypothesis tested is that the runtime of GOOGP with ADFs is better than the runtime of the OOGE with ADFs. The hypothesis was found to be significant at 1% level of significance.

Table 11.3. Performance comparison of the GOOGP and OOGE (each uses two ADFs)

Approach	**Success Rate**	**Average Fitness**	**Average Runtime (ms)**
GOOGP	13.33%	59.33	34349.43
OOGE	10%	55.33	102065.63

One of the solutions generated by GOOGP is shown in Figure 11.6. The solution correctly classifies all the given strings. It also can be used to check whether or not a string it has not seen before is a palindrome. When executed, the ADF1 correctly pushes all the characters in a given case onto a stack. The ADF2 pops and appends each character to the previously popped character. This forms a string which is returned to the main program at the end of the last iteration. The string returned by ADF2 is compared with the *arg* (the string supplied in the fitness case). The result of this comparison is a value of true or false which correctly classifies the given cases and any string that may be given to the solution.

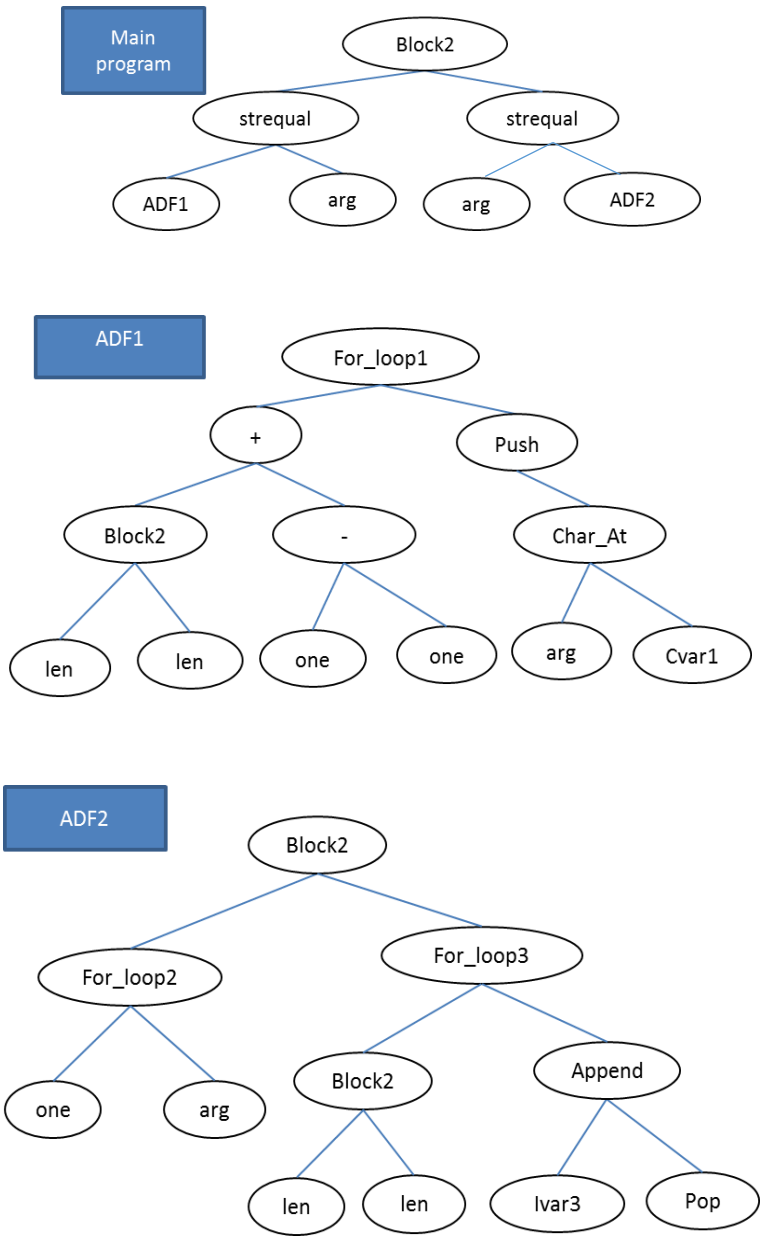


Figure 11.6 A solution generated by GOOGP with a main program and two ADFs, namely ADF1 and ADF2

11.3 The Queue Abstract Data Types (ADT) and Problem2

This section reports on the results obtained when applying OOGP, GOOGP and OOGE to produce code for the queue ADT and Problem2 which uses the queue ADT. Section 11.3.1 presents and compares the performance of OOGP, GOOGP and OOGE when applied to produce code for the queue ADT. Section 11.3.2 reports on the results obtained for Problem2.

11.3.1 Comparison of OOGP, GOOGP and OOGE Performances for the Queue ADT

Each of the OOGP, GOOGP and OOGE approaches was tested for producing code for the queue ADT. Thirty runs were performed for each approach using the parameters listed in chapter 9, section 9.3. In the 30 runs of OOGP, no solution was found. The best individual meets the criteria for all methods except the *dequeue*() method.

In the 30 runs of GOOGP, 2 solutions were found. Both the solutions were generated in the 2nd generations of run 1 and 21. As mentioned earlier, GOOGP found solutions early in the generations because the initial population is informed. The performance of the GOOGP approach deteriorated compared to its performance when tested for producing code for the stack ADT.

In the 30 runs of OOGE, 10 solutions were found. These solutions were found between generation 15 and 93. This is an indication that OOGE converges slower than GOOGP which found solutions early in the runs.

Table 11.4. Performance comparison of the approaches

Approach	**Success Rate**	**Average Fitness**	**Average Runtime (ms)**
OOGP	0%	198.4	-
GOOGP	6.67%	212.33	198698.5
OOGE	33.33%	215.77	78783.5

Table 11.4 shows the success rates, average fitness and average runtimes of OOGP, GOOGP and OOGE. OOGP has 0% success rate which means that no solution was found. As indicated by the success rate column, the performance of GOOGP deteriorated by 53.33% from its performance when tested for producing code for the stack ADT and the performance of the OOGE approach deteriorated by 16.67%. Both the success rate and the average fitness

of OOGE are high. The average runtime of the OOGE approach is less compared to that of GOOGP.

Statistical testing was conducted to determine the significance of the results. The hypotheses tested are:

- Hypothesis 1: OOGE performs better than GOOGP.
- Hypothesis 2: The runtimes of OOGE are better than GOOGP.

Table 11.5 Z-values for hypotheses tests

Hypothesis	**Z-Value**
Hypothesis 1	2.74
Hypothesis 2	2.09

Table 11.5 lists the Z-values. Hypothesis 1 was found to be significant at 1% level of significance while hypothesis 2 was found to be significant at 5% level but not at 1% level.

Figure 11.7 shows a solution generated by GOOGP. The *makeNull*() method contains introns. For example, the subtree rooted by the *for8* operator returns 0 given any value of *aux*. The *set_aux* node sets the pointer to this value, i.e 0. The second argument of *fblock2* decrements the pointer by 1. Hence the pointer becomes -1. The *enqueue*() method increments the pointer *aux*. It then writes the value *i* to the memory indexd by the pointer. In the *front*() method, the subtree, *set_aux* (*aux*), is an intron. It returns 0. Hence, the *front*() method reads the value stored in the 0th index of the queue, i.e the front of the queue. The *empty*() method returns the value of the pointer. Figure 11.8 shows the *dequeue*() method. The method uses the *for* operator to move each element to a new position in the queue. The *dequeue*() method is explained in details in Figure 11.9 which also highlights the importance of the *for* operator in moving the elements in the queue. The *dequeue*() method shown in Figure 11.8 is labelled with alphabets a, b, c, d and e which are referred to in the illustration presented in Figure 11.9.

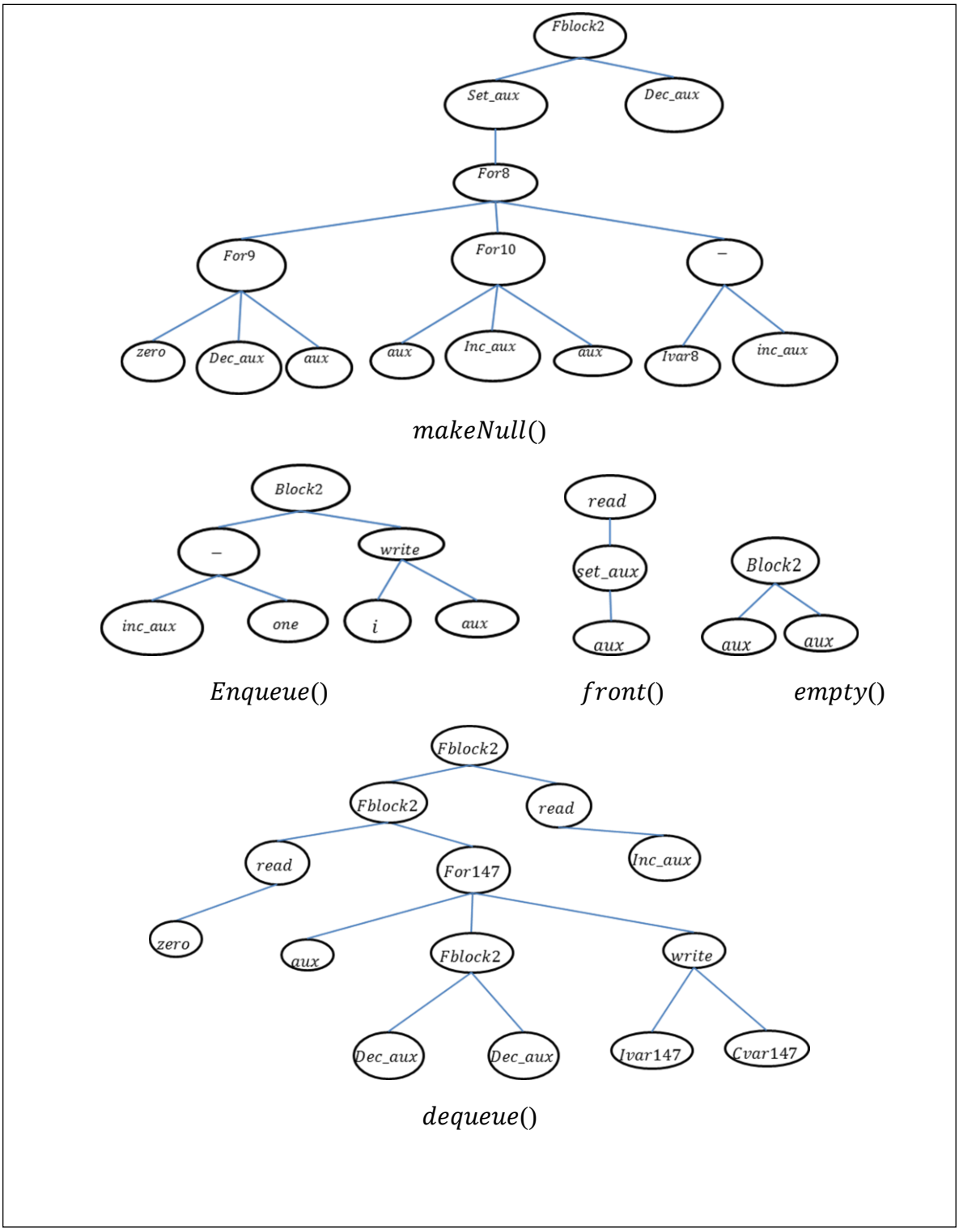


Figure 11.7: An example Queue solution generated by GOOGP

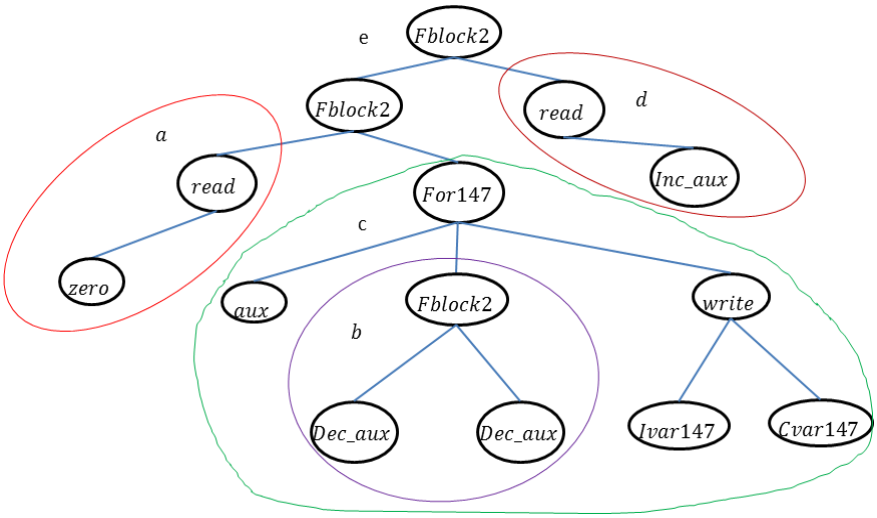
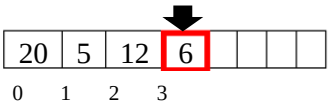


Figure 11.8: A *dequeue*() method generated by GOOGP

Assume the initial state of the queue as:



The pointer i.e., aux is 3. The *dequeue*() method is execute as shown below:

Code	Explanation
$a = read\ (0) = 20$	• Reads and returns the element at position 0, i.e., 20.
$b = for147(\ aux, c, write\ (Ivar147, Cvar147))$ $= for147(\ 3, c, write\ (Ivar147, Cvar147))$	• aux = 3. • From the cell below, c = 0.
But: $c = fblock2\ (dec\_aux, dec\_aux)$ $= fblock2(0,0) = 0$	• fblock2: executes its two arguments and returns the value of the first argument. • The first dec_aux decrements the pointer by 1. Again, the second decrements it by 1. • New value of aux = 1, i.e 3 – 1 – 1 = 1.

Hence: $b =$
$for147(\ 3, 0, write\ (Ivar147, Cvar147))$
 	
Pseudocode equivalent of b :
$for147\ (j = 3, j >= 0; j--)$
{
$Write\ (Ivar147, Cvar\ 147)$
}
Hence
$b = 20 = Ivar147$
	• Write(i, j): writes i to jth position and returns the element previously at jth position.
	• Ivar147: holds the value returned by the for147 operator after each iteration.
	• Cvar147: the counter for the for147.
	• Initial value of Ivar147 is 0.
	• Initial value of cvar147 is 3.
	The values of the variables after each iteration are:
	Action | Queue index | Value in the queue index | Ivar147 | Cvar147
	Write(0,3) | 3 | 0 | 6 | 2
	Write(6,2) | 2 | 6 | 12 | 1
	Write(12,1) | 1 | 12 | 5 | 0
	Write(5,0) | 0 | 5 | 20 | -1

$d = read\ (inc\_aux)$ = read(0) = 5	• The aux increments by 1. Hence aux = 2.
$e = fblock2\ (fblock2\ (a, b), d)$ $= fblock2\ (a, d)$ $= a$= 20	A value returned by the dequeue() method is 20. The final state of the queue is: 5 | 12 | 6 | 0 0 1 2 3

Figure 11.9: An illustration of the *dequeue()* method generated by GOOGP

```
makeNull()
for32933{zero,aux,write(Ivar32933,dec_aux)};

enqueue()
set_aux(aux);
write(i,−inc_aux − zero aux)

front()
read(− + + − aux − zero inc_aux one one one);
read(dec_aux)

dequeue()
one;
for32934{aux,dec_aux,write(Ivar32934,cvar32934)}

empty()
aux;
```

Figure 11.10: An example Queue solution generated by OOGE

Figure 11.10 shows a solution generated by OOGE. The solution is represented in prefix notation. As with the solution found by GOOGP, the *for* operator is used to update the queue elements. Rather than using the *set_aux* operator to set the pointer to -1, the *makeNull*() method uses the *for* operator to iteratively decrement the value of *aux* until it becomes -1. The first program statement of the *enqueue*() method is an intron. In the method, the statement $write\,(\,i\,,-\,inc\_aux\;-\;zero\;aux\,)$ increments the pointer, i.e., aux and writes the element represented by i to the position indexed by the pointer. The first program statement of the *front*() method increments the pointer which is decremented by *dec_aux* in the second program statement. Thus the pointer remains the same. The *dec_aux* returns the default value of the pointer, i.e 0. The *front*() method, therefore, returns the first element, i.e., element indexed by this value, in the queue. As with the *dequeue*() method generated by GOOGP, the *dequeue*() method generated by OOGE uses an iterative operator to move the elements in the queue. The method returns and removes the first element in the queue while the *empty*() method returns the value of aux which if negative indicates that the queue is empty.

11.3.2 Comparison of GOOGP and OOGE Performances for Problem2

OOGP was not able to produce code for the queue ADT and hence is not evaluated for Problem2.

In the 30 runs of the GOOGP, no solution was found. On inspection of the best individual in the last generation of each run, most of the individuals make use of the *for_loop* operator in

more than two places. Also, each of the individuals with a high fitness make use of one instance of the *while* operator. Conversely, the worst individuals of the runs do not make use of the *for_loop* operator. It therefore looks like GOOGP cannot solve Problem2 using a single algorithm and requires the problem to be broken down.

11.3.2.1 Improvement and changes

As indicated in Koza [7] and Banzhaf *et al*. [5], ADFs reduce code repetition and promote code reusability. Also, as stated in section 11.2.2.1, Langdon [6] used ADFs in solving difficult problems. For these reasons, the programming problem specification is changed to include an ADF. It is anticipated that a return value will not be needed because the program only needs to manipulate a memory. Hence, an ADF which returns a value is not used. The ADF included in the programming problem specification has no argument. The value returned is ignored. The new function and terminal sets for the GOOGP is shown in Table 11.6. OOGE also uses one ADF to allow a fair comparison of the two approaches. The modified grammar for OOGE is presented in **Appendix B.2.**

Table 11.6. The Revised GOOGP functions and terminals for Problem 2

	Function Set	Queue methods
Main program	for_loop, block2, block3, getArity, while, Not, root (the root of the tree), Cvar, Ivar, ADF, getElement	All the five methods evolved
ADF	for_loop, block2, block3, getArity, root (the root of the tree),Cvar, Ivar, getElement	All the five methods evolved (except the *empty*)

11.3.2.2 Performance comparison of GOOGP and OOGE

GOOGP found 9 solutions in 30 runs while OOGE found 29 solutions in 30 runs. Table 11.7 shows the success rates, average fitness and average runtimes of the GOOGP and OOGE approaches. OOGE has a success rate which is 66.67% better than the success rate of the GOOGP approach. Also, OOGE has a better average fitness of 57.87 while GOOGP has an average fitness of 33.17. The average runtime of the OOGE approach is slightly higher than the average runtime of the GOOGP approach. The average runtimes are 54965.17 and 50287.27 for the OOGE and GOOGP approaches respectively. This is expected as OOGE converges slower than the GOOGP which converges faster because the initial population is informed.

Table 11.7. Performance comparison of GOOGP and GE (each uses an ADF)

Approach	Success Rate	Average Fitness	Average Runtime (ms)
GOOGP	30%	33.17	50287.27
OOGE	96.67%	57.87	54965.17

Statistical testing was conducted to test the significance of the results. Two hypotheses were tested. The hypotheses are:

- Hypothesis 1: OOGE performs better than GOOGP.
- Hypothesis 2: The runtimes of GOOGP are better than OOGE.

Hypothesis 1 was found to be significant at 1% level. As shown in Table 11.8, the Z-value is 7.42. Hypothesis 2 has a Z-value of 0.10 and hence not significant at any level of significance.

Table 11.8. Z-values for hypothesis tests

Hypothesis	Z-Value
Hypothesis 1	7.42
Hypothesis 2	0.10

Figure 11.11 is among the 9 solutions found in 30 runs. It is also one of the simplest solutions found. The two *read* nodes generated as part of the subtree that forms the first argument of the *Block2* node are introns. The subtree that forms the first argument of the *Block2* node enqueues the root of the tree provided in the fitness case. The first argument of the *while* operator checks whether or not the queue is empty. If the queue is not empty, the second argument of the *while* node is executed. At the first check, the queue is not empty because an element has been enqueued. The second argument of the *while* operator calls the ADF. The *getArity* node is an intron.

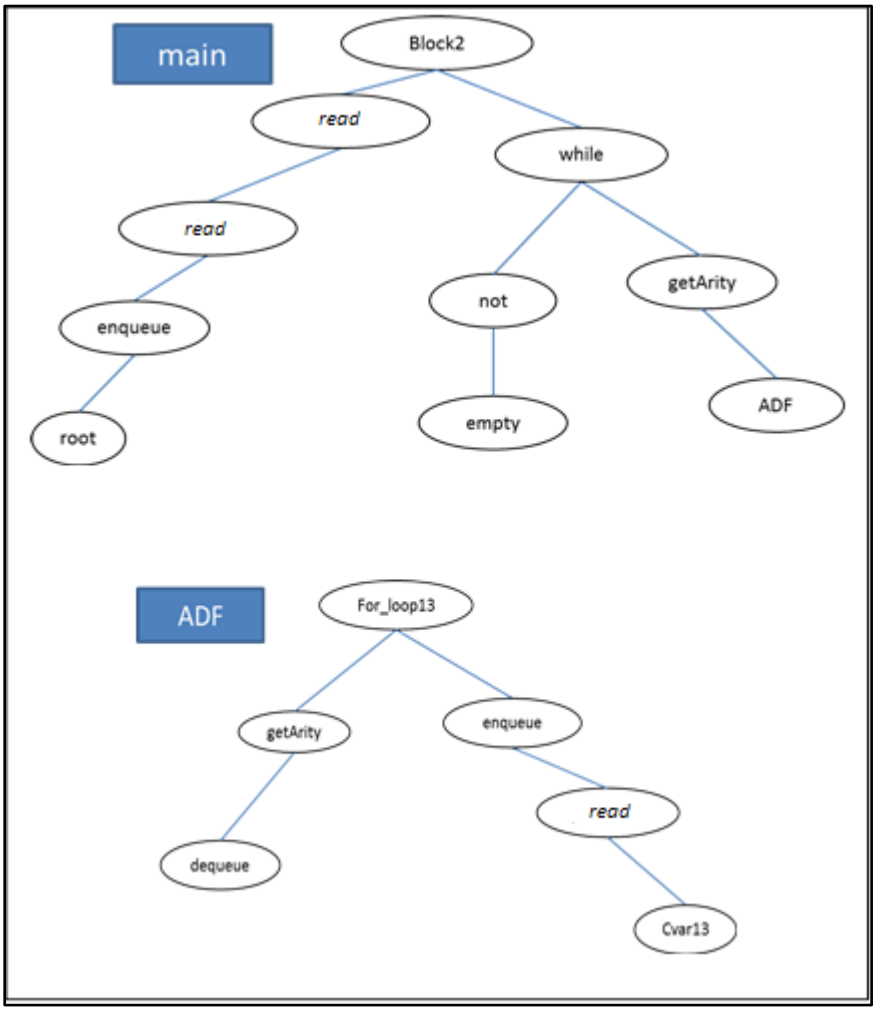


Figure 11.11: A GOOGP solution algorithm for the BFS problem

When the ADF is executed, the subtree that forms the first argument of the *for_loop13* operator dequeues an element. This element is, at first, the root of the tree provided in the fitness case. The arity of that root is returned by the *getArity* node. The second argument executes a number of times specified by the arity of the latest dequeued element. The counter variable, namely, *Cvar13* keeps track of the number of iterations of the *for_loop13* operator. It is initially set to 0. The *read* operator reads a child node of the latest dequeued node at the position indexed by the *Cvar13*. The child node is then enqueued. After the last iteration, the execution of the ADF ends. The main program continues by checking if the queue is empty. When the solution is executed, the queue can only be empty when the last node i.e., the right-most leaf node, of a tree provided in the fitness case has been dequeued. Figure 11.12 shows the pseudocode of the generated solution.

```
Main()
Enqueue (root);
While (! queue.isEmpty()){
        ADF();
}
ADF()
      int cvar13;
       Node currentNode = dequeue;
       int currentArrity = getArity( currentNode);
       For (cvar13 = 0; cvar13 < currentArrity; cvar13 + +){
             enqueue(currentNodeList. getElement(cvar13))
       }
```

Figure 11.12 The generated solution algorithm in Figure 11.11

11.4 The List Abstract Data Types (ADT) and Problem3

This section reports on the results obtained when applying OOGP, GOOGP and OOGE to produce code for the list ADT and Problem3 which uses the list. Section 11.4.1 presents and compares the performance of OOGP, GOOGP and OOGE to produce code for the list ADT while section 11.4.2 presents the results obtained for Problem3.

11.4.1 Comparison of OOGP, GOOGP and OOGE Performances for the List ADT

Each of the OOGP, GOOGP and OOGE approaches was tested for producing code for the list ADT. Thirty runs were performed for each approach using the parameters listed in chapter 10, section 10.3.

In the 30 runs of OOGP, no solution was found. Also, no solution was found in the 30 runs of GOOGP. Thus none of the GP approaches was able to produce code for the list ADT. On inspection of the best individuals generated by OOGP and GOOGP, it was found that the

insertAt() and *removeElement*() methods were not correctly induced. Both these methods are more complex because they require moving the elements in the list.

OOGE found 3 solutions in the 30 runs of the approach. These solutions were generated at generation 76, 74, and 94. This shows that OOGE takes long to find a solution and is reflected in the runtime given in Table 11.9. The table also shows the success rates and average fitness of the OOGP, GOOGP, and OOGE approaches. OOGP has the least average fitness. The average fitness of GOOGP is higher than the average fitness of OOGE without a corresponding higher success rate.

Table 11.9. Performance comparison of the approaches

Approach	**Success Rate**	**Average Fitness**	**Average Runtime (ms)**
OOGP	0%	176.73	62163.17
GOOGP	0%	208.57	47858.53
OOGE	10%	207.80	371369.97

Producing code for the list ADT is more difficult than producing code for the stack and queue ADTs. None of the GP approaches were able to produce code for the list ADT because the *insertAt*() and *removeEelement*() methods are complex to induce. Koza [7] stated that ADFs improve GP success rate when used for a complex problem. Langdon [6] used an ADF in producing code for the list ADT. Hence, the programing problem specification is changed to include an ADF. The ADF takes two arguments and returns a value. It can use none, one or both the arguments. This is done to allow GP decide how to use the ADF. The elements of the function and terminal sets for the ADF are {+, −, *, /, block2, for, write, Cvar, Ivar, zero, one, arg1, arg2}. OOGE also uses an ADF to allow (1) a comparison of OOGE with and without an ADF, and (2) a fair comparison of the OOGP, GOOGP and OOGE approaches. The modified grammar for OOGE is presented in **Appendix A.5.ii**.

Again, 30 runs were performed for each of OOGP, GOOGP and OOGE. In the 30 runs of OOGP, no solution was found. In the 30 runs of GOOGP, 4 solutions were found. OOGE found 2 solutions. Whereas the 4 solutions found by GOOGP use an ADF, only one of the two solutions found by GE uses an ADF. There is no notable improvement in the success rate

of OOGE with an ADF compared to OOGE without an ADF. Table 11.10 shows the success rates, average fitness and average runtimes of the approaches.

Table 11.10. Performance comparison of the approaches

Approach	Success Rate	Average Fitness	Average Runtime (ms)
OOGP	0%	195.96	20846.07
GOOGP	13.33%	211.83	50673.77
OOGE	6.67%	207.53	586688.37

A solution generated by OOGE without ADFs is shown in Figure 11.13. Introns have been partially removed to aid understanding of the solution. The *makeNull*() method sets *aux* to -1. The *insertAt*() method uses the *for* operator to adequately update the position of each element before it writes the element, say i, to a position specified by p. The *removeElement*() decrements the pointer by 1 when the second argument of the *for* operator is evaluated. The *for* operator then correctly updates each element and returns the element at position p.

```
makeNull( )
set_aux (+ − − aux + + inc_aux aux zero zero zero);

inserAt(p, i )
for1{+ − zero inc_aux p, aux, write ( Ivar1 , Cvar1 ) } ;
write ( i , p ) ;

getElement(p )
read ( p );

removeElement( )
for18{aux, + dec_aux p, write ( Ivar18 , Cvar18 ) } ;

empty( )
aux ;
```

Figure 11.13: A solution generated by GE without ADFs

11.4.2 Comparison of GOOGP and OOGE Performances for Problem3

OOGP was not able to produce code for the list ADT and hence is not evaluated for Problem3.

In the 30 runs of GOOGP, no solution was found. Figure 11.14 is the best individual and was obtained from the last generation of run 8. The fitness of the individual is 207. On inspection of the individual, it was found that the individual adds elements to the list but could not sort the list. It was anticipated that the individual failed to sort the list because of one or both of the following reasons. (1) The functions are not sufficient for GP to be used to sort the list. (2) Populating and sorting a list is a complex problem which GP cannot solve using a single algorithm. The problem therefore is required to be broken down. For these reasons, the programming problem specification was changed as specified in the next section.

block2 block2 for_loop340648 block2 block2 block2 len len block2 len len block2 block2 len len len insertat block2 block2 empty zero cvar340648 read cvar340648 for_loop340649 block2 block2 block2 len zero len len removeelement cvar340649 for_loop340649 block2 len block2 len len removeelement cvar340649

Figure 11.14 Best individual generated by the trial run of GOOGP for Problem3

11.4.2.1 Improvements and changes

In the study conducted by Kinnear [63], GP using different high order operators were investigated for evolving a sort algorithm. GP using the *swap* and *order* operators were compared. The study shows that GP using the *order* operator evolves a simple solution early in a run. Hence an *order* operator was included in the programing problem specification. The *order* operator takes two integer arguments. Each of the arguments is a memory index. The operator functions as follows: If the value, say x, stored in the memory indexed by the first argument is greater than the value, say y, stored in the memory indexed by the second argument, these values are swapped. The operator does nothing if x is less than or equal to y. If an argument of the *order* operator evaluates to a value less than zero or greater than the size of the memory structure of the program, (a) the operator does nothing and (b) a default value for the type of the operator is returned. Despite the inclusion of the *order* operator in the programing problem specification, no solution was found in the 30 runs of GOOGP.

Whereas previous studies [58–61] give GP a list in the memory to sort, GP implemented in this study constructs the list while sorting it. Hence the problem is more challenging. As a result, an ADF was included in the programming problem specification. ADFs improve GP

success rate when used for a complex problem [7]. As with Problem2, it is anticipated that a return value will not be needed because the program only needs to manipulate memory. Thus, the value returned is ignored. Also the ADF has no arguments. The new primitives for the GOOGP approach are given in Table 11.11. The grammar for the OOGE approach was changed to include an ADF and the *order* operator. This is done to allow a fair comparison of the two approaches. The grammar is given in **Appendix B.3.**

Table 11.11 The Revised GOOGP functions and terminals for Problem 3

	Functions and Terminals
Main program	+, -, for_loop, block2, read, order, one, len, Cvar, Ivar, ADF, the five methods of the list
ADF	+, -, for_loop, block2, read, one, order, len, Cvar, Ivar, the five methods of the list

11.4.2.2 Performance comparison of GOOGP and OOGE

In the 30 runs of GOOGP, one solution was found. The solution was generated in generation 3. OOGE found 4 solutions in 30 runs. The solution generated by GOOGP and the 4 solutions generated by OOGE use an ADF and the *order* operator. Table 11.12 shows the performance comparison of the two approaches. OOGE has a better success rate than GOOGP. Both the average fitness and runtime of the GOOGP approach are better than the OOGE's. The better average fitness of the GOOGP approach is as a result of the greedy approach used in the initial population.

Table 11.12 Performance comparison of GOOGP and OOGE for problem 3

Approach	**Success Rate**	**Average Fitness**	**Average Runtime (ms)**
GOOGP	3.33%	56.77	381160.3667
OOGE	13.33%	363.8	94790.86667

Statistical tests were conducted to test the statistical significance of the results. The two hypotheses tested are as below:

- Hypothesis 1: OOGE performs better than GOOGP.
- Hypothesis 2: The runtimes of GOOGP are better than OOGE.

Hypothesis 1 was found to be significant at 10% level but not at the other levels of significance. As shown in Table 11.13, the Z-value is 1.42. Hypothesis 2 has a Z-value of 7.34 which is significant at 1% level of significance.

Table 11.13 Z-values for hypothesis tests

Hypothesis	**Z-Value**
Hypothesis 1	1.42
Hypothesis 2	7.34

The solution generated by GOOGP is given in Figure 11.15. Introns have been removed to aid understanding of the solution.

```
ADF()
for_loop190955 len insertAt cvar190955 read cvar190955

Main()
block2 ADF for_loop251109 len for_loop251110 − len one order
cvar251109 cvar251110
```

Figure 11.15 A solution generated by the GOOGP approach

The variable *len* specifies the number of integers in the fitness case provided. For example, given a fitness case **A** = [3, 20, 7, 6], the value of *len* is 4. The ADF in Figure 11.15 contains a *for_loop* operator which iterates a number of times equal to *len*. Using the fitness case **A** and an empty list **L** = { }, Table 11.14 shows a trace for the ADF. On each iteration, a value is read from **A**. The value is read from a position indexed by the counter variable, i.e., *Cvar190955*, and is inserted at a similar index in the list. The insertion is carried out by the *insertAt()* method of the list ADT. Note that the initial value of the counter variable for the *for_loop* operator is 0. Hence the first value in **A**, i.e. 3, is inserted at a position in the list indexed by 0.The counter variable is incremented by 1 on each iteration. Thus the next index is 1 and the value to be inserted at the index is 20. This process continues until the *for_loop* operator iterates a number of times equal to *len* at which the list is populated. At the beginning of the main method, the ADF is invoked which causes the list to be populated. A nested loop and the *order* operator are then used to sort the list. Table 11.15 shows the trace for the main program.The outer loop in the main program uses the *for_loop251109* which

iterates a number of times specified by *len*. The inner loop uses the *for_loop251110* which iterates *len* – 1 times. Note that the initial values of *Cvar251109* and *Cvar251109* are the same. This value is 0. On each iteration of the inner loop, the *i* in the *order* function is replaced with *Cvar251109* while the *j* is replaced with *Cvar251110* as shown in Table 11.15. The *order* operator then swaps the values stored at *i* and *j* if the value stored at *i* is greater than the value stored at *j*. The operator does nothing if the value stored at *i* is less than or equal to the value stored at *j*. In lines number 4, 8, and 12 of Table 11.15, the value stored at *i* in the list **L** is greater than that stored at *j*. Hence these values were swapped. Any given populated list will be sorted at the end of the outer loop.

Table 11.14. A trace for the ADF generated by GOOGP

S/N	**Values of CVAR190955**	**L**
1	0	{3}
2	1	{3, 20}
3	2	{3, 20, 7}
4	3	{3, 20, 7, 6}

Table 11.15. A trace for the main program generated by GOOGP

S/N	**Values of** CVAR251109 (*i*)	**Values of** CVAR251110 (*j*)	**L (before executing the *order* function)**	**Oder(*i*, *j*)**	**L (after executing the *order* function)**
1	0	0	{3, 20, 7, 6}	Order(0,0)	{3, 20, 7, 6}
2	0	1	{3, 20, 7, 6}	Order(0,1)	{3, 20, 7, 6}
3	0	2	{3, 20, 7, 6}	Order(0,2)	{3, 20, 7, 6}
4	1	0	{3, 20, 7, 6}	Order(1,0)	{20, 3, 7, 6}
5	1	1	{20, 3, 7, 6}	Order(1,1)	{20, 3, 7, 6}
6	1	2	{20, 3, 7, 6}	Order(1,2)	{20, 3, 7, 6}
7	2	0	{20, 3, 7, 6}	Order(2,0)	{20, 3, 7, 6}
8	2	1	{20, 3, 7, 6}	Order(2,1)	{20, 7, 3, 6}
9	2	2	{20, 7, 3, 6}	Order(2,2)	{20, 7, 3, 6}
10	3	0	{20, 7, 3, 6}	Order(3,0)	{20, 7, 3, 6}
11	3	1	{20, 7, 3, 6}	Order(3,1)	{20, 7, 3, 6}
12	3	2	{20, 7, 3, 6}	Order(3,2)	{20, 7, 6, 3}

11.5 Conversion of the Solutions to a Programming Language

In this study, GP and GE produce code in an internal representation language. This facilitates the evolution of language independent programs. However, it is important that the code be compiled and run to achieve automatic programming. As mentioned in chapter 4, section 4.6, this study illustrates that the generated solutions can be converted to a programming language. Programming languages that support object-oriented programming include Ruby, Visual Basic, C++, Python and Java. Java has been chosen for this illustration because (1) it is widely used to teach object-oriented concepts and, (2) it is platform independent, .i.e., Java code need not to be recompiled to run on another machine. Thus, the code produced for the list ADT and Problem3 is converted to Java. This object-oriented programming problem is used for the illustration because it is of a hard difficulty level. Since OOGP found no solution for the list ADT, the solutions found by GOOGP are used for the illustration. Thus, the program implemented to generate code using GOOGP provides the option to convert the code for the list ADT and Problem 3 to Java. This can be seen in the user manual given in **Appendix E**.

A solution for the list ADT converted to Java is provided in **Appendix C**. It was generated in run 3 of GOOGP. As expected, there is redundancy in the Java code. This is as a result of introns in the solution. Also, the code generated by GP is not necessarily the same as that written by humans. This is in consistent with the study conducted by Koza [4] which revealed that GP can produce a solution which is not expected by humans. The solution generated by GOOGP for Problem3 is converted to Java and given in **Appendix D**. There is also some redundancy in the code. However, both the code presented in **Appendix C** and **Appendix D** prove that automatic programming is possible. Although this is illustrated for one run, other runs that found a solution can be tested. The GOOGP runs that found a solution for the list ADT and the seeds used for the runs are provided the **Appendix C**.

11.6 Performance Comparison with Other Studies

This section compares this study with other related work. The studies conducted by Bruce [8] and Langdon [6] are used for the comparison for the following reasons. First, both these studies implemented OOGP for producing code for ADTs. Second, this study uses the same representation used by both Bruce and Langdon. However, Langdon [6] aimed at improving GP scalability not automatic object-oriented programing while Bruce [8] aimed at automatic

object-oriented programming but did not examine producing code that uses the generated ADTs. The computer specifications are not the same in all the studies. Hence the runtimes cannot be compared. Only the success rates of the OOGP, GOOGP and OOGE approaches are compared with the success rate of the OOGP implemented by Langdon for the stack, queue and list ADTs. The success rates of OOGP, GOOGP and OOGE are also compared with the success rate of OOGP implemented by Bruce for the simultaneous induction of methods for the stack and queue ADTs.

Table 11.16 Comparison of OOGP, GOOGP and OOGE success rates with the success rates obtained from the related studies.

	Bruce [8]	**Langdon** [6]	**This study**		
			OOGP	**GOOGP**	**OOGE**
Stack ADT	5% (*)	6.67%	0%	60%	50%
Queue ADT	0% (*)	5.26% (*)	0%	6.67%	33.33%
List ADT	-	3.57% (*)	0%	13.33% (*)	10%

In Table 11.16, the cells marked with (*) means that an ADF was used. The success rates presented in the second column were obtained from an experiment termed Experiment 4 conducted by Bruce [8]. In the experiment, Bruce investigated simultaneous induction of methods and used typing to reduce the search space. Only one solution was found in 20 runs for the stack ADT. No solution was found for the queue ADT. Langdon [6] found 4 solutions in 60 runs for the stack ADT, 3 in 57 runs for the queue ADT and 2 in 56 runs for the list ADT. These give 6.67%, 5.26% and 3.57% success rates respectively as indicated in Table 11.16. OOGP implemented in this study has the lowest success rate for each of the three ADTs. This is expected as other implementations presented, with the exception of OOGE, used either an ADF or demes i.e., a population divided into subpopulations that exchange genetic material, or both to improve the success rate. For example, in Bruce [8], an ADF was used for each ADT. Langdon [6] used an ADF for the queue and list ADTs. The study also used demes i.e., the population was divided into subpopulations that exchange genetic materials. This has been reported to slow down convergence thereby helping the algorithm to escape local optima [6]. GOOGP uses a greedy method to direct the search during the initial population. Generally, OOGE performs better than other implementations. As specified in O'Neill [1], GE maintains diversity in the population. This contributes to the high success rate of OOGE for all the ADTs. It should, however, be noted that the number of methods

induced by Langdon for the list ADT is 10 while this study considers five as discussed in chapter 5, section 5.5.1. Producing code for 10 methods is more difficult compared to producing code for 5 methods. This may account for the low success rate in Langdon's implementation for the list ADT.

11.7 Chapter Summary

The chapter discussed the results obtained when applying OOGP, OOGP and OOGE to produce code for the stack, queue and list ADTs and problems, namely, Problem1, Problem2 and Problem3 that use the ADTs respectively. The chapter also discussed how the code produced by GOOGP is converted to Java. It then compared the performance of OOGP, GOOGP and OOGE with the performance of OOGP implemented in literature.

CHAPTER 12: CONCLUSION AND FUTURE WORK

12.1 Introduction

This chapter provides the summary of the findings of this dissertation and a conclusion to each of the objectives outlined in chapter 1. Section 12.2 presents the objectives outlined in chapter 1, the summary and conclusion while section 12.3 presents future work. The chapter summary is given in section 12.4.

12.2 Objectives and Conclusion

The objectives and the conclusion to each objective are provided below.

12.2.1 Objective 1: Evaluate Genetic Programming for Automatic Object-Oriented Programming.

The Object-Oriented Genetic Programming (OOGP) approach found in the literature was analysed. Based on the analysis, OOGP was developed with some changes made that distinguish some of its process from that found in the literature. For example, the genetic operators were changed to allow a proper mixture of genetic material between two parents. A variation of OOGP called Greedy OOGP (GOOGP) was also developed. GOOGP uses a greedy approach to generate the individuals in the initial population. Both the GP approaches, namely, OOGP and GOOGP were evaluated to produce code for classes. An internal representation language was defined to facilitate the translation of the generated code to a programming language. The function sets are subsets of the internal representation language. Both the approaches used a generational control model and represent each individual as a chromosome containing genes. Each gene is a parse tree representing a method of the class.

Three object-oriented programming problems were used to test each approach. Usually, an object-oriented program involves one or more classes. Each object-oriented programming problem used to test both the approaches involves two classes, one with the driver program and the Abstract Data Type (ADT) class. Thus, each approach was tested to produce code for the stack, queue and list ADTs, and the code that (1) uses the stack to determine whether or not a word, a phrase or a sentence is a palindrome, (2) uses the queue to perform a breadth-first traversal of any given parse tree, and (3) populates a list with integers and sorts the list. It was found that OOGP was not able to automatically produce complete code for the ADTs. For each of the stack, queue and list ADTs, it was also found that GOOGP was able to

produce code for both the classes and the driver programs when ADFs were implemented. Also, it was illustrated that the produced code can be converted to a programming language.

12.2.2 Conclusion to Objective 1

This study shows that GP can be used for automatic object-oriented programming. However, it requires (1) a mechanism, such as an ADF, to break down a complex problem into sub-problems and/or (2) a mechanism, such as the greediness in the initial population, to direct the search. GOOGP is successful at automatic programming because it uses ADFs and the initial population is informed.

12.2.3 Objective 2: Evaluate Grammatical Evolution for Automatic Object-Oriented Programming

The study investigated, for the first time, grammatical evolution for automatic object-oriented programming. Firstly, a thorough study of GE was conducted to determine the GE processes required to be adapted to the form needed to represent a class. Based on the study, Object-Oriented Grammatical Evolution (OOGE) was developed and evaluated to produce code for classes. OOGE uses a generational control model and represents each individual as a chromosome containing genes. Each gene is a binary string representing a method of the class. Again, genetic operators were changed to allow a proper mixture of genetic materials between two chromosomes. Like OOGP and GOOGP, each object-oriented programming problem used to test the approach involves two classes, one with the driver program and the Abstract Data Type (ADT) class. Thus, the approach was tested to produce code for the stack, queue and list ADTs, and code for programming problem that uses the ADTs. These programming problems are the same as those used to test OOGP and GOOGP. It was found that OOGE successfully produced code for the object-oriented programming problems tested. Code for the stack, queue and list ADTs was produced without requiring an ADF.

12.2.4 Conclusion to Objective 2

This study shows that GE can be used for automatic object-oriented programming. However, it requires both the chromosome representation and genetic operators to be adapted to a form suitable to represent a class. For complex problems, ADFs may be required for OOGE to produce code for the object-oriented programming problem.

12.2.5 Objective 3: Compare the Performance of Genetic Programming and Grammatical Evolution for Automatic Object-Oriented Programming

The OOGP, GOOGP and OOGE approaches were compared based on three criteria, namely, success rate, average fitness and average runtime. These criteria were used to compare the approaches at each level of difficulty as specified in chapter 5. The stack, queue and list represent a problem of easy, medium and hard difficulty levels respectively.

It was found that, for most of the problems, OOGE obtained a higher success rate compared to the OOGP and GOOGP success rates. This is because in OOGE, the search space and the program space are separated. Like in GE [1], this leads to the maintenance of diversity in the population and slows down convergence of the algorithm thereby allowing the algorithm to escape local optima. For the list ADT, the GOOGP success rate is slightly higher than the OOGE success rate when both the approaches use an ADF. OOGP obtained a 0% success rate at all the difficulty levels while GOOGP in most cases obtained a success rate competitive to that obtained by OOGE.

It was found that the average fitness of GOOGP is always better than the OOGE's average fitness. This is because the initial population of the GOOGP approach is informed. OOGP has the least average fitness at all the difficulty levels.

Several factors can increase or decrease the runtime of each approach. For example (1) how early in a run the approach finds a solution, (2) the process, such as conversion from binary to integer, involved in the OOGE approach. It was found that GOOGP finds solutions early in the runs. This reduces the average runtime of the approach. However, the initial generation of GOOGP is time consuming. OOGE converges slowly and in most cases finds solutions late in a run. This makes the runtime high. Also, as mentioned, the process of converting binary to integer increases OOGE runtime, thus making the runtime higher than GOOGP runtime. OOGP has the least runtime because it converges early to a local optimum.

12.2.6 Conclusion to Objective 3

Since the main aim is to produce code that correctly implements classes for the object-oriented programming problem at hand, the success rate is the most important criteria compared to the average runtime and average fitness. Hence GOOGE outperforms both GOOGP and OOGP while GOOGP outperforms OOGP.

12.3 Future Work

Based on the research presented in this dissertation, extension of the study will either be producing code that can be used by programmer or a black-box approach to automatic object-oriented programming. If the former is considered, a mechanism will be incorporated to remove the introns from the produced code before converting the code to a programming language. This will increase the efficiency of the program. In a black-box approach, the user will neither see how the code was evolved nor be able to see the evolved code for the methods. Only the output will be seen. Thus, introns need not be removed before converting the produced code to a programming language. Adding to the extension above, future work will also include an investigation into informed OOGE for automatic object-oriented programming. Thus, the initial population of OOGE will be informed to determine its effect on the OOGE success rate.

12.4 Chapter Summary

This chapter summarized the findings of this dissertation. It described how each objective was achieved and the conclusion to the objective. The objectives are to (1) evaluate GP for automatic object-oriented programming, (2) evaluate GE for automatic object-oriented programming and (3) compare the performance of GP and GE for automatic-object oriented programming. The initial aim was to investigate how good GP and GE are at automatic object-oriented programming. This study made the following contributions. Firstly, it provides a thorough survey of automatic Object-Oriented Genetic Programming (OOGP). It also extends and improves the previous work by introducing Greedy Object-Oriented Genetic Programming (GOOGP). It was found that GOOGP is able to increase the success rate of OOGP. Secondly, it is the first study investigating the use of GE for automatic object-oriented programming. Finally, the performance of genetic programming and grammatical evolution were compared for automatic object-oriented programming. It was found that grammatical evolution scales better than genetic programming when evolving code for classes. The field of automatic software development, specifically the field of automatic programming using evolutionary algorithms, is still in its infancy and has not seen many applications. This study forms part of an initial attempt in automating the process of writing code. This study has focused on automatically producing code for ADTs and programs that use the ADTs to solve problems. Hence, it is not yet widely applicable. However, this study has shown that GP and GE can be used for automatic object-oriented programming. This shows that both GP and GE have the potential to be used for a large scale production of code. This will be the focus in future work.

BIBLIOGRAPHY

1. 1. O'Neill, M., Ryan, C.: Grammatical evolution: evolutionary automatic programming in an arbitrary language. Springer (2003).

2. Rich, C., Waters, R.C.: Automatic programming: Myths and prospects. IEEE Computer. 21, 40–51 (1988).

3. Igwe, K., Pillay, N.: Automatic Programming Using Genetic Programming. In: Proceedings of the World Congress on Information and Communication Technologies. pp. 339–344. , Hanoi, Vietnam. (2013).

4. Koza, J.R.: Genetic Programming: On the programming of computers by means of natural selection. MIT press (1992).

5. Banzhaf, W., Nordin, P., Keller, R.E., Francone, F.D.: Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications (The Morgan Kaufmann Series in Artificial Intelligence). (1997).

6. Langdon, W.B.: Genetic programming and data structures: genetic programming+ data structures= automatic programming! Kluwe Academic Publisher (1998).

7. Koza, J.R.: Genetic programming II: automatic discovery of reusable programs. MIT press (1994).

8. Bruce, W.S.: The application of genetic programming to the automatic generation of object-oriented programs. Ph.D. thesis, Nova Southeastern University (1995).

9. Reynolds, C.W.: An evolved, vision-based behavioral model of coordinated group motion. From Animals to Animats. 2, 384–392 (1993).

10. Tackett, W.A., Carmi, A.: The donut problem: scalability, generalization and breeding policies in genetic programming. Adv. Genet. Program. 1, 143 (1994).

11. Pillay, N.: An Investigation into the Use of Genetic Programming for the Induction of Novice Procedural Programming Solution Algorithms in Intelligent Programming Tutors. Ph.D. thesis, University of Natal (2004).

12. Sywerda, G.: Uniform crossover in genetic algorithms. In: Proceedings of the third international conference on Genetic algorithms. pp. 2–9. Morgan Kaufmann Publishers Inc. (1989).

13. Icke, I., Bongard, J.C.: Improving genetic programming based symbolic regression using deterministic machine learning. In: Proceedings of 2013 IEEE Congress on Evolutionary Computation (CEC), pp. 1763–1770 (2013).

14. Xie, H., Zhang, M.: Impacts of sampling strategies in tournament selection for genetic programming. Soft Comput. 16, 615–633 (2012).

15. Goldberg, D.E.: Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989. NN Schraudolph J. 3, (1989).

16. Poli, R., Langdon, W.B.: An experimental analysis of schema creation, propagation and disruption in genetic programming. In: Proceedings of the Seventh International Conference on Genetic Algorithms (1997).

17. Poli, R., Langdon, W.B.: On the search properties of different crossover operators in genetic programming. Genet. Program. 293–301 (1998).

18. Beadle, L., Johnson, C.G.: Semantically driven crossover in genetic programming. In: Proceedings of the IEEE World Congress on Computational Intelligence. pp. 111–116 (2008).

19. Poli, R., Langdon, W.B., McPhee, N.F., Koza, J.R.: A field guide to genetic programming. Lulu Enterprises, UK Ltd (2008).

20. Montana, D.J.: Strongly typed genetic programming. Evol. Comput. 3, 199–230 (1995).

21. Koza, J.R., Bennett III, F.H., Stiffelman, O.: Genetic programming as a Darwinian invention machine. Springer (1999).

22. Pillay, N.: Evolving solutions to ASCII graphics programming problems in intelligent programming tutors. In: Proceedings of International Conference on Applied Artificial Intelligence (ICAAI'2003). pp. 236–243 (2003).

23. Teller, A.: Turing completeness in the language of genetic programming with indexed memory. In: Proceedings of the First IEEE Conference on Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence. pp. 136–141 (1994).

24. Finkel, J.R.: Using genetic programming to evolve an algorithm for factoring numbers. Genet. Algorithms Genet. Program. Stanf. 52–60 (2003).

25. Pillay, N.: A genetic programming system for the induction of iterative solution algorithms to novice procedural programming problems. In: Proceedings of the 2005 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries. pp. 66–77 (2005).

26. Li, X.: Utilising restricted for-loops in genetic programming, http://www.cs.rmit.edu.au/~vc/papers/li-phd.pdf, (2007).

27. Angeline, P.J.: A historical perspective on the evolution of executable structures. Fundam. Informaticae. 35, 179–195 (1998).

28. Brameier, M., Banzhaf, W.: Neutral variations cause bloat in linear GP. In: Proceedings of the European Conference on Genetic Programming. pp. 286–296. Springer (2003).

29. Silva, S., Costa, E.: Dynamic limits for bloat control. In: Proceedings of Genetic and Evolutionary Computation Conference –GECCO 2004. pp. 666–677. Springer (2004).

30. Nordin, P., Banzhaf, W., others: Complexity Compression and Evolution. In: Proceedings of the Sixth International Conference on Genetic Algorithms: (ICGA95 (1995).

31. Silva, S., Costa, E.: Dynamic limits for bloat control in genetic programming and a review of past and current bloat theories. Genet. Program. Evolvable Mach. 10, 141–179 (2009).

32. Ferreira, C.: Gene Expression Programming: A New Adaptive Algorithm for Solving Problems. Complex Syst. 13, 87–129.

33. Ahmadizar, F., Soltanian, K., AkhlaghianTab, F., Tsoulos, I.: Artificial neural network development by means of a novel combination of grammatical evolution and genetic algorithm. Eng. Appl. Artif. Intell. 39, 1–13 (2015).

34. Fenton, M., McNally, C., Byrne, J., Hemberg, E., McDermott, J., O'Neill, M.: Automatic innovative truss design using grammatical evolution. Autom. Constr. 39, 59–69 (2014).

35. Hidalgo, J.I., Colmenar, J.M., Risco-Martin, J.L., Cuesta-Infante, A., Maqueda, E., Botella, M., Rubio, J.A.: Modeling glycemia in humans by means of grammatical evolution. Appl. Soft Comput. 20, 40–53 (2014).

36. Chen, L.: Macro-grammatical evolution for nonlinear time series modeling—a case study of reservoir inflow forecasting. Eng. Comput. 27, 393–404 (2011).

37. Rothlauf, F., Oetzel, M.: On the locality of grammatical evolution. In: Proceedings of the European Conference on Genetic Programming, pp. 320-330. Springer (2006).

38. Thorhauer, A., Rothlauf, F.: On the Locality of Standard Search Operators in Grammatical Evolution. In: Proceedings of International Conference on Parallel Problem Solving from Nature–PPSN XIII. pp. 465–475. Springer (2014).

39. Hugosson, J., Hemberg, E., Brabazon, A., O'Neill, M.: Genotype representations in grammatical evolution. Appl. Soft Comput. 10, 36–43 (2010).

40. Ryan, C., O'Neill, M.: Grammatical evolution: A steady state approach. Late Break. Pap. Genet. Program. 1998, 180–185 (1998).

41. O'Neill, M., Ryan, C., Keijzer, M., Cattolico, M.: Crossover in grammatical evolution. Genet. Program. Evolvable Mach. 4, 67–93 (2003).

42. O'Neill, M., Ryan, C.: Crossover in grammatical evolution: A smooth operator? In: Proceedings of the European Conference on Genetic Programming. pp. 149–162. Springer (2000).

43. Byrne, J., O'Neill, M., Brabazon, A.: Structural and nodal mutation in grammatical evolution. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation. pp. 1881–1882. ACM (2009).

44. Byrne, J., O'Neill, M., McDermott, J., Brabazon, A.: An analysis of the behaviour of mutation in grammatical evolution. In: Proceedings of the European Conference on Genetic Programming. pp. 14–25. Springer (2010).

45. Castle, T., Johnson, C.G.: Positional effect of crossover and mutation in grammatical evolution. In: Proceedings of the European Conference on Genetic Programming. pp. 26–37. Springer (2010).

46. Harper, R., Blair, A.: Dynamically defined functions in grammatical evolution. In: Proceedings of 2006 IEEE Conference on Evolutionary Computation, CEC 2006. pp. 2638–2645. IEEE (2006).

47. Ryan, C.: Grammar based function definition in Grammatical Evolution. In: Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000). pp. 485–490 (2000).

48. Hemberg, E., O'Neill, M., Brabazon, A.: An investigation into automatically defined function representations in grammatical evolution. In: Proceedings of the 15th International Conference on Soft Computing, Mendel (2009).

49. Reinfelds, J.: A three paradigm first course for CS majors. In: ACM SIGCSE Bulletin. pp. 223–227. ACM (1995).

50. Ambler, A.L., Burnett, M.M., Zimmerman, B., others: Operational versus definitional: A perspective on programming paradigms. Computer. 25, 28–43 (1992).

51. Fahland, D., Lübke, D., Mendling, J., Reijers, H., Weber, B., Weidlich, M., Zugal, S.: Declarative versus imperative process modeling languages: The issue of understandability. In: Enterprise, Business-Process and Information Systems Modeling. pp. 353–366. Springer (2009).

52. Wang, Q., Wang, A.: Evolving Computing and Automatic Programming. In: Proceedings of the 2011 International Conference on Internet Computing & Information Services (ICICIS), pp. 213–214 (2011).

53. Pillay, N., Chalmers, C.K.: A hybrid approach to automatic programming for the object-oriented programming paradigm. In: Proceedings of the 2007 annual research conference

of the South African institute of computer scientists and information technologists on IT research in developing countries. pp. 116–124. ACM (2007).

54. Bruce, W.S.: Automatic generation of object-oriented programs using genetic programming. In: Proceedings of the First Annual Conference on Genetic Programming. pp. 267–272. MIT Press (1996).

55. Goodrich, M.T., Tamassia, R.: Data structures and algorithms in Java. Wiley (2001).

56. Johnson, C.: Basic Research Skills in Computing Science. Dep. Comput. Sci. Glasgow University, UK. (2001).

57. Inenaga, S., Takeda, M., others: Palindrome pattern matching. Theor. Comput. Sci. 483, 162–170 (2013).

58. Agapitos, A., Lucas, S.M.: Evolving modular recursive sorting algorithms. In: Proceedings of the European Conference on Genetic Programming. pp. 301–310. Springer (2007).

59. Kinnear Jr, K.E.: Evolving a sort: Lessons in genetic programming. In: Proceedings of the IEEE International Conference on Neural Networks, 1993. pp. 881–888. IEEE (1993).

60. O'Neill, M., Nicolau, M., Agapitos, A.: Experiments in program synthesis with grammatical evolution: A focus on Integer Sorting. In: Evolutionary Computation (CEC), 2014 IEEE Congress on. pp. 1504–1511. IEEE (2014).

61. Shirakawa, S., Nagao, T.: Evolution of sorting algorithm using graph structured program evolution. In: Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics, ISIC. pp. 1256–1261. IEEE (2007).

62. Griswold, R.: Some palindrome sentences, http://www.cs.arizona.edu/icon/oddsends/palinsen.htm.

63. Kinnear Jr, K.E.: Generality and Difficulty in Genetic Programming: Evolving a Sort. In: Proceedings of the European Conference on Genetic Programming ICGA. pp. 287–294. (1993).

APPENDIX A: THE OOGE GRAMMARS

A.1 Grammar for the Stack ADT

$< stmts > :: = < stmt >; \ | < stmt >; < stmts >$

$< stmt > ::= < set\_aux \ (< expr >) \ | \ write \ (< expr > < expr >) | \ read \ (< \ expr >)$

$| \ write \ (< element > < expr >) \ | < expr >$

$< expr > :: = \ < var > \ | + < var > < var > \ | - < var > < var >$

$< var > \ := \ one \ | zero | \ aux$

$< element > \ := \ i$

A.2. Grammar for Problem1

$< stmts > ::= < stmt >; \ | \ < stmt >; < stmts >$

$< stmt > ::= strequal(< strExpr > < strExpr >) \ | \ < strExpr >$

$| \ for\_loop_{j}(< expr > < loopstmt >)$

$< expr > ::= < var > \ | + < expr > < expr > \ | - < expr > < expr >$

$< loopStmt > :: = append \ (< loopExpr > < loopExpr > \ | \ < loopExpr >$

$| < stackMethod >$

$< loopExpr > ::= \ < loopVar > | \ char\_At \ (< strVar > < loopVar >)$

$< loopVar > ::= Cvar_{j} | arg | \ Ivar_{j}$

$< strVar > ::= \ arg$

$< strExpr > ::= \ strequal(< strExpr > < strExpr >) | < strVar >$

$< stackMethod > ::= \ push(< loopExpr >) | \ pop \ | peek \ | makeNull \ | empty$

$< var > ::= \ one \ | \ len \ | zero$

A.3 Grammar for the Queue ADT

$< stmts > :: = < stmt >; \mid < stmt >; < stmts >$

$< stmt > ::= < set\_aux\ (< expr >) \mid write\ (< expr > < expr >) \mid read\ (< expr\ ) \mid write\ (< element > < expr >) \mid < expr > \mid for_j(< expr > < expr > < loopstmt >)$

$< expr > :: = < var > \mid + < expr > < expr > \mid - < expr > < expr >$

$< element > ::= i$

$< loopStmt > ::= write\ (< loopElement > < loopExpr >) \mid read\ (< loopExpr >) \mid for_i(< expr > < expr > < loopstmt >) \mid loopExpr$

$< var > := one \mid zero \mid aux$

$< loopExpr > ::= < loopVar > \mid + < loopExpr > < loopExpr > \mid - < loopExpr > < loopExpr >$

$< loopVar > := one \mid zero \mid aux \mid Cvar_j$

$< loopElement > := i \mid Ivar_j$

A.4. Grammar for Problem 2

$< stmts > ::= < stmt >; \mid < stmt >; < stmts >$

$< stmt > ::= for\_loop_j(< expr > < loopstmt >) \mid while(< cond > < stmt >) \mid < queueMethod >$

$< expr > ::= < var > \mid getArity\ < var >$

$< loopStmt > :: = < loopQueueMethod > \mid < loopExpr >$

$< loopExpr > ::= getElement < loopVar > \mid < loopVar >$

$< loopVar > ::= Cvar_j \mid Ivar_j \mid$

$< var > ::= start \mid one \mid dequeue$

$< cond > ::= empty \mid not\ (cond)$

$< queueMethod > ::= enqueue(< expr >) \mid front \mid dequeue \mid makeNull \mid empty$

$< loopQueueMethod > ::= enqueue(< loopExpr >) \mid front \mid dequeue \mid makeNull \mid empty$

A.5.i Grammar for the List ADT (without an ADF)

$< stmts >$ ::= $< stmt >;$ | $< stmt >; < stmts >$

$< stmt >$ ::= $< set\_aux(< expr >)$ | $write(< expr >< expr >)$

| $read\ (< expr >)$ | $write(< element >< expr >)$ | $< expr >$

| $for_j(< expr >< expr >< loopstmt >)$

$< expr >$ ::= $< var >$ | $+ < expr >< expr >$ | $- < expr >< expr >$

$< element >$ :: = i

$< loopStmt >$ :: = $write\ (< loopElement >< loopExpr >$ | $< loopExpr >$

| $read(< loopExpr >)|for_j(< expr >< expr >< loopstmt >)$

$< var >$:: = $one\ |zero|aux|dec\_aux\ |inc\_aux\ |p$

$< loopExpr >$ ::= $< loopVar >$ | $+ < loopExpr >< loopExpr >$

| $- < loopExpr >< loopExpr >$

$< loopVar >$ ::= $one\ |zero|\ aux\ |dec\_aux\ |inc\_aux\ |p|\ Cvar_j$

A.5.ii Grammar for the List ADT (with an ADF)

$< stmts >$ ::= $< stmt >;$ | $< stmt >; < stmts >$

$< stmt >$ ::= $< set\_aux(< expr >)$ | $write(< expr >< expr >)$

| $read\ (< expr >)$ | $write(< element >< expr >)$ | $< expr >$

| $for_j(< expr >< expr >< loopstmt >)$ | $ADF0\ (< expr >< expr >)$

$< expr >$ ::= $< var >$ | $+ < expr >< expr >$ | $- < expr >< expr >$

$< element >$ :: = i

$< loopStmt >$ :: = $write\ (< loopElement >< loopExpr >$ | $< loopExpr >$

| $read(< loopExpr >)|for_j(< expr >< expr >< loopstmt >)$

$< var >$:: = $one\ |zero|aux|dec\_aux\ |inc\_aux\ |p$

$< loopExpr >$ ::= $< loopVar >$ | $+ < loopExpr >< loopExpr >$

| $- < loopExpr >< loopExpr >$

$< loopVar >$ ::= one |$zero$| aux |dec_aux |inc_aux |p| $Cvar_j$

ADF()

$< stmts >:: = < stmt >; | < stmt >; < stmts >$

$< stmt >: : = for_j(< expr >< expr >< loopstmt >) | < expr > | write(< expr > < expr >)$

$< expr >:: = < var > | + < expr >< expr > | - < expr >< expr >$

$< var >:: = arg1 | arg2 | zero | one$

$< loopStmt >:: = write(< loopElement >< loopExpr >)$

$| for_j(< expr >< expr >< loopstmt >) | < loopExpr >$

$< loopElement >:: = Ivar_j$

$< loopExpr >:: = Cvar_j | zero | one$

A.6. Grammar for Problem 3

$< stmts >$ ::= $< stmt >; | < stmt >; < stmts >$

$< stmt >$ ::= $for\_loop_j(< expr > < loopstmt >)$

$< expr >$ ::= $< var > | + < expr >< expr > | - < expr >< expr >$

$< loopStmt >$ :: = $for\_loop_j(< expr > < loopstmt >) | < listMethods >$

$| < loopExpr > | order(< loopExpr > < loopExpr >)$

$< loopExpr >$ ::= $getElement < loopVar > | < loopVar > | read < loopVar >$

$< loopVar >$ ::= $Cvar_j | Ivar_j$

$< var >$ ::= $zero | one | len$

$< listMethod >$ ::= $insertAt(< loopExpr > < loopExpr >) | makeNull | empty$

$| getElement < loopExpr > | removeElement(loopExpr)$

APPENDIX B: THE MODIFIED OOGE GRAMMARS FOR THE FINAL RUNS

B.1. Grammar for Problem1

$< stmts >$ ::= $< stmt >;$ | $< stmt >; < stmts >$

$< stmt >$ ::= $strequal(< strExpr >< strExpr >)$ | $< strExpr >$

| $for\_loop_j(< expr >< loopstmt >)$

$< expr >$ ::= $< var >$ | $+ < expr >< expr >$ | $- < expr >< expr >$

$< loopStmt >$ :: = $append$ $(< loopExpr >< loopExpr >$ | $< loopExpr >$

| $< stackMethod >$

$< loopExpr >$ ::= $< loopVar >$ | $char\_At$ $(< strVar >< loopVar >)$

$< loopVar >$ ::= $Cvar_j | arg | Ivar_j$

$< strVar >$ ::= arg

$< strExpr >$ ::= $ADF1 | ADF2 | strequal(< strExpr >< strExpr >) | < strVar >$

$< stackMethod >$ ::= $push(< loopExpr >) | pop | peek | makeNull | empty$

$< var >$ ::= $one | len | zero$

ADF1()

$< stmts >$ ::= $< stmt >;$ | $< stmt >; < stmts >$

$< stmt >$ ::= $for\_loop_j(< expr >< loopstmt >)$

$< expr >$ ::= $< var >$ | $+ < expr >< expr >$ | $- < expr >< expr >$

$< loopStmt >$ :: = $append$ $(< loopExpr >< loopExpr >$ | $< loopExpr >$

| $< stackMethod >$

$< loopExpr >$ ::= $< loopVar >$ | $char\_At$ $(< strVar >< loopVar >)$

$< loopVar >$ ::= $Cvar_j | arg | Ivar_j$

$< strVar >$ ::= $arg | Ivar_j$

$< stackMethod >$ ::= $push(< loopExpr >) | pop | peek | makeNull | empty$

$< var >$ ::= $one \mid len \mid zero$

ADF2()

$< stmt >$ ::= $< stmt >; \mid < stmt >; < stmts >$

$< stmt >$ ::= $for\_loop_j(< expr >< loopstmt >)$

$< expr >$ ::= $< var > \mid + < expr >< expr > \mid - < expr >< expr >$

$< loopStmt >$ :: = $append\ (< loopExpr >< loopExpr > \mid < loopExpr >$

$\mid < stackMethod >$

$< loopExpr >$ ::= $< loopVar > \mid char\_At\ (< strVar >< loopVar >)$

$< loopVar >$ ::= $< stackMethod > \mid Ivar_j$

$< strVar >$ ::= $Ivar_j$

$< stackMethod >$ ::= $push(< loopExpr >) \mid pop \mid peek \mid makeNull \mid empty$

$< var >$ ::= $one \mid len \mid zero$

(where j is a positive integer).

B.2. Grammar for Problem 2

$< stmts >$ ::= $< stmt >; \mid < stmt >; < stmts >$

$< stmt >$ ::= $for\_loop_j(< expr >< loopstmt >) \mid while(< cond >< stmt >)$

$\mid < queueMethod > \mid$ADF

$< expr >$ ::= $< var > \mid getArity\ < var >$

$< loopStmt >$ :: =$< loopQueueMethod > \mid < loopExpr >$

$< loopExpr >$ ::= $getElement < loopVar > \mid < loopVar >$

$< loopVar >$ ::= $Cvar_j \mid Ivar_j \mid ADF$

$< var >$ ::= $start \mid one \mid dequeue$

$< cond >$ ::= $empty \mid not\ (cond)$

$< queueMethod >$ ::= $enqueue(< expr >) \mid front \mid dequeue \mid makeNull \mid empty$

$< loopQueueMethod >$ ::= $enqueue(< loopExpr >) \mid front \mid dequeue \mid makeNull$

$| empty$

ADF()

$< stmts >$ ::= $< stmt >;$ | $< stmt >; < stmts >$

$< stmt >$ ::= $for\_loop_{\not j}(< expr > < loopstmt >)$

$< expr >$ ::= $< var >$ | $getArity$ $< var >$

$< loopStmt >$ :: =$< loopQueueMethod >$ | $< loopExpr >$

$< loopExpr >$ ::= $getElement < loopVar >$ | $< loopVar >$

$< loopVar >$ ::= $Cvar_{\not j}$ | $Ivar_{\not j}$

$< var >$ ::= $zero$ | one |$dequeue$

$< loopQueueMethod >$ ::= $enqueue(< loopExpr >)$| $front$| $dequeue$ | $makeNull$

$| empty$

B.3. Grammar for Problem 3

$< stmts > ::= < stmt >; \mid < stmt >; < stmts >$

$< stmt > ::= for\_loop_{\not j}(< expr > < loopstmt >)$

$< expr > ::= < var > \mid + < expr > < expr > \mid - < expr > < expr >$

$< loopStmt > :: = for\_loop_{\not j}(< expr > < loopstmt >) \mid < listMethods >$

$\mid < loopExpr > \mid order(< loopExpr > < loopExpr >)$

$< loopExpr > ::= getElement < loopVar > \mid < loopVar > \mid read < loopVar >$

$< loopVar > ::= Cvar_{\not j} \mid Ivar_{\not j}$

$< var > ::= zero \mid one \mid len \mid ADF0$

$< listMethod > ::= insertAt(< loopExpr > < loopExpr >) \mid makeNull \mid empty$

$\mid getElement < loopExpr > \mid removeElement(loopExpr)$

ADF0()

$< stmts > ::= < stmt >; \mid < stmt >; < stmts >$

$< stmt > ::= for\_loop_{\not j}(< expr > < loopstmt >)$

$< expr > ::= < var > \mid + < expr > < expr > \mid - < expr > < expr >$

$< loopStmt > :: = for\_loop_{\not j}(< expr > < loopstmt >) \mid$

$\mid < loopExpr > \mid order(< loopExpr > < loopExpr >)$

$< loopExpr > ::= getElement < loopVar > \mid < loopVar > \mid read < loopVar >$

$< loopVar > ::= Cvar_{\not j} \mid Ivar_{\not j}$

$< var > ::= zero \mid one \mid len$

$< listMethod > ::= insertAt(< loopExpr > < loopExpr >) \mid makeNull \mid empty$

$\mid getElement < loopExpr > \mid removeElement(loopExpr)$

APPENDIX C: THE GOOGP RUNS THAT PRODUCED CODE FOR THE LIST ADT AND A SOLUTION FOR THE LIST ADT CONVERTED TO JAVA.

Table C.1 Run numbers and seeds for the GOOGP runs that found a solution for the list ADT

S/N	Run Number	Seed
1	3	48885500493
2	11	-2600735935262183310
3	13	6009182497704641848
4	27	1788569723501689014l

The Java code provided below was converted from a solution generated in run 3 of GOOGP using the seed provided in the first row of Table C.1. The code was firstly generated in an internal representation language. Most of the functions in the internal representation language have an equivalent instruction in a particular programming language and can be converted. Some of the functions, for example, the *read*, *write* and *order*, have no equivalent. The code for such functions would be generated and are represented as private methods.

```
public class List{
public final int CAPACITY = 100;
private int capacity;
private int aux = -1;
private int size = aux + 1;
int[] L;

public List(){
capacity = CAPACITY;
L = new int[capacity];
}

public List(int cap){
capacity = cap;
L = new int[capacity];
```

```
}

private int ADF(int arg1, int arg2){
  int Ivar484 = 0;
 if(arg2 <= arg2 ){
  for(int Cvar484 = arg2; Cvar484 <= arg2; Cvar484++){
  Ivar484 = arg1;
 }
 }else{
  for(int Cvar484 = arg2; Cvar484 >= arg2; Cvar484--){
  Ivar484 = arg1;
 }
 }
  int Ivar485 = 0;
 if(arg1 <= arg2 ){
  for(int Cvar485 = arg1; Cvar485 <= arg2; Cvar485++){
  Ivar485 = Cvar485;
 }
 }else{
  for(int Cvar485 = arg1; Cvar485 >= arg2; Cvar485--){
  Ivar485 = Cvar485;
 }
 }
  int Ivar483 = 0;
 if(Ivar484 <= Ivar485 ){
  for(int Cvar483 = Ivar484; Cvar483 <= Ivar485; Cvar483++){
  int V2 = write(Ivar483 , Cvar483);
  Ivar483 = V2;
 }
 }else{
  for(int Cvar483 = Ivar484; Cvar483 >= Ivar485; Cvar483--){
  int V2 = write(Ivar483 , Cvar483);
  Ivar483 = V2;
 }
```

```
 }
  return Ivar483;
}

public void makeNull(){
  int V0 = set_aux(inc_aux());
  int V1 = aux - dec_aux() ;
  int V2 = write(V0 , V1);
}

public void insertAt(int pos, int N){
inc_aux();
  int V0 = aux;
  int V1 = ADF (pos , V0) ;
  int V2 = write(N , pos);
  int V3 = V2;
}

public int getElement(int pos){
  int V0 = read(pos);
  return V0;
}

public int removeElement(int pos){
  int V0 = aux - dec_aux() ;
  int Ivar159 = 0;
 if(0 <= 1 ){
  for(int Cvar159 = 0; Cvar159 <= 1; Cvar159++){
  Ivar159 = pos;
 }
 }else{
  for(int Cvar159 = 0; Cvar159 >= 1; Cvar159--){
  Ivar159 = pos;
 }
```

```
 }
  int V2 = ADF (V0 , Ivar159) ;
  return V2;
}

public boolean empty(){
  int Ivar10 = 0;
 if(aux <= aux ){
  for(int Cvar10 = aux; Cvar10 <= aux; Cvar10++){
  Ivar10 = aux;
 }
 }else{
  for(int Cvar10 = aux; Cvar10 >= aux; Cvar10--){
  Ivar10 = aux;
 }
 }
  return (Ivar10 < 0);
}

private int write(int val, int index){
    if (index < 0 || index >= capacity) {
        return 0;
    }else {
        int returnvalue = (index > aux)? 0: L[index];
        L[index] = val;
        return returnvalue;
    }
}

private int set_aux(int index){
    aux = index;
    return 0;
}
```

```
private int inc_aux(){
   aux = aux + 1;
   return 0;
}

private int dec_aux(){
   aux = aux - 1;
   return 0;
}

private int read(int index){
   if(index >= 0 && index < capacity){
      return (index > aux)? 0: L[index];
   } else {
      return 0;
   }
}
}
```

APPENDIX D: A SOLUTION FOR PROBLEM3 CONVERTED TO JAVA

```
public class IntegerSort{
private int len;
private int[] intArray;
private List list;

public IntegerSort(int[] arr){
len = arr.length;
list = new List(len);
intArray = new int[len];
System.arraycopy(arr, 0, intArray, 0, len);
}

private int ADF1(){
  int V0 = len;
  int Ivar295768 = 0;
  for(int Cvar295768 = 0; Cvar295768 <= V0; Cvar295768++){
  int V1 = Cvar295768;
  int V2 = read(Cvar295768);
  list.insertAt(V1 , V2) ;
int V3  = 0;
  Ivar295768 = V3;
  }
  return Ivar295768;
}

public int[] sort(){
ADF1();
  int V1 = 1;
  int V2 = len;
```

```
  int V3 = V2;
  int Ivar251109 = 0;
  for(int Cvar251109 = 0; Cvar251109 <= V3; Cvar251109++){
  int V4 = len;
  int V5 = V4 - 1 ;
  int Ivar251110 = 0;
  for(int Cvar251110 = 0; Cvar251110 <= V5; Cvar251110++){
  int V6 = order (Cvar251109 , Cvar251110) ;
  Ivar251110 = V6;
  }
  Ivar251109 = Ivar251110;
  }
  int V9 = Ivar251109;
  return list.L;
}

private int read(int index){
    if(index >= 0 && index < len){
        return intArray[index];
    } else {
        return 0;
    }
}
private int order(int index1, int index2){
  if(index1 >= 0 && index1 < len && index2 >= 0 && index2 < len){
   if(list.getElement(index1) > list.getElement(index2)) {
    int temp = list.getElement(index1);
    list.L[index1]= list.getElement(index2);
    list.L[index2] = temp;
   }
  }
  return 0;
}
}
```

APPENDIX E: THE USER MANUAL

E.1. Program Requirements

In order to run the automatic object-oriented program, Java must be installed. If Java is already installed, please ensure that it is updated. An update version can be obtained from http://java.com/en/download/

E.2. How to Run the Program

GUI is created to make the program easy to run. GP parameters can be entered and changed easily. Problems definitions can be seen before starting any run. The text files containing the fitness cases used by the program and the executable Java file (i.e., the .exe file) must be in the same folder. The following steps should be followed to run the algorithm.

E.2.1 Step 1 – Executable jar (.jar) file

The user starts by double clicking the Java icon  . This causes Figure E.1 to display. Once GP is selected by clicking on "GP", Figure E.2 is displayed. Similar interface is displayed if GE is selected. However, the GP interface is used for illustrations.

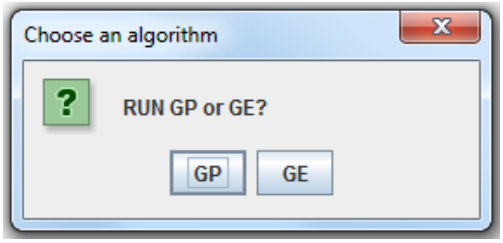


Figure E.1 GUI interface for selecting GP or GE

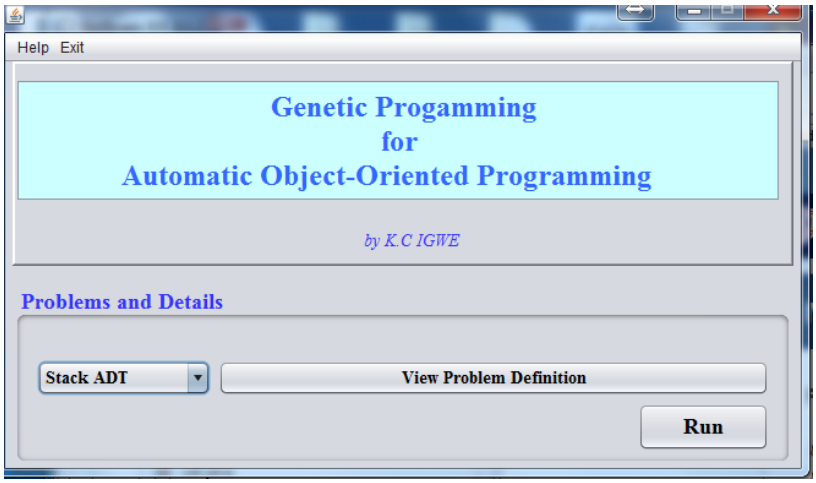


Figure E.2. The GUI interface for GP

E.2.2. Step 2 – Selecting a problem

OOGP, GOOGP and OOGE are applied to produce code for 3 object-oriented programming problems each involving 2 classes. Hence there are 6 different classes to be evolved. Figure E.3 shows how to select a problem while Figure E.4 shows how to view the problem definition by clicking on "view Problem Definition" button.

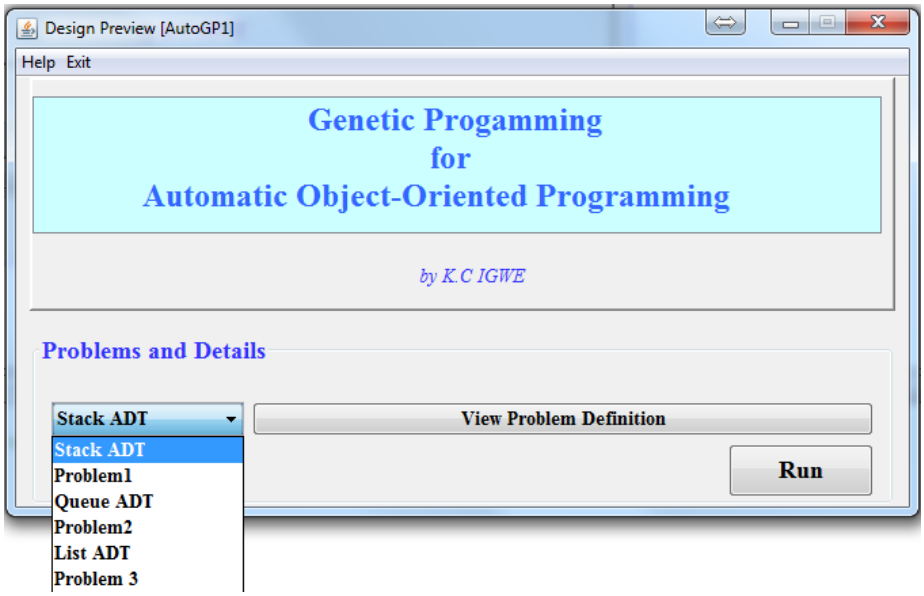


Figure E.3 The GUI interface for Problem selection

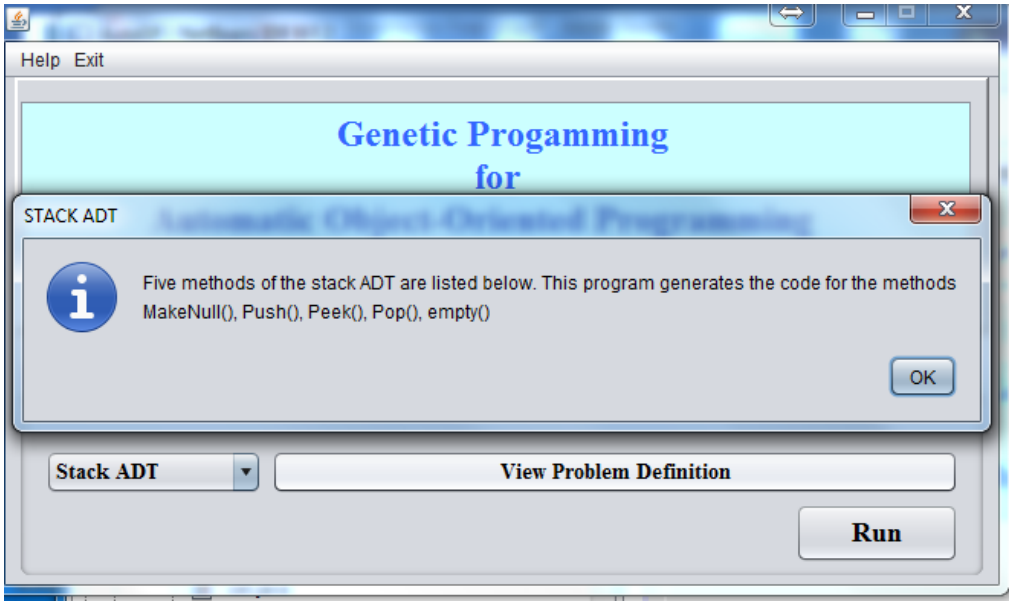


Figure E.4. GUI interface for viewing the problem definition

E.2.3. Step 3 –Selecting and applying an approach to produce code for a selected problem

Figure E.5 shows an example of an interface that allows the user to edit the GP parameters. The interface is displayed when the "run" button in Figure E.4 is clicked. GOOGP is selected for run if the check box labelled "Use Greedy OOGP" is checked. OOGP is selected by default. The run button is clicked to execute the selected approach.

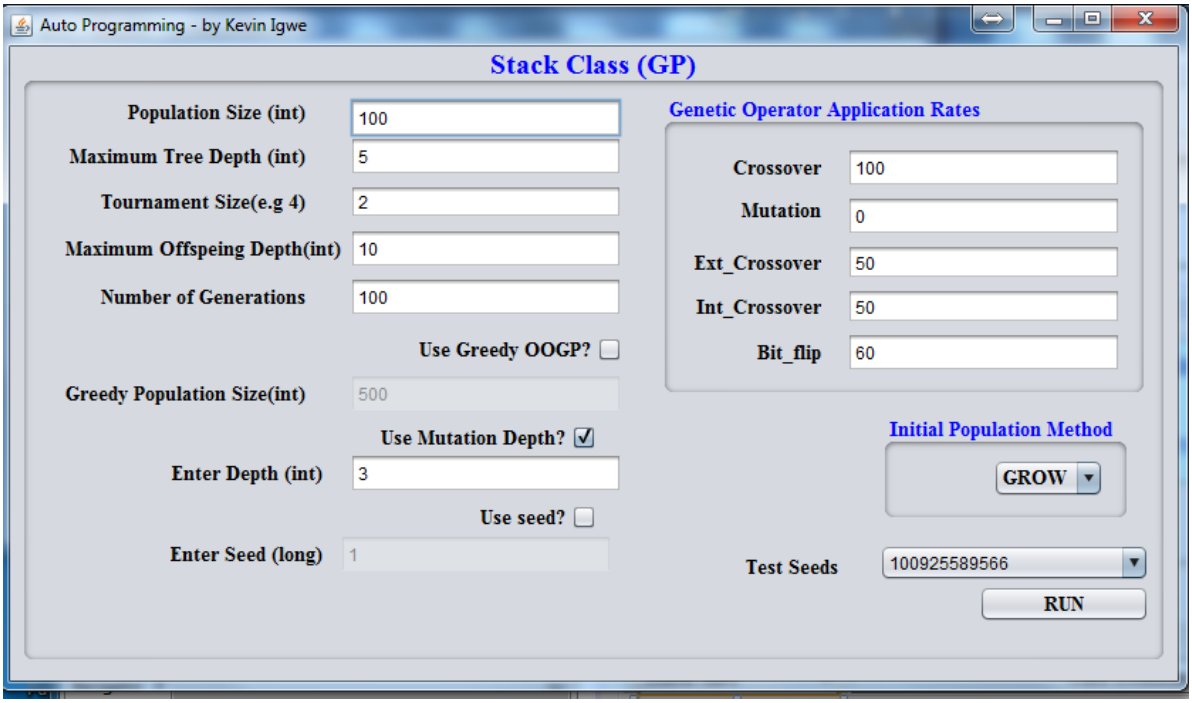


Figure E.5. The GUI Interface that allows editing of the GP parameters

E.3 Input Parameters and Editing

The interface shown in Figure E.5 contains fields and buttons. Each text field has a default value which can be edited. A field is disabled if is not available for editing or use for a particular approach. Below is the explanation of each filed and button.

E.3.1 Text fields

Population Size: It contains an integer value that specifies the number of individuals that must be created during the initial population generation.

Maximum Tree Depth: This contains an integer value that specifies the maximum level of nodes (depth) that can be reached when creating the individuals in the initial population generation.

Tournament Size: It specifies the number of individuals that must compete for fitness. It must be an integer.

Maximum Offspring depth: This specifies the maximum level of nodes the offspring created by genetic operators are allowed to have.

Crossover: it specifies the percentage of the offspring that must be created by applying crossover operator. For instance, if the population size is 500, an input value 80 will allow (80*500)/100 = 400 offspring to be created using crossover operator. It must be an integer value.

Mutation: This specifies the percentage of the offspring that must be created by applying mutation operator. It must be an integer value.

Ext_Crossover: This specifies the probability that external crossover will be applied to an individual in the population. It must be an integer value. A value of 80 implies 0.8 probabilities.

Int_Crossover: This specifies the probability that internal crossover will be applied to an individual in the population. It must be an integer value. A value of 50 implies 0.5 probabilities.

Bit_flip: This field is used when GE in Figure E.1 is selected. It specifies the probability that a bit in a GE chromosome will be flipped. It must be an integer value.

Enter Depth: It is recommended that this field is always used. Otherwise the value will be randomly generated. Mutation creates new individual using the grow method of the initial population generation. The individual replaces the subtree that must be deleted from the parent. Instead of using the value specified in the "Maximum Tree Depth" field. The value specified in the "Enter Depth" is used.

Enter Seed: This filed allows the user to specify a seed to be used. Once this field is selected, the default seed will not be used. This field must contain an Integer or Long (data type) numbers. It should be noted that the same solution may not be generated if the seed or any other parameters changes.

E.3.2. Check Buttons

This is used to allow the user make a decision whether or not to use a particular operation. The "Use Mutation Depth**"** enables or disables the "Enter depth" field while the "Use seed" enables or disables the "Enter Seed" field.

E.3.3. Combo Box

The combo box labelled "Test seeds" is useful for selecting one of the default seeds used for testing the approaches. The system can only make use of the selected seed if the "Enter Seed**"** field is disabled. The grow method of initial population generation was used for the final runs of each approach. Hence, both the ramped half-and-half and full methods were disabled.

E.4 Indicators

The system displays a message to indicate whether or not a solution has been found. Figure E.6 shows this message. If a solution is found, it is writing to a text file named using the format *problemApproch*Solution.txt. For example, a solution for the list ADT produced by GOOGP will be written to a file named ListGOOGPsolution.txt. The file will be written to the same folder containing the .jar file. As shown in Figure E.7, the system provides the option to convert the generated solution to Java, if the selected approach is GOOGP and the selected problem is the list ADT or Problem3. The option displays when a solution has been found and the user clicked "OK". Choosing "Yes" in Figure E.7 will write Java code to a file named using the format *problem*Class.java, where *problem* is either List or Problem3. For example, a file named ListClass.java is produced when *problem* is List.

Figure E.6. Solution status indicators

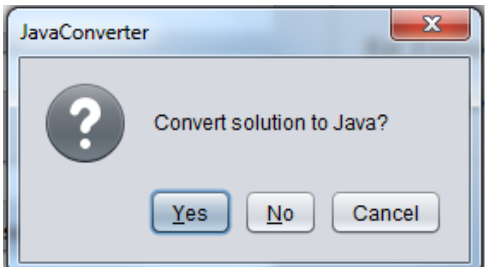


Figure E.7 The option to convert a solution to Java

APPENDIX F: FITNESS CASES FOR THE FINAL RUNS

Table F. 1. The fitness cases used for the final runs for Problem1

Fitness case	**Expected outcome**
Noel sees Leon	TRUE
Pen tip	FALSE
Sore was I ere I saw Eros	TRUE
deepened	FALSE
Euston saw I was not Sue	TRUE
going	FALSE
No evil Shahs live on	TRUE
On the go	FALSE
murdrum	TRUE
Pump	FALSE

Table F. 2. The fitness cases used for the final runs for Problem2

Input case	**Target arrangement**
0/3 1 1/2 4 6 7 2/8 9 3/ 4/5 5/ 6/ 7/ 8/ 9/	0 3 1 2 4 6 7 8 9 5
13/4 6 12 4/1 2 3 6/5 12/8 9 10 11 1/ 2/ 3/ 5/ 8/7 9/ 10/ 11/ 7/	13 4 6 12 1 2 3 5 8 9 10 11 7
8/2 7 1/ 2/1 3/ 4/3 5/ 6/ 7/4 5 6	8 2 7 1 4 5 6 3
1/2 3 2/4 5 3/6 7 8 4/ 5/9 10 6/ 7/ 8/ 9/ 10/	1 2 3 4 5 6 7 8 9 10
4/2 6 2/1 3 6/5 7 1/ 3/ 5/ 7/	4 2 6 1 3 5 7