

**An Investigation into the Use of Genetic Programming for the Induction of  
Novice Procedural Programming Solution Algorithms in Intelligent  
Programming Tutors**

by

**Nelishia Pillay**

Submitted in fulfilment of the academic requirements of the degree of Doctor of  
Philosophy in the School of Geological and Computer Sciences, Faculty of Science,  
University of KwaZulu-Natal, Durban

All work for this thesis was completed at the former University of Natal

April 2004

## **Abstract**

Intelligent programming tutors have proven to be an economically viable and effective means of assisting novice programmers overcome learning difficulties. However, the large-scale use of intelligent programming tutors has been impeded by the high developmental costs associated with building intelligent programming tutors. The research presented in this thesis forms part of a larger initiative aimed at reducing these costs by building a generic architecture for the development of intelligent programming tutors. One of the facilities that must be provided by the generic architecture is the automatic generation of solutions to programming problems. The study presented in the thesis examines the use of genetic programming as means of inducing solution algorithms to novice programming problems. The scope of the thesis is limited to novice procedural programming paradigm problems requiring the use of arithmetic, string manipulation, conditional, iterative and recursive programming structures.

The methodology employed in the study is proof-by-demonstration. A genetic programming system for the induction of novice procedural solution algorithms was implemented and tested on randomly chosen novice procedural programming problems. The study has identified the standard and advanced genetic programming features needed for the successful generation of novice procedural solution algorithms. The outcomes of this study include the derivation of an internal representation language for representing procedural solution algorithms and a high-level programming problem specification format for describing procedural problems, in the generic architecture. One of the limitations of genetic programming is its susceptibility to converge prematurely to local optima and not find a solution in some cases. The study has identified fitness function biases against certain structural components that are needed to find a solution, as an additional cause of premature convergence in this domain. It presents an iterative structure-based algorithm as a solution to this problem.

This thesis has contributed to both the fields of genetic programming and intelligent programming tutors. While genetic programming has been successfully implemented in various domains, it is usually applied to a single problem within that domain. In this study the genetic programming system must be capable of solving a number of different programming problems in different application domains. In addition to this, the study has also identified a means of overcoming premature convergence caused by fitness function biases in a genetic programming system for the induction of novice procedural programming algorithms. Furthermore, although a number of studies have addressed the student modelling and pedagogical aspects of intelligent programming tutors, none have examined the automatic generation of problem solutions as a means of reducing developmental costs. Finally, this study has contributed to the ongoing research being conducted by the artificial intelligence in education community, to test the effectiveness of using machine learning techniques in the development of different aspects of intelligent tutoring systems.

## **Preface**

The experimental work described in this thesis was carried out in the School of Geological and Computer Sciences, University of KwaZulu-Natal Durban, from January 1998 to January 2004, under the supervision of Professor Alan Sartori-Angus.

The 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 work of others it is duly acknowledged in the text.

Angus 22/04/04

### **Acknowledgements**

I would like to thank Professor Alan Sartori-Angus for supervising this project. I would also like to thank Professor Hugh Murrell for his encouragement and support. A big thank you to Mr Soren Greenwood for his invaluable technical support throughout this project. Thank you to the University of Natal for financing my studies. Last but not least, I would like to thank my parents for their tolerance and support during the past six years.

## Table of Contents

|   |           |
|---|-----------|
| <b>Chapter 1 - Introduction</b>                         | Page -1-  |
| 1. Purpose of the Study                                 | Page -1-  |
| 2. Objectives   | Page -2-  |
| 3. Scope of the Study                                   | Page -2-  |
| 4. Contributions of the Study                           | Page -3-  |
| 5. Thesis Layout  | Page -5-  |
| 5.1 Chapter 2 - Background Information                  | Page -5-  |
| 5.2 Chapter 3 - An Overview of Genetic Programming      | Page -5-  |
| 5.3 Chapter 4 - Methodology                             | Page -5-  |
| 5.4 Chapter 5 - The Proposed Genetic Programming System | Page -5-  |
| 5.5 Chapter 6 - Results and Discussion                  | Page -5-  |
| 5.6 Chapter 7 - Conclusions and Future Work             | Page -6-  |
| <b>Chapter 2 - Background Information</b>               | Page -7-  |
| 1. Introduction   | Page -7-  |
| 2. Intelligent Tutoring Systems                         | Page -7-  |
| 2.1 Introduction  | Page -7-  |
| 2.2 Intelligent Tutoring System Architectures           | Page -8-  |
| 2.3 The Development of Intelligent Tutoring Systems     | Page -8-  |
| 2.4 Reusability and Interoperability                    | Page -9-  |
| 3. Intelligent Programming Tutors                       | Page -10- |
| 3.1 Introduction  | Page -10- |
| 3.2 PROUST  | Page -11- |
| 3.3 The Lisp Tutor                                      | Page -13- |
| 3.4 RAPITS  | Page -15- |
| 3.5 INTELLITUTOR  | Page -16- |
| 3.6 SIPLeS  | Page -17- |
| 3.7 SIPLeS II   | Page -19- |
| 3.8 MoleHill  | Page -20- |
| 3.9 Summary   | Page -21- |
| 3.9.1 Functions of an Intelligent Programming Tutor     | Page -21- |
| 3.9.2 Critical Analysis                                 | Page -22- |
| 4. Proposed Generic Architecture                        | Page -24- |
| 4.1. Student Module                                     | Page -25- |
| 4.2. The Interface Module                               | Page -27- |
| 4.3. The Pedagogical Module                             | Page -28- |
| 4.4. The Domain Module                                  | Page -29- |

|   |  |                  |
|---|--|------------------|
| 4.5.  | The Expert Module .....  | Page -29-        |
| 4.6.  | The Instructional Strategies Module .....                            | Page -31-        |
| 4.7.  | The Problems Module .....  | Page -31-        |
| 4.8.  | Code Specification Module .....                                      | Page -31-        |
| 4.9.  | The Explanation Module .....   | Page -31-        |
| 4.10.   | Learning / Experience Module .....                                   | Page -32-        |
| 4.11.   | Summary of the Interaction Between Modules .....                     | Page -32-        |
| 5.  | The Automatic Derivation of Solution Algorithms .....                | Page -33-        |
| 6.  | Summary .....  | Page -34-        |
| <b>Chapter 3 - An Overview of Genetic Programming .....</b> |  | <b>Page -35-</b> |
| 1.  | Introduction .....   | Page -35-        |
| 2.  | The Standard Genetic Programming System .....                        | Page -36-        |
| 2.1   | The Genetic Programming Algorithm .....                              | Page -36-        |
| 2.2   | Control Models .....   | Page -37-        |
| 2.2.1   | The Generational Control Model .....                                 | Page -37-        |
| 2.2.2   | The Steady-state Control Model .....                                 | Page -37-        |
| 2.2.3   | Varying Population-Size Control .....                                | Page -38-        |
| 2.3   | Functions and Terminals .....  | Page -38-        |
| 2.3.1   | Functions .....  | Page -39-        |
| 2.3.2   | Terminals .....  | Page -39-        |
| 2.3.3   | Closure and Sufficiency .....  | Page -39-        |
| 2.3.3.1   | The Closure Property .....   | Page -39-        |
| 2.3.3.2   | The Sufficiency Property .....                                       | Page -40-        |
| 2.3.4   | Extraneous Functions and Terminals .....                             | Page -40-        |
| 2.3.5   | The Automatic Evolution of the Function and<br>Terminal Sets .....   | Page -40-        |
| 2.4   | Program representation .....   | Page -41-        |
| 2.4.1   | Tree Representation .....  | Page -41-        |
| 2.4.2   | Linear Representation .....  | Page -42-        |
| 2.4.3   | Graphical Representation .....                                       | Page -42-        |
| 2.4.4   | Representing a GP Population Using a Directed<br>Acyclic Graph ..... | Page -43-        |
| 2.4.5   | Stack-based Genetic Programming .....                                | Page -43-        |
| 2.4.6   | Multi-Tree Representation .....                                      | Page -44-        |
| 2.5   | Initial Population Generation .....                                  | Page -44-        |
| 2.5.1   | The Full Method .....  | Page -45-        |
| 2.5.2   | The Grow Method .....  | Page -45-        |
| 2.5.3   | Ramped Half-and-Half .....   | Page -45-        |
| 2.5.4   | Seeding and/or Biasing the Initial Population .....                  | Page -45-        |
| 2.5.5   | Variety .....  | Page -46-        |
| 2.6   | Evaluation .....   | Page -46-        |
| 2.6.1   | Fitness Cases .....  | Page -46-        |
| 2.6.2   | Fitness Function .....   | Page -48-        |

|      |         |   |           |
|------|---------|---|-----------|
|      | 2.6.2.1 | Raw fitness   | Page -49- |
|      | 2.6.2.2 | Standardized Fitness  | Page -50- |
|      | 2.6.2.3 | Adjusted Fitness  | Page -50- |
|      | 2.6.2.4 | Normalized Fitness  | Page -50- |
|      | 2.6.2.5 | Multi-Objective Fitness Functions                                 | Page -51- |
| 2.7  |         | Selection Methods   | Page -51- |
|      | 2.7.1   | Fitness-proportionate Selection                                   | Page -52- |
|      | 2.7.2   | Rank Selection  | Page -53- |
|      | 2.7.3   | Tournament Selection  | Page -53- |
|      | 2.7.4   | Over-Selection  | Page -53- |
|      | 2.7.5   | Truncation or $(\mu, \lambda)$ Selection                          | Page -54- |
| 2.8  |         | Genetic Operators   | Page -54- |
|      | 2.8.1   | Reproduction  | Page -54- |
|      | 2.8.2   | Crossover   | Page -54- |
|      | 2.8.3   | Mutation  | Page -57- |
|      | 2.8.4   | Permutation   | Page -58- |
|      | 2.8.5   | Editing   | Page -59- |
|      | 2.8.7   | Decimation  | Page -60- |
|      | 2.8.8   | The Inversion Operator  | Page -60- |
|      | 2.8.9   | The Hoist Operator  | Page -60- |
|      | 2.8.10  | The Create Operator   | Page -60- |
|      | 2.8.11  | Smart Operators   | Page -60- |
| 2.9  |         | Termination Criteria and Result Designation                       | Page -61- |
| 2.10 |         | Implementing a GP System  | Page -61- |
| 3.   |         | Advanced Genetic Programming Features                             | Page -62- |
|      | 3.1     | Turing Completeness   | Page -63- |
|      |         | 3.1.1 Memory  | Page -63- |
|      |         | 3.1.1.1 Indexed Memory  | Page -64- |
|      |         | 3.1.1.2. Data Structures  | Page -65- |
|      |         | 3.1.1.3 Automatically Defined Storage                             | Page -66- |
|      | 3.1.2   | The Halting Problem   | Page -67- |
|      | 3.1.3   | Iteration   | Page -68- |
|      |         | 3.1.3.1 Iteration Primitives                                      | Page -68- |
|      |         | 3.1.3.2 Automatically Defined Iterations and<br>Loops             | Page -69- |
|      |         | 3.1.3.2.1 Automatically Defined Loops                             | Page -69- |
|      |         | 3.1.3.2.2 Automatically Defined Iterations                        | Page -69- |
|      | 3.1.4   | Recursion   | Page -70- |
| 3.2  |         | Modularization  | Page -71- |
|      | 3.2.1   | Automatically Defined Functions (ADFs)                            | Page -71- |
|      |         | 3.2.1.1 Representation of each Individual                         | Page -71- |
|      |         | 3.2.1.2 ADF References  | Page -74- |
|      |         | 3.2.1.3 Changes that Need to be Made to the<br>Standard GP System | Page -75- |
|      |         | 3.2.1.4 Typing  | Page -75- |
|      |         | 3.2.1.5 The Automatic Evolution of the ADF<br>Architecture        | Page -76- |
|      | 3.2.3   | Encapsulation   | Page -77- |

|  |  |            |
|--|--|------------|
| 3.2.4  | Module Acquisition   | Page -78-  |
| 3.3  | Strongly-Typed Genetic Programming                             | Page -79-  |
| 3.4  | Architecture-Altering Operations                               | Page -80-  |
| 3.4.1  | Structure Duplication  | Page -81-  |
| 3.4.2  | Argument Duplication   | Page -82-  |
| 3.4.3  | Structure Creation   | Page -82-  |
| 3.4.4  | Argument Creation  | Page -83-  |
| 3.4.5  | Subroutine Deletion  | Page -83-  |
| 3.4.6  | Argument Deletion  | Page -83-  |
| 3.5  | Cultural Learning  | Page -83-  |
| 4.   | Problems Encountered in the Implementation of GP Systems       | Page -84-  |
| 4.1  | Introns and Bloat  | Page -84-  |
| 4.2  | Premature Convergence of the Genetic Programming System        | Page -87-  |
| 4.2.1  | Causes of Premature Convergence                                | Page -87-  |
| 4.2.1.1  | Lack of Genetic Diversity                                      | Page -87-  |
| 4.2.1.2  | Selection Noise or Variance                                    | Page -88-  |
| 4.2.1.3  | The Existence of Alpha Individuals                             | Page -88-  |
| 4.2.1.4  | The Destructive Effects of Genetic Operators                   | Page -89-  |
| 4.2.2  | Preventing Premature Convergence                               | Page -89-  |
| 4.2.2.1  | Maintaining Genetic Diversity                                  | Page -89-  |
| 4.2.2.2  | Multiple Iterations  | Page -90-  |
| 4.2.2.3  | Restricted Mating  | Page -90-  |
| 4.2.2.4  | Parallel Population  | Page -90-  |
| 4.2.2.5  | Preventing the Destructive Crossover                           | Page -91-  |
| 4.2.2.6  | The Introduction of Dissimilar Individuals into the Population | Page -92-  |
| 4.2.2.7  | Niching Methods  | Page -93-  |
| 4.2.2.8  | Weighting Fitness Cases  | Page -94-  |
| 4.2.2.9  | The Races Genetic Algorithm (RGA)                              | Page -95-  |
| 4.2.2.10   | Demes  | Page -95-  |
| 5.   | Evolutionary Structured Programming                            | Page -96-  |
| <b>Chapter 4 - Methodology</b>                             |  | Page -99-  |
| 1.   | Research Methodologies   | Page -99-  |
| 2.   | The Proof by Demonstration Methodology                         | Page -100- |
| 3.   | Implementation Details   | Page -101- |
| 4.   | Summary  | Page -101- |
| <b>Chapter 5 - The Proposed Genetic Programming System</b> |  | Page -102- |
| 1.   | Programming Problem Specification                              | Page -102- |
| 2.   | Strongly-Typed GP  | Page -104- |
| 3.   | Program Representation   | Page -105- |
| 4.   | Control Model  | Page -106- |



|   |  |                   |
|---|--|-------------------|
| 5.  | Initial Population Generation .....                    | Page -107-        |
| 6.  | Primitives .....                                       | Page -108-        |
| 6.1   | Arithmetic Operators .....                             | Page -108-        |
| 6.2   | String Manipulation Operators .....                    | Page -109-        |
| 6.3   | Logical Operators .....                                | Page -110-        |
| 6.4   | Memory Manipulation Operators .....                    | Page -111-        |
| 6.4.1   | Named Memory Operators .....                           | Page -112-        |
| 6.4.2   | Indexed Memory Operators .....                         | Page -113-        |
| 6.5   | Conditional Control Structures .....                   | Page -113-        |
| 6.6   | Iterative Control Structures .....                     | Page -115-        |
| 6.7   | Input and Output Operators .....                       | Page -117-        |
| 6.8   | Recursive Operators .....                              | Page -119-        |
| 6.9   | Multiple Statements .....                              | Page -120-        |
| 6.10  | Errors .....   | Page -121-        |
| 7.  | Population Evaluation .....                            | Page -122-        |
| 8.  | Selection Methods .....                                | Page -123-        |
| 9.  | Genetic Operators .....                                | Page -124-        |
| 9.1   | Mutation .....   | Page -124-        |
| 9.2   | Crossover .....  | Page -125-        |
| 10.   | Escaping Local Optima .....                            | Page -125-        |
| 11.   | GP Parameters .....                                    | Page -126-        |
| 12.   | Summary .....  | Page -126-        |
| <b>Chapter 6 - Results and Discussion .....</b> |  | <b>Page -128-</b> |
| 1.  | Changes to the Program Representation .....            | Page -128-        |
| 2.  | Escaping Local Optima .....                            | Page -129-        |
| 2.1   | Problems with Methods Used in the Proposed System .... | Page -130-        |
| 2.1.1   | Maintaining Diversity .....                            | Page -131-        |
| 2.1.2   | Multiple Runs .....                                    | Page -132-        |
| 2.1.3   | Non-Destructive Operators .....                        | Page -132-        |
| 2.2   | The Iterative Structure-Based Algorithm (ISBA) .....   | Page -132-        |
| 2.2.1   | An Overview of the ISBA .....                          | Page -132-        |
| 2.2.2   | Similarity Indexes .....                               | Page -135-        |
| 2.2.2.1   | Calculating the <i>gsim</i> Similarity Index ....      | Page -135-        |
| 2.2.2.2   | Calculating the <i>lsim</i> Similarity Index ....      | Page -137-        |
| 2.2.3   | ISBA Parameters .....                                  | Page -137-        |
| 2.2.4   | Application of the ISBA .....                          | Page -138-        |
| 3.  | Observations .....                                     | Page -139-        |
| 4.  | Summary .....  | Page -140-        |

|   |            |
|---|------------|
| <b>Chapter 7 - Conclusions and Future Work</b>  | Page -141- |
| 1.    Conclusions                               | Page -141- |
| 2.    Future Research                           | Page -143- |
| 3.    Summary                                   | Page -145- |
| <b>References</b>                               | Page -146- |
| <b>Appendix A - Novice Programming Problems</b> | Page -159- |
| 1.    Sequential Problems                       | Page -159- |
| 2.    Conditional Problems                      | Page -159- |
| 3.    Iterative Problems                        | Page -160- |
| 4.    ASCII Graphics Problems                   | Page -161- |
| 5.    Recursive Problems                        | Page -163- |
| <b>Appendix B - Running Simulations</b>         | Page -164- |
| 1.    System Requirements                       | Page -164- |
| 2.    Running the Program                       | Page -164- |
| 3.    Running Simulations                       | Page -165- |
| <b>Appendix C - Simulation Durations</b>        | Page -169- |
| <b>Appendix D - Sequential Problems</b>         | Page -175- |
| <b>Appendix E - Conditional Problems</b>        | Page -203- |
| <b>Appendix F - Iterative Problems</b>          | Page -277- |
| <b>Appendix G - ASCII Graphics Problems</b>     | Page -315- |
| <b>Appendix H - Recursive Problems</b>          | Page -395- |

## List of Figures

### Chapter 2 - Background Information

|  |    |
|--|----|
| Figure 2.3.2.1: The Rainfall Problem presented in [JOHN87] ..... | 12 |
| Figure 2.3.2.2: The Bank Problem presented in [JOHN90] .....     | 12 |
| Figure 2.4.1: Proposed Generic Architecture .....                | 25 |

### Chapter 3 - An Overview of Genetic Programming

|  |    |
|--|----|
| Figure 3.2.2.2.1: The Steady-State Control Model .....                               | 38 |
| Figure 3.2.4.1.1 .....   | 41 |
| Figure 3.3.1.1.3.1: ADS Example .....  | 67 |
| Figure 3.3.1.4.1: ADR Example .....  | 70 |
| Figure 3.3.2.1.1.1: ADF Program Structure .....                                      | 72 |
| Figure 3.3.2.1.1.2: Lisp program with ADFs presented by Koza [KOZA94] .....          | 73 |
| Figure 3.3.2.1.1.3: Alternative ADF Representation .....                             | 73 |
| Figure 3.3.2.1.2.1: Program with two ADFS .....                                      | 74 |
| Figure 3.3.2.4.1: Individual to which the compression operator will be applied ..... | 78 |
| Figure 3.3.2.4.2: Diagram indicating the chosen module .....                         | 78 |
| Figure 3.3.2.4.3: Result of the compression operator .....                           | 79 |

### Chapter 5 - The Proposed Genetic Programming System

|   |     |
|---|-----|
| Figure 5.1.1: ASCII Graphics Problem .....  | 103 |
| Figure 5.3.1: Multi-tree representation of a single individual with three outputs .....   | 105 |
| Figure 5.3.2: A parse tree presenting an individual with one output .....   | 105 |
| Figure 5.3.3: The vector representation of a parse tree .....   | 106 |
| Figure 5.3.4: An individual and its corresponding memory structure given that<br>x=1 and y=1 .....  | 106 |
| Figure 5.6.4.1.1: Using the <i>write</i> operator. Assume that x=1 and y=1 .....  | 112 |
| Figure 5.6.4.1.2: Using the <i>change</i> operator .....  | 113 |
| Figure 5.6.5.1: Parse tree that calculates the maximum of two numbers .....   | 114 |
| Figure 5.6.5.2: Represents a list of options in a <i>switchi</i> instance .....   | 114 |
| Figure 5.6.5.3: An instance of the <i>switchc</i> operator .....  | 115 |
| Figure 5.6.6.1: The use of the <i>for</i> operator in calculating the factorial of the<br>integer N .....   | 117 |
| Figure 5.6.6.2: An instance of the <i>while</i> operator .....  | 117 |
| Figure 5.6.6.3: An instance of the <i>dowhile</i> operator .....  | 117 |
| Figure 5.6.7.1: An individual containing the <i>place</i> operator and the corresponding<br>screen representation maintained by the GP system ..... | 119 |
| Figure 5.6.8.1: A parse tree representation of the algorithm calculating the factorial<br>of the integer N using recursion .....                    | 120 |
| Figure 5.6.9.1: An individual using instances of <i>block</i> operators .....   | 121 |
| Figure 5.6.10.1: Solution algorithm for the square root problem .....   | 122 |

## Chapter 6 - Results and Discussion

|  |     |
|--|-----|
| <b>Figure 6.1.1:</b> Multi-tree representation of a single individual with three outputs . . . . .   | 128 |
| <b>Figure 6.1.2:</b> Example of a solution algorithm for an output and the corresponding set of functions . . . . .                        | 129 |
| <b>Figure 6.2.2.1.1.:</b> Iterative Structure-Based Algorithm (ISBA) for escaping local optima caused by fitness function biases . . . . . | 136 |

## Appendix B - Running Simulations

|   |     |
|---|-----|
| <b>Figure B.1.1:</b> Screen Resolution and Colour . . . . .                             | 164 |
| <b>Figure B.2.1:</b> GP Panel . . . . .   | 165 |
| <b>Figure B.3.1:</b> Problem Category Dialog Box . . . . .                              | 165 |
| <b>Figure B.3.2:</b> Working Directory Dialog Box . . . . .                             | 166 |
| <b>Figure B.3.3:</b> Directory Dialog Box . . . . .                                     | 166 |
| <b>Figure B.3.4:</b> Simulation Dialog Box . . . . .                                    | 167 |
| <b>Figure B.3.5:</b> Solution Dialog Box . . . . .                                      | 167 |
| <b>Figure B.3.6:</b> Solution Dialog Box displayed if a solution is not found . . . . . | 168 |
| <b>Figure B.3.7:</b> System Parameters Dialog Box . . . . .                             | 168 |

## List of Tables

### Chapter 3 - An Overview of Genetic Programming

|  |    |
|--|----|
| Table 3.3.1.1.3.1: Memory types and dimensions ..... | 66 |
|--|----|

### Chapter 5 - The Proposed Genetic Programming System

|  |     |
|--|-----|
| Table 5.1.1: Problem specification for the factorial problem .....                   | 103 |
| Table 5.1.2: Problem specification for an ASCII Graphics Problem .....               | 104 |
| Table 5.6.1.1: Arithmetic Operators .....  | 108 |
| Table 5.6.2.1: String Manipulation Operators .....                                   | 110 |
| Table 5.6.3.1: Logical Operators .....   | 111 |
| Table 5.6.7.1: Programming Problem Specification for an ASCII Graphics Problem ..... | 119 |
| Table 5.6.10.1: Problem specification for the square root problem .....              | 121 |
| Table 5.11.1: GP Parameters for the Proposed Genetic Programming System .....        | 127 |

### Chapter 6 - Results and Discussion

|  |     |
|--|-----|
| Table 6.2.1: Performance of the Proposed Genetic Programming System .....                                  | 130 |
| Table 6.2.1.1: Best individuals for each seed for sequential <b>Problem 4</b> .....                        | 131 |
| Table 6.2.1.2: Best individuals for each seed for recursive <b>Problem 1</b> .....                         | 131 |
| Table 6.2.2.1.1: Best individuals for seed 1034007003030 for sequential <b>Problem 4</b> .....             | 133 |
| Table 6.2.2.1.2: Best individuals for each generation of a single run for recursive <b>Problem 1</b> ..... | 134 |
| Table 6.2.2.2.1.1: Primitive Types .....   | 137 |
| Table 6.2.2.3.1: ISBA Parameters .....   | 138 |
| Table 6.2.3.4.1: Indexes and parameters to detect system convergence .....                                 | 139 |

### Appendix C - Simulation Durations

|   |         |
|---|---------|
| Table C.1: Approximate Duration of Simulations for <b>Sequential Problems</b> .....     | 169-170 |
| Table C.2: Approximate Duration of Simulations for <b>Conditional Problems</b> .....    | 171     |
| Table C.3: Approximate Duration of Simulations for <b>Iterative Problems</b> .....      | 172     |
| Table C.4: Approximate Duration of Simulations for <b>ASCII Graphics Problems</b> ..... | 173     |
| Table C.5: Approximate Duration of Simulations for <b>Recursive Problems</b> .....      | 174     |

### Appendix D - Sequential Problems

|   |     |
|---|-----|
| Table D.1.1: GP Parameters for <b>Problem 1</b> .....         | 175 |
| Table D.1.2: Problem Specification for <b>Problem 1</b> ..... | 176 |
| Table D.1.3: Solutions to <b>Problem 1</b> .....              | 177 |
| Table D.2.1: GP Parameters for <b>Problem 2</b> .....         | 178 |
| Table D.2.2: Problem Specification for <b>Problem 2</b> ..... | 179 |
| Table D.2.3: Solutions to <b>Problem 2</b> .....              | 180 |
| Table D.3.1: GP Parameters for <b>Problem 3</b> .....         | 181 |
| Table D.3.2: Problem Specification for <b>Problem 3</b> ..... | 182 |
| Table D.3.3: Solutions to <b>Problem 3</b> .....              | 183 |

|   |     |
|---|-----|
| <b>Table D.4.1: GP Parameters for Problem 4</b>           | 184 |
| <b>Table D.4.2: Local Optima Parameters for Problem 4</b> | 184 |
| <b>Table D.4.3: Problem Specification for Problem 4</b>   | 185 |
| <b>Table D.4.4: Solutions to Problem 4</b>                | 186 |
| <b>Table D.5.1: GP Parameters for Problem 5</b>           | 187 |
| <b>Table D.5.2: Problem Specification for Problem 5</b>   | 188 |
| <b>Table D.5.3: Solutions to Problem 5</b>                | 189 |
| <b>Table D.6.1: GP Parameters for Problem 6</b>           | 190 |
| <b>Table D.6.2: Problem Specification for Problem 6</b>   | 191 |
| <b>Table D.6.3: Solutions to Problem 6</b>                | 192 |
| <b>Table D.7.1: GP Parameters for Problem 7</b>           | 193 |
| <b>Table D.7.2: Problem Specification for Problem 7</b>   | 193 |
| <b>Table D.7.3: Solutions to Problem 7</b>                | 194 |
| <b>Table D.8.1: GP Parameters for Problem 8</b>           | 195 |
| <b>Table D.8.2: Problem Specification for Problem 8</b>   | 196 |
| <b>Table D.8.3: Solutions to Problem 8</b>                | 197 |
| <b>Table D.9.1: GP Parameters for Problem 9</b>           | 198 |
| <b>Table D.9.2: Problem Specification for Problem 9</b>   | 198 |
| <b>Table D.9.3: Solutions to Problem 3</b>                | 199 |
| <b>Table D.10.1: GP Parameters for Problem 10</b>         | 200 |
| <b>Table D.10.2: Problem Specification for Problem 10</b> | 201 |
| <b>Table D.10.3: Problem Specification for Problem 10</b> | 202 |

## Appendix E - Conditional Problems

|   |         |
|---|---------|
| <b>Table E.1.1: GP Parameters for Problem 1</b>         | 203     |
| <b>Table E.1.2: Problem Specification for Problem 1</b> | 204     |
| <b>Table E.1.3: Solutions to Problem 1</b>              | 205-207 |
| <b>Table E.2.1: GP Parameters for Problem 2</b>         | 208     |
| <b>Table E.2.2: Problem Specification for Problem 2</b> | 209     |
| <b>Table E.2.3: Solutions to Problem 2</b>              | 210-211 |
| <b>Table E.3.1: GP Parameters for Problem 3</b>         | 212     |
| <b>Table E.3.2: Problem Specification for Problem 3</b> | 213-214 |
| <b>Table E.3.3: Solutions to Problem 3</b>              | 215-216 |
| <b>Table E.4.1: GP Parameters for Problem 4</b>         | 217     |
| <b>Table E.4.2: Problem Specification for Problem 4</b> | 218     |
| <b>Table E.4.3: Solutions to Problem 4</b>              | 219-225 |
| <b>Table E.5.1: GP Parameters for Problem 5</b>         | 226     |
| <b>Table E.5.2: Problem Specification for Problem 5</b> | 227     |
| <b>Table E.5.3: Solutions for Problem 5</b>             | 228-233 |
| <b>Table E.6.1: GP Parameters for Problem 6</b>         | 234     |
| <b>Table E.6.2: Problem Specification for Problem 6</b> | 235     |
| <b>Table E.6.3: Solutions for Problem 6</b>             | 236-244 |
| <b>Table E.7.1: GP Parameters for Problem 7</b>         | 245     |
| <b>Table E.7.2: Problem Specification for Problem 7</b> | 246     |
| <b>Table E.7.3: Solutions to Problem 7</b>              | 247-249 |
| <b>Table E.8.1: GP Parameters for Problem 8</b>         | 250     |
| <b>Table E.8.2: Problem Specification for Problem 8</b> | 251     |

|   |         |
|---|---------|
| <b>Table E.8.3: Solutions to Problem 8</b>                | 252-261 |
| <b>Table E.9.1: GP Parameters for Problem 9</b>           | 262     |
| <b>Table E.9.2: Problem Specification for Problem 9</b>   | 263     |
| <b>Table E.9.3: Solutions to Problem 9</b>                | 264-266 |
| <b>Table E.10.1: GP Parameters for Problem 10</b>         | 267     |
| <b>Table E.10.2: Problem Specification for Problem 10</b> | 268     |
| <b>Table E.10.3: Solutions to Problem 10</b>              | 269-276 |

## Appendix F - Iterative Problems

|   |         |
|---|---------|
| <b>Table F.1.1: GP Parameters for Problem 1</b>             | 277     |
| <b>Table F.1.2: Problem Specification for Problem 1</b>     | 278     |
| <b>Table F.1.3: Solutions to Problem 1</b>                  | 279     |
| <b>Table F.2.1: GP Parameters for Problem 2</b>             | 280     |
| <b>Table F.2.2: Local Optima Parameters for Problem 2</b>   | 280     |
| <b>Table F.2.3: Problem Specification for Problem 2</b>     | 281     |
| <b>Table F.2.4: Solutions to Problem 2</b>                  | 282-284 |
| <b>Table F.3.1: GP Parameters for Problem 3</b>             | 285     |
| <b>Table F.3.2: Local Optima Parameters for Problem 3</b>   | 285     |
| <b>Table F.3.3: Problem Specification for Problem 3</b>     | 286     |
| <b>Table F.3.4: Solutions to Problem 3</b>                  | 287-288 |
| <b>Table F.4.1: GP Parameters for Problem 4</b>             | 289     |
| <b>Table F.4.2: Problem Specification for Problem 4</b>     | 290     |
| <b>Table F.4.3: Solutions to Problem 4</b>                  | 291-295 |
| <b>Table F.5.1: GP Parameters for Problem 5</b>             | 296     |
| <b>Table F.5.2: Local Optima Parameters for Problem 5</b>   | 296     |
| <b>Table F.5.3: Problem Specification for Problem 5</b>     | 297     |
| <b>Table F.5.4: Solutions to Problem 5</b>                  | 298     |
| <b>Table F.6.1: GP Parameters for Problem 6</b>             | 299     |
| <b>Table F.6.2: Local Optima for Problem 6</b>              | 299     |
| <b>Table F.6.3: Problem Specification for Problem 6</b>     | 300     |
| <b>Table F.6.4: Solutions to Problem 6</b>                  | 302     |
| <b>Table F.7.1: GP Parameters for Problem 7</b>             | 301-303 |
| <b>Table F.7.2: Local Optima Parameters for Problem 7</b>   | 303     |
| <b>Table F.7.3: Problem Specification for Problem 7</b>     | 304     |
| <b>Table F.7.4: Solution to Problem 7</b>                   | 304     |
| <b>Table F.8.1: GP Parameters for Problem 8</b>             | 305     |
| <b>Table F.8.2: Local Optima Parameters for Problem 8</b>   | 305     |
| <b>Table F.8.3: Problem Specification for Problem 8</b>     | 306     |
| <b>Table F.8.4: Solution to Problem 8</b>                   | 307     |
| <b>Table F.9.1: GP Parameters for Problem 9</b>             | 308     |
| <b>Table F.9.2: Problem Specification for Problem 9</b>     | 309     |
| <b>Table F.9.3: Solution to Problem 9</b>                   | 309     |
| <b>Table F.10.1: GP Parameters for Problem 10</b>           | 310     |
| <b>Table F.10.2: Local Optima Parameters for Problem 10</b> | 310     |
| <b>Table F.10.3: Problem Specification for Problem 10</b>   | 311     |
| <b>Table F.10.4: Solutions to Problem 10</b>                | 312-314 |

## Appendix G - ASCII Graphics Problems

|  |          |
|--|----------|
| <b>Table G.1.1:</b> GP Parameters for <b>Problem 1</b> .....           | 315      |
| <b>Table G.1.2:</b> Problem Specification for <b>Problem 1</b> .....   | 316      |
| <b>Table G.1.3:</b> Solutions to <b>Problem 1</b> .....                | 317- 324 |
| <b>Table G.2.1:</b> GP Parameters for <b>Problem 2</b> .....           | 325      |
| <b>Table G.2.2:</b> Problem Specification for <b>Problem 2</b> .....   | 326      |
| <b>Table G.2.3:</b> Solutions to <b>Problem 2</b> .....                | 327-332  |
| <b>Table G.3.1:</b> GP Parameters for <b>Problem 3</b> .....           | 333      |
| <b>Table G.3.2:</b> Problem Specification for <b>Problem 3</b> .....   | 334      |
| <b>Table G.3.3:</b> Solutions to <b>Problem 3</b> .....                | 335-337  |
| <b>Table G.4.1:</b> GP Parameters for <b>Problem 4</b> .....           | 338      |
| <b>Table G.4.2:</b> Problem Specification for <b>Problem 4</b> .....   | 339      |
| <b>Table G.4.3:</b> Solutions to <b>Problem 4</b> .....                | 340-344  |
| <b>Table G.5.1:</b> GP Parameters for <b>Problem 5</b> .....           | 345      |
| <b>Table G.5.2:</b> Problem Specification for <b>Problem 5</b> .....   | 346      |
| <b>Table G.5.3:</b> Solutions to <b>Problem 5</b> .....                | 347-351  |
| <b>Table G.6.1:</b> GP Parameters for <b>Problem 6</b> .....           | 352      |
| <b>Table G.6.2:</b> Problem Specification for <b>Problem 6</b> .....   | 353      |
| <b>Table G.6.3:</b> Solutions to <b>Problem 6</b> .....                | 354-363  |
| <b>Table G.7.1:</b> GP Parameters for <b>Problem 7</b> .....           | 364      |
| <b>Table G.7.2:</b> Problem Specification for <b>Problem 7</b> .....   | 365      |
| <b>Table G.7.3:</b> Solutions to <b>Problem 7</b> .....                | 366-373  |
| <b>Table G.8.1:</b> GP Parameters for <b>Problem 8</b> .....           | 374      |
| <b>Table G.8.2:</b> Problem Specification for <b>Problem 8</b> .....   | 375      |
| <b>Table G.8.3:</b> Solutions to <b>Problem 8</b> .....                | 376-383  |
| <b>Table G.9.1:</b> GP Parameters for <b>Problem 9</b> .....           | 384      |
| <b>Table G.9.2:</b> Problem Specification for <b>Problem 9</b> .....   | 385      |
| <b>Table G.9.3:</b> Solutions to <b>Problem 9</b> .....                | 386-388  |
| <b>Table G.10.1:</b> GP Parameters for <b>Problem 10</b> .....         | 389      |
| <b>Table G.10.2:</b> Problem Specification for <b>Problem 10</b> ..... | 390      |
| <b>Table G.10.3:</b> Solutions to <b>Problem 10</b> .....              | 391-394  |

## Appendix H - Recursive Problems

|  |         |
|--|---------|
| <b>Table H.1.1:</b> GP Parameters for <b>Problem 1</b> .....           | 395     |
| <b>Table H.1.2:</b> Local Optima Parameters for <b>Problem 1</b> ..... | 395     |
| <b>Table H.1.3:</b> Problem Specification for <b>Problem 1</b> .....   | 396     |
| <b>Table H.1.4:</b> Solutions to <b>Problem 1</b> .....                | 397-398 |
| <b>Table H.2.1:</b> GP Parameters for <b>Problem 2</b> .....           | 399     |
| <b>Table H.2.2:</b> Local Optima Parameters for <b>Problem 2</b> ..... | 399     |
| <b>Table H.2.3:</b> Problem Specification for <b>Problem 2</b> .....   | 400     |
| <b>Table H.2.4:</b> Solutions to <b>Problem 2</b> .....                | 401-402 |
| <b>Table H.3.1:</b> GP Parameters for <b>Problem 3</b> .....           | 403     |
| <b>Table H.3.2:</b> Local Optima Parameters for <b>Problem 3</b> ..... | 403     |
| <b>Table H.3.3:</b> Problem Specification for <b>Problem 3</b> .....   | 404     |
| <b>Table H.3.4:</b> Solution to <b>Problem 3</b> .....                 | 404     |
| <b>Table H.4.1:</b> GP Parameters for <b>Problem 4</b> .....           | 405     |



|  |         |
|--|---------|
| <b>Table H.4.2:</b> Problem Specification for <b>Problem 4</b> .....   | 405     |
| <b>Table H.4.3:</b> Solutions to <b>Problem 4</b> .....                | 406-410 |
| <b>Table H.5.1:</b> GP Parameters for <b>Problem 5</b> .....           | 411     |
| <b>Table H.5.2:</b> Local Optima Parameters for <b>Problem 5</b> ..... | 411     |
| <b>Table H.5.3:</b> Problem Specification for <b>Problem 5</b> .....   | 412     |
| <b>Table H.5.4:</b> Solutions to <b>Problem 5</b> .....                | 412     |

## Chapter 1 - Introduction

### 1. Purpose of the Study

The research presented in this thesis tests the hypothesis that **genetic programming** can be used to induce solution algorithms to novice procedural programming problems. This research forms part of a larger initiative to reduce the developmental costs associated with building intelligent programming tutors.

According to Proulx [PROU00] a first course in programming is a “major stumbling block” for novice programmers. The difficulties experienced by first time programmers is further emphasised by Johnson [JOHN90] who describes the process of learning to program to be both time-consuming and frustrating. Odekirk [ODEK00] states that while one-on-one tutoring has proven to be an effective means of helping novice programmers to learn how to program, it is not economically feasible.

Research conducted by Anderson et al. [ANDE85] revealed that intelligent tutoring systems (ITSs) are a cost-effective means of providing novice programmers with the individualised attention needed to overcome learning difficulties. The effectiveness of intelligent programming tutors (IPTs) is illustrated by the Lisp tutor which not only improved the learners’ performance but also equipped learners with the skills to solve problems within a shorter time span [MURR96].

In spite of the proven effectiveness of IPTs, not many intelligent programming tutors have been developed thus far. This can be attributed to the high developmental costs associated with creating intelligent tutoring systems and hence intelligent programming tutors [SUTH96]. Murray [MURR99] states that although ITSs have proven to be both powerful and effective learning systems they are both difficult and expensive to build. The intelligent programming tutors created thus far ([ANDE96], [APLE95], [CHEE97], [DUBO88], [JOHN90], [UENO96], [WOOD95], [XU99]) have been developed independently of each other and concentrate on only one or two of the functions of an ITS. This is consistent with the findings of both Murray [MURR97] and Freedman et al. [FREE00] which indicate that ITSs usually concentrate on one or two of the main components that an ITS should cater for, due to the large amount of time and effort needed to create just one module of an ITS.

Mizoguchi [MIZO00] ascribes the high developmental costs associated with building ITSs to be the lack of shareable or reusable components for the construction of ITSs. Thus, intelligent tutoring systems are built from scratch. The potential of reusable components is further stressed by Arruarte et al. [ARRU96] who propose that domain specific intelligent tutoring system architectures consisting of reusable, authorable components be created as a means of reducing developmental costs.

The work presented in this thesis is aimed at developing a reusable **expert module**. This module will form part of a generic architecture for the development of intelligent programming tutors. The thesis concentrates on the automatic induction of solutions to programming problems in an attempt to facilitate the reusability of the expert module and hence reduce the developmental costs associated with building intelligent programming tutors. More specifically, the thesis evaluates genetic programming as a means of generating solutions to novice procedural programming problems.

The following section describes the objectives of this study. Section 3 defines the scope of the study. A summary of the results obtained is presented in section 4. Finally, section 5 provides a summary of each chapter.

## 2. Objectives

This thesis tests the hypothesis that genetic programming can be used to induce solution algorithms to novice procedural programming problems. In order to prove this hypothesis the following objectives must be met:

- A problem specification must be defined - The genetic programming system will need a description of the function of the program to be induced and the application domain as input. A programming problem specification will be defined for this purpose.
- An internal representation language must be defined - In order to ensure the reusability of the expert module, the solutions generated by the genetic programming system must be language independent. Thus, solution algorithms will be expressed in an internal representation language. Defining this language will involve identifying the primitives needed by the GP system to successfully induce solution algorithms to novice procedural programming problems.
- The standard genetic programming features needed to derive novice procedural programming algorithms must be identified - This entails identifying the following: control model/s, program representation, initial population generation method/s, genetic operators, selection methods, and the fitness measure.
- Any advanced features needed by the genetic programming system to induce novice procedural programming algorithms must be identified - Examples of such features include the use of memory, strongly-typed GP, cultural learning and architecture-altering operations.
- In some cases genetic programming systems are unable to evolve solutions to a problem. The cause of this is premature convergence of the genetic programming algorithm. Methodologies for escaping local optima in the domain of novice procedural programming problems must be identified.

The next section defines the scope of the study.

## 3. Scope of the Study

The main aim of the study presented in this thesis is to test the hypothesis that genetic programming can be used to generate solution algorithms to novice procedural programming problems in intelligent programming tutors. The study has been delimited as follows:

- Programming problems

Section 4.5 of **Chapter 2** describes the different types of programming problems that must be catered for by an intelligent programming tutor. This study focuses on deriving solutions to those problems that require the derivation of an algorithm or computer program as a solution.

- Programming paradigms

The study concentrates on the procedural programming paradigm.

- Programming concepts

Based on a survey of literature on introductory programming courses ( [BISH87], [DEIT01], [GINA00], [HAGA98], [KNOX98], [KOFF99], [WALK98], and [WEBE00]) it was decided that the study would be confined to the following introductory programming concepts:

- ▶ Arithmetic
- ▶ Character and string manipulation
- ▶ Input and output structures
- ▶ Conditional control structures
- ▶ Iterative control structures
- ▶ Recursion

- The removal of redundant code

The algorithms generated by genetic programming systems contain redundant code referred to as **introns**. Section 4.1 of **Chapter 3** discusses a number of methods for reducing introns. It is evident from the discussions presented in **Chapter 3** that while some studies have indicated that introns are a hindrance to the genetic programming process and should be removed, others have revealed that introns arise as a response to the destructive effects of the genetic operators and ensure the success of a genetic programming system. An investigation into the usefulness of introns and methods for eliminating or reducing them has been left for future research.

- The algorithms evolved by the genetic programming system will only be evaluated with respect to correctness. The evolution of efficient programming algorithms, and the evolution of multiple algorithms, each taking a different approach to solve the same problem, will be examined as part of future research.

The results of the study presented in this thesis are briefly summarised in the next section.

#### 4. Contributions of the Study

The study presented in this thesis uses the **proof-by-demonstration** computing methodology [JOHNb]. This essentially involves iteratively refining the GP system. In order to test the success of the system, it was applied to a set of randomly selected novice procedural programming problems. The study produced the following output:

- A programming problem specification for describing novice procedural programming problems in an intelligent programming tutor. This problem specification describes both the function of the program to be derived and the application domain and forms input to the genetic programming system.

- An internal representation language for the generation of novice procedural programming solution algorithms in intelligent programming tutors. The programming constructs defined by this language are contained in the function and terminals sets of the genetic programming system.
- The following standard genetic programming features are sufficient to facilitate the induction of novice procedural programming problems:
  - ▶ Control model: The generational control model
  - ▶ Program representation: Each program was represented as a parse tree with its own memory structure.
  - ▶ Method of initial population generation: Grow, full or ramped-half-and-half. This was dependent on the problem domain.
  - ▶ Genetic operators: The crossover and mutation operators. The application rates of these operators are problem dependent.
  - ▶ Selection method: Tournament selection. The size of the tournament is problem dependent.
  - ▶ Fitness measure: For all problems, except ASCII graphics problems, one of two fitness measures were used in each case, namely, the error function [KOZA92] or the number of fitness cases for which the program produces the correct output. The choice of fitness measure is problem dependent. The second fitness measure is also used as part of the termination criterion of the GP system in all cases. In ASCII graphics problems the fitness measure is defined in terms of the number of missing characters, number of screen locations containing the correct character, the number of screen locations at which the incorrect character has been written, the number of locations which display a character when the location should be blank, the number of screen locations written to more than once and the number of times the program attempted to access a screen location beyond the boundaries of the screen.
- Strongly-typed genetic programming and the use of named and indexed memory is needed for the successful derivation of novice procedural programming problems.
- In some cases the genetic programming system converged prematurely and was unable to evolve a solution algorithm. The causes of premature convergence were identified to be selection noise, the destructive effects of the genetic operators and fitness function biases against essential components of solution algorithms. Selection variance was dealt with by performing multiple successive runs for each seed. The use of non-destructive genetic operators, which produce offspring with neutral fitness, were implemented to counter the destructive effects of the mutation and crossover operators. An iterative structure-based algorithm (ISBA) was developed to escape local optima caused by fitness function biases against the existence of certain structural components.

This genetic programming system was able to successfully induce solutions to the forty five randomly chosen novice procedural programming problems.

## **5. Thesis Layout**

This section provides a brief summary of the rest of chapters in the thesis.

### **5.1 Chapter 2 - Background Information**

This chapter firstly provides a brief introduction to intelligent tutoring systems. A description of those intelligent programming tutors that are relevant to this study is presented. The chapter also proposes a generic architecture for the development of procedural and object-oriented programming tutors. An examination of the automatic generation of solutions and solution algorithms in intelligent programming tutors is provided.

Finally, this chapter discusses why genetic programming should be considered as a means of inducing novice procedural solution algorithms in intelligent programming tutors.

### **5.2 Chapter 3 - An Overview of Genetic Programming**

Chapter 3 presents a detailed overview of genetic programming. The chapter provides an account of both the standard genetic programming system as well as later advancements in the field. This chapter also describes the limitations of genetic programming and problems that have been encountered in the application of genetic programming. Finally, the chapter discusses other studies that have used genetic programming for the induction of procedural programming algorithms.

### **5.3 Chapter 4 - Methodology**

This chapter describes the methodology employed to test the hypothesis that genetic programming can be used to induce solution algorithms to novice procedural programming problems. Implementation details of the system employed for purposes of testing this hypothesis is also provided.

### **5.4 Chapter 5 - The Proposed Genetic Programming System**

A description of both the standard and advanced genetic programming features of the system proposed for the induction of novice procedural solution algorithms is presented. Details of the mechanisms employed to overcome those limitations of a GP system which may prevent it from converging to a solution is also provided.

### **5.5 Chapter 6 - Results and Discussion**

This chapter presents the results of applying the genetic programming system proposed in **Chapter 5** to the induction of a set of randomly chosen novice procedural programming problems. The chapter provides a description of the refinements made to the proposed genetic programming system, and observations regarding the solutions induced by the system.

## **5.6 Chapter 7 - Conclusions and Future Work**

Chapter 7 discusses the conclusions that can be drawn with respect to the objectives outlined in section 2 of **Chapter 1** and thus the hypothesis that genetic programming can be used to induce novice procedural programming solution algorithms. This chapter also describes future work that will be conducted as an extension to the study presented in this thesis.

## **Chapter 2 - Background Information**

### **1. Introduction**

This chapter provides details regarding the overall research project which led to the study presented in this thesis. The research described in the thesis forms part of a larger initiative to develop a generic architecture for the development of intelligent programming tutors, for the procedural and object-oriented programming paradigms, in an attempt to reduce the developmental costs of building IPTs.

An introduction to intelligent tutoring systems is provided in section 2. A survey of the intelligent programming tutors developed thus far is presented in section 3. Section 4 proposes a generic architecture for the development of intelligent programming tutors for the procedural and object-oriented programming paradigms. Section 5 examines the automatic generation of solutions to one of the types of programming problems listed in section 4.5 of this chapter, namely, the induction of solution algorithms, and motivates why genetic programming should be investigated as a means of deriving solution algorithms.

### **2. Intelligent Tutoring Systems**

This section provides a brief introduction to intelligent tutoring systems.

#### **2.1 Introduction**

Intelligent tutoring systems (ITS), also known as Intelligent Computer-Aided Instruction (ICAI) systems or Intelligent Educational Systems (IES) [MIZO97] have been described as “computer programs that use artificial intelligence techniques to help a person learn” [WONG98]. Schank [BECK99] describes an intelligent system as one that can adapt its behaviour to better accomplish a task. Adaptation is an essential characteristic of an ITS. Intelligent tutoring systems adapt their instruction of a particular domain according to the needs of each learner.

ITSs generally carry out the functions of a human tutor. Kimball et al. [KIMB82] and O'Shea [OSHE82] describe the basic characteristics of an ITS to be the following:

- An ITS can use problem solving heuristics to present the user with appropriate examples to work on based on what the user knows and what needs to be learnt.
- Intelligent tutoring systems can successfully tutor students with diverse backgrounds.
- Intelligent tutoring systems can learn superior problem solving heuristics from the users of the system.
- Intelligent tutoring systems can run experiments to improve their own teaching performance.

The first intelligent tutoring system created is the GUIDON system [CLAN87] which was developed at Stanford University to tutor medical students. Some of the earlier ITSs developed include the WHY system which was used to tutor students on the climatology of Oregon and Ireland; SOPHIE I, II, III were developed for the domain of electric circuits and the BUGGY program for the building of models of students' arithmetic skills.



The next section describes the different components that an ITS can be composed of.

## **2.2 Intelligent Tutoring System Architectures**

An ITS has a modular architecture. Kopec et al. [KOPE92] describe the basic architecture of an ITS as consisting of four main modules:

- Student module/modeller - this module tries to understand the knowledge/skill level of the learner.
- Expert module - this module contains domain specific knowledge which it uses to solve problems that are presented to the user.
- Tutoring module - contains teaching and tutoring strategies.
- Curriculum module - this module specifies the scope, form and content of the instructional material.

An alternative ITS architecture proposed by Freedman et al. [FREE00] is comprised of the following modules:

- Domain module - this module contains domain specific declarative and procedural knowledge .
- Student module - maintains a model of the student's knowledge and preferences; instruction is based on the content of this module.
- Teaching module - contains teaching and tutoring strategies.
- Interface module - this module facilitates student-tutor communications.

According to Murray [MURR97] it is essential that an intelligent tutoring system contains knowledge regarding problem-solving, how to teach, and student modelling. Thus, the basic architecture of an ITS is comprised of a student module, a pedagogical/instructional module and a domain module. Additional modules in an architecture and the interaction between modules are dependant on the function of the ITS and the domain in which it will be used. Thus, the number of modules that an ITS is comprised of and the connection between these modules are domain specific. Each of these modules are implemented using the appropriate artificial intelligence structures and the associated algorithms.

## **2.3 The Development of Intelligent Tutoring Systems**

The development of an ITS requires the implementation of a number of steps. Suthers [SUTH96] describes the processes involved in developing an ITS to include the following:

- Acquiring and encoding domain and instructional knowledge from a number of experts.
- Developing knowledge representation strategies and reasoning mechanisms to represent the knowledge acquired.
- Designing and implementing all the system components of the architecture.

Building an ITS requires input from a number of experts. Hsieh et al. [HSIE96] emphasise that it is essential that experts in the relevant fields must be included in the developmental process. Information contained in the domain and expert modules need to be acquired from an expert in the particular domain.

The instructional strategies and meta-strategies specified in the pedagogical or tutoring module must be developed by instructional designers and educationalists. In order to adapt the tutoring style to a particular student it is essential that a student's learning styles and the mental models developed by the student be taken into consideration. This requires input from cognitive scientists and psychologists. Computer scientists are needed to design the architecture of an ITS and implement or derive artificial intelligence techniques to represent the content of each module and the interaction between modules.

According to Suthers [SUTH96] when a developer creates an ITS, the system is developed from "scratch". There is no foundation which developers can use when creating an ITS and hence each component has to be redeveloped. Thus, the developmental costs associated with creating ITSs, especially with respect to time and manpower, are high.

A possible means of reducing the developmental costs associated with building ITSs is the creation of ITS architectures that consist of reusable, interoperable components. The concepts of reusability and interoperability are discussed below.

## **2.4 Reusability and Interoperability**

Murray [MURR99] and Mizoguchi et al. [MIZO97] describe the development process associated with creating intelligent tutoring systems to be work intensive and hence time consuming. This can be attributed to the fact that each ITS is built from scratch. Mizoguchi et al. [MIZO00] state that shareable/ reusable components are needed in order to reduce the developments costs associated with building an ITS. The effectiveness of reusable components is further highlighted by Grandbastien [GRAN96] who is of the opinion that both design and implementation time can be saved by developing common components that can be easily revised.

Components of an ITS can be reused directly or indirectly. Arruarte et al. [ARRU96] define the direct reuse of components as the reuse of a component, as is, in an ITS different from the one it was originally implemented in. For example, a tutoring module consisting of instructional strategies and meta-strategies may be easily directly reused. Indirect reusability is described by Arruarte [ARRU96] et al. as involving the editing of an existing module.

Authoring tools or shells are required to facilitate the indirect reuse of components. According to Murray [MURR99] authoring tools not only reduce the time and effort required to build ITSs, but also reduce the bottle neck associated with eliciting knowledge from experts and hence facilitate the creation of an ITS by more than one person.

In order to enable the reusability of an ITS architecture it is essential that the modules comprising an ITS architecture are interoperable, i.e. a means via which the modules can communicate and co-operate must be provided. Interoperability facilitates the replacement of one or more modules in a generic architecture. The communication infrastructure must allow for communication links to be maintained between modules that have been directly and indirectly reused to create a new ITS.

According to Cheikes [CHEI95] interoperability is achieved by implementing an ITS architecture as a multi-agent system. Each module of the architecture is an agent or a team of agents. Agents communicate using an agent communication language.

Section 4 of this chapter proposes a generic architecture, consisting of reusable and interoperable components, for the development of intelligent programming tutors for the procedural and object-oriented programming paradigms. The next section reviews the intelligent programming tutors most relevant to this study.

### **3. Intelligent Programming Tutors**

A survey of intelligent programming tutors for the procedural and object-oriented programming paradigms was conducted in order to:

- Determine the facilities and functions that an intelligent programming tutor must provide.
- Identify the shortcomings of existing intelligent programming tutors and the mistakes made in implementing these systems so that these errors are not replicated in the generic architecture developed.

The results of this survey are presented in this chapter.

#### **3.1 Introduction**

DuBoulay [DUBO88] in his paper, “Intelligent Systems for Teaching Programming”, defines two main categories of intelligent programming tutors:

- **Tutors**  
  
Tutors are systems that monitor a student’s progress in grasping a particular concept and based on this, make decisions on how and in what order programming concepts should be introduced. Examples of such tutors include Anderson’s Lisp tutor and RAPITS.
- **Bug-finders**  
  
These tutoring systems debug student programs and present the student with a diagnoses of their errors. A bug-finder that has made a valuable contribution to this field is the PROUST system which debugs Pascal programs.

Some of the first intelligent programming tutors described by DuBoulay [DUBO88] include the MALT tutor, SPADE-0 and LAURA. These systems were considered to be intelligent as they provided individualised feedback. This feedback was dependant on the particular learner’s attempt at a solution. The MALT tutor was used to assist students in overcoming difficulties experienced when programming in machine code. MALT generated a problem which was presented to the student. As the student typed in responses the MALT tutor prompted the student and commented on the response. SPADE-0 was a tutor developed for novice Logo programmers. This tutoring system tutored students on solving the “whishing well” problem. Program planning formed the basis of this system. LAURA was a bug-finder developed to debug Fortran programs.

This system converted both the student solution and a model answer to graphs. These graphs were then standardised and compared.

Most of the earlier programming tutors were developed primarily for experimental purposes and not for classroom use. Those intelligent programming tutors that are most relevant to this study and that have made a valuable contribution to the field are discussed in sections 3.2 to 3.8. Section 3.9 presents an analysis of these tutors.

### 3.2 PROUST

PROUST is a bug-finder that assists novice programmers to debug their programs. PROUST was developed by Johnson et al. [JOHN87] to detect semantic errors in novice Pascal programs. PROUST was implemented in a language similar to Lisp called T and runs on a VAX70. The PROUST system takes a student program as input and produces a list of bugs, including a description of each error, as output.

According to Johnson [JOHN90] an intention-based approach is taken to detect program bugs. Intention-based analysis involves firstly creating a problem description. Each description is composed of a set of goals that must be attained and a description of the data objects that the goals are applied to. Data objects can be either constant-valued or variable-valued. For example, if a problem required a program to read in numbers until a sentinel value of 99999 was entered, one of the data objects would be a constant data object, namely, the sentinel value while the other would be a variable representing the values read in. Each goal is decomposed into a plan or set of alternative plans that must be implemented in order to achieve the goal. This process is referred to as goal decomposition. Frames are used to represent each goal and its corresponding plan/s in the knowledge base.

These goal decompositions are matched against the plans represented by the student program. A plan mismatch can be attributed to an error in the student program. Alternatively, a mismatch can occur as a result of the student taking a different approach to solving the problem than that predicted by the PROUST system. In the latter case goal-reformulation rules are applied in order to adapt PROUST's plans to better reflect the intentions of the student.

The knowledge-base contains plan-difference rules to deal with mismatches. Each plan-difference rule consists of a test component and an action component. There are essentially three types of plan-difference rules:

- Plan-difference rules that describe the bugs or misconceptions which mismatches between plans and the student's code can be attributed to.
- Plan-difference rules that change the mismatched plan so that it more closely reflects the student's intentions.
- Plan-difference rules which change the student's code to clarify the associated goal decomposition.

When a plan-difference rule detects a bug it generates a bug description. This description specifies the kind of bug or misconception that has occurred as well the context in which the bug has occurred. These bug descriptions then serve as input to the process of explanation generation. Once PROUST has completed analysing the problem, the list of errors that it has detected is sorted according to their severity (from most severe to least severe). An explanation is generated for each bug.

The Rainfall problem in **Figure 2.3.2.1** is one the problems PROUST was tested on. PROUST was applied to 206 programs. Eighty nine percent of these programs had bugs. PROUST was able to completely analyse 81%. Fifteen percent of the programs were analysed partially while 4% were aborted. PROUST was able to identify 94% of the bugs in programs. Sixty six of the programs were incorrectly diagnosed as having bugs.

**Rainfall Problem**

Write a Pascal program that will prompt the user to input numbers from the terminal; each input stands for the amount of rainfall in New Haven for a day. Note: since rainfall cannot be negative, the program should reject negative input. Your program should compute the following statistics from this data:

- the average rainfall per day
- the number of rainy days
- the number of valid inputs (excluding any invalid data that might have been read in)
- the maximum amount of rain that fell on any one day.

The program should read data until the user types 99999. This is the sentinel value signalling the end of input. Do not include 99999 in the calculations. Assume that if the input is non-negative, and not equal to 99999, then it is valid input data.

**Figure 2.3.2.1:** The Rainfall Problem presented in [JOHN87]

PROUST was also tested on a more complex problem, namely the Bank Problem in **Figure 2.3.2.2**.

**Bank Problem**

Write a Pascal program that processes three types of bank transactions: withdrawals, deposits, and a special transaction that says no more transactions are to follow. Your program should start by asking the user to input his/her account id and his/her initial balance. Then your program should prompt the user to input:

- The transaction type
- If it is an END-PROCESSING transaction, the program should print out the final balance of the user's account, the total number of transactions, the total number of each type of transaction, and the total amount of service charges and then stop.
- If it is a DEPOSIT or a WITHDRAWAL, the program should ask for the amount of the transaction and then post it appropriately.

Use a variable of type CHAR to encode the transaction types. To encourage saving, charge the user 20 cents per withdrawal, but nothing for a deposit.

**Figure 2.3.2.2:** The Bank Problem presented in [JOHN90]

The results of this analysis was not as positive as the analysis of the Rainfall Problem. PROUST was able to analyse only 50% of the 64 programs completely. The 64 programs had 420 bugs. PROUST only detected 50% of these bugs. According to Johnson et al. [JOHN87] the PROUST system must be tested on a wider range of problems and its accuracy improved before use in a classroom.

### 3.3 The Lisp Tutor

The Lisp tutor developed by Carnegie Mellon University (CMU) is one of the first programming tutors developed. The purpose of this tutor was to provide individualised tuition for students taking Lisp courses at CMU. Students were required to read a booklet on each topic and then use the tutor to work through examples on these topics. The tutor assists the student to plan and code the solution to a problem. The Lisp tutor runs under FRANZLISP.

This tutor is based on the ACT cognitive theory developed by Anderson et al. [ANDE90] and uses the model-tracing methodology [ANDE86] to tutor students. This involves constantly monitoring the student's program. Each symbol typed in is matched against a set of production rules. As soon as the tutor detects that the student is experiencing difficulties it guides the student back onto the right path.

The Lisp tutor provides students with a structured editor via which they can enter Lisp commands. The editor automatically balances parentheses and provides place holders for arguments. For example, if the student types in (**defun** the tutor will display

```
( defun <NAME> <PARAMETERS>
  <process>
)
```

on the screen. The student will have to replace the contents of the angled brackets with code.

This structured editor enables the student to concentrate on the semantics of the Lisp function instead of trying to balance parentheses and checking the syntax of the language. Anderson et al. [ANDE86] have found that this accelerates the learning without depriving students of syntax knowledge.

The interface for interaction purposes is composed of a number of windows. The student types the code in the code window. Feedback is displayed in the tutoring window and hints or reminders are presented in the goals window.

According to Anderson et al. [ANDE90] the use of natural language understanding to obtain student input regarding algorithms is both difficult and expensive. Thus, the Lisp tutor provides the student with menus to express his/her responses when the user is designing an algorithm. The menus are constructed from English descriptions of correct and buggy production rules.

Upon input from the student the tutor tries to determine whether a correct or buggy production rule is being entered by the student. If the input is an error the tutor responds by providing advice. If the input corresponds to a correct rule the tutor stays silent and waits for further input. The Lisp tutor provides two mechanisms for assisting students.

The first provides the student with hints. These hints take the form of queries and reminders about current goals. If necessary the Lisp tutor provides the next piece of code. The next section of code is provided :

- If the student requests the next piece of code.
- If the student has made more than the maximum number of errors, i.e. two errors per portion of code.

The second assists the student to develop the corresponding algorithm if the student has problems with coding. When the system detects that the student is experiencing difficulty coding a problem the student is taken from “coding mode” to “planning mode”. During planning mode the tutor works through the algorithm with the student. After the algorithm is constructed the student can return to coding.

After the student has finished coding the Lisp function the student is taken into Lisp where he or she can trace through the function if necessary.

The Lisp tutor contains a simulation of programming knowledge that the ideal student uses in problem solving called the ideal model. This model is based on studies of how students learn to program. GRAPES (Goal-Restricted Production System) [ANDE85] is used to generate the rules that programmers use for writing programs. A problem solution is comprised of a number of rules where each rule is represented as a production rule. Each production rule contains an IF-part and a THEN-part. The IF-part contains a set of conditions that are used to determine whether the rule applies.

The ideal model also contains a large set of buggy rules, i.e. rules that represent misconceptions novices often develop during learning. The tutor must keep track of what the student currently does or does not know and the student’s approach to each problem. The Lisp tutor contains approximately 325 production rules about planning and writing Lisp programs and 475 buggy versions of the above rules. These production rules cover the following topics: basic syntax of Lisp, design of iterative and recursive functions, use of data structures, means-end planning of code.

Templates are associated with each production rule. These templates form the basis of English explanations which are constructed when the student requests assistance or when an error arises. The explanation templates allow the tutor to describe an error or provide a hint.

A class of twenty students were used for purposes of evaluating the Lisp tutor. These students were divided into two homogeneous groups. All students attended the same lectures and did the same problems as homework. One group used the Lisp tutor to do the problems while the other used FRANZLISP. A proctor was available to assist the students. Students that used FRANZLISP required more assistance than those that used the Lisp tutor.

According to the authors students obtaining help from a human tutor took 11.4 hours to complete the assigned exercises while students obtaining assistance from a Lisp tutor took 15 hours. Students not receiving any help from a tutor took 26.5 hours to complete exercises.

All the students wrote two examinations. Tutored students with the Lisp tutor scored 43% higher on the final examinations.

Students were required to program in a top-down and left to right manner. Students found this restrictive. Furthermore, the tutor reacted to every symbol entered by the student. The student could not type in a sequence of symbols and edit it before getting feedback. Anderson et al. [ANDE86] describe one of the problems associated with immediate feedback to be that the student is not given the chance to work out why a particular piece of code is wrong. Students also found the menu-driven approach to algorithm construction restrictive and time-consuming.

Anderson et al.[ANDE86] state that in order to speed up the development cycle of the tutor, the modularity of the system needs to be improved. As a solution to this problem they propose the PUPs architecture for future versions of the tutor. This architecture essentially consists of three modules: the student, pedagogical and interface module.

### **3.4 RAPITS**

Woods et al. [WOOD95] have developed the RAPITS (Rapid Prototyping Adaptive Intelligent Tutoring System) system, an intelligent programming tutor that teaches Pascal looping concepts. RAPITS is a Windows-based system developed in Asymetrix ToolBook.

Woods et al. [WOOD95] state that research in the domain of teaching programming has suggested a variety of models for programming instruction, there is no one best way. Thus, RAPITS uses a variety of teaching strategies to teach students from diverse cultural backgrounds. It automatically changes its teaching strategy, e.g. the topic to be presented next, according to the student history maintained by the system.

The RAPITS tutor's basic approach to tutoring is learning by analogy. The tutor displays the syntax of a particular looping construct as well as the code of a demonstration example and illustrates the execution of the example code. The learner is then presented with an exercise similar to the demonstration example and is required to predict the outcome of the code. The tutor responds to the students solution to the exercise. The tutor also provides the student with the option of viewing the piece of code as part of a complete program.

The domain knowledge used by RAPITS is stored in a hierarchy of topics broken into concepts. Each concept is stored as an electronic book page. The teaching strategies are built into the domain knowledge. Lesson material and domain knowledge is entered using Microsoft Excel and Word and is linked to Toolbook via the Microsoft Windows Dynamic Data Exchange Protocol.

The student history is essentially a list of page numbers of topics in the electronic book already covered. This history is stored in an array during a session and as a spreadsheet upon exit of the program. This enables the student to return to the position he/she was at when he/she next accesses the system.

Only a prototype of RAPITS has been developed. The students that it will be tested on are novice programmers from a diverse cultural background.

The authors describe an advantage of the system to be that components of the system can be "built rapidly" using commonly used software such as the spreadsheet and the wordprocessor without using a programming language such as Lisp or Prolog.



After evaluating the RAPITS system in tutoring Pascal looping constructs Woods et al. [WOOD95] hope to apply this system to tutoring in other subject areas.

### 3.5 INTELLITUTOR

INTELLITUTOR is a bug-finder developed by Ueno [UENO96]. This tutoring system serves the same function as the PROUST system described above, namely, the detection of logical bugs in novice Pascal programs and the provision of advice on how to rectify these errors.

The difference between these systems lies in the methodology employed to comprehend student programs. The PROUST system takes a plan-based approach, i.e. each program is developed as a set of plans or subprograms, while INTELLITUTOR's interpretation of novice programs is algorithm-based. INTELLITUTOR is implemented in ZERO and runs off a UNIX server.

The INTELLITUTOR system takes a Pascal program as input and provides the learner with a list of bugs and an explanation of each bug. This explanation includes the location of the bug in the program, the type of bug (algorithm-oriented, technique-oriented or language-oriented), the cause of the bug, and advice on how to rectify the error. INTELLITUTOR provides an editor, GUIDE, via which students can enter their programs. GUIDE has a help function to assist students in writing programs. A second component of INTELLITUTOR is ALPUS which interprets novice programs and identifies programming bugs.

INTELLITUTOR utilizes four types of knowledge during the interpretation process, namely, an algorithm for each problem, knowledge on different programming techniques, knowledge of the Pascal programming language syntax, and information on the programming bugs commonly made by students. The latter includes knowledge on how novices make errors as well as details describing relations between errors made by a novice programmer and the programmer's intentions.

Ueno [UENO96] uses a Hierarchical Procedure Graph (HPG) to represent the algorithm for each problem. According to Ueno[UENO96] the use of this data structure enables INTELLITUTOR to interpret more complex problems than the PROUST system is able to. Each node in the graph represents a process and contains a link to a semantic network describing the programming technique (e.g. a while loop) needed to perform the process.

The interpretive process assumes that all solution algorithms to a particular problem will have the same overall structure. Thus, if a deviation from the structure of the corresponding stored solution algorithm is detected this is attributed to poor programming style or an error in the student's program. ALPUS uses pattern matching to interpret student programs. The pattern matching process involves three tasks: normalization, identifying the role of each variable, identifying each process. During normalization the student program is converted to a set of language independent statements. The process of identifying the role of each variable essentially involves identifying the attributes(e.g. type) of the variables. Identification of each process is performed by matching each segment of the student program against standard template patterns in the knowledge base. If a match is found the system concludes that the segment is complete.

If a match is not found the segment is firstly matched against acceptable pattern templates in the knowledge base. If a match is found in this case it means that the student's program is inefficient and the system advises the student on how to improve the program.

If the program segment does not correspond to any of the acceptable pattern templates it is matched against the buggy templates. A match indicates an error in the program and the tutoring system analyses the segment to determine the intention of the student and advises the student accordingly. If the system cannot find a match for the segment this suggests that INTELLITUTOR is unable to comprehend the segment and the knowledge base needs to be updated.

INTELLITUTOR was tested on a class of intermediate level college students. Three programming problems namely, the quicksort algorithm, the “straight” sort algorithm and the Rainfall problem (which PROUST was tested on), was used for this purpose. INTELLITUTOR was able to correctly interpret 68.6% of the quicksort programs and 77.5% of the “straight sort” programs. Ueno [UENO96] is of the opinion that the accuracy of INTELLITUTOR can be improved by using one HPG to represent each subprocess of the algorithm instead of one HPG to represent the entire solution algorithm.

INTELLITUTOR experienced difficulties in interpreting solution programs to the Rainfall problem. The author attributes this to the fact that ALPUS requires all solution algorithms to a particular problem to have the same overall algorithmic structure. While this is the case with both the sorting problems, solutions to the Rainfall problem can take the form of a number of different overall structures. Ueno [UENO96] proposes a combination of the both the HPG-based approach employed by INTELLITUTOR and plan-based approach taken by PROUST as a solution to this problem.

Later studies conducted by Ueno [UENO00] have extended INTELLITUTOR to diagnose errors in both Pascal and C programs. ALPUS II was developed for this purpose. In this version of INTELLITUTOR GUIDE is implemented in Java in order to facilitate portability. ALPUS II caters for a subset of C programming constructs namely, those for which equivalent Pascal primitives exist. A “generalized abstract syntax” (AL) was developed to enable ALPUS II to cater for both Pascal and C programs. For example, AL contains a representation that is equivalent to both a Pascal *while-do* loop structure and a C *while* loop structure. INTELLITUTOR has not as yet been tested on C programs.

### 3.6 SIPLoS

SIPLoS is an intelligent tutoring system for intermediate Smalltalk programming. SIPLoS was developed using the ParcPlace Digitaltalk VisualWorks programming environment on an Apple Machintosh. Users are assumed to have a knowledge of introductory Smalltalk concepts and a familiarity with the Smalltalk programming environment. The main aim of SIPLoS is to help the learner to acquire the skills necessary to reuse existing libraries in a collaborative environment. SIPLoS concentrates on the pedagogical function of an ITS.

The instructional strategies employed by SIPLoS are GBS (Goal-Based Scenarios) and cognitive apprenticeship. According to Chee et al. [CHEE97] the GBS methodology is used to “teach” skills in a particular domain.

The learner has a choice of selecting one of three tasks, namely object-oriented interface (OOI) construction, object-oriented design (OOD) or object-oriented programming (OOP). The learner can perform this task for one of the domains documented by the system. Currently, the system contains one domain, namely, the Album domain.

In this domain each of the tasks needs to be carried out in the context of developing a program that creates a database containing personal details of friends and colleagues. The database also contains photographs and movies for each person. Completed solutions for each of these tasks already exist in the knowledge base. If the student has to complete the OOI task for example, he or she can assume that the other two tasks will be completed by other team members working on the same project (in this case the OOD and OOP task solutions will be retrieved from the knowledge base). The student has to ensure that the solution he or she creates can be integrated with the solutions to the other two tasks for the particular domain. Thus, the student appears to be working in collaboration with other team members to complete the overall project.

The problem specification for each task consists of a work requirement and a list of tasks which must be completed in order to obtain a solution. SIPLoS contains a facility which enables the user to view the methodology employed to solve similar problems.

The learner constructs solutions to a particular task, i.e. OOI, OOD or OOP by reusing or adapting code components that exist in the Smalltalk repository maintained by SIPLoS. The user can access these components by typing in relevant keywords from the problem statement, or using the content index. Both of these actions results in a list of case components being displayed on the screen. The corresponding code for each component is also displayed. The user can request a variant of a particular component to be displayed. These case components are the building blocks from which a solution can be built. After selecting different case components or variations thereof the learner copies the code for each component to a workspace and edits these components to create a solution. If the student feels that the search for keywords was not successful or the solution cannot be derived from the chosen components, he or she can initiate another search by typing in a new combination of keywords. Chee et al. [CHEE97] refer to this process as active learning.

A number of tools are available to assist the learner:

- The Advisor - Provides access to code fragments, explanations, etc based on the keywords entered by the user. The learner can activate the Advisor at any stage during the process of constructing a solution. The grain size of the information displayed by the Advisor is dependent on the student level, i.e. beginner, intermediate or advanced, entered by the user.
- Program Case Components Selection - Assists the user in selecting useful case components from the repository of case components maintained by the system. This tool is activated by the user typing in keywords or using the context index.
- Program Task Explanations - Provides explanations of the programming task at hand.
- Tutorial Examples - Provides access to Smalltalk code examples. The user has to type in keywords describing the examples he or she is looking for.
- Case Retriever - Allows the user to retrieve completed case components for OOI, OOD and OOP.
- Case Recorder - Once the student has successfully completed a task the student can add his/her solution to the particular task, i.e. OOI, OOD or OOP, to the SIPLoS repository. The learner is presented with a template for this purpose.
- Online documentation - Evokes a link to the online version of the VisualTools Cookbook, a reference manual for Smalltalk.

SIPLeS is a case-based system. Information contained in the SIPLeS knowledge base includes:

- Smalltalk knowledge - This includes information accessed via the Case Retriever, programming task explanations and tutorial examples.
- Project knowledge - Information regarding a particular task, i.e. OOI, OOD or OOP.
- Domain related knowledge, e.g. information about the Album domain.

The knowledge base is represented as a multi-level hierarchical tree structure and contains both project knowledge and Smalltalk knowledge. Each node is a frame. Each type of knowledge node has a different slot structure. Knowledge nodes are indexed by keyword. Domain-related knowledge nodes and case components are indexed by context as well.

An evaluation of SIPLeS is still to be performed.

### 3.7 SIPLeS II

Xu et al. [XU99] describe the SIPLeS II system which diagnoses semantic errors in novice programs written in Smalltalk. SIPLeS II is implemented in Smalltalk VisualWorks 2.5. According to Xu et al. there are a number of methods that can be used to diagnose errors in computer programs. The approach taken by Xu et al. [XU99] is the source-to-source approach. This essentially involves matching the student program against a model program. The methodology implemented for the purposes of error detection is not language dependent and can be used to assess programs in other object-oriented languages such as C++ or Java. However, the system has been tested using only Smalltalk programs.

According to Xu et al. [XU99] a semantic-level comparison of the student and model programs is conducted as follows:

- Parse tree versions of these programs are converted to abstract syntax trees.
- The abstract syntax trees are then standardized.
- A flow-graph is generated for both trees. The purpose of this is to elicit information regarding the definition of program components.
- Advanced standardization is then performed on both the programs.
- The trees are converted to AOPDGs (Appended Object-Oriented Program Dependence Graphs) in order to eliminate syntax level differences such as a difference in the ordering of program statements.
- A comparison algorithm is then applied. The algorithm identifies and resolves differences of variable names and control structures.
- A report describing the errors detected is generated.

Redundant code is also removed from the both the programs prior to the comparison of programs. SIPLeS II identifies expressions which are syntactically different but are semantically the same and reduces these to a common form. This system is also capable of recognising whether the algorithm implemented by a student differs from that implemented by the model program.

The system was tested on a hundred programs. These programs were student solutions to three problems. The first required the student to calculate the taxi fare given the mileage and taxi charge. The second problem involved testing whether a number is a perfect square while the third required a program that could identify a palindrome. Each program contained ten to twenty statements.

The system correctly diagnosed 88% of the programs. When the system detected a difference in algorithms the instructor was required to enter another model program implementing an algorithm similar to that of the student's. In this particular case study the instructor was required to enter 4 to 5 model programs implementing different algorithms for each problem.

### 3.8 MoleHill

Alpert et al. [ALPE95] (and [ALPE99]) have developed the MoleHill tutoring system to assist novice programmers write Smalltalk programmers. First time Smalltalk programmers not only experience difficulties with learning how to program in Smalltalk, but also struggle with using the Smalltalk environment and available utilities for constructing programs. MoleHill is aimed at students who have previously been exposed to the procedural programming paradigm but are novice object-oriented programmers. The programming tasks presented to the learner by MoleHill are at an introductory level. The MoleHill tutor concentrates on the instructional/pedagogical function of an ITS. The design of the MoleHill system was based on:

- The results of research conducted on instructional strategies.
- The authors' observations of problems encountered by novice Smalltalk programmers.
- The authors' observation of human tutoring in the Smalltalk domain.
- The authors' intuitions as computer scientists and research psychologists.

The main pedagogical strategy employed by MoleHill is instruction by demonstration. This methodology is based on the idea that one learns best if an expert either demonstrates a particular procedure or concept to the learner or an expert provides input on the learner's demonstration of a procedure or concept. In order to facilitate this instructional methodology MoleHill uses what Alpert et al. [ALPE95] describe as "software gurus". These are essentially animated agents that perform the demonstrations. Two such agents exist, namely the Language guru and the Interface guru. The Language guru assists students with problems they experience with programming in Smalltalk. The Interface guru handles student queries with regard to using the Smalltalk environment to access the different Smalltalk classes. Explanations given by the gurus during demonstrations are verbal instead of text-based. Every effort has been made by Alpert et al. [ALPE95] to ensure the agents are easily and clearly distinguishable. As a result there is a marked difference in the physical appearance and "voices" of both the agents.

The demonstrations given by both the gurus are a combination of bitmap animations, verbal commentaries and dynamic animations of the interface components. Bitmap animations involve the animation of a component of the agent, e.g. an agent frowning or pointing at a particular feature of the Smalltalk environment. Dynamic animations are simulations of a particular task in the Smalltalk environment.

The programming tasks presented to the students require the learners to use existing Smalltalk classes to create a solution. At any stage during the problem-solving process the learner can evoke either of the gurus.

When a learner chooses a new project from the menu the project content and required output is demonstrated by the Language guru by means of a dynamic animation. The learner can evoke this demonstration at any stage during the processes of deriving a solution. The gurus also provide diagnostic advice when they detect errors made by the student when using the Smalltalk environment or programming in Smalltalk.

Currently, MoleHill has one programming task in its knowledge base. This project requires the learner to create a Text Pane which displays keyboard input in upper case. According to Alpert et al. [ALPE95] this problem was chosen because the output requirements of the problem are easily assessable. Furthermore, in solving this particular problem it was felt that the learner will acquire the knowledge of a large number of Smalltalk concepts.

MoleHill keeps track of the student's actions by performing intention-based analysis of the students plans. Model-tracing is used to diagnose student programming errors. This methodology involves matching student plans against expert programmers programming plans for the same task.

The GoalPoster is a component of the MoleHill system that maintains a goal-plan tree of the students goals and plans and the student's progress in achieving these goals. GoalPoster infers a student's goals from the student's actions, e.g. browsing the Smalltalk class hierarchy. A blackboard displaying the students goals and plans form part of the MoleHill interface. At the root of the goal-plan tree is a specification of the programming problem posed to the student. If the tutor detects that the student is in need of assistance or is formulating a non-optimal solution an icon representing the relevant guru is placed on the blackboard in line with the goal that the student is currently working on. If the learner wishes to obtain assistance the student can click on the icon. Alpert et al. [ALPE95] have built this mechanism into the system to prevent stifling of the learning process if the learner does not wish to receive assistance at that particular stage of the problem-solving process. The tutor maintains a repetition heuristic to keep track of how often the student requires advice for the same goal. If advice is required more than once the tutor uses a different approach each time, e.g. the tutor may provide a more detailed explanation than previously or if the previous response was text-based it will perform a demonstration instead.

At the end of a task the user can request a replay of the steps he/she performed to derive a solution. The replay essentially performs a trace of the goals and plans used in finding a solution. In addition to the replay the student can also reexamine those goals for which non-optimal solutions were found. Non-optimal solutions are marked by MoleHill for this purpose. Thus, the student can attempt to find a more efficient means of attaining the goal/s in question. This facility can also be used at any stage during the problem solving process.

Informal evaluations of the tutor have been performed. This has involved obtaining feedback from colleagues who have used the system. Alpert et al.[ALPE95] describe the feedback as positive.

### **3.9 Summary**

This section summarises the aspects highlighted by the review of the intelligent programming tutors above. Section 3.9.1 describes the functions that must be provided by an intelligent programming tutor and hence catered for by the generic architecture. A critical analysis of the IPTs described is presented in section 3.9.2.

#### **3.9.1 Functions of an Intelligent Programming Tutor**

From the survey of intelligent programming tutors it is evident that the following functions should be provided by an IPT:

- Presenting and explaining programming concepts. The need for this is illustrated by the RAPITS, SIPLoS and MoleHill IPTs.
- Providing students with different types of programming problems to work through. Each type of problem will test a different aspect of programming. For example, the RAPITS system provides exercises which determine whether the student is able to comprehend and predict the output of a code fragment. The SIPLoS system presents students with problems that test the student's ability to use existing libraries and classes while the Lisp tutor and MoleHill test the student's ability to develop algorithms and computer programs.
- Assisting students to develop solutions to the programming problems.
- Assisting students in debugging their programs for semantic errors. This basically involves detecting logical errors, identifying student misunderstandings, explaining these to students and advising the student on how to fix programming bugs.

The generic architecture proposed in section 4 will cater for all these functions.

### 3.9.2 Critical Analysis

A human tutor, assisting a student to program, usually performs all of the functions listed in section 3.9.1. The IPTs presented implement just one or two of these functions. This can be attributed to the time required to develop a system to carry out just one of these functions. In addition to these functions a tutor often needs to illustrate to a student how the student can debug his/ her program for semantic errors. According to du Boulay [DUBO88] this is an essential function of an intelligent programming tutor. None of the IPTs provide this option and the generic architecture needs to cater for this. Furthermore, the IPTs described do not implement all the functions of an intelligent tutoring system highlighted in section 2. Most of the IPTs presented have concentrated on the student modelling aspects and/or the instructional strategies of an ITS. This further emphasises the high developmental costs associated with building an ITS and the need for a generic architecture for the creation of IPTs.

The intelligent programming tutors examined above are platform dependent and hence not portable. For example, INTELLITUTOR is Unix-based and RAPITS runs in a Windows environment. A generic architecture for the development of IPTs will need to be both portable and reusable. Hence, the generic architecture will be implemented in Java.

It is evident from the studies presented that an important decision that needs to be made during the tutoring process is when to provide the student with feedback. Immediate feedback ensures that the situation in which a student struggles with a problem for hours is avoided. Furthermore, the student can deal with errors as they occur instead of having to work through a list of errors after completing a program. However, as illustrated in the evaluation of the Lisp tutor, immediate feedback can be restrictive in that the student is not given the chance to fully formulate his or her ideas or detect errors on his or her own, which is a learning experience in itself.

When the MoleHill tutor detects that a student is experiencing difficulties, a help icon is placed on the screen and its left up to the student to decide whether he or she requires assistance at that point. Instead of possessing an overall strategy regarding feedback, as was the case in the IPTs presented, the generic architecture will tailor the frequency of feedback to the needs of the individual student, taking into account factors such as the student's experience, rate of work, the student's history and learning style.

In a number of the IPTs presented, the knowledge representation structures used to describe solution algorithms as well as the methodologies employed to assess student programs do not generalise well. For example, the representation used in INTELLITUTOR assumes that all student algorithms will take the same overall approach as that employed in the stored solution algorithm in solving the given problem. Furthermore, the PROUST and Lisp tutors use production rules to represent programming knowledge. As the complexity of problems increase this could result in these systems becoming more and more brittle. Knowledge representations to represent solution algorithms and methodologies to assess student programs, that are not brittle need to be identified or developed for the generic architecture.

In order to cater for more than one solution algorithm for each programming problem, the PROUST and the SIPLeS II systems store alternative algorithms for each problem. For example, the SIPLeS II system stores 4 to 5 solutions for each problem. This contributes to the developmental costs of the intelligent programming tutor. A methodology that automatically evolves solution algorithms for a programming problem, each taking different approaches, needs to be identified or derived.

It is evident from the review of the RAPITS system that more than one instructional strategy can be used to teach a particular programming concept. Similarly, multiple meta-strategies can be employed for the overall tutoring process. In the generic architecture the instructional and meta-strategies implemented will be dependent on the particular student and his or her history, learning style, etc.

INTELLITUTOR and MoleHill evaluate student programs with respect to correctness and efficiency. The same approach will be taken in the generic architecture.

The importance of program design is illustrated in both the Lisp tutor and SIPLeS. Whenever the Lisp tutor detects that the student is experiencing problems coding a particular program, the student is taken into planning mode and the tutor helps the student create the corresponding algorithm, i.e. design the program. One of the task that must be completed by students using the SIPLeS tutor is object-oriented design. According to Ziegler et al. [ZIEG99], Grove [GROV98], Hagan et al. [HAGA98] and Ginat [GINA00] a fundamental component of any programming course is program design. Koffman et al. [KOFF99] and Gaona [GAON00] emphasize that the most important aim of an introductory programming course is teaching a programming methodology rather than teaching a programming language. Lidtke et al. [LIDT98] and Gaona [GAON00] state that once learners have acquired the necessary problem solving skills, expressing the solution algorithms in a particular programming language is a simple task. Thus, the introductory programming courses run by Lidtke et al. [LIDT98], Ziegler [ZIEG99] and Gaona [GAON00] consist of two main components the first of which teaches program design and algorithm development. Once students have completed this component and acquired the necessary programming skills they can move on to the second component which involves translating algorithms to a particular programming language.

The importance of design is further emphasized by Barr et al. [BARR99] who describe object-oriented design to be the crux of object-oriented programming. Thus, it is essential that novice programmers understand the design process and create object-oriented designs prior to writing any computer programs. One of the functions of an intelligent programming tutor, and hence the generic architecture, described in the previous section is to assist students write solution programs to programming problems.



This must be achieved by firstly helping the student design their program.

All the intelligent programming tutors presented in this chapter have provided students with a textual environment in which to express solution algorithms. However, research conducted by Reiser et al. [REIS88] has indicated that graphical representation is more suitable for program construction in an intelligent programming tutor. Furthermore, research conducted by Calloni et al. [CALL97] indicated that the use of an iconic programming interface instead of a textual interface improved student performance. An investigation into the relationship between factors such as a student's learning style, history and experience and interface preferences needs to be conducted. The results of this investigation must be taken into consideration when deciding which interface to use.

The study presented by Alpert et al. [ALPE99] has indicated that the use of pedagogical agents enhances the learning process and motivates students. In addition to this the instructional methodology employed to incorporate the use of pedagogical agents, namely, instruction by demonstration, has proven to be effective in the programming domain.

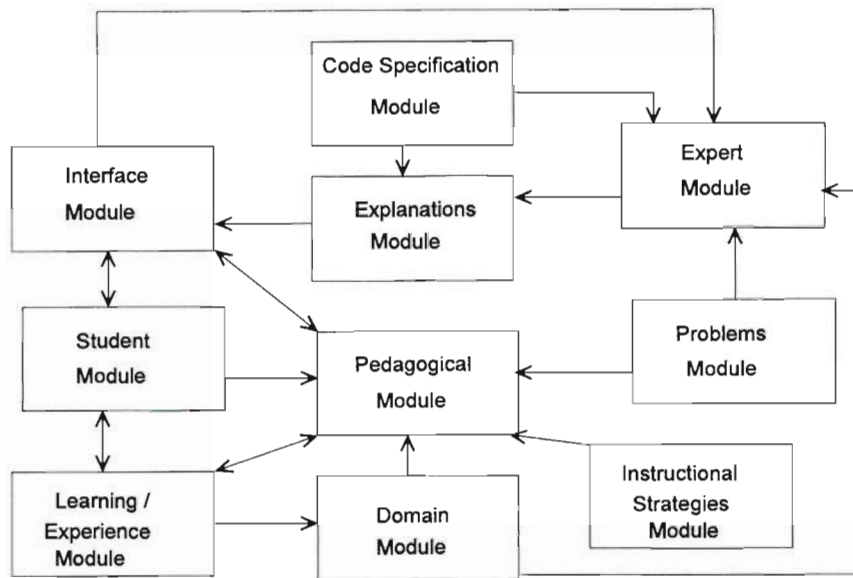
The next section proposes a generic architecture for the development of intelligent programming tutors for the procedural and object-oriented programming domains. The factors highlighted in section 3.9.1 and 3.9.2 have been taken into consideration when deriving the architecture.

#### **4. Proposed Generic Architecture**

This section proposes a generic architecture for the development of intelligent programming tutors for the procedural and object-oriented programming paradigms. The following surveys provided input to the process of deriving the generic architecture:

- The survey of existing intelligent programming tutors presented in the previous section.
- A survey of studies on teaching programming.
- A survey of intelligent tutoring systems.

**Figure 2.4.1** illustrates the proposed generic architecture. Details regarding the derivation of the architecture have been published in [PILL00] and [PILL03b]. In order to facilitate the reusability and interoperability of each module in the architecture, each module will function as an agent or team of agents. Authoring tools will be developed to facilitate indirect reuse of the architecture. Sections 4.1 to 4.10 describe each of these modules in detail. Section 4.11 summarises the interaction between the different modules.



**Figure 2.4.1: Proposed Generic Architecture**

#### 4.1. Student Module

One of the features of an intelligent tutoring system is the provision of individualized tuition. In order to tailor instructions to the needs of a particular student the ITS must determine the student's level of knowledge and monitor the student's progress. This is the function of the student module.

The knowledge acquired by the student module forms the basis of the decisions made by the pedagogical module. These decisions include which instructional strategies to use, when to provide feedback, which problem to present, etc.

The student module in the generic architecture will keep track of the following information:

- The students's ability, knowledge of the subject and level of understanding of the subject

According to Hsieh et al. [HSIE96] “a student module should keep track of the student’s ability, knowledge, and understanding”. Beck et al. [BECK96] state that it is essential that a grain size for this understanding be established, e.g. the student knows the domain versus the student does not know the domain. Barton [BART95] suggests that an ITS can determine a student’s level of knowledge by using an initial probability. This probability can either be determined by the developer and entered during the authoring process or it can be calculated by the system.

Various methods have been used to acquire knowledge needed by the student module. For example, Mizoguchi et al.[ MIZO97] have constructed student modules based on the observation of student's solutions to problems.

The AIDA system developed by Hsieh et al.[HSIE96] asks the student practice questions in order to determine the student's level of knowledge. The student module in the SELTS system, also created by Hsieh et al.[HSIE96], essentially consists of a list of lessons computed by the students and the scores obtained by the students on tests.

This knowledge will effect decisions such as which topic should be presented to the student next.

- A list of misconceptions made by the student [BECK96]. This list must be constantly updated.
- The student's learning style, cognitive style, problem solving skills, gender and personality differences.

According to Dale et al. [DALE97] and Beck et al. [BECK96] each student has an individual learning style. When programming concepts are presented to students a particular instructional style or methodology is used. Whether the instructional style is effective as a means of helping a particular student grasp the concepts being taught, is dependent on the student's learning style. Thus, certain instructional methodologies will only be effective for certain learning styles. Consequently, it is essential that a student's learning style is taken into consideration when tutoring students. As outlined in [DALE97] a number of learning style models can be used for this purpose. The most frequently used model in the instruction of programming courses is Kolb's model. Studies need to be conducted to firstly identify the different instructional and meta-strategies that can be used to tutor programming and then determine the relationship between the student's learning style and pedagogical preferences. The results of these studies can be used by the student module of the generic architecture.

In addition to learning styles Goold [GOOL00] describes the student's cognitive styles, problem-solving skills, gender and personality differences as characteristics that effect the student's programming ability. The relationship between these characteristic and the a student's success at programming needs to be determined and used by the student module.

Section 3.9.2 identifies three types of interfaces that can be used to construct algorithms, namely, textual, graphical and iconic. The relationship between the above characteristics and the student's interface preference also needs to be established.

- The student's previous programming experience

When learning a new programming paradigm students that have been previously exposed to different paradigms experience additional learning difficulties in understanding the new paradigm. This problem is stressed by Declue [DECL96] who states that if a student who has learned the procedural paradigm is presented with a C++ program he/she will not be able to see the objects. Similarly, if a student who is well versed on the object-oriented paradigm is presented with a C program the student will most definitely find objects in the program.

Thus, the student module needs to ascertain which programming paradigms the student has already been exposed to and use this information during the tutoring process.

- The student module needs to ascertain the student's acquisition and retention rates.

Beck et al.[BECK96] define the acquisition rate as measuring how fast a student learns new topics. Retention refers to how well the student recalls material over time.

Mizoguchi et al. [MIZO97] explain that during the tutoring process a student's knowledge on a particular concept will change. This could result in contradictions during the student modeling process. Thus, it is essential that the student module monitors this monotonic change and is able to identify and deal with the resulting contradictions.

Knowledge representation structures for representing the knowledge and methodologies for ascertaining the necessary inferences must be identified. Some of the artificial intelligence techniques used for student modeling include model-tracing, Bayesian networks, and fuzzy set modeling [MITR99]. Machine learning strategies have also been used for this purpose in some ITSs [BECK99].

#### 4.2. The Interface Module

Beck et al. [BECK96] describe the functions of the interface module as controlling interactions with the student. This essentially involves presenting the student with the correct screen layout for the task at hand and facilitating communication between the student and the other components of the ITS.

Research conducted by Anderson et al. [ANDE86] revealed that the effectiveness of the ITS is largely dependent on the design of the interface. Anderson et al.[ANDE86] list the following points that must be taken into consideration when designing an interface for an intelligent programming tutor:

- The interface must clearly indicate to the student where he or she is in a problem. This was illustrated in the MoleHill tutor.
- Errors made must be clearly visible to the student.

Hsieh [HSIE96] states that the student should not need to spend lots of time in learning how to use the interface. The importance of presenting a student with an interface that is easy to learn is further emphasized by Anderson et al.[ANDE96] who state that the interface must be both easy to learn and use. Anderson et al. describe an interface that is easy to use as one that minimizes the number of actions that the student has to perform to communicate with the tutor.

The interface module in the generic architecture must cater for the above characteristics.

According to Patel et al. [PATE96] natural language processing is still too "remote" to be used in intelligent tutoring systems for communication purposes. More specifically, Anderson et al. [ANDE86] describe natural language processing as being both difficult and expensive (in terms of system resources) as a means via which student programming algorithms can be entered into an ITS. Thus, the use of natural language processing will not be catered for by the interface module.

A programming interface refers to the interface via which a student enters a programming algorithm or code. **Section 3.9.2.** describes three different types of programming interfaces, namely, textual, graphical and iconic interfaces.

Based on input from the pedagogical module and student module the interface module must present the student with the appropriate interface.

Rosenberg et al. [ROSEN97] state that students learning the object-oriented paradigm require adequate facilities for testing algorithms and programs. This need is further emphasized by Kolling et al. [KOLL96] who are of the opinion that graphical visualization mechanisms are necessary in order for students to identify the relationships between the different object-oriented concepts. Furthermore, according to Robling et al. [ROBL00] algorithm animation plays a crucial role in assisting students to understand the functioning of complex algorithms.

Thus, it is essential that an intelligent programming tutor, and hence the interface module of the generic architecture, provide the user with tools for debugging and tracing through algorithms and for code and algorithm animation.

The effectiveness of the use of pedagogical agents for demonstration of concepts is illustrated in the MoleHill system. The interface module needs to cater for the use of such agents.

The interfaces of the necessary tools must be maintained by the interface module and displayed upon request from the pedagogical module.

#### **4.3. The Pedagogical Module**

Intelligent tutoring systems provide individualized instruction. This instruction is in the form of meta-strategies and instructional strategies. Instructional strategies refer to the methods that are used to teach a particular concept. Instructional strategies are represented in the instructional strategy module described in section 4.6..

Meta-strategies on the other hand refer to the overall teaching or tutoring strategy employed. For example, Woods et al. [WOOD95] employ a learning by analogy strategy in the RAPITS system.

A similar approach, namely, the programming-by-example meta-strategy is described by Johansson [JOHA98]. This methodology involves students learning how to program by studying existing program examples. Another example of a meta-strategy would be presenting the student with a topic and a problem to solve on that topic.

Tennison [TENN94] attributes the lack of the widespread use of intelligent tutoring systems in the classroom to the fact that tutoring systems usually provide only one teaching strategy. From the studies described by Woods et al. [WOOD96] there is not one best strategy to teach or tutor programming, multiple strategies are needed. Thus, an intelligent programming tutor should provide multiple meta-strategies and instructional strategies.

Decisions regarding which teaching strategy to employ, frequency of feedback, topic selection, etc. will be based on input from the student module in order to tailor tutoring to the particular student.

This input is also used by the pedagogical module to communicate with the instructional strategy module to choose the most appropriate instructional strategies for the particular student and the topic at hand.

Low level issues that the pedagogical module needs to consider as part of the meta-strategy include:

- Topic selection - According to Beck et al. [BECK96] the pedagogical module needs to obtain input from the student module to determine which topics the student needs to be tutored on.
- Problem generation - Beck et al. [BECK96] state that once a topic is chosen the problem must be generated for a student to work on. The pedagogical module needs to determine which problem to present to the student according to the topic and the level of difficulty.
- Feedback - According to Beck et al. [BECK96] and Barton [BART95] another decision that needs to be made by the pedagogical module is how frequently to provide the student with feedback such as hints and error listings. It is evident from the discussion presented in section 3.9.2 that there is much controversy surrounding the issue of feedback. Beck et al. [BECK96] state that too little feedback can lead to frustration while too much of feedback can interfere with the learning process. The pedagogical module in the proposed generic architecture will devise a feedback strategy for each student according to factors such as the student's history, acquisition and retention learning rates and the student's learning style.

#### **4.4. The Domain Module**

According to Beck et al. [BECK96] the domain module essentially stores the knowledge that the student will be tutored on. Chappell [CHAP95] states that this information usually consists of a combination of procedural, declarative and heuristic knowledge. For example, the system developed by Woolf [WOOL87] consisted of procedural knowledge describing how problems can be solved, declarative knowledge describing concepts in the domain and heuristic knowledge which describes actions taken by the expert.

From the discussion presented in section 3 it is evident that the knowledge required by an intelligent programming tutor includes :

- Procedural knowledge on how to write programs.
- Knowledge describing errors commonly made by novice programmers.
- Declarative knowledge on the different programming concepts.

The expert module will use the knowledge contained in the domain module when solving problems. The declarative knowledge contained in this module will also be used by the pedagogical module when tutoring the student on a particular concept.

#### **4.5. The Expert Module**

The functions of the expert module are to solve the problems presented to the student and also debug and assess student programs for semantic errors and programming efficiency.

According to Anderson et al. [ANDE86] problem solutions can either be generated prior to the use of the tutor or during run-time. Factors that need to be taken into consideration when deciding when the problem solving process should take place include processing power and the availability of storage space. Generating prior solutions to problems will reduce the processing power needed at run-time but will require more disk space.

The intelligent programming tutors discussed in the previous section store the solutions to the programming problems presented to the learner. These solutions are usually developed by the instructor and stored in a knowledge base. In order to reduce the development costs associated with building IPTs it was decided that expert module in the generic architecture should automatically generate solutions to programming problems from a problem specification. This module needs to cater for the derivation of solutions to the following types of programming problems:

- Problems that help students develop their debugging skills. For example, presenting the student with a program consisting of semantic errors which the student must rectify.
- Problems that test students' comprehension of programs and the different control structures. For example, presenting a student with a program that he or she has to perform a trace of and predict the output. Dale [DALE98] describes program comprehension as an essential programming skill. Furthermore, Zeller [ZELL00] states that students need practice in assessing programs written by others for both correctness and efficiency of design. An IPT's repository of programming problems must include problems that test program comprehension and the expert module needs to generate solutions to such problems.
- The derivation of solution algorithms to programming problems. Each programming problem will be described as a set of requirements that must be met by the solution program. The expert module must be capable of generating both procedural and object-oriented solution programs. In order to ensure the reusability of this module solution algorithms will be generated in an internal representation language. The induced algorithm will then be translated to the language syntax specified in code specification module<sup>1</sup>.
- The importance of program design is highlighted in section 3.9.2. Thus, the expert module must generate the program design for each of the solution programs induced. Depending on the design methodology used, in the case of procedural programming problems this could involve merely converting the solution algorithm derived in the internal representation language to the notation of the design methodology. In the case of object-oriented programs a separate design must be developed. Each object-oriented design will form input to the process of inducing a solution program for the corresponding problem.
- Problems that test students knowledge of the object-oriented programming paradigm by requiring students to develop solution algorithms that require the use of existing classes. The research presented by Singley et al. [SING91] and Chee et al. [CHEE97] clearly indicates that in learning the object-oriented programming paradigm students must be able to solve a programming problem by using existing classes. Thus, given such a task the expert module must be capable of choosing and adapting suitable components for reuse.

The expert module will receive input from the domain, code specification and problems modules and will communicate with the explanations module in order to generate explanations to present to the student.

---

<sup>1</sup>The code specifications module is described in section 4.8.

#### **4.6. The Instructional Strategies Module**

As mentioned in section 4.3 both meta-strategies and instructional strategies are required in order to tutor a student. While meta-strategies concentrate on the actual tutoring process, instructional strategies are concerned with the presentation of a particular concept, e.g. polymorphism. In most ITS architectures this module forms part of the pedagogical module.

However, in order to allow for the reuse of the proposed generic architecture for other programming paradigms and problem solving domains in the future, it was decided that the instructional methods should be represented by a separate module.

According to Hsieh et al. [HSIE96] it is essential that an intelligent tutoring system has multiple teaching strategies. The importance of providing multiple teaching strategies, especially in the programming domain, is stressed by Woods et al. [WOOD96] who state that there is no best way to teach programming, but that a number of teaching strategies are needed to provide for effective learning. Thus, this module will contain a number of different strategies for each programming concept.

The pedagogical module will choose the most appropriate instructional strategy for the particular student according to input from the student module. Studies will be conducted to determine the relationship between the student's characteristics and preference of instructional strategy.

#### **4.7. The Problems Module**

The problems presented to the student will be represented in the problems module. The representation of each problem will include an indication of the level of difficulty of the problem.

The possibility of automatically generating programming problems from general rules built into this module must be investigated. Murray [MURR99] describes expert systems as being suitable for this purpose. The automatic problem generator developed by Kumar [KUMA00] requires the general description of problem types to be specified by a template which is then used to generate problems in a particular domain. These methodologies need to be examined.

#### **4.8. Code Specification Module**

If the student communicates with the intelligent programming tutor via a textual interface the student will enter his/her algorithms using a specific programming language or pseudo-code. Declarative knowledge describing the language or pseudo-code used will be represented in the code specification module. Similarly, if the student uses an iconic or graphical user interface the intelligent programming tutor will need to know the syntax of the concepts underlying the graphics or icons used and this information needs to be specified in this module.

The code specification module provides input to the expert and explanations module.

#### **4.9. The Explanation Module**

According to Jerinic [JERI97] it is essential that ITSs provide detailed explanations regarding errors made by the user. In the PROUST system plan-difference rules are used to generate bug descriptions.



An English description for each bug description is stored in the knowledge base and forms part of the explanation produced. The Lisp tutor stores template mappings corresponding to each buggy production rule.

In addition to providing an explanation of the errors made by students, the intelligent programming tutor also needs to generate explanations in the form of hints and advice.

The SIPLoS system [CHEE97] system contains a component called the Advisor that provides explanations. MoleHill [ALPE95] provides advice in the form of textual and graphic demonstrations with verbal commentary.

The explanation module will obtain input from the expert module and generate explanations based on this input. These explanations will be presented to the student via the interface module.

#### **4.10. Learning / Experience Module**

An ITS must be able to improve its performance over time. Based on the tutors interactions with the student this module will:

- Improve and add to the collection of problem solving techniques in the domain module.
- Identify frequently made errors that are not specified in the domain module and update the domain module accordingly.
- Monitor the relationships between a students' learning style and cognitive style and the student's preferences of meta-strategies, instructional strategies and programming interfaces. Both the student module and the pedagogical module will be updated accordingly.

According to Beck et al. [BECK99] artificial intelligence techniques can be incorporated into the design of ITSs in order to facilitate self-improvement. Suitable machine learning techniques will be identified or derived for this purpose.

#### **4.11. Summary of the Interaction Between Modules**

**Figure 2.4.1** depicts the interaction between modules in the proposed generic architecture.

The interface module obtains input from the student and communicates this information to the student and the pedagogical modules. Both the student and the pedagogical modules interact with the student via the interface module. The explanation module provides the student with explanations of concepts and errors via the interface module.

Based on the decisions made by the pedagogical module and the behavior of the student the learning / experience module updates the student, pedagogical and domain module accordingly if necessary.

The pedagogical module receives input from student, domain, instructional strategies and problems module and uses this knowledge to tutor the student via the interface module.

The domain module provides input to the expert module. The expert module uses this information together with information from the problems and code specifications modules to generate solutions to the problems presented to students.

This information is also used in conjunction with input from the interface module to assess the student's attempt at a solution. Output from this process is then communicated to the explanations module which generates the necessary explanations to provide feedback to the student via the interface module.

The code specification module describes the medium via which the student enters algorithmic solutions to problems, e.g. a particular programming language. Thus, this module provides input to both the expert and the explanations modules.

The following section examines the automatic generation of novice solution algorithms.

## **5. The Automatic Derivation of Solution Algorithms**

One of the functions of the expert module in the generic architecture is the generation of solutions to programming problems requiring a computer program or algorithm as a solution. This section looks at why genetic programming was considered as an option for this purpose.

Like genetic algorithms, genetic programming is an evolutionary algorithm, that takes a problem specification as input and produces a function or algorithm as output. The foundations of genetic programming lie in Darwin's theory of evolution and each genetic programming run is composed of a number of generations. It is assumed that the fitness of the population will improve from one generation to the next resulting in the algorithm converging to a solution.

The reasons as to why genetic programming has been hypothesised as a suitable means of inducing novice solution algorithms are listed below:

- The genetic programming algorithm is domain-independent. Furthermore, according to Banzhaf et al. [BANZ98] and Koza [KOZA99] genetic programming has been successfully applied to problems in various problem domains.
- Genetic programming is fault tolerant and can deal with both inconsistent and incomplete information.
- Genetic programming can induce solutions to problems from a high-level specification of the requirements of the problem. [KOZA99].
- According to Koza [KOZA99] genetic programming makes provision for internal store by means of named memory, indexed memory, state memory and relational memory. Internal storage is needed for storing variable values and data structures declared in computer programs.
- Experiments conducted by Koza [KOZA92] (and [KOZA99]) have revealed that genetic programming is capable of successfully implementing conditional, iterative and recursive control structures. These low-level structures are essential components of procedural and object-oriented programs.
- Genetic programming facilitates the generation of modular solutions. Koza [KOZA94] describes how automatically defined functions (ADFs) can be used by a GP system to cater for modularization. The encapsulation and the compression operators defined by Koza [KOZA92] can also be used for this purpose.
- Research conducted by Langdon [LANG98b] indicates that genetic programming can generate data structure algorithms and use the evolved abstract data type in the program induction of other solution algorithms.

- Studies conducted by Bruce [BRUC95] and Langdon [LANG98b] are indicative of the fact that genetic programming can successfully generate object-oriented solution algorithms.
- In addition to producing correct solution algorithms, genetic programming can also produce efficient programs. This is illustrated in [KOZA92] and [LANG98b].
- Based on the discussion in section 3.9.2 more than one solution algorithm, each taking a different approach to solving the problem, must be generated. Genetic programming systems are capable of generating multiple solutions, each taking a different problem solving approach to the problem.
- From Koza's evaluation of genetic programming [KOZA99] it is evident that in some domains the solutions produced by genetic programming are competitive with those produced by humans.

## 6. Summary

Intelligent programming tutors have proven to be an effective and economically viable means of assisting novice programmers overcome learning difficulties. However, the developmental costs associated with building intelligent programming tutors are high. As a solution to this problem, this chapter has proposed a generic architecture for the development of intelligent programming tutors for the procedural and object-oriented programming paradigms. A function of the expert module of the generic architecture is the automatic generation of solutions to problems presented to the student. The study presented in the thesis focuses on the derivation of solution algorithms to novice procedural programming problems. The thesis investigates the use of genetic programming as a means of inducing novice procedural solution algorithms. The next chapter provides an overview of genetic programming.

## Chapter 3 - An Overview of Genetic Programming

### 1. Introduction

The concept of genetic programming (GP) was firstly suggested by Cramer in 1985, and then formalized by Koza in 1992. The foundations of genetic programming lie in genetic algorithms. Hence, like genetic algorithms, genetic programming is also based on Darwin's theory of the survival of the fittest.

Whereas most search techniques move from one single point to another in the search space, genetic programming moves from one population of points to another [KOZA99]. Banzhaf et al.[BANZ98] describe genetic programming as a supervised machine learning system which implements a beam search, where the population size represents the size of the beam, and the fitness function is used to determine which candidate solutions are kept in the beam and which are not. The genetic programming algorithm is essentially a stochastic search method that is initiated by randomly generating an **initial population** of potential program solutions. Each program is represented as a parse tree instead of a binary string or chromosome as in the case of genetic algorithms. The population is evaluated on a set of **fitness cases** specified as part of the program specification. The evaluation process involves applying a **fitness function** to the output generated by an individual in order to calculate a **fitness measure** for the individual. A **selection method** is then applied to identify individuals in the population, based on the program's fitness measure, that will be used to create the next generation. The selection criteria is not elitist. This together with the fact that genetic programming does not perform a point-to-point search ensures that genetic programming avoids greedy hill climbing [KOZA99]. **Genetic operators** are applied to the chosen individuals to create the next generation. The process of evaluation, selection and recreation is continued from one generation to the next until the specified **termination criteria** is met.

Koza [KOZA92] emphasizes the generality of genetic programming by successfully applying it to a number of different types of problems. He states that genetic programming provides a natural representation of the different problem domains. Furthermore, genetic programming does not require preprocessing although it may require some postprocessing depending on the problem domain.

Genetic programming requires very little foreknowledge regarding the shape and size of the problem and its output is an algorithm that can be executed as a computer program. Experiments conducted by Koza [KOZA92] have revealed that genetic programming can still find solutions in cases where the information provided to the system is inaccurate, inconsistent or incomplete. Furthermore, Koza [KOZA92] has found genetic programming to be fault tolerant. However, genetic programming is computationally intensive and requires large amounts of processing time and memory.

The chapter provides an overview of the field of genetic programming. The following section describes the standard genetic programming system introduced by Koza [KOZA92]. Section 3 examines extensions that have been made to the standard genetic programming system since its inception. A discussion of the limitations of genetic programming is presented in section 4. Section 5 reviews a study conducted to design a genetic programming system for the evolution of procedural programs.

## 2. The Standard Genetic Programming System

This section provides a description of the standard genetic programming system. **Section 2.1** presents the genetic programming algorithm. The genetic programming algorithm can be implemented using one of three control models. These control models are discussed in **Section 2.2**.

Each individual in a genetic programming population is composed of terminals and functions. These concepts are examined in **Section 2.3**. These terminals and functions are combined to form a particular structure which represents a computer program.

The most common representation of a computer program in a genetic programming population is a parse tree. However, other representations including linear genomes, and directed acyclic graphs have been used for this purpose. The different structures that have been used to represent genotypes in genetic programming systems are outlined in **Section 2.4**.

The first step performed by a genetic programming system is the generation of an initial population of computer programs. **Section 2.5** discusses three different methods that can be used to generate the initial population.

During each generation of a genetic programming run the population is evaluated on a training set, referred to as a set of fitness cases, using a fitness function specified by the user. The effects of different fitness case sets and an account of the fitness functions commonly used in GP systems is provided in **Section 2.6**.

Once the individuals in a population are evaluated, one of the selection methods described in **Section 2.7** are implemented to select those individuals which will be used to create the next generation. A combination of two or more of the genetic operators outlined in **Section 2.8** are applied to the chosen individuals to create the population of the new generation.

The genetic programming algorithm terminates when the termination criteria specified by the user is met. At this point the designated result of the GP run is reported. These concepts are discussed in **Section 2.9**.

Finally, **Section 2.10** describes the steps that must be performed by a user in order to implement a genetic programming system.

### 2.1 The Genetic Programming Algorithm

Koza [KOZA92], and Banzhaf et al. [BANZ98] present the following standard genetic programming algorithm:

- Create an initial population: This population is generated by constructing parse trees by randomly choosing elements from the function set and terminal set.
- Repeat
  - Evaluate the population using the fitness function.
  - Create a new population by applying genetic operators to individuals selected according to their fitness.

Until the termination criterion has been met.

- The individual representing the best individual in a generation is returned as the result of the generation.

A GP algorithm can implement one of three types of control models. The two commonly used models are the generational model and the steady-state model. In a generational control model a generation is one iteration of the repeat loop. However, in the steady-state control model a generation consists of those steps executed from one evaluation of the population to the next. More details regarding control models are outlined in **Section 2.2**.

A genetic programming run involves creating an initial population, implementing a specified number of generations and designating the result of a run. A different random number generator seed is usually used for each run. More than one run is performed for a problem due to the randomness associated with genetic programming.

## **2.2 Control Models**

Bruce [BRUC95] describes essentially three types of control models that can be employed by a GP system: generational, steady-state, and varying population-size control. The first two control models are most frequently used by GP systems.

### **2.2.1 The Generational Control Model**

This is the control model that is traditionally used by GP systems. There are a distinct number of generations performed by the algorithm. In each generation there is a complete population of individuals. The size of the population remains constant throughout a run of the genetic programming system. The new population is created from the old population which it replaces. Two population arrays, one for the current population and one for the new population, are maintained during each generation. According to Bruce [BRUC95] the advantage of this method is that it is easy to implement. Banzhaf et al. [BANZ98] present the following algorithm for creating the new population at each stage of the GP algorithm (see **Section 2.1**) using the generational control model:

Repeat

- Choose individuals or an individual using a selection algorithm.
- Apply the genetic operators to the selected individual or individuals.
- The offspring are members of the new population.

Until a maximum number of individuals for the new population are created.

### **2.2.2 The Steady-state Control Model**

This model does not implement a fixed number of generations. According to Bruce [BRUC95] a single population of a fixed size is maintained during a genetic programming run. A single array is used to store individuals of the population. Newly created individuals replace the individuals with poor fitness in the population. The newly created individuals can immediately be chosen as parents to create new offspring. Inverse selection methods are used for choosing which members of the population to replace. Banzhaf et al. [BANZ98] describe a generation in this model as those steps performed from one evaluation of the population to the next.

According to Manjunath et al. [MANJ97] the amount of memory used by the genetic programming system can be reduced by using the steady-state control model rather than the generational control model. Kent [KENT97] describes steady-state GP as being less susceptible to premature convergence. He states that in addition to this the use of a steady-state model produces solutions of a better quality. Furthermore, individuals with a poorer fitness are less likely to be passed onto the next generation when using the steady state approach than when a generational control model is implemented [RYAN98]. A summary of the overall algorithm is listed in **Figure 3.2.2.1**.

- |   |
|---|
| <ol style="list-style-type: none"> <li>1. Create an initial population</li> <li>2. Evaluate the population</li> <li>3. Select the individuals to apply the genetic operators to. These are called winners.</li> <li>4. Use inverse selection methods to select members of the population to replace. These are referred to as losers.</li> <li>5. Replace the losers in the population with the offspring of the winners.</li> <li>6. Repeats steps 1 to 6 until the termination criteria are met.</li> </ol> |
|---|

**Figure 3.2.2.1:** The Steady-State Control Model

### 2.2.3 Varying Population-Size Control

In this case a single population is maintained, the size of which changes throughout a GP run. Bruce [BRUC95] describes GAVaPS as one of the many algorithms that implement this model. This algorithm enables the population size to grow and shrink according to the state of the search.

If the population does not contain many fit individuals the population grows. If the neighborhood of an optimal solution is found the population size is decreased. In the neighbourhood of a local optimum the size of the population is increased.

## 2.3 Functions and Terminals

Each potential solution in a GP population is represented as a parse tree. The nodes that the parse tree is comprised of are called genes [GRAN00].

Each parse tree is composed of elements from the function set  $F = \{f_1, f_2, \dots, f_n\}$ , where  $n$  is the number of functions, and elements of the terminal set  $T = \{t_1, t_2, \dots, t_m\}$ , where  $m$  is the number of terminals. These are collectively referred to as primitives and form a representation language in which programs are expressed. Each element of the function set takes a specified number of arguments. We refer to this as the arity of the function. The set  $A = \{a_1, a_2, \dots, a_n\}$  consists of the corresponding arity for each member of the function set. According to Langdon [LANG98b] the choice of the correct function and terminal set elements is crucial to the success of the genetic programming system.

Banzaf et al. [BANZ98] describe self-modification primitives that can be included in the function set. These functions evolve together with the solution, thus allowing the architecture of individuals to be changed during a run.

### 2.3.1 Functions

The following examples of elements that are contained in function sets are listed by Koza [KOZA92] and Banzhaf [BANZ98]:

- Arithmetic functions: +, -, \*.
- Conditional operators: If-then-else.
- Variable assignment functions : ASSIGN.
- Loop statements: WHILE...DO; REPEAT...UNTIL, FOR...DO.
- Depending on each problem certain domain specific functions may be defined.

The function set can include both low level and high level operators. For example, the function set could contain the multiplication operator to enable the GP system to generate an algorithm to calculate the cube of a number. The multiplication operator is a low level operator. Alternatively, the function set could contain a cube operator which takes a single number as an argument and returns the cube of the argument. The cube operator is a high-level operator. According to Grant [GRAN00] and Banzhaf et al. [BANZ98] the choice of low level and high level operators has a large impact on whether the GP system will be successful or not. Thus, these should be chosen with care. The choice of elements to include in a function set is problem dependant.

### 2.3.2 Terminals

Elements of a terminal set include:

- Variables representing the input to functions, or state variables of a system
- Single constants.
- Banzhaf et al. [BANZ98] describe a set of real-numbered constants which can be chosen for the entire population. These are called random ephemeral constants and remain constant throughout a genetic programming run. The ephemeral constant is represented by the symbol  $\mathfrak{R}$  in the terminal set. An ephemeral constant has a specific range, e.g. integers between 1 and 10. If the ephemeral constant is chosen as a node when creating a tree, a value in the specified range is randomly chosen.
- Functions with an arity of zero that have certain side effects.

Banzhaf et al. [BANZ98] suggest that a parsimonious approach should be taken when choosing constants. It is not necessary to include all the constants needed as a GP system can generate the necessary constants from the subset provided.

### 2.3.3 Closure and Sufficiency

According to Koza [KOZA92], for each problem the function set and the terminal sets chosen must satisfy the closure property and the sufficiency property.

#### 2.3.3.1 The Closure Property

A function set satisfies the closure property if each function in the set accepts any value that the elements in the function set may return, as well as any value that each member of the terminal set may take on as input.



In some problems, in order to satisfy the closure property some functions may be defined as protected functions. Langdon [LANG98b] defines protected functions as those which are able to deal with illegal arguments and return valid computations. For example, the division operator cannot accept a denominator of zero. Similarly, the square root operator cannot accept negative values. In both these cases the operators are defined as protected functions. These protected functions either return a value of one or an error message if an unacceptable input value is passed to them.

### **2.3.3.2 The Sufficiency Property**

The sufficiency property is satisfied if the solution to the problem can be expressed in terms of the members of the function set and the terminal set.

However, in some problems it may be impossible to identify the variables needed to solve the problem, e.g. predicting the interest rate, predicting the results of an election. **Section 2.3.5** describes a means of dealing with these cases.

### **2.3.4 Extraneous Functions and Terminals**

In some cases the genetic programming algorithm may find a solution to the problem without using all the functions specified in the function set. In this case the functions that are not used are referred to as extraneous functions. The run-time of the GP system is increased in the presence of extraneous primitives [LANG98b]. Similarly, a terminal set may contain extraneous variables or constants. According to Koza [KOZA92] and Grant [GRAN00] system performance will be degraded if the function and terminal sets contain extraneous primitives.

Furthermore, Banzhaf et al. [BANZ98] and Bruce [BRUC95] state that it is important that too large a function set is not chosen as this will increase the search space and hence the time the search takes. This is further emphasized by Kent [KENT97] who states that the search space grows exponentially with an increase in the number of primitives specified in the function and terminal sets. Thus, only those primitives needed to find a solution should be specified in these sets.

In studies conducted by Bruce [BRUC95] initially the solution language contained only those primitives that were necessary to find a solution. This was done in order to reduce the search space of programs. However, it was found that this may reduce the genetic diversity of the population and cause the premature convergence of the genetic programming algorithm. Thus, although the sizes of the function and terminal sets must be kept to a minimum, the content of each set must be sufficient to promote genetic diversity<sup>2</sup>.

### **2.3.5 The Automatic Evolution of the Function and Terminal Sets**

In some problems it may not be possible to specify the exact function and terminals sets that will be needed to solve the problem. Koza [KOZA94] describes how the set of primitive functions can be evolved together with the solution.

---

<sup>2</sup>The concept of genetic diversity is discussed in more detail in section 4.2.1.1.

Primitive functions (PF) are evolved in the same way that automatically defined functions<sup>3</sup> (ADFs) are evolved. Koza [KOZA94] starts the process with at least one primitive function in the overall program. The initial primitive function or set of primitive functions is then used to define the automatically defined functions. Each program will consist of a branch representing the main program and one or more branches representing each primitive to be evolved. Both the primitive functions specified and the ADFs representing the PFs to be evolved are used in the branch of the tree representing the main program.

The function set of the main program branch will consist of  $n$  primitive functions and  $m$  ADFs, where  $n$ ,  $m$ , and the ADFs will be evolved as part of the GP process. The function set for each function-defining branch will consist of  $n$  primitives and the ADFs that it can hierarchically reference.

The method for generating an initial population is adapted so that the number of primitive function branches that an individual contains is randomly chosen. The random number generated is greater than or equal to one and less than or equal to a specified limit. A random number is also generated to determine the number of arguments that a primitive-defining branch can take.

Koza [KOZA94] describes the process of simultaneously evolving the terminal set together with the solution to the problem. This essentially involves selecting the necessary terminals from a “sufficient” superset of terminals in a manner similar to that of automatically extending the function set described above.

## 2.4 Program representation

In most genetic programming systems each program is represented as a parse tree. Other program representations include linear genomes and graphs. According to Banzhaf et al. [BANZ98] the linear structure has the fastest execution time. This section describes commonly used program representations.

### 2.4.1 Tree Representation

The most common representation of computer programs in genetic programming systems is a parse tree [LANG98b]. Each individual in the population is represented as a parse tree as illustrated in Figure 3.2.4.1.1.

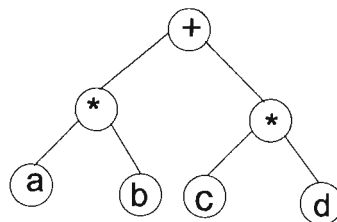


Figure 3.2.4.1.1

---

<sup>3</sup>Automatically defined functions (ADFs) are described in Section 3.2.1.

The program is executed by firstly performing a prefix or postfix traversal of the tree. The results of the traversal are then interpreted by an interpreter built into the system or compiled and executed. A prefix traversal has an advantage over a postfix traversal in that the execution time can be reduced when executing certain functions such as a conditional statement, e.g. an if-else statement.

### 2.4.2 Linear Representation

Banzhaf et al. [BANZ98] describe a linear structure as one consisting of set of instructions that can be executed in a top to bottom or left to right manner. Linear representations are used in GP systems evolving solutions in machine code.

This representation is used in the system AIMGP (Automatic Induction of Machine Code with Genetic Programming ) implemented by Banzhaf et al. [BANZ98]. In this representation registers are used to store global memory values when the sequence of instructions is executed. The advantage of this representation is a decrease in the runtime of the GP system during solution induction [LANG98b].

### 2.4.3 Graphical Representation

The PADO (Parallel Algorithm Discovery and Orchestration) system was developed by Teller et al. [TELL96a] for the purpose of signal processing. PADO uses genetic programming to induce algorithms to recognise objects that fall into one of a number of different classes. Each individual consists of a main program which is represented as a directed graph. Mechanisms are built into the PADO system to control the execution of a program.

Each main program has access to its own set of Mini programs. These Mini programs are equivalent to the automatically defined functions used by Koza [KOZA94]. The PADO system maintains a set of library programs which are accessible by all programs induced. During each generation the library programs are updated according to how many programs access them and how often they are used. The fitness value of the programs which call the libraries are also taken into consideration when updating these library programs.

Each node in the graph representing a main program has an action component and a branch-decision component. Each program has its own private stack and indexed memory<sup>4</sup>. The stack is used by the action components of the graph to pop their inputs off the stack and push their outputs onto the stack. The main program contains a number of different nodes, including a start node, a stop node, and nodes representing mini and library programs.

According to Banzhaf et al. [BANZ98] graphs can be used to store complex programs “compactly”. Furthermore, recursion and iteration can be easily represented in a graph structure compared to a tree or linear representation.

---

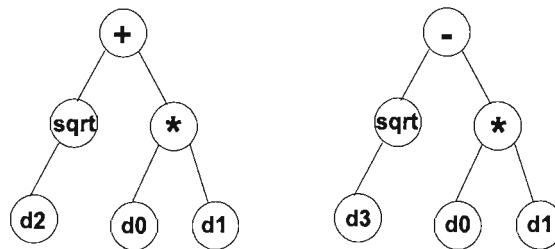
<sup>4</sup>The concept of indexed memory will be discussed in more detail in **Section 3.1.1.1**.

#### 2.4.4 Representing a GP Population Using a Directed Acyclic Graph

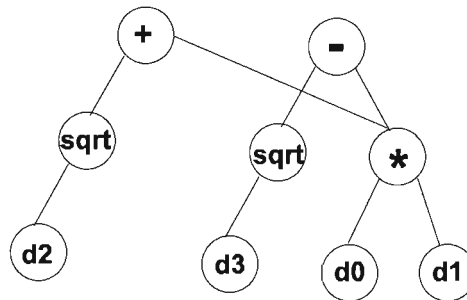
In the system implemented by Ehrenburg [EHRE96] the population is represented as a single directed acyclic graph(DAG) instead of a forest of trees as in conventional GP systems. Each individual program tree is rooted at a node in the DAG.

Example:

A forest of two trees:



The corresponding DAG:



Erhenburg [EHRE96] uses a method called creation-evaluation to evaluate the population with respect to fitness. According to Erhenburg [EHRE96] this representation reduces redundancy in a population. Furthermore, Langdon [LANG98b] states that the use of directed acyclic graphs reduces both the memory requirements and the runtime needed to find a solution.

#### 2.4.5 Stack-based Genetic Programming

In the study conducted by Perkis [PERK94] each program is represented as an expression in Reverse Polish Notation. A stack-based interpreter is needed to evaluate/execute each program stored in this form. The interpreter operates as follows. The program parameters are pushed onto the stack. The interpreter then pops values off the stack as needed. If a function node is popped off the stack a number of values equal to the number arguments of that function takes ( i.e. the arity of the function) are popped off the stack.

The result of applying this function to these arguments is then pushed back onto the stack. If a variable or constant is popped off the stack, its value is located and pushed onto the stack. If a function needs more arguments to be popped off the stack then there are values on the stack, the function is treated as a null operation.

Research conducted by Koza [BRUC95] has revealed that if the same function sets are used for both the tree and stack representations there is no change in the performance of the system. However, if stack operations are added to the function set fewer evaluations were needed to find a solution using the stack representation.

#### **2.4.6 Multi-Tree Representation**

Bruce [BRUC95] and Langdon [LANG98b] use a multi-tree representation for each program in the genetic programming population. Each program is required to perform more than one function, thus each chromosome contains multiple trees, one representing each function that the program must perform. Each tree is referred to as a gene.

All the genes in the chromosome usually access the same instance of indexed memory<sup>5</sup>. Each fitness case is comprised of a sequence of function calls. The corresponding tree in the genome is evaluated for each function call.

According to Langdon [LANG98b] this representation facilitates the creation of building blocks. The procedures for initial population generation, crossover and mutation need to be adapted to cater for this representation of individuals. The crossover and mutation operators are applied to a single tree, i.e. gene, in the chromosome. Crossover is performed between like trees, i.e. trees which represent the same function. Crossover produces a single offspring. The reproduction operator essentially makes a copy of each tree in the chromosome.

The mutation operator firstly makes a copy of the chromosome. A tree in the genome is randomly selected. A mutation point is then randomly selected. The mutation operator generates a node (and corresponding subtree) and replaces the old node (and corresponding subtree) with it [BRUC95].

#### **2.5 Initial Population Generation**

The initial population essentially consists of parse trees representing each potential solution program. These parse trees are created by randomly choosing elements from a combination of the function set and the terminal set. Koza [KOZA92] suggests that a uniform random distribution be used to choose elements from the terminal and function sets.

One of the parameters<sup>6</sup> of the GP system is the maximum tree size. This maximum is either the maximum depth of the tree or the maximum number of nodes a tree can be composed of. Banzhaf et al. [BANZ98] define the depth of a node as the number of nodes between the particular node and the root of the tree. Langdon [LANG98b] states that the performance of the GP system will be improved by including shorter trees in the initial population.

In the system implemented by Koza [KOZA92] a function node is firstly randomly selected from the function set. This ensures that a hierarchical structure is generated and not a trivial tree. If a terminal is chosen for a point of the tree, this node becomes the endpoint of the tree. In this case the node is not generated any further. Each function node is generated further recursively in a left to right manner until the maximum depth is reached.

---

<sup>5</sup>The concept of indexed memory will be discussed in more detail in **Section 3.1.1.1**.

<sup>6</sup>The parameters of a genetic programming system are described in **Section 2.10**.

Koza [KOZA92] defines three different methods that can be used to generate the initial population: the full method, the grow method and the ramped half-and-half method.

### **2.5.1 The Full Method**

This method ensures that the length of every path between an endpoint and the root of each tree is equal to the specified maximum depth. Labels for nodes at a depth less than the maximum depth are chosen from the function set at random. Labels for nodes at a depth equal to the maximum depth are chosen from the terminal set.

### **2.5.2 The Grow Method**

Trees generated using this method are of variable shape. In each individual the length of a path between the root and an endpoint is less than or equal to the maximum depth.

Labels for nodes at a level less than the maximum depth are randomly chosen from a combination of the terminal and function sets. Labels for the nodes at a level equal to the maximum depth are randomly chosen from the terminal set.

### **2.5.3 Ramped Half-and-Half**

This method produces trees of various shapes and sizes. It combines the full and grow methods. An equal number of trees of each depth value in the range of two to the maximum depth is generated.

Half the trees for each depth are created using the grow method and the other half using the full method. Koza [KOZA92] presents the following example to clarify the implementation of this method: If the maximum depth is six then twenty percent of the trees will have a maximum depth of two, twenty percent a maximum depth of three, twenty percent a maximum depth of four, twenty percent a maximum depth of five, and twenty percent a maximum depth of six. For each depth value fifty percent of the trees will be generated using the full method and fifty percent using the grow method.

According to Koza [KOZA92] the ramped half-and-half method has proven to perform the best over a wide spectrum of problems. Banzhaf et al. [BANZ98] state that the ramped half-and-half method can be used to promote genetic diversity in the population while the grow and full methods could result in a population of uniform individuals being produced. It is important that genetic diversity is maintained in the population to prevent the premature convergence of the algorithm.

### **2.5.4 Seeding and/or Biasing the Initial Population**

Studies conducted by Bruce [BRUC95], Grant [GRAN00] and Soule [SOUL98] et al. have indicated that in some domains it is beneficial to seed the initial population with either programs created by hand or previously simulated. In the Odin system developed by Holmes [HOLM95] the programs used to seed the initial population were developed by the user.

Alternatively, a genetic programming system can be used to generate the individuals with which to seed the initial population of another genetic programming run. This approach is referred to as population enrichment.

The implementation of this method involves firstly making a number of genetic programming runs in which the initial population is created randomly. The best individual found in each run is saved and used as a seed value for the final run of the genetic programming system. This method is suitable for problem domains in which there are a large number of primitives.

Whigham [WHIG96] proposes a method for biasing the initial population with respect to the syntax and typing of individuals. Whigham [WHIG96] states that according to the “no free lunch” theorem a system will perform poorly when applied to certain problems. Thus, a system needs to be tailored to the needs of a specific problem. According to Whigham [WHIG96] this can be achieved by means of a bias. In the system implemented by Whigham [WHIG96] a context-free grammar is used to generate the initial population and serves the purpose of a bias. This system learns the syntax and typing of computer programs from a grammar. In this way the search space is narrowed thereby increasing the probability of finding a solution.

### **2.5.5 Variety**

Koza [KOZA92] describes the existence of duplicate individuals in an initial population as being unproductive and a waste of computational resources. Furthermore, duplicates in an initial population reduce the genetic diversity of the population. Thus, Koza [KOZA92] suggests that duplicate trees must not be included in the initial population.

Koza [KOZA92] defines the variety of the population to be the percentage of individuals in the population for which there are no duplicates. The higher the variety of the initial population, the greater the chance of the population containing the components necessary to find a solution. The variety of the initial population must be a hundred percent. In later generations duplicate individuals could possibly result due to the application of the genetic operators and cannot be avoided. Thus, Langdon [LANG98a] states that the developer needs to keep a close watch on the GP system as it evolves solutions to ensure that the genetic diversity is maintained and that the primitives that are essential to solve the problem remain in the population. Mechanisms must be built into the system to keep track of the frequency of primitives and the population variety. The population variety must be at least ninety percent throughout a run. The concept of variety is discussed in more detail in section 4.2.

## **2.6 Evaluation**

In order to create the next generation each individual in a population is assigned a scalar value referred to as a fitness measure. Genetic operators are then applied to the fitter individuals of the population. This fitness measure is calculated by executing the individual on the input values in the fitness cases specified by the user and applying a fitness function to the output of the individual and target output in the fitness cases. Section 2.6.1 discusses the different methods employed for choosing and using fitness cases while section 2.6.2 provides an overview of commonly used fitness functions.

### **2.6.1 Fitness Cases**

Fitness cases are input-output pairs describing the target output of a program given the input value specified in the fitness case. An individual's fitness is usually calculated by executing the program with the input values specified in each of the fitness cases and comparing the output of program with the target value of the corresponding fitness case.

The fitness measure of a program is a function of the differences between the target value and program output for each fitness case.

An entire problem domain is usually very large or infinite. Fitness cases are a representational sample of the entire problem domain. In small domains, e.g. the domain of Boolean functions it is feasible to use all possible input combinations. However, in cases where the number of fitness cases is large or infinite a sample of the fitness cases must be taken. Statistical methods can be employed to effectively select fitness cases.

According to Banzhaf et al. [BANZ98] if the training set is too small then it will not be reliable and solutions generated will not be able to generalize. Once a solution is generated the GP system can test how well the solution generalizes. This can be achieved by dividing the training set into two sets, one set is used to obtain a best solution. Once the best solution is obtained, it is then tested on the second set of fitness cases. According to Koza [KOZA92] if the set of fitness cases contains irrelevant cases this will degrade system performance.

The set of fitness cases can be fixed, i.e. the same fitness cases are used for each generation. Alternatively, the fitness cases can be dynamic. According to Grant [GRAN00] this involves using a different set of fitness cases for each generation. One of the main reasons for using dynamic fitness cases is to reduce the time taken for the purposes of evaluation of a genetic programming population when a large number of fitness cases are needed to represent the problem domain sufficiently.

A number of methods exist for implementing the use of dynamic fitness cases:

- Stochastic Sampling

Banzhaf et al. [BANZ98] propose stochastic sampling as a means of reducing the time required for evaluating individuals with respect to fitness. Stochastic sampling involves testing each individual on a different set of fitness cases. According to Banzhaf et al. [BANZ98] solutions produced by this method have been found to generalize more easily.

- Dynamic Subset Selection(DSS)

Both Banzhaf et al. [BANZ98] and Gathercole [GATH98] describe dynamic subset selection (DSS) as follows. A run of the GP system is firstly performed. During this run the evaluation of the individuals on the different fitness cases are observed and the fitness cases are categorized as being difficult or easy. The more difficult fitness cases are then selected and used in the next run. This method has been found to accelerate learning.

- Limited Error Fitness (LEF)

According to Banzhaf et al. [BANZ98] and Gathercole [GATH98] in this case the system keeps track of the cumulative error of an individual during the process of evaluation for each fitness case. If this cumulative error is found to exceed a certain threshold at any stage the process is aborted and the individual is not evaluated on any of the other fitness cases.



- Rational Allocation of Trails (RAT)

This method involves allocating a number of fitness cases to each individual [GATH98]. Fitness cases are allocated to individuals based on the expected significance of the new information that will be provided by evaluating the individual on another fitness case.

- Banzhaf et al. [BANZ98] state that the efficiency of a genetic programming system containing a large set of fitness cases can be improved by dividing the set into a number of smaller sets or chunks. A solution will be formed by concatenating the solutions obtained using each subset of fitness cases.

In some instances the solutions developed by genetic programming systems have been criticized for being brittle [MOOR97]. Moore et al. [MOOR97] attribute this to the poor selection of training sets. Banzhaf et al. [BANZ98] state that in order to reduce the effect of using a particular set of fitness cases, different fitness cases may be chosen for each generation in a run. Furthermore, Moore et al. [MOOR97] suggest that the use of a new set of randomly generated fitness cases for each generation helps reduce the brittleness of the solution generated by a GP system.

Bruce [BRUC95] describes the cross-validation method which can be used to generate programs that are not brittle. This method essentially involves splitting the training set into  $n$  partitions. A number of partitions, usually  $n-1$ , are used to train the population while the  $n$ th partition is used to test the program obtained. The process of partitioning, training and testing is repeated until every partition has formed part of the training set at some stage. The  $q$  best programs are designated as the result of the algorithm. A number of variations of the cross-validation method exist (see [BRUC95]).

## 2.6.2 Fitness Function

A fitness measure can be used to represent a single object. Bruce [BRUC96] describes four classes of objectives that fitness functions commonly test for:

- How well the individual actually solves the problem, i.e. the correctness of the solution algorithm.
- A minimization of the size of the programs induced.
- A minimization of the run time of programs induced.
- A minimization of the cost of solving the problem.

Koza [KOZA92] describes four measures of fitness that can be used to measure correctness of an individual in a GP system:

- Raw fitness
- Standardized fitness
- Adjusted fitness
- Normalized fitness

These measures are described in sections 2.6.2.1 to 2.6.2.4.

According to Soule [SOUL98], instead of evaluating an individual based on one objective only, a fitness function can also give some weight to secondary or tertiary factors such as parsimony of programs, the efficiency of programs, the compliance to initial conditions, etc. For example, the fitness function used by Langdon [LANG98b] in studies involving the generation of abstract data types included components which penalize individuals that utilize a lot of memory and CPU time in addition to testing whether the individual provides a correct solution to the problem. Multi-objective fitness functions are used for this purpose. Although a single scalar fitness measure can be the output of a multi-objective function, Pareto optimality has proven to be more effective at evaluating individuals on multiple criteria [LANG98b]. Multi-objective fitness functions are described in section 2.6.2.5.

Dynamic fitness measures have been used by some genetic programming systems. Langdon [LANG98b] states that this involves using dynamic fitness cases from one generation to the next, i.e the fitness cases are not fixed from generation to generation or even from one run to another. An example of a dynamic fitness function presented by Grant [GRAN00] firstly evaluates an individual on the evolution of a solution to a sub-task of a complex problem. Once a solution to the sub-task has been evolved, the fitness function is adapted to evolve a solution for the whole problem.

#### 2.6.2.1 Raw fitness

According to Koza [KOZA92] raw fitness is a measurement of fitness defined in “natural terms”. For example, in the artificial ant problem<sup>7</sup> the raw fitness is the number of pieces of food eaten by the ant. For problems which have real-valued target values, Koza [KOZA92] defines raw fitness as the sum of the differences between the output value of a fitness case and the result of evaluating the individual using the input values as the corresponding arguments. A common form of raw fitness is the fitness error function or the Minkowski difference [KOZA98a] which is defined by.

$$r(i, t) = \sum_{j=1}^{N_e} |s(i, j) - C(j)| \quad \dots\dots\dots (3.2.6.2.1.1)$$

where:  $s(i, j)$  is the value returned by the program  $i$  for fitness case  $j$ .

$N_e$  is the number of the fitness cases

$C(j)$  is the correct value of the fitness case  $j$ .

The result of a program can be Boolean-valued, integer-valued, floating-point valued, or symbolic-valued. For integer and floating point values the absolute values of the distances are summed. If the program is Boolean-valued or symbolic-valued the sum of the distances is equal to the number of fitness case mismatches. If the program is complex-valued the sum of the distances for each component is computed. These are then summed.

---

<sup>7</sup> This problem involves navigating an artificial ant along a trail so that it picks up the maximum amount of food on the trail.

Alternatively, the Euclidean distance [KOZA98a] can be used to assess the error factor in an individual for problems with real-valued output:

$$f_p = \sum_{i=0}^n (p_i - o_i)^2 \dots\dots\dots (3.2.6.2.1.2)$$

where:  $f_p$  is the raw fitness for individual  $p$   
 $p_i$  is the target value for fitness case  $i$   
 $n$  is number of fitness cases minus 1  
 $o_i$  is the value output by the individual for fitness case  $i$

Whether a scaled or a squared fitness measure may improve system performance is problem dependent. Based on the problem a better raw fitness may be a smaller value, e.g. the fitness error or a bigger value, e.g. the amount found eaten by the artificial ant.

### 2.6.2.2 Standardized Fitness

Standardized fitness, denoted by Koza [KOZA92] by  $s(i,t)$ , restates the raw fitness so that a lower numerical value is always better. The best value of the standardized fitness is zero. In problems where a smaller fitness value means a better fitness the standardized fitness and the raw fitness are equal, i.e.  $s(i,t) = r(i,t)$ .

In problems where a larger value means a better fitness the standardized fitness is calculated by subtracting the raw fitness from the maximum raw fitness value that can be attained. The adjusted and the normalized fitness is calculated directly from the raw fitness if the raw fitness does not have an upper bound.

### 2.6.2.3 Adjusted Fitness

The adjusted fitness is calculated using the standardized fitness as follows:

$$a(i,t) = \frac{1}{1+s(i,t)} \dots\dots\dots (3.2.6.2.3.1)$$

where  $s(i, t)$  is the standardized fitness of program  $i$  at time  $t$ .

The range of the adjusted fitness is zero to one. The adjusted fitness is higher for fitter individuals of the population. The adjusted fitness measure is only used if the selection method employed by the GP system is fitness proportionate selection<sup>8</sup>. The adjusted fitness can be used to distinguish between a good individual and a very good individual.

### 2.6.2.4 Normalized Fitness

This fitness measure is used if the method of selection of individuals is fitness proportionate selection. The normalized fitness is calculated from the adjusted fitness as follows:

---

<sup>8</sup>Selection methods are described in **Section 2.7**.

$$n(i,t) = \frac{a(i,t)}{\sum_{k=1}^M a(k,t)} \dots\dots\dots (3.2.6.2.4.1)$$

The normalized fitness lies between zero and one. It is larger for fitter individuals in the population. The sum of the normalized fitness values for each individual in the population is one. This fitness measure evaluates an individual in relation to the fitness of the rest of the population.

### 2.6.2.5 Multi-Objective Fitness Functions

Bruce [BRUC95] lists the following frequently used multi-objective fitness functions:

- The fitness measure for each individual objective is combined into one numerical value. Different functions can be used for this purpose. The most commonly used function involves taking a weighted average of each objective score. In this function more important objectives can be assigned a higher weight than the other objectives.

In the studies conducted by Langdon [LANG98b] a single scalar fitness value is used to represent multiple objectives. Scaling factors are used to allow certain dimensions of the fitness function to carry more weight than others. One of the problems experienced by Langdon [LANG95a] with using a scalar fitness function was that an increase in the fitness measure of an individual representing one objective often resulted in a decrease in the fitness measure of the same individual representing another objective. This usually leads to premature convergence of the algorithm.

- Define the fitness of an individual as a set of values instead of a single value. Each member of the set is the fitness value corresponding to a particular objective. Pareto optimality is often used for this purpose.

Langdon [LANG98b] describes Pareto optimality as an efficient means of evaluating the fitness of an individual using multiple criteria. Instead of comparing a single scalar value, Pareto optimality involves comparing each dimension of the fitness function, e.g. correctness and efficiency. Langdon [LANG98b] defines a fitter individual to be one that scores the same as the other individuals on most of the objectives, but scores better on at least one dimension.

## 2.7 Selection Methods

According to Banzhaf et al. [BANZ98] the selection method employed by a GP system effects the speed of evolution and can cause the algorithm to converge prematurely. Each selection method exerts a different amount of selection pressure<sup>9</sup>.

---

<sup>9</sup>Bruce [BRUC95] defines selection pressure to be the extent to which the selection method is biased towards the highly fit individuals in the population.

Koza [KOZA92] describes three methods that are commonly used to select individuals to apply the genetic operators to:

- Fitness-proportionate selection
- Rank selection
- Tournament selection

Koza [KOZA92] has applied the tournament selection, fitness proportionate selection, and fitness proportionate selection with over-selection methods to the 6-multiplexor problem. Tournament selection proved to be better than fitness proportionate selection but not as good as fitness proportionate selection with over-selection. The concept of over-selection is discussed below. Fitness-proportionate selection, rank selection and tournament selection are described in sections 2.7.1 to 2.7.3 respectively. Section 2.7.4 discusses over-selection. Truncation selection is presented in section 2.7.5.

### 2.7.1 Fitness-proportionate Selection

Fitness-proportionate selection, also known as roulette wheel selection, expresses the fitness of a particular individual as a ratio of all the individuals in the population. Genetic operators are applied to the elements randomly chosen from a mating pool. Fitness proportionate selection is applied as follows:

- For each individual calculate the probability that the individual will be copied into the mating pool.

$$P = \frac{f(s_i(t))}{\sum_{j=1}^M f(s_j(t))} \dots\dots\dots (3.2.7.1.1)$$

where  $f(s_i(t))$  is the adjusted fitness of an individual  $s_i$  in the population at generation  $t$ . According to Kent [KENT97] the sum of the probabilities should equal one.

- Create the mating pool.  $P$  times  $M$  occurrences of each individual will be copied into the mating pool, where  $M$  is the population size.
- Randomly select an individual, using a uniform random distribution, from the mating pool to apply the genetic operators to.

When using the steady state control model the individuals with the poorest fitness need to be located and replaced with newly created individuals. The fitness proportionate selection method can be changed as follows to cater for this. The probability for selection is calculated using the following formula .

$$1.0 - \frac{f(s_i(t))}{\sum_{j=1}^M f(s_j(t))} \dots\dots\dots (3.2.7.1.2)$$

This is referred to as inverse proportionate selection.

### **2.7.2 Rank Selection**

This method involves ranking individuals according to their fitness values. Individuals are chosen based on their rank. Thus, selection is based on a qualitative measure instead of a quantitative measure.

Rank selection prevents highly fit individuals from dominating the selection process and hence preserves genetic diversity. Koza [KOZA92] describes the following general rank selection algorithm:

- Compute the fitness values for each individual in the population.
- Sort the individuals according to fitness values and assign each individual a rank accordingly. The range of the ranks is one to the population size. Linear or exponential ranking can be used for this purpose [BRUC95].
- Individuals are selected according to rank so that the best individual is given a better chance of being chosen.

Selection is performed with replacement. This means that an individual can be selected more than once as a parent.

### **2.7.3 Tournament Selection**

The tournament selection method is implemented as follows:

- A group of individuals, usually two or more, are chosen at random from the population. The number of individuals in the group is referred to as the tournament size.
- The fitness value is calculated for each individual.
- The individual with the best fitness is chosen.

Selection is performed with replacement. According to Banzhaf et al. [BANZ98] adjusting the tournament size in a tournament selection results in the selection pressure being adjusted. The smaller the tournament size the lower the selection pressure. Soule [SOUL98] describes a common range for the tournament size to be two to five.

The tournament selection method has an advantage over fitness proportionate selection in that all the members of the population do not have to be evaluated to choose a particular individual. According to Langdon [LANG98b] the tournament selection method is commonly used due to the fact that it does not require statistics for the entire population to be computed.

Inverse tournament selection maybe needed if the steady-state control model is employed. Inverse tournament selection can be used to locate those individuals of the population with poor fitness. This involves choosing the individual with worst fitness in the tournament. This individual is then replaced with a newly created individual.

### **2.7.4 Over-Selection**

According to Koza [KOZA92] a population size of five hundred individuals has proven to be sufficient to solve most problems. However, more complex problems may require a larger population size.

Koza [KOZA92] states that the performance of a genetic programming system can be improved by “greedily over-selecting” the fitter individuals of a population. The fitness function used needs to be adjusted to ensure that the fittest individuals are given a better chance of being selected. The method of over-selection is implemented where the population size is a 1 000 or larger.

The method is implemented as follows:

- Individuals in the population are sorted in ascending order according to their normalized fitness.
- The fittest individuals accounting for thirty two percent of the normalized fitness are placed in Group1 and the remaining individuals in Group 2.
- Eighty percent of the time an individual is chosen from Group1 based on its fitness.
- Twenty percent of the time an individual is chosen from Group 2 based on its fitness.

### **2.7.5. Truncation or $(\mu, \lambda)$ Selection**

According to Bruce [BRUC96] this selection method has the strongest selection pressure. Each individual in the population reproduces a number of times, usually by mutation, until  $\lambda$  offspring are created, where  $\lambda > \mu$ . The new population contains the best  $\mu$  of  $\lambda$  offspring.  $(\mu, \lambda+\mu)$  is a variation of this method in which the  $\mu$  best individuals are chosen from the union of the old population and the new offspring.

## **2.8 Genetic Operators**

The genetic operators used by Koza [KOZA92] to create the next generation at each stage of the algorithm are Darwinian reproduction and crossover. Koza [KOZA92] also describes a number of what he terms “secondary genetic operators” which can be used in genetic programming systems. A description of both types of operators is provided in this section.

### **2.8.1 Reproduction**

The reproduction operator is applied to a single individual. An individual is randomly selected from the population using one of the selection methods described in section 2.7. This individual is then copied into the population of the next generation.

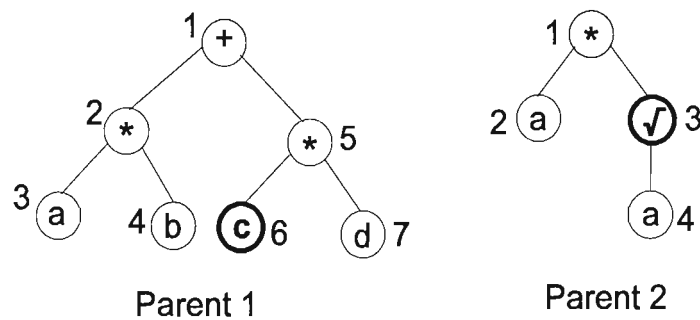
If an individual appears in a generation as a result of the application of the reproduction operator on an individual from the population of the previous generation, then there is no need to recalculate its fitness value. In this case the fitness value would have been previously computed. This reduces the amount of computer time needed. In most genetic programming systems the reproduction operator is usually applied to ten percent of the population.

### **2.8.2 Crossover**

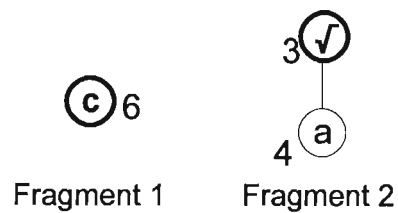
Banzhaf et al. [BANZ98] define the crossover operator as a conservation operator that builds on the information that the system has already learned. The crossover operator has been described by Poli et al. [POLI97] as a local search operator due to the fact that most of the material inherited by an offspring comes from one of the parents chosen for crossover. The crossover operator is applied as follows:

- Two parents are selected from the current population using one of the selection methods described above.
- A random point is selected in each of the parents using a uniform random distribution. This point is referred to as a crossover point. The subtrees rooted at these points are called crossover fragments.
- Both the crossover fragments are swapped generating two new offspring.

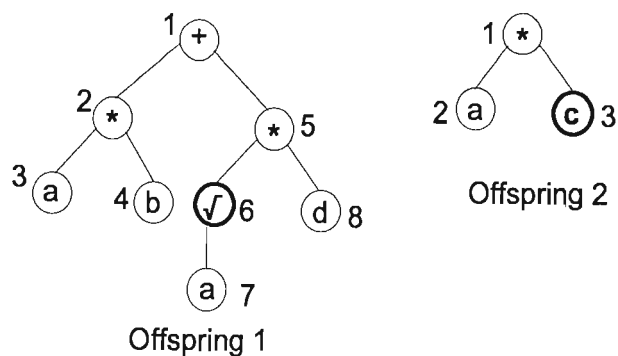
Example:



The nodes of each tree are numbered in a depth-first left-to-right manner. The crossover points selected are six in the first parent and three in the second parent. Thus, the crossover fragments are:



The corresponding offspring are:





As illustrated in the diagrams above, if a terminal is located at a crossover point in one parent, say the first parent, the subtree representing the crossover fragment in the second parent is inserted at the location of the terminal node in the first parent and the terminal is inserted at the location of the subtree in the second parent. If terminals are located at both crossover points in both the parents then the terminals are merely swapped. If both the terminals are the same this operation has no effect and the offspring are identical to the parents.

If the terminals are different then this operation has the same effect as the mutation operator. In the case where the root of both parents are chosen as crossover points then the crossover operation has the same effect as the reproduction operation.

Most GP systems place a limit on the size of the trees that can be induced using the crossover operator. A maximum permissible size, measured by the depth of the tree or the number of nodes the tree is composed of, is established for offspring created by crossover. Gathercole et al. [GATH96] describe the different ways in which a system can deal with a tree produced by crossover that exceeds the specified maximum tree size:

- Choose only the one offspring, i.e. the one that is legal.
- Reselect the crossover points until both the children produced are legal.
- Pruning of over-sized trees.
- If an offspring is over-sized replace it with one of its parents.

However, according to Gathercole [GATH98] the use of the crossover operator together with a depth limit can lead to a degradation of system performance.

In a number of studies [KOZA92], [BANZ98], and [GRAN00] crossover has been implemented with a bias towards choosing function nodes instead of terminal nodes in order to prevent the crossover operator from merely swapping terminals. In studies conducted by Koza [KOZA92] the selection process is biased so that at least ninety percent of the nodes chosen for crossover are non-leaf nodes. Research conducted by Angeline [ANGE96] has indicated that the choice of a suitable function node selection frequency is problem dependent and thus an understanding of the problem domain is necessary before selecting such a bias. Experiments conducted by Angeline [ANGE96] and O' Reilly et al. [OREI94] have indicated that a constant function or leaf frequency value does not suffice and a leaf or function frequency needs to be selected randomly for each parent prior to choosing a crossover point.

The system implemented by Langdon [LANG96a] keeps track of which trees are executed when testing fitness cases. This system also keeps track of scores that were obtained by the tree on fitness tests. This information is then used to bias the selection of trees when applying the crossover operator. Once the trees are chosen the crossover points are then chosen randomly.

A number of variations of the standard crossover operator exist. These include the following:

- Langdon [LANG94] describe a variation of crossover that produces one offspring. The crossover operator implemented by Langdon [LANG94] selects a subtree in one individual and replaces it with a randomly selected subtree from another individual. If the program is larger than the maximum program size it is discarded and another is created.

- Langdon [LANG96a] is of the opinion that in cases where the individual is fit it is a waste of resources to apply the crossover operator. He suggests a directed crossover operator which will only be applied to trees which are performing poorly in order to improve performance. An alternative provided by Rosca [LANG96a] determines the fitness for each subtree during the execution of a tree. In this way fit subtrees are prevented from being subject to the destructive affects of crossover<sup>10</sup>.
- Grant [GRAN00] describes a variation of crossover, namely depth-dependent crossover. This operator only permits crossover between subtrees of the same depth.
- Poli et al. [POLI97] use one-point crossover in their experiments. Both parents are firstly traversed to identity subtrees of the same type. According to Poli et al. [POLI97] subtrees are of the same type if the arity of the nodes in each subtree are the same from the root node through to each leaf node. Two subtrees, one from each parent, are then chosen randomly from the identified subtrees in each parent and swapped. Strict one-point crossover requires that each of the subtrees of the same type have the same function nodes from the root to each leaf node.
- The uniform crossover operator firstly identifies a set of subtrees in each chosen parent which have the same structure, i.e. the nodes of subtrees from the root to the leaf nodes have the same arity. The **sets** of subtrees are then swapped. Strictly uniform crossover swaps trees that have exactly the same structure, i.e. the subtrees have the same function nodes from the root to each leaf node.

### 2.8.3 Mutation

Banzhaf et al. [BANZ98] describe the mutation operator as an innovation operator that considers new areas of the search space and thus increases the diversity of the population. The mutation operation involves making random changes to an individual in the population. An individual is selected from the population using one of the selection methods described above.

A mutation point is selected at random. This point can be an internal point, i.e. a function or an external point, i.e. a terminal point. The subtree at the mutation point is removed. A randomly generated subtree is inserted at the mutation point. The main purpose of the mutation operator is to maintain the genetic diversity of the population. According to Bruce [BRUC95] the crossover operator can reduce the genetic diversity of a system and hence the mutation operator is used to counter the effect caused by crossover.

Although in earlier studies conducted by Koza [ANGE97] mutation is treated as a redundant operator, more recent studies conducted by Poli et al. [POLI98] and Langdon [LANG98b] indicate that the use of the mutation operator is a crucial means of preventing the premature convergence of the GP system.

A number of variations of the mutation operator exist. These include the following:

- Koza [KOZA92] describes a special case of the mutation operator which involves inserting a single terminal at a randomly selected position in the tree.
- Fernandez et al. [FERN99] explain that the mutation operator can also be used to replace a function node with another legal function node.

---

<sup>10</sup>The destructive effects of crossover are discussed in section 4.2.1.4.

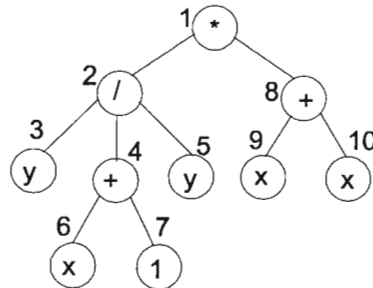
- Grant [GRAN00] and Angeline [ANGE97] describe swap mutation. Swap mutation involves replacing a chosen function or terminal with a function or terminal that has the same arity as the chosen node.
- Shrink mutation replaces the subtree rooted at the chosen node with a subtree of smaller depth from within the original subtree (see [GRAN00], [LANG98b] and [ANGE97]).
- Grow mutation has the opposite effect of shrink mutation [ANGE97]. Grow mutation replaces a randomly selected terminal with a randomly generated subtree rooted at a function node.
- Angeline [ANGE97] defines cycle mutation. A randomly selected function node is replaced with a randomly selected function symbol with the same number of arguments. Only the function node is changed, the arguments remain unchanged.
- According to Poli et al. [POLI97] and Soule [SOUL98] point mutation involves replacing a randomly selected internal or external point with a randomly generated subtree of the same arity.
- Angeline [ANGE97] and Langdon [LANG98b] define a form of mutation which randomly replaces numerical constants with randomly chosen constants. Gaussian noise can be added to a randomly chosen real value in an individual. A variation of this involves replacing the input variables by constants [LANG98b].

As with the crossover operator a depth limit can be placed on the offspring produced by mutation (see [LANG97a] and [LANG98b]). The methods for dealing with oversized offspring described in the previous section can also be applied to offspring evolved by the mutation operator. A further option for the mutation operator described by Langdon et al. [LANG97a] involves reducing the maximum height of the randomly generated tree by one until the offspring produced meets the required depth. If a depth of two is reached and the depth limit is still exceeded, the parent is copied into the next population.

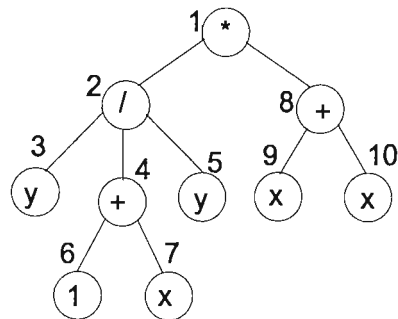
#### 2.8.4 Permutation

According to Koza [KOZA92] and Langdon [LANG98b] this method involves choosing one parent from the population using one of the selection methods described in section 2.7. The permutation operator is a type of mutation and produces a single offspring. A function node in the parent selected is randomly chosen. A permutation of the arguments of the function is randomly selected. If a function has  $n$  arguments then  $n!$  permutations of these arguments exist.

Example :



Suppose that four is chosen as a permutation point. The resulting offspring is:



### 2.8.5 Editing

The editing operation is used to edit and simplify programs. This operator is applied to one parent. The result of the operation is a single offspring. This operation recursively applies a set of editing rules to the program. These editing rules are domain-independent.

The Universal Independent Editing Rule states that if a subtree representing a function has no side effects and is composed of only constants as arguments, then the editing operation will evaluate the function and replace the function with the result of the evaluation.

Examples:        (+ 1 2) will be replaced with 3  
                   ( AND T T) will be replaced with T.

Domain-specific editing rules may also be applied.

The editing operator is used to:

- Display output in a more readable form.
- Produce simplified output during a run.
- Improve the overall performance of genetic programming.

A frequency parameter, denoted by Koza [KOZA92] as  $f_{ed}$ , specifies whether the editing operator must be applied:

- To every generation,  $f_{ed} = 1$ .
- To none of the generations,  $f_{ed} = 0$ .
- With a certain frequency,  $f_{ed} = n$ , where  $n$  is greater than 1. The editing operator is applied to every generation  $t$  where  $t$  modulo  $n$  is equal to zero.

The editing operator produces parsimonious programs from non-parsimonious programs. According to Koza [KOZA92] the application of the editing function can affect the overall performance of the algorithm negatively by reducing the variety of individuals when applied too early in a run. Another drawback of using the editing operator is that it is time consuming.

### **2.8.7 Decimation**

The decimation operator is used when the initial population has a large number of individuals with poor fitness values. In such populations the fitter individuals could end up dominating each population and hence reduce the variety of the population. The following two parameters control the application of the decimation operator:

- A percentage, Koza [KOZA92] denotes this by  $p_d$ .
- A condition stating when the decimation operator must be applied.

The effect of the decimation operator is that the entire population, except  $p_d$  percent, is deleted on generation specifying when the decimation operator must be applied. The fitter individuals are selected to remain in the population.

### **2.8.8 The Inversion Operator**

The inversion operator swaps randomly selected subtrees within an individual. Subtrees to be swapped must not be contained within each other.

### **2.8.9 The Hoist Operator**

Langdon [LANG98b] describes the hoist operator to be a variation of mutation. The hoist operator basically controls the size of trees generated from generation to generation. The hoist operator randomly selects a subtree of an individual and copies it into the population of the next generation.

### **2.8.10 The Create Operator**

The create operator generates a new parse tree, in the same manner as the trees of the initial population were generated, as part of the population of the next generation. In this way it maintains the genetic diversity of the system.

### **2.8.11 Smart Operators**

Angeline [LANG96a] proposes evolving the crossover operator together with the program it will be applied to. A similar mechanism is implemented by Teller et al. [TELL96a] in the PADO system. This system uses what Teller et al. [TELL96a] describe as SMART operators. Teller et al. [TELL96b] regard these genetic operators as programs. These operators are co-evolved together with the programs representing potential solutions.

According to Teller [TELL96b] the SMART operator programs learn to evolve programs more successfully than the conventional genetic operators as they select crossover and mutation points intelligently rather than randomly.

The population is divided into two sub-populations: one for evolving SMART operators and one for evolving potential solutions to the problem. Initially, programs representing the SMART operators are randomly generated. However, the SMART operator initial population can contain randomly generated or seeded individuals. The fitness of the SMART population and the main population have an effect on each other. Hence, this process is referred to as co-evolution.

These operators are evaluated by applying them to the main population to recombine programs. The fitness values of the programs representing the SMART operators are calculated relative to the fitness of programs representing potential solutions to the problem.

The SMART recombination operators examine their input programs in detail in order to intelligently choose fragments to exchange so that the offspring produced has a high fitness. Experiments conducted by Teller [TELL96b] have revealed that the use of SMART operators do not reduce the speed of the system or degrade the performance of individuals in the population but instead have helped to improve the overall performance of the system and the population. Essentially two SMART operators were induced by the PADO system, an intelligent crossover operator and an intelligent mutation operator.

## **2.9 Termination Criteria and Result Designation**

The genetic programming algorithm stops when the termination criteria specified by the user are met. Langdon [LANG98b] describes the following commonly used termination criteria:

- when an exact solution is found, or
- an approximate solution has been found, or
- a certain number of generations have been completed.

The first two options are often referred to as the success predicate. A problem dependent predicate can be defined as a termination criterion.

In a study conducted by Koza [KOZA92] the best individual over all the generations is designated as the result of a run. Alternatively, the best individual of the last generation can be designated as the result. Both methods have been found to return the same result. In some cases better individuals are found in later generations after the success predicate is satisfied. Thus, the population of the last generation of a run can also be designated as the result of the run.

## **2.10 Implementing a GP System**

Banzhaf et al. [BANZ98] and Langdon [LANG98b] list the following preparatory steps that must be performed before implementing a genetic programming system. These steps essentially involve setting the control parameters for the GP system:

- Choose the terminal and function sets. Terminals and functions are randomly selected to form each parse tree. Bruce [BRUC95] describes the combination of terminal and function sets as a representation language.
- Define the fitness function.
- Specify parameters such as the population size, the number of generations in a run, the maximum individual size and the maximum offspring size.
- Specify probabilities indicating the application rate of each genetic operator. These values are usually set at the beginning of a run and remain static throughout the run. However, studies conducted by Koza [KOZA99] used dynamic application rates which were changed at different stages during a run.
- Choose a selection method.
- Define the termination criteria of a run and how the result of a run will be designated.

In addition to this a set of fitness cases describing the target output for particular input values must also be specified by the user.

Bruce [BRUC95] and Grant [GRAN00] suggest that a large population size should be chosen to maintain the diversity of the population. However, studies conducted Gathercole et al. [GATH96] have revealed that a smaller population over many generations was more successful than a larger population. Thus, the choice of population size and the number of generations is problem dependent. The size of a GP population is also dependant on the available system resources.

According to Langdon [LANG98b] the evolution process usually stops after generation fifty. Thus, the maximum number of generations in a run is often chosen to be fifty. However, the maximum number of generations is problem dependent.

The control parameters and overall program architecture are usually specified at the beginning of a run. However, Angeline [ANGE97] and [KOZA99] states that these values can be changed at any stage during a run. These changes are referred to as adaptive computations.

### **3. Advanced Genetic Programming Features**

This section describes studies that have brought to light extensions of the standard genetic programming system since its inception in 1992.

The incorporation of some form of memory and iteration or recursion into a GP system will ensure that the system is Turing complete. Details regarding Turing completeness, memory, iteration and recursion are explained in section 3.1. Turing complete systems cannot solve the halting problem. Thus, a GP system will not be able to determine whether an algorithm will stop or not and hence a program can run infinitely, especially if it contains an iterative or recursive control structure, during the evaluation process. Mechanisms for dealing with infinite and time-consuming iterations and recursions are also outlined in section 3.1.

According to Koza [KOZA99] and Langdon [LANG98b] the creation and reuse of modules is essential for the successful application of a genetic programming system to complex problems. Section 3.2 describes the concept of modularization in the context of genetic programming.

Experiments conducted by Montana [HAYN97] have revealed that strongly-typed genetic programming has been successful, and in some cases more successful than standard genetic programming, in problem solving. Strongly-typed genetic programming is discussed in section 3.3.

While the standard genetic programming system automatically determines the size of a solution, the architecture of individuals in the population has to be specified by the user. Koza [KOZA98b] describes architecture-altering operations for the purpose of automatically determining the architecture of each overall program. Architecture-altering operations are examined in section 3.4.

According to Banzhaf et al. [BANZ98] cultural learning reduces the computational effort needed by a genetic programming system to evolve a solution. The concept of cultural learning is defined in section 3.5.

### 3.1 Turing Completeness

According to Teller [TELL94a] the output of the genetic programming algorithm proposed by Koza [KOZA92] is a function and not an algorithm. He describes these functions as mappings from inputs to outputs while algorithms are procedures that include iteration and/or recursion and utilize memory. Algorithms may return an output or perform some side effect. Teller [TELL94a] adapts the standard GP system presented by Koza to generate algorithms instead of functions.

According to Teller et al. [TELL96a] the use of functions instead of algorithms in the PADO system degrades system performance. A version of PADO using functions required 10 times more memory space and took approximately five times as many hours of computation. Furthermore, an object recognition rate of only seventy five percent was attained compared to the above ninety percent of the PADO system.

In his paper entitled “Turing Completeness in the Language of Genetic Programming with Indexed Memory” Teller [TELL94b] proves that the standard GP system incorporating both iteration and the use of indexed memory is Turing complete. Teller [TELL94b] states that the genetic operators need to be adapted in order to search the space of algorithms successfully.

This section illustrates how the standard GP system can be extended to incorporate the use of memory, iteration and recursion. Section 3.1.1 describes how the use of memory can be incorporated into the standard genetic programming system. The use of iterative and recursive control structures can result in infinite and time-consuming program executions. Section 3.1.2 discusses methodologies that can be employed to prevent infinite and time-consuming iteration and recursion. A summary of the studies incorporating the use of iteration into algorithms induced by GP systems is presented in section 3.1.3. Section 3.1.4 reviews studies conducted to evolve recursive algorithms.

#### 3.1.1 Memory

According to Teller [TELL94a] algorithms require some form of memory. Koza [KOZA99] defines the following types of memory that have been utilized by GP systems:

- Koza [KOZA99] and Langdon [LANG98b] describe named or scaled memory - This refers to a memory location that is accessed by a name, e.g. num1. *Read* and *write* commands are used to access this memory location. The variable representing the memory location is usually included in the terminal set [BANZ98].
- A one-argument memory setting function, e.g. SETM0 stores its argument in the memory location represented by M0. The return value of the function is its argument. In some studies the type of the SET function is “void”. One such function exists for each memory location. This is a form of named memory.
- Indexed memory - Consists of two or more memory locations stored in an array or vector. A variable is used to index the array or vector.
- Matrix memory - Is similar to indexed memory, however the memory does not take the form of a single one-dimensional array, but instead is represented by a two-dimensional array. A pair of variables represents each memory location.
- Relational memory - This form of memory consists of a fixed set of points. These points are initially unrelated. In this case the *write* operator stores values in such a way that a relation is established between the different points.



When values are read from memory, the read command determines whether there is a relation between the relevant points or not.

- Data structures, e.g stacks, queues and lists.

In addition to these forms of memory, a memory structure more recently defined is automatically-defined storage [KOZA99]. Those memory structures that have made a significant contribution to this field are discussed below.

### 3.1.1.1 Indexed Memory

According to Bruce [BRUC95], Teller [TELL94a] and Langdon [LANG98b] indexed memory is commonly represented as an integer array indexed over integers. Two functions are used to access memory, *read* and *write*, are added to the function set. The *read* function takes one argument, the index of the array. It returns the element stored at the index in the array.

The *write* function takes two arguments, an index and an integer value to be stored in the array at the index. The *write* function returns the current value stored at the index and stores the integer value passed to it at the index position in the array.

Teller [TELL94a] uses indexed memory as a means of enabling the GP system to save past inputs and utilize them in processes at a later stage. In his paper "The Evolution of Mental Models" Teller [TELL94c] describes how indexed memory can be incorporated into the GP system. In the study presented by Teller [TELL94c] each individual in the population consists of a tree as well as an array of elements indexed from 0 to  $M - 1$ , where  $M$  is the number of memory elements. The elements of the array are integers in the range 0 to  $M - 1$ . The *read* and *write* functions described above are added to the function set. The terminals are constants between 0 and  $M - 1$  and the variables representing the system inputs. All the functions in the function set are defined to return integers in the range 0 to  $M - 1$ . This ensures that each value computed is a legal memory index. Thus, a wrapper is needed in this implementation.

In the study conducted by Langdon [LANG98b] to induce abstract data types indexed memory consists of 63 memory elements. If an individual tries to access a memory element that is not within the range -31 to 31 the program is aborted and penalized accordingly. Such an individual is not tested any further.

Experiments conducted by Teller [TELL93] indicate that systems using indexed memory performed better than systems not using indexed memory in solving the Tartarus problem<sup>11</sup>. In this particular problem the number of memory elements  $M$  was chosen to be 20. The effect of choices of smaller or larger values of  $M$  on the performance of the system was examined by Teller [TELL93]. Values less than eight or nine and values greater than 40 or 50 resulted in a degradation of system performance. Teller [TELL93] explains that in the case of a lower values of memory elements there are insufficient memory elements while in the case of a larger number of memory elements there may be too many elements and hence memory elements which are written to may not be accessed again.

---

<sup>11</sup>The Tartarus problem involves an agent that is presented with the task of pushing all the boxes from the centre of the grid to the parameter of the grid. The agent is awarded two points for every box that is pushed into a corner and one point for each box that is pushed into an edge position.

### 3.1.1.2. Data Structures

According to Langdon [LANG98b] data abstraction is essential to enable genetic programming to generate solutions to more complex problems. Studies conducted by Bruce [BRUC95] and Langdon [LANG98b] have revealed that genetic programming is capable of generating methods for abstract data types (ADTs). Furthermore, studies conducted by Langdon [LANG98b] have illustrated that data abstraction is more effective than indexed memory in solving certain problems.

The research conducted by Bruce [BRUC95] involves applying genetic programming to the induction of methods for integer array-based stack, queue, and priority queue abstract data types. Essentially five methods were induced for each data structure, a constructor, a method to determine whether the data structure is empty, a method to determine whether the data structure is full, a method to add a data element to a data structure, and a method to remove an element from a data structure. Bruce [BRUC95] found that GP could generate the methods of the stack and queue individually. However, only four of the five methods were correctly induced for the priority queue. In the strongly-typed GP system implemented all the methods were induced in this experiment. Bruce [BRUC95] is of the opinion that this indicates the advantage that a strongly typed genetic programming system has over an un-typed genetic programming system. None of the experiments conducted by Bruce [BRUC95] to simultaneously induce methods were successful. Bruce [BRUC95] attributes this failure to the fact that the induction problem was too difficult to solve given the limited population size and number of generations.

However, the GP system implemented by Langdon [LANG98b] was able to successfully induce methods for the stack, circular queue and the list abstract data types. The methods for each data structure was induced simultaneously. A multi-tree structure was used to represent each program.

Five methods each were induced for the stack and queue ADTs. Indexed memory formed the basis of the stack. Ten methods for an integer linked list were simultaneously induced. Thus, in this case each chromosome was composed of ten genes. The data structures induced by Langdon's [LANG98b] system did not cater for stack, queue or list overflow and underflow.

The data structures generated in the studies conducted by Langdon [LANG98b] were then used in the generation of the solutions to the Dyck Language problem and the Reverse Polish Expression Evaluation problems. The results obtained were compared with those obtained using a GP system in which data abstraction was replaced by indexed memory. All the runs using data abstraction correctly evolved solutions to the Dyck Language problem. However, none of the runs using indexed memory generated a solution. Similar results were obtained for the Reverse Polish Expression Evaluation problem. Thus, the problems appeared to be more difficult to solve when using indexed memory than when using data abstraction.

Based on experiments conducted by Langdon [LANG96b] and Bruce [BRUC95] it is evident that genetic programming is unable automatically evolve its own data structures as needed while simultaneously inducing the solution to a problem. Thus, abstracts data types must be developed first and then form high-level members of the function set used to induce the solution to the overall problem.

The results obtained by Bruce [BRUC95] and Langdon [LANG98b] illustrate the ability of genetic programming to generate abstract data types and the effectiveness of incorporating the use of data abstraction into a GP system.

However, the appropriateness of using abstract data types is problem dependent.

### 3.1.1.3 Automatically Defined Storage

Instead of presetting an architecture incorporating the use of memory, the architecture of an individual's memory structure can be automatically evolved using automatically-defined storage. Koza [KOZA99] proposes the use of automatically defined stores (ADSs) as a means of automatically determining:

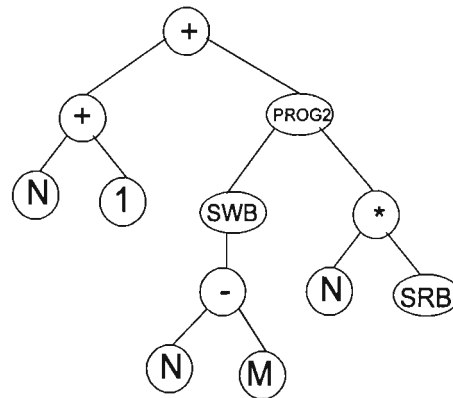
- The amount of internal memory that should be used.
- The type of internal memory to be used. The different types of memory include named memory, indexed memory, n-dimensional arrays where n is greater than or equal to two, and different data structures, e.g. stack, queue and lists.
- The dimensionality of the memory structures.
- How the memory is utilized.

The use of ADSs is incorporated into a system by adding two branches to each individual. One of the branches is a storage writing branch (SWB) while the other is a storage reading branch (SRB). The function of the SWB and SRB are analogous to that of the write and read commands described above. Each ADS consists of a unique name, size, type and dimensionality. **Table 3.3.1.1.3.1** lists each type of memory and its corresponding dimensionality. The type of the ADS is randomly chosen from this list of preset categories.

| Dimension | Types                                     |
|-----------|---|
| 0         | Named memory, pushdown stack, queue       |
| 1         | Indexed memory, list                      |
| 2         | Two-dimensional memory, relational memory |
| 3         | Three-dimensional array                   |
| 4         | Four-dimensional array                    |

**Table 3.3.1.1.3.1:** Memory types and dimensions

The dimensionality of the memory structure effects the number of arguments each SWB and SRB takes. This is type-dependent. For example, if the memory type is named memory, then the SWB will have one argument and SRB will have no arguments. Similarly, if the memory represented by an ADS is indexed memory SWB will have two arguments and SRB one argument. The number of elements of each indexed memory store and array-based store is chosen randomly. Each SWB and SRB in an ADS is also given a unique name, e.g. SRB0 and SWB0. An example of an individual containing an ADS structure is illustrated in **Figure 3.3.1.1.3.1**.



**Figure 3.3.1.1.3.1: ADS Example**

The function performed by the SWB and SRB branches is dependent on the type of internal memory represented by the ADS. For example, if the ADS represents named memory SRB1 takes no arguments and reads the contents of the first memory cell while SWB1 writes the result of evaluating its argument to the first cell returning the previous contents of the cell.

However, if the ADS represents indexed memory then SWB takes two arguments, the subtree representing the value to be written to the cell and the cell address, the contents of which is returned.

ADSs can be incorporated into the population by specifying an architecture for each individual which includes ADSs during the process of initial population creation. Alternatively, architecture-altering operators can be used for this purpose. The latter option is discussed in detail in section 3.4.

### 3.1.2 The Halting Problem

Banzhaf et al.[BANZ98] describe one of the limitations of a Turing machine to be that it cannot solve the halting problem. The halting problem is the problem of developing a program that determines whether another program halts or not. Thus, the GP system will not be able to determine whether a program can halt or not. This problem can be dealt with in GP systems by performing time-bounded executions of programs containing loops and recursion. The following methods have been used for this purpose:

- Teller [TELL94a] and Langdon [LANG98b] describe the “popcorn” method. If there are  $N$  different parse trees representing  $N$  different algorithms, all the algorithms are run at once on a particular fitness case. The algorithms are stopped when a certain time threshold is exceeded. Trees that represent algorithms that were aborted and which did not terminate are not assigned a fitness value of zero. For each individual the system maintains an average of the fitness scores that the individual has obtained on the previous fitness cases, prior to the fitness case it did not terminate on. The fitness of the individual is calculated using this average.
- Langdon [LANG98b] and Teller [TELL94a] suggest that the anytime algorithm be used to prevent infinite program execution. This algorithm is implemented as follows. For each individual the corresponding input is placed in memory.

After the algorithms have run for a certain amount of time they are stopped and are required to return a response based on what they have calculated thus far, i.e. the algorithms are not required to terminate [TELL94a]. Usually, the entire algorithm cannot be executed during the allocated time. At the end of the time period the contents of the relevant memory positions are extracted and interpreted as the answer.

- The aggregate computation time ceiling method is an example of a time-bounded execution methodology. It involves distributing the execution time over the fitness cases. Thus, each fitness case will have a slice of execution time [KOZA99].
- According to Koza [KOZA99] and Langdon [LANG98b] computer time can be assigned to each individual in the population in a round-robin fashion. The time allocated must not be static but must be decreased over a run.
- Koza [KOZA99] suggests that a limit on the number of executions that can be performed by an iteration be set. In order to cater for nested iterations and multiple iterations in an individual a limit can also be set on the number of executions per individual.
- The function and terminal sets can be constructed so as to prevent the occurrence of long and infinite loops [KOZA99].
- A method described by Langdon [LANG98b] involves restricting evolution to the body of a loop while the starting or termination condition is preset and remains static.

### 3.1.3 Iteration

The standard GP system can be extended to cater for iteration by either specifying iterative primitives in the function set or incorporating an iterative component directly into the architecture of each individual. **Section 3.1.3.1** examines the use of iterative primitives while **Section 3.1.3.2** discusses automatically-defined iterations and loops.

A problem associated with implementing iteration and recursion is the possibility of infinite programs and lengthy programs which consume a large amount of computer resources. One of the methodologies presented in the previous section must be employed to prevent infinite and time consuming iterations .

#### 3.1.3.1 Iteration Primitives

In the system implemented by Koza [KOZA92] the function set contains the iteration primitive *DU* (do until). *DU* takes two arguments, a body to be executed on each iteration, and a condition. The iteration stops when the condition specified as the second argument of the *DU* is met. A maximum number of 25 iterations can be performed by each *DU* instance and a maximum of a 100 iterations per individual is permitted.

A similar primitive, namely *forwhile*, is used in the GP system developed by Langdon [LANG98b] to automatically induce methods for the list abstract data type. Nested *forwhile* loops were not allowed, and only 32 iterations per *forwhile* was permitted.

In the GP system proposed by Pringle [PRIN95] iteration is catered for by the *WHILE* primitive. Like the *DU* operator the *WHILE* takes two arguments, a body to be executed on each iteration and a condition. However, in this case the iteration is terminated when the condition specified is no longer satisfied.

### **3.1.3.2 Automatically Defined Iterations and Loops**

This section describes how iterative structures can be incorporated into the architecture of individuals. Automatically Defined Iterations (ADIs) and Automatically Defined Loops (ADLs) are used for this purpose. An iteration forms the basis of both ADIs and ADLs. An iteration, as defined by Koza [KOZA99], consists of essentially four components:

- An initialization.
- A termination or continuation condition.
- The body of the iteration.
- An increment or decrement step.

ADLs and ADIs are described below.

#### **3.1.3.2.1 Automatically Defined Loops**

According to Koza [KOZA99] automatically defined loops are used to implement a general iteration in a genetic programming system. ADLs are comprised of four branches, namely a loop initialization branch (LIB), a loop condition branch (LCB), a loop body branch (LBB) and a loop update branch (LUB). An invocation of the loop results in the LIB branch being evaluated followed by the LCB branch. Depending on the result of the evaluation of the LCB branch the LBB will be executed followed by an evaluation of the LUB branch or the loop will be terminated. An ADL is assigned a name and an argument list. Koza [KOZA99] suggests that an indexing variable is needed to keep track of the number of executions performed in an iteration.

ADLs can be incorporated into a genetic programming system by using architecture-altering operations or creating individuals during initial population generation that include ADL structures in their architecture. Architecture-altering operations are discussed in more detail in section 3.4.

#### **3.1.3.2.2 Automatically Defined Iterations**

A special instance of the general iteration structure is problem-specific iteration, i.e. iteration needed to access the elements of an array, a finite sequence or a vector. The initialization step, the continuation/termination condition and the increment or decrement step is fixed in this case. Automatically-defined iterations are used to represent these problem-specific implementations of an iteration while automatically-defined loops are used to represent the general iteration structure.

The structure of an ADI consists of a name, argument list and body. The body of an ADI is executed more than once, i.e. for each element of the array, vector or finite list. ADIs can be incorporated into the GP system by creating individuals specific to the ADI architecture, i.e. with a results-producing branch (representing the main program) with zero or more ADI branches, during the process of generating the initial population. Alternatively, architecture-altering operations can be used for this purpose. Architecture-altering operations are discussed in section 3.4.

3.1.4 Recursion

Recursion has not been used as widely in GP systems as iteration. Like iteration, recursion can be incorporated into a GP system by adding a recursion operator to the function set or including an automatically-defined recursion component in the architecture of each individual.

In a study conducted by Koza [KOZA92] to automatically induce the Fibonacci sequence using genetic programming, the SRF primitive was used to cater for recursion. SRF takes two integer arguments, K and D. As each element of the Fibonacci sequence was generated this value was stored and used by SRF to calculate successive elements of the sequence. If the K-th element of the sequence has already been computed SRF returns the value of this element otherwise it returns D.

Brave [BRAV96] has implemented a GP system to evolve a recursive algorithm that performs a binary tree search. The system developed by Brave [BRAV96] uses automatically defined functions (defined in section 3.2.1) that are allowed to call themselves to perform recursion. An automatically defined function can be called recursively until the maximum depth of the tree being searched is reached. Further recursive calls result in the value of the leaf node at the current position being returned. This system proved to be more successful at evolving the binary search algorithm than the standard GP system and the standard GP system with non-recursive ADFs.

A structure more recently used to incorporate recursion into a GP system is Automatically Defined Recursion (ADR). ADRs consists of four components, namely, a recursion condition branch (RCB), a recursion body branch (RBB), a recursion update branch (RUB) and a recursion ground branch (RGB). ADRs can be incorporated into the population by specifying an architecture inclusive of ADRs for the generation of the initial population or by using architecture-altering operators. **Figure 3.3.1.4.1** illustrates an example of an individual using an ADR. Architecture-altering operations are discussed in section 3.4.

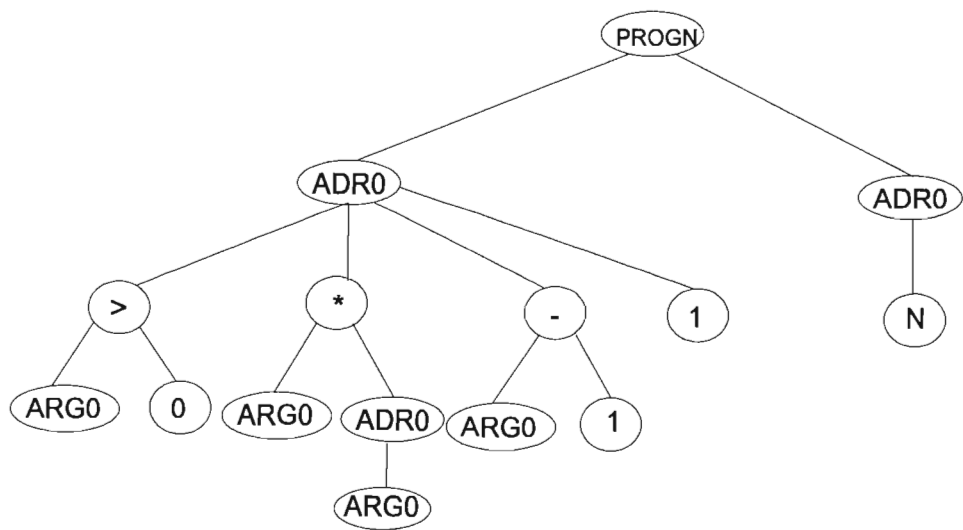


Figure 3.3.1.4.1: ADR Example

An invocation of an ADR results in the RCB being firstly executed. If the evaluation reveals that the condition represented by the branch holds, the RBB branch is then evaluated followed by the execution of the RUB. When the recursion terminates the RGB is implemented. Thus, if the specified condition is met the result of the RBB is returned while if the specified condition is not met the result of the RGB is returned. The RBB branch of the ADR may contain a call to itself. A unique name and an argument list is assigned to each ADR branch. Invocations of an ADR may be in a results-producing branch (representing the main program), or an ADF<sup>12</sup>, ADI, or ADL branch.

### **3.2 Modularization**

According to Banzhaf et al. [BANZ98] and Langdon [LANG98b] modularization is a technique commonly used by human programmers in problem-solving. In order for the genetic programming algorithm to induce efficient solutions and deal with larger programming problems, it is essential that modularization is catered for [BANZ98]. A number of attempts have been made to extend the standard genetic programming system proposed by Koza [KOZA92] to include modularization. These include automatically defined functions, encapsulation, and module acquisition. This section examines each of these methods.

#### **3.2.1 Automatically Defined Functions (ADFs)**

Koza [KOZA94] introduces the concept of subroutines within a program by means of automatically defined functions. According to Langdon [LANG98b] the incorporation of ADFs in GP systems has enabled genetic programming to solve problems previously unsolved by the standard GP system. Automatically defined functions enable a genetic programming system to solve a problem by decomposing it into subproblems. The GP system implemented by Koza [KOZA94] simultaneously induces a main program together with the subroutines called by the main program.

Genetic programming does not generate programs in the same way as a human programmer writes programs. Thus, the automatically defined functions induced may not be defined in the same way as those created by a human programmer.

ADFs are not beneficial for simple problems. The use of automatically defined functions decreases the computational effort needed to find a solution as well as increases the parsimony of solutions provided that the problem is of sufficient difficulty [KOZA98b]. The benefits of using ADFs increases with the complexity of the problem.

Koza [KOZA94] has found that a GP system incorporating the use of ADFs has the lens effect, i.e. it tends to find individuals that have extreme fitness scores.

Sections 3.2.1.1 through to 3.2.1.5 examines the implementation details of using ADFs.

##### **3.2.1.1 Representation of each Individual**

Each individual of the population is represented as a tree containing one or more function-defining branches, representing each ADF, and a results-producing branch, representing the main program. The results-producing branch can call the functions defined by the function-defining branches.

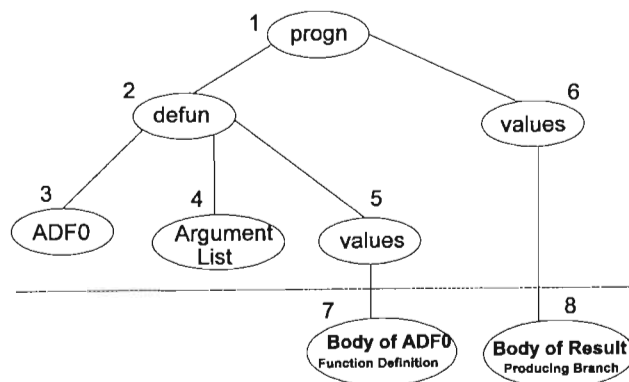
---

<sup>12</sup>The concept of an ADF is presented in Section 3.2.1.



The evolutionary process determines how many times and in what manner each of the functions defined by the function-producing branches are called.

Koza [KOZA94] presents the following example tree consisting of one function-defining branch and one results-producing branch. This tree represents a program in Lisp.



**Figure 3.3.2.1.1.1: ADF Program Structure**

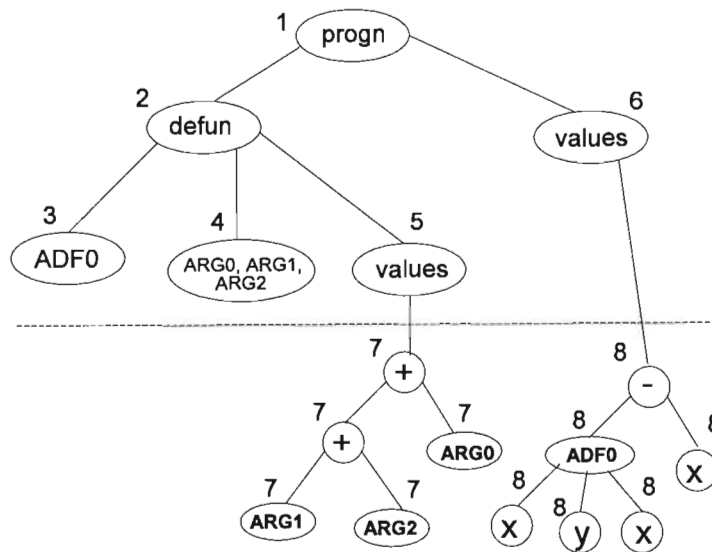
Koza [KOZA94] describes the first six points in this tree as invariant and the remaining two as non-invariant. Each node is defined as follows:

1. The first node in the tree is the root of the tree. In this case the Lisp connective *progn* is the root of the tree and combines together the statements representing the automatically defined function and the main program.
2. The Lisp primitive *defun* which marks the beginning of each function in Lisp.
3. The name of the automatically defined function.
4. The argument list of the automatically defined function.
5. Returns value/s outputted by the automatically defined function.
6. Returns value/s outputted by the results-producing branch,
7. The body of the automatically defined function.
8. The body of the results-producing branch.

If the minimalist approach is taken each individual in the initial population consists of a single function-defining branch which takes a single argument [KOZA98b].

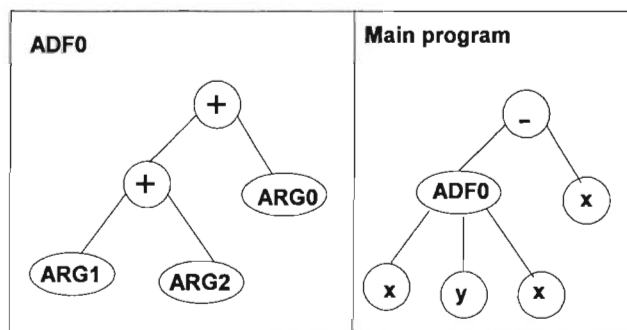
If more than one value is returned by the results-producing branch, a number of sub-branches exists under the values node in the results-producing branch. The actual variables of the problem are defined by the results-producing branch and can be passed to the function defining branches. The different scenarios for function calls between ADFs are discussed in section 3.2.1.2. Koza [KOZA94] presents the Lisp program in **Figure 3.3.2.1.1.2** as an example.

Bruce [BRUC95] suggests that instead of using a single tree to represent the main program and the ADFs the genotype can consist of  $n$  trees stored in a fixed-size array, one representing the main program and the  $n-1$  trees representing the  $n-1$  ADFs. The user has to specify the number of abstract trees in each genotype. The user must also specify how the evaluation of each tree will contribute to calculating the overall fitness of the individual.



**Figure 3.3.2.1.1.2:** Lisp program with ADFs presented by Koza [KOZA94]

Bruce [BRUC95] states that this representation has a number of advantages over the representation used by Koza. Firstly, it simplifies the representation of programs including ADFs. As we are now dealing with separate trees and not a single tree with constraints on which branches are modifiable, the genetic operators do not need to be adapted. This representation is more general, e.g. the genetic programming system without the use of ADFs is essentially this system with the size of the genotype being one. **Figure 3.3.2.1.1.3** illustrates the equivalent representation for the program depicted in **Figure 3.3.2.1.1.2**.



**Figure 3.3.2.1.1.3:** Alternative ADF Representation

In the system implemented by Koza [KOZA94] a call to an ADF is treated as an element of the function set, i.e. the tree representing the ADF is applied to the arguments passed to the ADF. Langdon [LANG98b] further extends the ADF concept by catering for pass-by-reference modules. An automatically defined function is able to change the argument passed to it, in the same way that variables are passed by reference in a program. The ADF is not allowed to return a constant or return an output value that is equal to its input value.

The use of call-by-reference ADFs eliminate the need to update a memory location with the value calculated by the ADF.

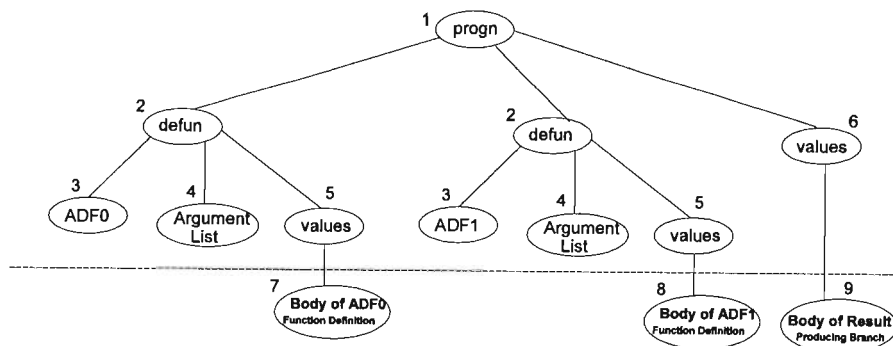
Another alternative described by Grant [GRAN00] and Naemura et al. [NAEM98] involves maintaining two breeding pools, one to evolve ADFs and the other to evolve main programs. ADFs which result in the main programs having a good fitness are added to a library. Both pools can access the library. The system implemented by Naemura et al. [NAEM98] consists of three components, one for module generation, one for storing modules, and one for main trees.

The module generation component has its own primitive set. Subtrees contained in ADFs which appear to improve the fitness of individuals are added to the module storage component. All the programs in the main tree generation component can access this library. Modules are removed from the library if they appear to reduce the fitness of individuals and are replaced with newly found modules. A module hierarchy is developed over a run. The first layer of the hierarchy (layer-0) consists of modules composed only of elements of the GP primitive set. Individuals can access the modules in layer-0. Modules in layer-1 of the hierarchy are composed of primitives and the modules in layer-0. Individuals can now access modules in layer-0 and layer-1. This method was found to produce more parsimonious solutions.

### 3.2.1.2 ADF References

In programming a function can usually call other functions that have been defined. If there is more than one function-defining branch, these branches can have one of the following relationships:

- There are no references between the function-defining branches.
- References among function-defining branches are not restricted. In this case an ADF can call itself recursively either directly or indirectly. The name of the ADF is added to its function set.
- A function may hierarchically call those functions that have been defined before it, e.g. in **Figure 3.3.2.1.2.1** ADF0 can call ADF1 but not vice versa.



**Figure 3.3.2.1.2.1:** Program with two ADFS

### 3.2.1.3 Changes that Need to be Made to the Standard GP System

When incorporating the use of ADFs into a GP system the user has to specify the number of function-defining branches and the number of arguments that each ADF will take. Koza [KOZA94] suggests that the number of variables of the problem should be used as an upper limit of the number of arguments that an ADF can take. If each individual will contain more than one ADF the user will have to choose one of the methods in section 3.2.1.2 to define legal calls between functions.

For each of the function-defining branches and the results-producing branch a terminal and a function set must be specified. Experiments conducted by Koza [KOZA94] have indicated that the effect of using different architectures with ADFs, e.g. individuals with four ADFs with two arguments each instead of individuals with three ADFs with four arguments each, etc, does not have a significant effect on system performance.

The method used to randomly generate the initial population must be edited to create individuals that are composed of a results-producing branch and one or more function-defining branches. The crossover operator needs to be adapted in order to ensure that crossover is performed on like branches, i.e. structure-preserving crossover must be applied to individuals so that the offspring produced are syntactically correct.

The invariant components of each individual are not altered during crossover, i.e. structure-preserving crossover is restricted to the non-invariant points of the individual. Each non-invariant point is assigned a type. Structure-preserving crossover is implemented as follows:

- A non-invariant point in the first parent is randomly chosen.
- A point in the second parent is randomly chosen from the points of the same type as the point chosen in the first parent.

To ensure that genetic operators are applied to the same branches in each individual, types have to be assigned to the nodes in each individual. Typing is discussed in section 3.2.1.4. Crossover occurring in the function-defining branches indirectly affects the main program whilst the opposite is not true.

One of the disadvantages of using ADFs is that parameters, e.g. the number of ADFs and the number of arguments of each ADF needs to be preset prior to evolution [NAEM98].

### 3.2.1.4 Typing

The following methods can be used to assign types to the non-invariant nodes:

- Branch typing - According to Koza [KOZA94] there is one type for each branch. A different type is assigned to the non-invariant points of each branch. In the tree in **Figure 3.3.2.1.1.2**, the function-defining branch is of type seven while the results-reproducing branch is of type eight.
- Point typing - Each individual non-variant point in the overall individual is assigned a type. Each type has a function set, a terminal set, an argument map, and a description of the syntactic constraints of the branch where the node is located, associated with it.

Koza [KOZA94] suggests that point typing be used in systems where the overall program architecture is evolved during the run (see section 3.2.1.5 ).

- Like-branch typing - Points in branches that have the same terminal and function sets are assigned the same type [KOZA98b].

According to Koza [KOZA99] branch typing is the most commonly used typing method. In GP systems which automatically evolve the architecture of the individuals together with evolution of a solution, it is essential that point typing is used.

### 3.2.1.5 The Automatic Evolution of the ADF Architecture

One of the methods described by Koza [KOZA94] which can be used to determine how many ADFs each individual should have, the number of arguments of each ADF, and the terminal and function set of each function-defining branch is prospective analysis of the nature of the problem. The method of prospective analysis involves using our knowledge of the problem domain to choose the number of ADFs and the number of arguments for each ADF. The amount of prospective analysis that needs to be done is dependant on the user's goals. Alternatively, the method of using affordable capacity can be employed for this purpose. In this case the computer resources available dictate the architecture of the system. A third alternative is the evolution of the ADF architecture. Instead of the user specifying the number of ADFs, the number of arguments each ADF should take, and the nature of hierarchical references between ADFs, these architectural characteristics can be developed together with the solution to the problem.

The function set of the results-producing branch consists of the primitives of the problem as well as the  $n$  automatically defined functions, where  $n$  and the ADFs will be generated by the GP system. The function set of each function-defining branch will consist of primitive functions as well as the ADFs that the particular ADF can access hierarchically, i.e. all the ADFs defined before it. The terminal set of the results-producing branch will consist of the variables of the problem.

These can be passed to the ADFs but are not contained in the terminal sets of the ADFs. The terminal set of each function-defining branch contains  $n$  dummy variables. For each ADF the number of arguments for the ADF is randomly chosen from a specified range.

The architecture of each offspring will be inherited from the offspring's parent/parents. An automatically defined function appearing in more than one parent can possibly have a different number of arguments in each of the parents although it will have the same name. Structure preserving crossover together with point typing is used to ensure that both syntactically and semantically correct offspring are produced.

Automatically Defined Functions (ADFs) can be incorporated into the architecture of each individual during the process of initial population generation, i.e. each program in the population is composed of one results-producing branch and zero or more function-defining branches. Changes have to be made to the process used to generate the initial population randomly. The number of ADFs that an individual contains is randomly chosen from a specified range, e.g. 1 to 3. A number of arguments, within a specified limit, for each ADF is randomly chosen. Zero is included in the limit range. The ADFs are systematically named, ADF0, ADF1, etc. According to Koza [KOZA94] the maximum number of arguments permissible in an ADF can be made equal to the number of variables for the problem. In some problems having an ADF with 0 arguments may be meaningless and hence the range of arguments must begin at one.

Alternatively architecture-altering operators can be used for this purpose. The latter is discussed in section 3.4.

Koza [KOZA94] found that when applied to the Boolean even-5-parity problem this simultaneous evolution of both the solution and the ADF architecture required more system resources.

### 3.2.3 Encapsulation

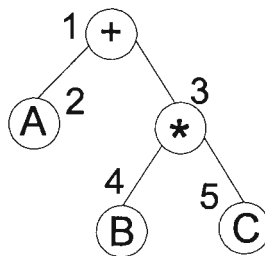
An individual is selected using one of the methods of selection described in section 2.7. The result of applying this operator is a single offspring.

The encapsulation operator identifies a subtree that increases the possibility of finding a solution if left in tact and not divided by the crossover operator. The operator assigns a name to the subtree and replaces the subtree with this name so that it can be used again later.

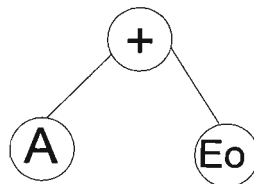
The encapsulation operator used by Koza [KOZA92] is applied as follows:

- A copy of the selected parent is made.
- Select a function node in the copy at random.
- The subtree located at the selected point is removed.
- A new function is defined to represent the deleted tree. This function has no arguments.
- The new encapsulated function is labeled, E0, E1, E2,...,etc. Because the function has an arity of zero the new function label is added to the terminal set.

Example: Consider the following parse tree



Suppose that point three was randomly chosen. The function E0 is then created. The offspring produced is :



The subtree is no longer in danger of being subjected to the disruptive effects of crossover. This will be beneficial if the encapsulated subtree is a good building block. The original parent is not semantically changed by the application of the encapsulation operator.

An intelligent encapsulation operator will choose a function node based on whether the subtree rooted at this node is a good building block or not, instead of randomly choosing a node.

3.2.4 Module Acquisition

Module acquisition or compression performs a similar function to that of the encapsulation. A function node is randomly selected in a chosen individual. A part of the subtree rooted at the chosen function node up until a specified cutoff-depth is defined as a module. The nodes at the level below the cut-off depth are considered as arguments to the module. The module is named and added to the function set. Alternatively, the module can be added to a library of such functions which can be accessed by all the individuals in the population. The module remains in the library until there no longer exists any individuals that reference it.

Consider the following parse tree:

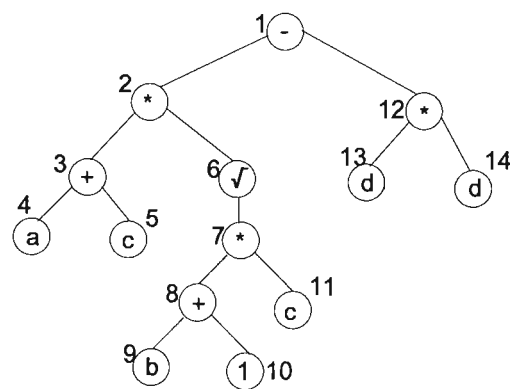


Figure 3.3.2.4.1: Individual to which the compression operator will be applied

Suppose that the cutoff-depth is three and that node six, i.e. the square root operator, has been randomly selected. The new module will consist of the nodes that are boxed in the figure below. This new function will take two arguments, in this case b and 1.

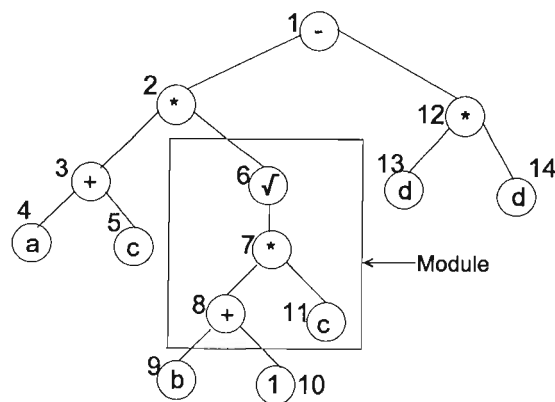
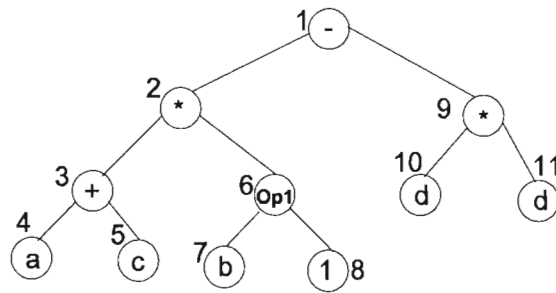


Figure 3.3.2.4.2: Diagram indicating the chosen module

The effect of applying the compression operator to the above parse tree is:



**Figure 3.3.2.4.3:** Result of the compression operator

**Op1** is the name of the new module. A library of the modules created using compression is often maintained [NAEM98]. All the individuals in the population can access these modules.

In addition to the application rate of the module acquisition operator, another parameter, namely, the cutoff-depth must be specified. Module acquisition allows for a module to be protected from the destructive effects of crossover.

An expand operator can be incorporated into the GP system to undo the effects of module acquisition if necessary. The expand operator can only be applied to an individual that contains at least one node that represents a function created by the compress operator. The expand operator expands such a node into the function it represents. In the case of nested modules, the chosen module is expanded one level only.

### 3.3 Strongly-Typed Genetic Programming

According to Bruce [BRUC95] Montana introduced the concept of “typing” into genetic programming systems. This later became known as strongly-typed genetic programming (STGP). In strongly-typed genetic programming systems each variable, constant, function argument, argument return value and program return value contains a type label. Based on these labels only syntactically correct programs are created. Studies conducted by Bruce [BRUC95] and Montana [HAYN97] have illustrated the effectiveness of strongly-typed genetic programming. Bruce [BRUC95] describes the following advantages of strongly-typed genetic programming over untyped/conventional programming:

- Programs produced by a strongly-typed GP system can be directly translated into a computer programming language such as C.
- According to Banzhaf et al. [BANZ98], Bruce [BRUC95], and Haynes [HAYN97] typing reduces the size of the search space by reducing the number of syntactically correct program trees that can be generated using the language elements. This can be attributed to the fact that syntactical constraints are placed on the architecture of each individual. This enables strongly-typed genetic programming systems to converge to a solution quicker.



Furthermore, Langdon [LANG98b] states that typing enables the genetic programming system to choose a better solution path than the standard GP system as syntactically incorrect programs are avoided. In strongly-typed genetic programming systems only syntactically correct programs, i.e. with respect to data types, must be created by the initial population generator and the genetic operators. Kent [KENT97] states that the mechanism used to generate the initial population needs to be adapted to take typing into consideration. Furthermore, the genetic operators need to be adapted in order to cater for strongly-typed GP. For example, when the mutation operator is applied to a new node the mutation operator must replace the node with one of the same type. The crossover operator must be modified so that only subtrees of the same type can be swapped .

According to Langdon [LANG96a] the strongly-typed genetic programming system implemented by Montana performed better than the standard GP system. He attributes this superior performance to the fact that search space was reduced in size. However, experiments conducted by Kent [KENT97] have revealed that the strongly-typed GP has an additional time overhead associated with it.

In a strongly-typed genetic programming system some primitives may be defined to have a generic type [HAYN96] (and [BANZ98] ). The types for these primitives are instantiated when their arguments are randomly generated during the initial population generation process.

For example, the conditional *IFTE* control structure is defined as a generic type. The type of the subtree rooted at the *IFTE* primitive, i.e. the return type and the type of its third argument, is instantiated to the type of the subtree representing its second argument. This instantiation only takes place once the subtree representing the second argument of the particular instance of the *IFTE* primitive is generated. The use of generic functions prevent the need for primitives to be redefined for each type, e.g an IF-statement for reals, integers and strings. In the STGP developed by Haynes et al. [HAYN96] both generic primitives and a type hierarchy are used. The type hierarchy allows more than two levels of typing. The type hierarchy represents inheritance relationships between the different types.

Changes had to be made to the procedure for generating the initial population, crossover and mutation to cater for generic typing and the use of the type hierarchy. The type of the root of the replacement subtree generated during mutation must be the same type or a subtype as that of the root of the subtree it is replacing. Similarly, the roots of subtrees rooted at the selected crossover points during crossover must be of the same type.

### 3.4 Architecture-Altering Operations

The concept of architecture-altering operations is presented in [KOZA99]. Architecture-altering operations are used by a genetic programming system to automatically evolve the program architecture together with the solution and in doing so relieves the developer of specifying the program architecture prior to the run. According to Koza [KOZA98b] the use of architecture-altering operations enable the GP system to automatically determine the shape of the solution while simultaneously evolving the solution.

The application of architecture-altering operations results in a diverse population of individuals which differ with respect to the number of subroutines, arguments for each subroutine, iterations, recursions, and memory structures that they contain.

Bennett III [BENN96] has successfully used the following architecture-altering operations in a GP system developed to automatically evolve multi-agent architectures:

- The Branch Duplication Operation

The application of this operation involves making a copy of a branch in a multiple agent program. Bennett [BENN96] applies this operator to three percent of the population for each generation. The tournament selection with a tournament size of seven is used to choose an individual. A branch is randomly chosen and duplicated. The copy of the branch is inserted into the individual. If the individual already has a maximum number of branches then this operation has no effect.

- Branch Deletion Operation

This operator deletes a particular branch of a parallel multiple agent program. This operator is applied to one percent of the population during each generation. An individual is selected from the population using tournament selection as in the case of the branch duplication operation. A branch is randomly chosen in the individual and deleted.

This operator has no effect if the individual has only one branch, or if the deleted branch is a copy of another branch in the individual, or if the deleted branch performs the same function as another branch in the individual.

Koza [KOZA99] defines architecture-altering operations for automatically defined functions (described in section 3.2.1), automatically defined iterations (described in section 3.1.3.2.2), automatically defined loops (described in section 3.1.3.2.1), automatically defined recursion (described in section 3.1.4) and automatically defined storage (described in section 3.1.1.3). The following architecture-altering operations can be applied to each of these structures: structure<sup>13</sup> duplication, argument duplication, structure creation, argument creation, structure deletion and argument deletion.

For each of the architecture-altering operations an individual is selected from the population using one of the selection methods. The architecture-altering operation is applied to a copy of the chosen individual to create an offspring. An application rate must be specified by the user for each operator (a separate rate must be defined for each operation for each structure).

### 3.4.1 Structure Duplication

Structure duplication involves duplicating a structure branch<sup>14</sup> in the chosen individual and adding the chosen branch to a copy of the selected individual to create an offspring. If the chosen individual has no structure branches or already has the maximum number of structure branches permissible, the reproduction operator is applied to the individual.

---

<sup>13</sup>In the sections that follow a structure refers to an automatically defined function, an automatically defined iteration, an automatically defined loop, an automatically defined store or an automatically defined recursion.

<sup>14</sup>A structure branch refers to a function-defining branch, an iteration branch, a loop branch a recursion branch or a storage branch.

References to the copied branch in the offspring are randomly distributed between the newly created branch and the original branch. Furthermore, if hierarchical referencing between the particular structure type is allowed, the name of the new function is added to the function set of all the relevant branches and the results-producing branch. Structure duplication is a semantics-preserving operator.

### **3.4.2 Argument Duplication**

The argument duplication operator extends the argument list of a chosen structure branch, in a copy of the selected parent, to include a copy of one of the arguments already in the list.

If the chosen branch does not have any arguments or has the maximum number of arguments the chosen individual is merely copied into the next generation. The references to the copied argument in the original structure branch is randomly distributed between the copied argument and the original argument in the offspring. An additional argument has to be passed to an invocation of the structure branch in the rest of the individual. The additional argument is a copy of the subtree passed to the structure for the original argument. The offspring produced is semantically equivalent to the parent. If the structure is an ADS the arguments of both the storage writing branch and the storage reading branch are increased.

### **3.4.3 Structure Creation**

A node in the body of any of the structure branches or results-producing branch is randomly chosen. This node will be referred to as N. The chosen node will form the root of the body of the new structure branch. The chosen node will also be replaced with the new structure label. The subtree rooted at the chosen node is traversed to determine the arity of the new structure. During the traversal each node is randomly chosen as being an argument or not of the new structure (in the branch containing N). If a function node is chosen as an argument, its children are not visited. Zero or more points maybe chosen. The new branch is given a unique name. The arity of the new structure is equal to the number of nodes randomly chosen during the traversal. N is replaced with the name of the newly defined structure and the arguments chosen during the traversal form the arguments of this node. The body of the structure is a modified version of the subtree rooted at N in which each node (and its corresponding subtree) in the subtree chosen during the traversal to be an argument, is replaced by the corresponding dummy variable (e.g. ARG0, ARG1, etc).

If the structure being created is an automatically defined loop or an automatically defined recursion, the four branches of the ADL or ADR are randomly chosen from the existing branches in the copy of the selected individual. The choices of the four branches are independent of each other.

The creation procedure for generating a new ADS differs from the creation operator of the other structures. The dimension for each ADS is randomly chosen. A memory type for the chosen dimension is also randomly selected. If necessary, an integer value representing the capacity of the structure is selected randomly. A connective, PROG2, or BLOCK2, is used to combine SWB and SRB branches on the offspring. An ADS is incorporated into an individual by firstly selecting a subtree of a chosen branch in the original individual. The subtree is replaced with the connective and SWB and SRB as its children. The chosen subtree becomes an argument for SWB. Further arguments for SWB and SRB are chosen by randomly selecting a subtree in a randomly chosen branch.

#### **3.4.4 Argument Creation**

A node is selected in the body of a randomly chosen structure branch. This node will be referred to as N. An argument is added to the argument list of the branch containing N. N, and the subtree rooted at N, is replaced with the new argument label. Every call to the chosen structure containing N in the individual must be updated to include another subtree representing the new argument. The additional subtree consists of the subtree rooted at N, with the dummy variables replaced with the subtrees representing each dummy variable for that particular occurrence of the structure. The terminal set of the branch containing N is extended to include the new argument.

In the case of automatically defined stores this operator increases the number of arguments of the SWB and the SRB by one.

A unique argument name is added to the argument list of the ADS to represent the copy of an existing argument subtree that is added on as a new argument to the invocation of SWB and SRB.

#### **3.4.5 Subroutine Deletion**

The subroutine deletion operator removes a chosen structure branch from a copy of the selected individual to create a new offspring. Invocations of the deleted branch must be replaced. One of three methods, namely consolidation, random regeneration, or macro expansion, can be used for this purpose.

#### **3.4.6 Argument Deletion**

Argument deletion involves removing an argument from the argument list of a chosen structure branch. If the branch has no arguments or a minimum number of arguments the reproduction operator is applied to the individual. The subtree representing the deleted argument is removed in each invocation of the structure.

References to the deleted argument are replaced with an existing argument in the chosen branch. Either consolidation, or random regeneration or macro expansion can be used for this purpose.

Storage argument deletion reduces the number of arguments of the SWB and SRB of a chosen ADS by one. The number of arguments passed to an invocation of the ADS must also be reduced by one.

### **3.5 Cultural Learning**

Spector et al. [SPEC96] have introduced the concept of cultural learning into GP systems. They define the term culture in the context of GP systems as information transferred among individuals via “non-genetic means”. Analogous to the term gene this information is called a meme. According to Banzhaf et al. [BANZ98] the use of cultural learning can reduce the computational effort needed to find a solution.

Spector et al. [SPEC96] use an adaptation of Teller’s indexed memory to implement culture. In the system developed by Spector et al. [SPEC96] indexed memory is initialized once at the beginning of a run. All the individuals in the population share the same memory. The contents of memory at the end of one fitness evaluation is not erased but is used by the next individual and the next generation.

This allows an individual to pass information to itself from one fitness case to the next. Furthermore, programs can use information left in memory by previously evaluated programs. The contents of memory can be removed at the end of evaluating an individual or the entire population. In the latter case, the order in which the population is evaluated can have an effect on the success of a solution and this order is hence randomized for each generation.

The correct functioning of a solution found in a GP system incorporating culture is dependent on the initial state of memory. Thus, the best-of-generation individual should be reported in each generation together with its initial memory state. If the solution program is used again its memory must firstly be initialized to this initial memory state.

Spector et al. [SPEC96] applied cultural learning to three problems, a symbolic regression problem, the lawn mower problem, and wumpus world. When applied to the symbolic regression problem the GP system using culture required the least amount of computational effort while the GP system using indexed memory required more computational effort than the standard GP system. The amount of computational effort needed to find a solution was reduced by sixty one percent when incorporating culture instead of just memory. Similar results were found for the lawnmower problem and wumpus world. However, cultural learning is not necessarily beneficial in all problem domains [LANG98b].

#### **4. Problems Encountered in the Implementation of GP Systems**

This section provides a detailed discussion of the shortcomings of genetic programming that one must be aware of when implementing a GP system.

Algorithms evolved by genetic programming systems usually contain redundant code. The redundant code is referred to as introns. The number of introns usually increases rapidly during later generations. This is referred to as bloat. Bloat generally results in degradation of system performance. Section 4.1 discusses the causes of introns and bloat and methods for reducing bloat.

A common limitation of all search methods, and hence genetic programming, is their susceptibility to converge prematurely to one or more local optimum. The causes of premature convergence are presented in section 4.2. This section also provides an account of studies conducted to prevent premature convergence in systems implementing evolutionary algorithms.

##### **4.1 Introns and Bloat**

Introns occur as part of the evolutionary process and the rapid growth of introns is referred to as bloat. Soules [SOUL98] state that bloating prevents genetic programming from successfully solving more complex problems.

According to Banzhaf et al. [BANZ98], Soule [SOUL98] and Altenberg [ALTE96] code growth results as a protective measure against the destructive effects of crossover. An increase in the destructive effects of the genetic operators results in an increase in the growth of introns. Although these introns help to reduce the destructive effects of crossover during the early and middle stages of a GP run, they grow exponentially towards the end of a run.

This exponential growth of introns is called bloat and causes the GP algorithm to stagnate. Bloat has also been referred to as fluff or structural complexity [LANG97a]. Experiments conducted by Banzhaf et al. [BANZ98] have revealed that if destructive crossover is removed, less bloat occurs.

Soule [SOUL98] describes two other possible causes of code growth, the semantics of the solution space and removal bias. Code growth results because the solution space contains a number of programs that are semantically the same but syntactically different. Larger programs with introns are semantically equivalent to smaller programs. The change in size of individuals produced as a result of crossover is assumed to be zero as the insertions and removals are expected to balance off. However, the size change in offspring is greater than zero. This is referred to as removal bias. This bias results due to a lack in symmetry of the code removed and inserted by the crossover operator.

It is evident from experiments conducted by Langdon et al. [LANG97a] that like crossover, mutation also causes bloat. Langdon et al. [LANG97a] suggest that the over fitting of a training set can also cause code growth.

According to Langdon et al. [LANG97b] there are more longer programs than shorter programs with the same fitness, thus longer programs get chosen and this leads to bloat. Langdon et al. [LANG97c] attribute the code growth to fitness-based selection. Experiments carried out by Langdon et al. [LANG97c] have revealed that the removal of fitness-based selection resulted in a removal of code growth.

Introns are pieces of code that do nothing and have no effect on the fitness of an individual. They are referred to as exons [SOUL98]. The following are examples of introns presented by Banzhaf et al. [BANZ98]:

(NOT(NOT(X)), (IF (2==1)...X), (+ X 0).

A number of different types of introns exist. Soule et al. [SOUL96] have found two types of non-functional code:

- Code which did nothing.
- Code which was not executed, e.g. an alternative in an if-statement.

Andre et al. [ANDR96] describe three types of function introns:

- Local introns - These are functions that have no effect except to pass on the values that were passed to them.
- Hierarchical introns - These are functions which contain one or more arguments that will be ignored, e.g. an if-else statement.
- Sibling (horizontal) introns - These functions undo the effect of a function represented by a sibling tree, e.g. when a subtree has its contributions to a memory location overwritten by another subtree executed after it.

Research in this area has led to the discovery of a number of ways of dealing with the problem of bloat:

- Kent [KENT97], Koza [KOZA94] and Soule [SOUL98] suggest that bloat can be reduced by using the editing operation during or after a run. According to Soule [SOUL98] such code modification is not very effective at removing bloat. He states that the problem of incomplete editing is inevitable. According to Soule et al. [SOUL96] the application of the editing operator to remove non-functional code will be able to reduce the amount of non-functional code but will not be able to remove it completely. Koza [KOZA94] proposes a method in which the best-of-run program is simplified during a post-run process.
- Andre et al. [ANDR96] are of the opinion that an increased application of the mutation operator is a solution to the problem of intron growth. Systems applying the mutation operator stagnated a lot later, the number of introns were reduced and better overall solutions were found. However, contradictory results were found by Langdon [LANG97a] which indicates that mutation causes bloat.
- Kent [KENT97], Langdon [LANG98b] and Soule [SOUL98] state that in order to reduce the amount of redundant code in programs generated, a measure of parsimony can be incorporated into the fitness function by penalizing an individual for the number of nodes that make up the individual. Soule et al. [SOUL96] have found the use of a penalty function to be the most effective means of removing bloat.

Koza [KOZA94] states that a parsimony measure can be incorporated into the fitness function. However, the problem with this is deciding what weighting to give the different objectives (i.e. parsimony and correctness) represented by the fitness function. Koza [KOZA94] is of the opinion that the removal of extraneous code too early in a run can result in the reduction of the diversity of the population. Thus, Koza [KOZA94] suggests incorporating parsimony into a fitness function late in a run or alternatively after a certain number of solutions have been found.

This is further stressed by Zhang et al. [ZHAN95] who state that the incorporation of a parsimony measure into a fitness function must be done with caution so as to prevent it from causing premature convergence of the genetic programming algorithm.

Soule [SOUL98] states that the use of a parsimony measure may degrade system performance and that parsimony pressure merely limits code growth and cannot completely remove it. Experiments conducted by Soule [SOUL98] have indicated that the use of parsimony pressure has proven to be effective in some cases and ineffective in others. The latter occurs when the use of such pressure decreases genetic diversity or there is a negative correlation between a smaller tree size and a better fitness.

- Hooper et al. [HOOP96] propose that a process called expression simplification be used to generate simpler, smaller solution programs. Expression simplification basically has the same effect as the editing operation. The simplification process involves applying a number of transformation rules to programs. The GP system implemented by Hooper et al. [HOOP96] utilizes 200 such rules. The application rate of the expression simplification operator specifies the number of individuals that will be simplified.

Experiments conducted by Hooper et al. [HOOP96] have revealed that expression simplification eliminates and simplifies bloated expressions. This results in smaller simpler programs being produced.

- Both Koza [KOZA 94], Kent [KENT97] and Zhang et al. [ZHAN95] describe the use of automatically-defined functions as a means of producing more parsimonious programs.
- Research conducted by Soule [SOUL98] has indicated that the use of a non-destructive crossover operator, i.e. an operator that only retains offspring that have a higher fitness than their parents or the same fitness as their parents, reduces code growth. Furthermore, this does not have a negative effect on the fitness of the population.

According to Banzhaf et al. [BANZ98] the increase in bloat can be used to automatically detect the end of a run. In experiments conducted by Banzhaf et al. [BANZ98] this reduced the run time by fifty percent. Langdon et al. [LANG97a] suggest that once bloat sets in it is more beneficial to abort the current run and start again.

## **4.2 Premature Convergence of the Genetic Programming System**

The genetic programming process begins with an initial population of dissimilar trees. As the evolutionary process proceeds from one generation to the next, elements of the population become more similar due to selection pressure until eventually the algorithm converges to a particular program structure, i.e. the best individual of each generation has basically the same structure and fitness. This decrease in genotypic and phenotypic diversity is referred to as genetic drift. If the program structure converged to is a solution program or a solution algorithm can easily be derived from it, then the GP system has achieved its aim. However, this does not always happen. Often the solution algorithm cannot be induced from the program structure converged to. In this case we say that the GP algorithm has converged to a local optimum or converged prematurely. Genetic programming systems using multimodal fitness functions (i.e. fitness functions with landscapes that contain many peaks, or depressions) are guaranteed to converge prematurely.

Section 4.2.1 examines the causes of premature convergence. Methodologies for escaping from local optima are presented in section 4.2.2.

### **4.2.1 Causes of Premature Convergence**

A survey of the relevant literature has revealed that there are essentially four main causes of premature convergence, namely, a lack of genetic diversity, selection noise or variance, the existence of alpha individuals and the destructive effects of genetic operators. The following sections describe each of these causes.

#### **4.2.1.1 Lack of Genetic Diversity**

Kent [KENT97] emphasises the importance of genetic diversity in a GP system. He states that in order to converge to a global optimum it is essential that the genetic diversity of the population is maintained as a loss in genetic diversity will lead to premature convergence to a sub-optimal solution. Genetic diversity refers to the variety of the population. If the population contains a number of duplicate individuals or individuals with a similar structure, the variety and hence genetic diversity of the population will be low. In such cases the population is not representative of most of the search space, and hence may not contain those components that are essential to find a solution.



In those domains where there are a large number of primitives the chosen population size may be too small to represent the search space sufficiently resulting in the GP system converging prematurely due to a lack of diversity.

Cloning is another cause of premature convergence. According to Soule et al. [SOUL96] the use of crossover together with a depth limit can result in the crossover operator merely producing clones of the parents and hence causing the GP process to stagnate. Grant [GRAN00] and Langdon et al. [LANG97a] describe cloning as a protective reaction to the destructive effects of crossover. Cloning leads to a large amount of system resources being utilized and can degrade system performance [LANG97a]. Langdon [LANG96c] describes two types of crossover that reduce the variety of a population, namely, crossover which swaps terminals and crossover which replaces whole trees. Where variety is low both these types of crossover lead to a further production of clones.

If crossover produces copies of parents at a high rate this leads to a reduction of the variety of the population and the eventual extinction of certain primitives.

Langdon [LANG96c] states that the cloning of parents is found to occur most frequently when the crossover operator is applied to trees which are short or identical to their parents. Furthermore, if the trees involved in crossover have repeated subtrees, the chance of producing clones is greater. Poli et al. [POLI98] criticise crossover of being both biased and a local search operator. Although crossover is expected to exchange an equal amount of genetic material when creating offspring, an offspring usually contains a majority of genetic material from the first parent only which could lead to cloning in later generations.

Mawhinney [MAWH00], Bersano-Begey [BERS97] and Mahfoud [MAHF95] attribute premature convergence to the lack of diversity in the population as a result of selection pressure. While fitness-based selection is essential for the convergence of an evolutionary algorithm, selection pressure often leads to convergence to a local optimum. Selection pressure usually leads to the elimination of certain primitive and subtrees from the population. In a study conducted by Langdon [LANG96c] this resulted in those primitives which were essential to find a solution disappearing from the population during the evolutionary process. Langdon [LANG96c] attributes this to the fact that a number of partial solutions with high scores existed early in the run.

Studies conducted by Langdon [LANG98b] indicate that the loss of primitives occur when there is a negative correlation between the frequency of a particular primitive and the fitness of the individual. If the selection method used is elitist, the chances of premature convergence as a result of selection pressure is very high.

#### **4.2.1.2 Selection Noise or Variance**

Mahfoud [MAHF95] describes selection noise or variance as a cause of premature convergence. As a result of the random nature of genetic programming, the populations of some runs contain the components necessary for the induction of a solution algorithm while others may not.

#### **4.2.1.3 The Existence of Alpha Individuals**

Bersano-Begey [BERS97] ascribes premature convergence to the existence of what he terms alpha individuals. Bersano-Begey describes alpha individuals to be very fit individuals which have a very low structural complexity and from which a solution algorithm cannot be evolved.

These individuals usually have a better fitness than a majority of the population. The existence of alpha individuals usually creates a negative correlation between the raw fitness and the components necessary for the induction of a solution. As a result the GP algorithm converges to these alpha individuals prematurely.

#### 4.2.1.4 The Destructive Effects of Genetic Operators

Mahfoud [MAHF95] states that the destructive effects of the genetic operators, especially crossover, can lead to premature convergence. Banzhaf et al. [BANZ98] have described crossover as being both constructive and destructive. One of the destructive effects of crossover is that it breaks up good building blocks that could form part of a solution. Another destructive effect is that it may insert a good building block into an individual that does not make proper use of it.

According to Soule [SOUL98] and Langdon et al. [LANG97b] the offspring produced by crossover usually has the same or a lower fitness than its parents. In experiments conducted by Langdon et al. [LANG97b] only 19.5% of the offspring created by crossover during early generations had the same fitness as their parents and as the evolution process proceeded the fitness of offspring produced by crossover deteriorated. Furthermore, Langdon et al. [LANG97a] state that the crossover operator does not produce offspring of improved fitness but instead produces offspring that are semantically the same although syntactically different. Crossover essentially finds a longer version of an individual already produced [LANG98b]. Experiments conducted by Banzhaf et al. [BANZ98] have revealed that crossover is destructive to offspring approximately seventy five percent of the time. The closer a tree is to a solution the more susceptible it is to the destructive effects of crossover.

### 4.2.2 Preventing Premature Convergence

The previous section identified the causes of premature convergence of evolutionary algorithms. This section discusses studies conducted to escape local optima in evolutionary algorithms. Discussion is restricted to those methodologies that are domain-independent and hence problem-independent.

#### 4.2.2.1 Maintaining Genetic Diversity

Langdon [LANG98b] and Bruce [BRUC95] suggest that the use of **higher mutation rates** will promote genetic diversity and thus possibly prevent premature convergence. A higher mutation rate will slow down the convergence of a genetic programming system and possibly introduce new, fitter individuals into the population from which a solution algorithm can be derived. However, Mawhinney [MAWH00] states that very high mutation rates will introduce large amounts of random code into the population which may stunt convergence and reduce the evolutionary process to a random search.

Bersano-Begey [BERS97] and Ryan [RYAN96] suggest that using **larger population sizes** could prevent premature convergence. The larger the population size the greater the chance of the population containing building blocks that are necessary for a solution. Mawhinney [MAH00] states that a larger population size will also reduce the chances of a single individual dominating the population.

According to Ryan [RYAN96] the **steady-state control model** has proven to maintain genetic diversity. Using inverse selection methods, the steady-state control model locates weaker elements of the population and replaces them with newly created offspring. The replacement is only performed if the offspring has a better fitness than the individual that it is replacing. This approach will to some extent reduce the destructive effects of the genetic operators if the offspring are required to be fitter than the individuals that they replace.

Langdon [LANG96c] suggests that the use of a dynamic fitness function will force the population to continually improve and **prevent the elimination of primitives**. In addition to this Langdon [LANG96c] states that the use of a number of small general purpose primitives instead of the use of highly abstract primitives will also reduce the chances of the removal of certain primitives.

Bersano-Begey [BERS97] and Ryan [RYAN96] state that **disabling the production of clones** will promote diversity and possibly prevent premature convergence. Langdon [LANG96c] states that this can be achieved by not using the reproduction operator and detecting when an offspring is identical to one of its parents and discarding such an offspring. In addition to this, an increase in selection pressure will also reduce cloning.

#### 4.2.2.2 Multiple Iterations

According to Mawhinney [MAWH00] a local optimum caused by selection variance can be escaped by performing multiple iterations for the same seed. He states that this may prove to be computationally expensive in systems in which fitness evaluation is time consuming. Beasley et al. [BEAS93] state that while multiple runs of a genetic algorithm may visit more than one region of a search space, there is no guarantee that a new area of the search space will be explored on each iteration and that the global optimum will be found.

#### 4.2.2.3 Restricted Mating

Ryan [RYAN96] reports that restricted mating has been used to prevent premature convergence. While some studies restrict mating by not allowing similar individuals to mate ([BERS93] and [RYAN96]) others require both parents to be similar ([RYAN96]) in an attempt to improve the evolutionary process.

Applying genetic operators to similar individuals localises the search and will increase the rate of convergence of the algorithm and could reduce diversity. The use of dissimilar parents will slow down convergence and maintain diversity to some extent in early generations.

#### 4.2.2.4 Parallel Population

Instead of performing a number of iterations sequentially to detect local optima, a population can be divided into a number of subpopulations which run in parallel to speed up the process. Ryan [RYAN96] and Beasley et al. [BEAS93] describe the evolution of parallel subpopulations as a method of locating more than one optimum and thus possibly the global optimum. However, there is no guarantee that each of the subpopulations will not explore the same area or converge to the same individual. In some studies these subpopulations communicate by means of interbreeding. According to Beasley et al. [BEAS93] such interbreeding reduces diversity of the entire population causing it to converge to a single structure.

#### 4.2.2.5 Preventing the Destructive Crossover

A review of the literature has indicated that the following methods can be used to overcome the destructive effects of crossover:

- The use of non-destructive operators

Mahfoud [MAHF95] describes the destructive effects of genetic operators as one of the causes of premature convergence. Non-destructive genetic operators can be used to ensure that offspring are fitter than or at least as fit as their parents [SOUL96]. These operators perform essentially the same function as the standard mutation and crossover operator with an exception that they discard offspring that are weaker than their parents and reapply the genetic operator until a fitter offspring is produced.

The application of non-destructive operators has proven to increase the number of successful runs of genetic programming systems. One of the drawbacks of these operators is that they are computationally expensive.

- Using the Macromutation Operator

Banzhaf et al. [BANZ98] and Angeline [ANGE97] describe the macromutation operator, also known as “headless chicken crossover”, as a possible improvement over the standard crossover operator. The macromutation operator selects a single parent. It then generates a new individual randomly. The selected individual is crossed over with the new individual to produce a single offspring. If the offspring has a fitness value greater than or equal to the fitness of its parent it forms part of the new population, else it is discarded.

Angeline [ANGE97] describes two variations of headless chicken crossover, namely strong headless chicken crossover and weak headless chicken. With strong headless chicken crossover the offspring that form part of the next generation are those which are created by replacing a subtree in the selected parents. Weak headless chicken crossover involves randomly selecting one offspring from each pair of offspring created.

Experiments conducted by Angeline [ANGE97] using the 6-multiplexor, the intertwined spiral, and the sunspot problems revealed that the headless chicken crossover operator performed just as well as, and in some cases better than, the standard crossover operator.

- Banzhaf et al. [BANZ98] are of the opinion that the problems experienced with the crossover operator can be attributed to the fact that there are differences between biological crossover and GP crossover. Biological crossover is only applied to two individuals that are homologous with respect to their function and structure. However, GP crossover is applied to any two individuals.

- Brood Recombination

According to Banzhaf et al. [BANZ98] brood selection reduces the destructive effects of crossover. Brood selection requires that a brood size  $N$  be specified. Two parents are selected randomly. Standard crossover is applied to the parents  $N$  times creating  $N$  pairs of offspring. The offspring are evaluated for fitness and sorted according to their fitness values.

The best two offspring are selected as the result of brood selection and the rest of the offspring are discarded. One of the disadvantages of this method is the time required to evaluate all  $2N$  offspring. Tacket [BANZ98] overcomes this problem by evaluating the offspring on a small portion of the fitness cases. This slightly degrades system performance.

- Intelligent Crossover

An intelligent crossover operator is used in the PADO system developed by Teller et al. [BANZ98]. This operator learns how to choose good crossover points. The intelligent crossover operator resulted in an increase in the performance of the PADO system.

Another form of the intelligent crossover described by Banzhaf et al. [BANZ98] involves calculating a performance evaluation for each subtree. This performance value is used to determine which subtrees to insert into other trees, and which to replace. This form of the crossover operator was found to improve the performance of the system “substantially”.

Banzhaf et al. [BANZ98] state that the result of using intelligent crossover is not as good as that obtained using brood selection.

- Context-Sensitive Crossover

According to Banzhaf et al. [BANZ98] the crossover operator is not context preserving. The application of a strong context preserving crossover operator in combination with the standard crossover operator in the system developed by D’haeeleer resulted in an improvement in system performance.

- Explicitly Defined Introns

This approach involves adding introns to an individual to reduce the destructive effects of crossover. These introns are called EDIs, explicitly defined introns, and usually take the form of real or integer values inserted between every two nodes in an individual.

- According to Langdon [LANG98b] the use of genetic libraries consisting of building blocks selected from individuals in the population will prevent these building blocks from being destroyed by crossover.
- Poli et al. [POLI98] describe a uniform crossover as a global search operator that is not biased. Experiments conducted by Poli et al. [POLI98] indicated that uniform crossover has performed better than standard crossover in certain problem domains.

#### **4.2.2.6 The Introduction of Dissimilar Individuals into the Population**

Keller et al. [KELL96] and Mawhinney [MAWH00] propose that premature convergence can be prevented by introducing new individuals into the population which are dissimilar to the existing elements of the population.

Keller et al. [KELL96] attribute the lack of diversity to the population not representing a sufficient number of genotypes that make up the genospace for a particular problem, i.e. although there may not be duplicates in the population the individuals in the population may be similar. They propose an algorithm which employs a diversity checker to determine the diversity of the population after  $n$  generations. The diversity checker ascertains how many different genotypes of the genospace are represented by the current population. It uses a structural measure for this purpose. This structural measure determines whether two individuals represent the same genotype by calculating the minimum number of edit operations, e.g. deleting a node, inserting a node, necessary to transform one individual into the other. If the diversity is below a certain threshold the diversity of the population is restored by replacing all except one occurrence of each genotype in the population with an individual representing a genotype not represented in the population.

A similar approach is taken by Mawhinney [MAWH00]. The GP system implemented by Mawhinney [MAWH00] replaces a percentage of the most similar individuals in the population by newly created individuals. The system maintains two parameters for this purpose, namely, the percentage of trees to replace and how often to perform the replacement, e.g. once every generation, once every five generations. The system uses the UNIX utility *diff* to determine how similar two programs are. The similarity of an individual is calculated by applying the *diff* utility to the individual in question and every other element of the population. The similarity index is the sum of the number of lines output by *diff* for each comparison. The system implemented by Mawhinney [MAWH00] was applied to two different problem domains. In both cases the system produced more successful runs. It was found that in certain domains the approach taken by Mawhinney [MAWH00] proved to be more computationally expensive than the standard genetic programming system. The high computational effort needed could possibly be attributed to the fact that replacement individuals are merely added to the population without their similarity to the rest of population being taken into consideration and hence a replacement individual maybe more similar to the population than the original individual.

#### 4.2.2.7 Niching Methods

Niching methods aim at enabling evolutionary algorithms to locate all possible optima by identifying all peaks or depressions in fitness landscapes of multimodal fitness functions. In doing so niching methods also assist the evolutionary algorithm from escaping local optima. According to Mahfoud [MAHF95] niches can be developed in parallel (parallel niching) or sequentially (sequential niching). Niching methods that have been used by **genetic algorithm** systems include crowding, fitness sharing and sequential niching.

Crowding implements the steady-state control model but instead of replacing the individual with the worst fitness, crowding replaces the individual which is most similar to the newly created offspring. Each offspring is compared to a subset of the existing population. The size of the subset is specified by the crowding factor (CF). The offspring replaces the element of the subset that it is most similar to it. The Hamming distance is used as a similarity measure. A drawback of crowding is the fact that the individual replaced by crowding may have a better fitness than its replacement or maybe a near-solution and contain components that are essential to the derivation of a solution. Ryan [RYAN96] and Mahfoud [MAHF95] report that crowding has not proven to be very successful at preventing premature convergence.

Fitness sharing algorithms penalise individuals that are phenotypically similar. The raw fitness of phenotypically similar individuals are replaced by a shared fitness. Each group of similar individuals that share a fitness value form a niche and each niche represents a peak or depression on the fitness landscape. Beasley et al. [BEAS93] state that fitness sharing assumes that optima are evenly spread over the fitness landscape and this may result in solutions algorithms being missed and the genetic algorithm converging to a local optimum. Furthermore, if the search space has many optima a fairly large population size is needed. According to Beasley et al. [BEAS93] if niches are too small the genetic algorithm is still susceptible to premature convergence. On the other hand if niches are too large a solution maybe missed.

Sequential niching attempts to locate all the optima of problems using multimodal fitness functions. Information from previous runs is used in the current run in order to prevent the genetic algorithm from revisiting areas of the search space.

If the best individual at the end of the run is not a solution algorithm, sequential niching applies a derating function to the raw fitness of the best individual to depress (if the fitness function is being maximised) those areas of the problem space that have already been visited. Sequential niching performs multiple runs until all optima have been located. The fitness function is modified on each iteration. During the first iteration the raw fitness is used to calculate fitness measures for each individual. During successive iterations the fitness function is modified using one of two derating functions. The derating function is defined in terms of the distance between two chromosomes and a niche radius. The distance between the chromosomes is the Euclidean distance between binary vectors. Each peak or depression is represented by a niche and the niche radius determines the boundary of the niche. The niche radius is defined in terms of the dimension of the problem space and an estimate of the number of optima. One of the problems encountered with sequential niching is the choice of the size of the niche radius. If too small a value is chosen for the niche radius, the modified fitness function will converge to a local optimum. On the other hand if too large a value is chosen a solution may be missed altogether. Other problems encountered with sequential niching include the derivation of inaccurate solutions and halting of runs prior to the convergence of the genetic algorithm.

#### **4.2.2.8 Weighting Fitness Cases**

The system implemented by Bersano-Begey [BERS97] escapes from local optima by changing the fitness landscape. The system performs half the runs using the standard fitness function. During these runs the system ascertains which fitness cases individuals have performed poorly on. The rest of the runs are performed with a modified fitness function. The fitness function is adjusted to add a bonus to those fitness cases which the population has performed poorly on. The bonus is multiplied by the number of generations for which few or no individuals have solved the particular fitness case. When applied to the 11-multiplexor problem the system produced more successful runs than the standard genetic programming system.

In problems where the error function [KOZA92] is used to calculate an individual's fitness, the fitness function is minimised and in most cases the hits ratio, i.e. the number of fitness cases for which the individual computes the correct value, is zero. Thus, in these cases a direct comparison of how the population performs on each of the fitness cases cannot be implemented. Furthermore, there may not always be a correlation between the population's performance on the fitness cases and those genotypes missing from the population that are crucial to the induction of solution algorithms.

#### 4.2.2.9 The Races Genetic Algorithm (RGA)

The research conducted by Ryan [RYAN96] is aimed at locating all the peaks (when maximising the fitness function) or depressions (when minimising the fitness function) in a search space. Ryan [RYAN96] views the points belonging to each peak or depression as belonging to different races. The search space is divided into a number of real-valued racial perfects. The number of racial perfects is equal to twice the number of estimated peaks or depressions in the search space. The fitness function used evaluates individuals with respect to correctness as well as how close the individual is to an existing racial perfect. The closeness between an individual and a racial perfect is the difference between the racial perfect and the phenotype of the individual.

During the process of initial population generation an individual is created and added to a particular race, depending on its racial perfect. The races algorithm takes a steady-state approach to evolution. Upon randomly selecting a race, an individual is probabilistically chosen from the race. The algorithm employs the meta-strategy described in [RYAN96] to determine whether the individual should inbreed or outbreed. If the individual is to inbreed an individual from the same race which also wishes to inbreed is selected, otherwise an individual of a different race which also wishes to outbreed is chosen. Each newly created offspring is added to the appropriate race. If a suitable race is not found the offspring is rejected. This process continues until the termination criterion is met.

As is the case with sequential niching one of the drawbacks of the RGA is that the number of optima in the search space must be estimated. Ryan [RYAN96] states that if there are a large number of peaks or depressions in the problem space or if the estimate of the number of optima is far off from the actual number of peaks or depressions the RGA will experience difficulties.

#### 4.2.2.10 Demes

The genetic programming paradigm has been described as “panmictic” [GRAN00], i.e. evolution is not species-based and all individuals can breed with each other. The use of demes enforces breeding between individuals that belong to the same group (deme). A small percentage of interbreeding is permitted.

Demes have been described as means of maintaining the genetic diversity of a population [LANG98b]. Demetic groupings consist of similar individuals intra-breeding in groups independently from each other. The immigration operator is used for inter-breeding between demes.

Manjunath et al. [MANJ97] describe the advantages of demetic grouping to be:

- A number of areas within a search space is searched at once. This is especially useful when trying to find solutions to difficult problems.
- The use of demes reduces the speed at which the genetic programming algorithm converges and hence prevents premature convergence of the GP system.
- Demes facilitate parallelization.

Furthermore, research conducted by Langdon [LANG95a] and [LANG98b] has revealed that the division of the population into a number of sub-populations improves the performance of the system.



Langdon [LANG95a] attributes this success to the fact that the use of demes promotes the breeding of similar individuals.

Disassortive mating is a variation of the deme approach that is also used to prevent the premature convergence of the genetic programming algorithm to a sub-optimal program [GRAN00]. Disassortive mating involves maintaining two breeding pools. One pool focuses on generating good but lengthy solutions while the other pool develops poorer more parsimonious solutions. Migration between pools is allowed.

## 5. Evolutionary Structured Programming

This section provides an account of a study conducted to investigate the evolution of procedural programming algorithms. Pringle [PRIN95] proposes such a system which he refers to as Evolutionary Structured Programming (ESP). This system has not been implemented or tested.

The ESP GP system will induce structured imperative programs. Pringle [PRIN95] criticises the standard GP system for producing solution trees that contain code that does not perform any function. The ESP system proposed by Pringle[PRIN95] will induce trees that are easily readable.

The proposed ESP language contains the following constructs which promote the generation of modular, structured programs:

- **STATEMENT**

This is equivalent to a single programming statement in a programming language, e.g. an assignment statement, a subroutine call, or a call to any of the other ESP constructs.

- **BLOCK**

Consists of a number of programming statements in sequential order.

- **IF *boolean block***

This is equivalent to the IF-statement in most imperative programming languages. If the boolean statement evaluates to true the statements in the *block* component are executed.

- **IFELSE *boolean block1 block 2***

If the boolean expression evaluates to true the statements in block1 are executed else the statements in *block2* are executed.

- **CASE *casevar range1 block1...,ranken blockn***

This statement is equivalent to the case or switch statements used in imperative programming languages. If casevar is in rangei then block i will be evaluated, where i = 1,...,n. Casevar is a scalar.

- WHILE *boolean block*

Is equivalent to the while loop in imperative programming languages.

- LOOPTHRU *array loopvar block*

Is a loop used specifically for traversals through arrays.

- IFCONTINUE *boolean*

Is called from within loops. A non-zero value for *boolean* results in the loop being terminated and restarted. If this happens within the LOOPTHRU loop the *loopvar* is incremented.

- IFBREAK *boolean*

Is called from within loops. If *boolean* is a non-zero value the loop is terminated and the statement directly after the loop is executed.

- FUNCTION *type name model type1 argument1,..., moden typen argumentn*

*type* - refers to the return type of the function

*name* - refers to the name of the function

*model...moden* - specifies whether the variable is an input or output variable

*argument1...argument2* - names of the arguments

*type1...typen* - specifies the type of each of the arguments

Pringle's [PRIN95] system will cater for the following data types: integer, floating point, character, structure, void(only used to describe the return type of a function).

The system will allow the user to define a set of global variables, constants and a set of functions that can be used by the genetic operators and the procedure that generates the initial population. Block statements can contain other block statements and any of the other ESP constructs. Functions can be defined to contain both static and dynamic local variables. Functions can contain constants as well.

According to Pringle [PRIN95] human programmers do not program by making random changes to their program, but instead use certain rules of thumb. He describes how these rules of thumb can be incorporated into the system by assigning context-sensitive selection probabilities to the different constructs. These probabilities will change from context to context.

As with the standard GP system the initial population will be generated randomly. Each individual will consist of a main program and a number of functions. Some individuals may not include functions. The main program may include constants, global variables, and any number of the function calls. Pringle [PRIN95] describes both the main program and functions as consisting of a single block statement. The length of each block will be randomly generated.

Crossover involves choosing two groups of statements within each of the parents. Offspring are created by swapping the two sets of statements. Both the offspring are checked for syntax errors.

The mutation operator is defined to be similar to that applied by Teller [TELL94b]. The mutation operator performs “small” changes such as converting  $<$  to  $<=$ . This operator can also insert program statements into or delete programs statements from the program trees.

Pringle [PRIN95] deals with the halting problem by defining a maximum execution time for program runs. Programs that have not completed their execution will be terminated at the end of this time threshold. However, the fitness function will still be applied to such programs.

Pringle [PRIN95] claims that due to the “structured nature” of programs induced by the ESP system the solution program will be easier to understand and interpret. Furthermore, it will be easy to convert the programs induced by ESP to imperative programming languages such as C.

## Chapter 4 - Methodology

This chapter describes the methodology employed to meet the objectives listed in section 2 of **Chapter 1**. Section 1 provides a summary of the computer science research methodologies commonly used and identifies the methodology that is most appropriate for the study presented in the thesis. Application details of the chosen methodology are presented in section 2. This methodology requires a genetic programming system to be implemented, tested and revised if necessary. The implementation details of the genetic programming system are listed in section 3.

### 1. Research Methodologies

In the papers entitled “What is Research in Computing” [JOHNa] and “Basic Research Skills in Computer Science”[JOHNb] Johnson identifies four types of research methodologies commonly used in computer science, namely, proof by demonstration, empiricism, mathematical proof techniques, and hermeneutics or observational studies.

**Proof by demonstration** essentially involves iteratively improving the performance of a computer system in order to answer a research question. The reasons identified for failure on one iteration form input to the refinement process of the next iteration. If the system is still unable to induce solutions after numerous steps of refinement reasons for this failure must be identified.

**Empiricism** is used to test a specific hypothesis, e.g. comparison between two information retrieval engines. A methodology needs to be defined for this purpose. Conclusions are drawn from the results of implementing the methodology in order to support or refute the proposed hypothesis. It may be necessary to utilize statistical methods to determine the significance of the conclusions arrived at. **Mathematical proof techniques** use formal mathematical methods to either prove or refute a hypothesis. **Hermeneutics** entail observing and evaluating a computer system, developed for purposes of the study in question, in the working environment that it is intended for.

The main aim of the study presented in this thesis is to determine whether genetic programming can induce solutions to novice procedural programming problems. The process of determining whether a genetic programming system is capable of generating solutions to a particular problem entails identifying the GP parameters for this purpose ( Grant [GRAN00] and Langdon [LANG98b]). This is done by implementing a GP system with an initial set of parameters and varying the parameters in an attempt to obtain a system that is capable of generating solutions. Thus, the **proof by demonstration** methodology appears to be the most suitable for the study at hand. In order to meet the objectives outlined in section 2 of **Chapter 1**, in addition to varying the GP control parameters it may also be necessary to make changes to the overall functioning or architecture of the GP system, i.e. the standard and advanced features, the programming problem specification, the internal representation language and the methodologies employed to prevent premature convergence. The following section describes the details of this methodology with respect to the problem presented in this thesis.

## 2. The Proof by Demonstration Methodology

The previous section identified the proof by demonstration methodology as the most appropriate for this study. In this study only one refinement iteration will be performed if necessary. The steps involved in applying this methodology are:

- Develop and implement a genetic programming system for the induction of novice procedural solution algorithms.
- Test this system on the randomly chosen set of novice procedural programming problems listed in **Appendix A**. These problems are placed into the following categories: sequential, conditional, iterative, recursive and ASCII graphics.

Sequential problems require the use of arithmetic operators and simple character and string manipulation. Conditional problems require the use of the standard conditional control structures, namely, **if-then-else** statements and/or **case/switch** statements in the solution algorithms. The iterative category of problems are problems which test the student's knowledge of the standard iterative control structures, e.g., the **for**, **while** and **dowhile** looping constructs. The recursive problems require solution algorithms to be recursive. Finally, ASCII graphics problems are commonly used in introductory programming courses to test students' understanding of simple input and output and iteration. In this study the group of ASCII graphics problems require the use of iteration and nested iteration.

- Due to the randomness associated with genetic programming, the system may not find a solution as a result of the random choices made. Thus, the system will be tested using ten different random number generator seeds for each problem. If the system is unable to evolve a solution for at least one seed for each problem, changes will be made to one or more of the following:
  - ▶ The internal representation language.
  - ▶ The standard genetic programming parameters and features. These include the control model, program representation, method of initial population generation, application rates of the genetic operator, the selection method and associated parameters, calculation of the fitness measure.
  - ▶ Parameters of the advanced genetic programming features implemented as part of the system. The parameters will be defined in **Chapter 5**.
  - ▶ Advanced GP features - The GP system may need to be extended by incorporating the use of additional advanced features such as the use of cultural learning.
  - ▶ Methods for overcoming limitations of the GP algorithm that may prevent the system from finding a solution - The mechanisms built into the initial system may not be sufficient and additional methodologies may need to be identified or developed.
- Test the revised system and report on the results obtained. If the system was unable to induce solutions to any of the problems, discuss the reasons for this failure.

The following section describes the implementation details of the genetic programming system.

### **3. Implementation Details**

In order to test the proposed GP system a prototype of the system was developed in the Professional version of JBuilder 4.0, using JDK 1.3. The random number generator provided by Random class in the JBuilder library was used for purposes of random number generation. This random number generator implements the linear congruential method to modify the initial seed.

The simulations in this study were run on two different types of systems, namely, a Pentium III with Windows XP and a Pentium 4 with Windows 2000. Implementation details of the test simulations performed can be found in **Appendix B**.

### **4. Summary**

This chapter describes the methodology employed to investigate the use of genetic programming as a means of inducing novice procedural solution algorithms. The following chapter proposes a genetic programming system for the evolution of such problems. The results of applying this system to the problems listed in **Appendix A** are discussed in **Chapter 6**.

## Chapter 5 - The Proposed Genetic Programming System

This chapter proposes a genetic programming system for the induction of solutions to novice procedural programming problems. [PILL01], [PILL02] and [PILL03a] describe the proposed system and some of the results obtained. The format of the programming problem specification, which forms the input to the genetic programming system for each problem, is defined in section 1. In order to facilitate the translation of the algorithms derived by the GP system into a programming language, the genetic programming system implemented in this study is strongly-typed. Details regarding this typing are presented in section 2. Section 3 describes the structure used to represent each individual. An account of the control models utilized by the system is provided in section 4. Section 5 discusses the process used to generate the initial population. In order to ensure the reuse of the expert module of the generic architecture proposed in **Chapter 2**, solution algorithms will be generated in an internal representation language. Section 6 describes the terminals and functions that will form the internal representation language. A fitness measure needs to be calculated for each individual. The fitness functions used for this purpose are examined in section 7. Section 8 discusses the selection method employed by the system. The genetic operators implemented by the system are described in section 9. It is evident from the discussion presented in section 4.2 of **Chapter 3** that a genetic programming system may not find a solution if it converges prematurely. Mechanisms that have been built into the system to escape local optima are discussed in section 10. According to Koza et al. [KOZA99a] one of the preparatory steps that must be performed when implementing a GP system is the determination of major and minor system parameters to be used by the system. Section 11 specifies these parameters for the GP system presented. A brief summary of the chapter is provided in section 12.

### 1. Programming Problem Specification

For each problem, the input to the genetic programming system is a problem specification. Each problem specification contains the following information:

- A description of the input to the problem – A variable is used to represent each input to the problem. The type and source of each variable must be specified. The source of the input can be the keyboard, a file, or memory (i.e. a variable).
- A description of the output of the problem – A variable describing each problem output must also be provided. Both the type and the destination (i.e. the screen, a file, or memory) of the variable must be specified.
- A set of fitness cases – Each fitness case provides a value (or list of values) for each input variable and the corresponding output values for each output variable defined.
- Details regarding the application domain – In order to write a program for a specific application domain, a programmer must have a knowledge of that domain, e.g. to write a program that converts years to months the programmer must know that there are twelve months in a year. Such domain-specific knowledge needs to be made accessible to the genetic programming system. This knowledge takes the form of constant values in the program specification.
- Screen output – This must be specified in the case of ASCII graphics problems. Each screen output specification contains a set of x and corresponding y values that correspond to positions on the screen, and the character found at each position.

- The function set that must be used. From the discussion presented in section 2.3.4 of **Chapter 3** it is evident that extraneous functions result in degradation in the performance of a GP system and usually lead to the system not finding a solution. Thus, the function set will be a subset of the internal representation language. This subset will consists of those functions that a student should have knowledge of to solve the particular problem.

**Table 5.1.1** tabulates the programming problem specification for the factorial problem while **Table 5.1.2** defines the programming problem specification for the ASCII graphics problem which requires the right-angled triangle illustrated in **Figure 5.1.1** to be displayed on the screen.

|                          |   |
|--------------------------|---|
| Input variables          | N   |
| Types of input variable  | Integer                                     |
| Source of input variable | Screen                                      |
| Input values for N       | 0, 1, 2, 3, 4, 5                            |
| Constants                | one : 1                                     |
| Target values            | fact  |
| Target type              | Integer                                     |
| Target destination       | Screen                                      |
| Target values for fact   | 1, 1, 2, 6, 24, 120                         |
| Function set             | +, -, *, /, if, for, <, >,<br><=, >=, =, != |

**Table 5.1.1:** Problem specification for the factorial problem

```

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

**Figure 5.1.1:** ASCII  
Graphics Problem



|                                   |                              |
|-----------------------------------|------------------------------|
| <b>x-coordinates</b>              | 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 |
| <b>y-coordinates</b>              | 1, 1, 2, 1, 2, 3, 1, 2, 3, 4 |
| <b>Character at each position</b> | *                            |
| <b>Constant/s</b>                 | ch: *                        |
| <b>Constant/s type</b>            | Char                         |
| <b>Function set:</b>              | place, block2, block3, for   |

**Table 5.1.2:** Problem specification for an ASCII Graphics Problem

## 2. Strongly-Typed GP

According to Bruce [BRUC95] a genetic programming system should be strongly-typed in order to facilitate the direct translation of the algorithms generated by the genetic programming system into a programming language. The genetic programming system implemented in this study is required to generate solutions to novice procedural programming problems which will eventually be translated into a specific programming language and thus programs generated by the GP system must have a legal structure, e.g. the condition of an if-statement cannot be the sum of two numbers. Strongly-typed GP systems implement structure-preserving genetic operators and methods of initial population generation. Thus, in order to ensure the evolution of structurally correct individuals, the system implemented in the study presented in this thesis is strongly-typed.

Each terminal, constant, memory location, operator and operator argument is of a specific type. The types catered for by the system are Integer, Real, Boolean, Char, and String. The Integer type is defined as a subtype of the Real type. Thus, during the process of initial population generation a node of the type Integer can be inserted wherever a node of the type Real is needed. Similarly, the mutation operator can replace a chosen subtree of type Real with a newly created subtree of the type Real or Integer. In order to cater for the induction of solutions to ASCII graphic problems, the type Output is also defined. Any primitive of type Output does not return a value but updates the screen maintained by the system. Details regarding this screen output is presented in section 6.7.

The function set and terminal set are represented as a collection of subsets, one for each type. When a node of a particular type is needed during tree creation, an element is randomly selected from the terminal or function subset corresponding to that type. When a tree is created, the root of the tree is chosen to be of the same type as the output that it must generate, e.g. if the system is required to generate an algorithm to calculate the factorial of a given positive integer, the root of all the trees will be of the type Integer.

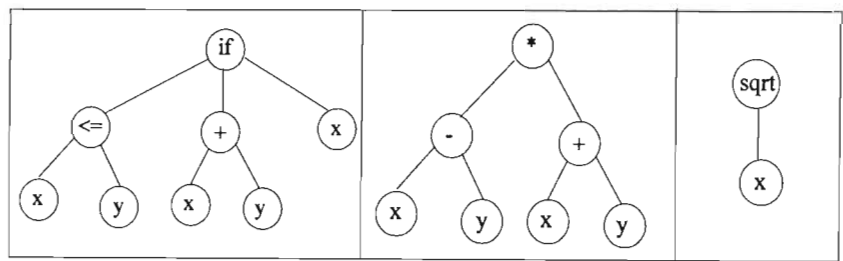
Some operators and operator argument types are defined to be generic and are only instantiated during the initial population generation process. Haynes [HAYN98] and Banzhaf et al. [BANZ98] describe the advantage of using generic types to be the elimination of the need to define an operator multiple times to perform the same task for different types (discussed in section 3.3 in **Chapter 3**). For example, the type of an instance of the *if* operator and its second and third arguments are only instantiated once a subtree representing its second argument is generated. Similarly, the type of the *for* operator is only determined when its third argument has been induced.

The procedures for initial population creation, mutation and crossover have been implemented so as to facilitate typing. The following section describes the structure used to represent each individual.

### 3. Program Representation

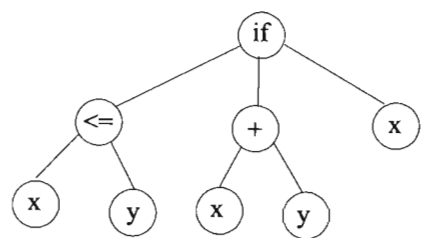
It is evident from the discussion presented in section 2.4 of **Chapter 3** that a number of different structures have been used to represent an individual in genetic programming populations. According to Langdon [LANG98b] the most commonly used representation is a parse tree. The AIMGP system uses a linear genome to represent each program [BANZ98] while the PADO system implemented by Teller et al. [TELL96] uses graphs. Other representations include directed acyclic graphs [EHRE96], stack representation [PERK94] and the multi-tree representation described by Bruce [BRUC95] and Langdon [LANG98b].

Novice procedural programming solution algorithms often have more than one output. For this reason each individual is represented using the multi-tree genome. The length of the genome is problem dependent. For example, if the GP system is required to generate an algorithm that has three outputs, the length of the genome will be three. Each gene is represented by a parse tree, one for each output. An example of this genome is illustrated in **Figure 5.3.1**.



**Figure 5.3.1:** Multi-tree representation of a single individual with three outputs

In the case where the algorithm to be derived has a single output, the multi-tree genome is reduced to single parse tree as illustrated in **Figure 5.3.2**.



**Figure 5.3.2:** A parse tree presenting an individual with one output

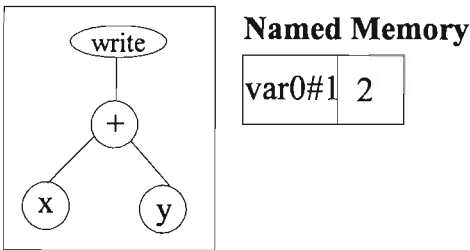
Koza et al. [KOZA99a] describe the most efficient representation of a parse tree to be a vector storing the tree in Polish notation. In this study each individual is thus represented as an array of parse trees with each parse tree stored in prefix notation using a vector. The corresponding vector notation of the individual in **Figure 5.3.2** is illustrated in **Figure 5.3.3**.

|    |    |   |   |   |   |   |   |
|----|----|---|---|---|---|---|---|
| 0  | 1  | 2 | 3 | 4 | 5 | 6 | 7 |
| if | <= | x | y | + | x | y | x |

**Figure 5.3.3:** The vector representation of a parse tree

According to Koza et al. [KOZA99b] generally computer programmers use internal memory to store immediate results and thus prevent re-calculation. Hence, each individual has its own memory structure. An individual may contain zero or more named memory locations. **Figure 5.3.4** illustrates an individual with one memory location. The 0 in *var0#1* refers to the number of the individual in the population, in this case the first individual, and the 1 refers to the number of the variable in the particular individual.

An individual’s memory may also contain named memory locations accessed by an instance of a particular operator, e.g. the *for* operator. Section 6.4 provides a detailed account of the use of memory by the GP system.



**Figure 5.3.4:** An individual and its corresponding memory structure given that x=1 and y=1

Each parse tree is constructed by randomly choosing elements from the terminal and function sets. The function set together with the terminal set essentially forms the internal representation language in which solution algorithms will be expressed. Section 6 examines the terminal and function sets utilized by the genetic programming system.

4. Control Model

From the research surveyed it is evident that the control models commonly implemented by genetic programming systems are the generational and steady-state control models. While the generational control model is easy to implement [BENN96], the steady-state control model has been described as being less susceptible to the problem of premature convergence [KENT97] and requires less systems resources [MANJ97]. However, a review of the literature has revealed that the most suitable model to use is problem dependent.

Thus, both these models will be available for use by the system. Inverse tournament selection will be used for the purpose of identifying replacement individuals when implementing the steady-state control model.

## 5. Initial Population Generation

Koza [KOZA92] proposes three methods for generating the initial population, namely, the full method, the grow method and the ramped half-and-half method. Studies conducted by Koza [KOZA92] have revealed that the ramped half-and-half method has had the best performance over a wide spectrum of problems. Furthermore, Banzhaf et al. [BANZ98] describe the ramped half-and-half method as a means of promoting genetic diversity in the population, and thus possibly preventing premature convergence of the GP algorithm, while the full and grow methods can result in a population of uniform individuals being produced. However, in the studies conducted by Poli et al. [POLI97] the results were significantly better when the full method was used. Hence, the best method to use for the purpose of initial population generation is problem dependent. The GP system implemented in this study caters for all three methods.

According to Langdon [LANG98b] the performance of a GP system can be improved by including shorter trees in the initial population. Thus, an initial depth bound is set on each tree. This value is problem dependent. Furthermore, consistent with the study conducted by Koza [KOZA92], the root of each tree is randomly selected from the function set in order to ensure that trivial trees are not generated.

The process of initial population generation caters for strong-typing. Each function argument is defined to be of a specific type. A child node is chosen by randomly selecting a node from a set of primitives of the correct type, i.e primitives of the same type or subtype of the specified argument. If the node is to be inserted at a level less than the maximum permitted depth, the generic operators<sup>15</sup> are also included in this set. If a generic operator is randomly chosen as the child, it is instantiated to be the type defined for that particular argument. The root of the tree is randomly selected from the set of functions including the generic operators and those operators that are of the same type as the output the tree represents. Upon selecting a node of the correct type, two checks are performed to ensure that the choice of this node at the particular position will result in a legal tree being eventually formed. The first check is performed if one more level of the tree is left to be created. This check determines whether a terminal node of the correct type exists for each argument of the selected node. If one or more terminal nodes of the correct type do not exist a new node is selected. The process of re-selection is continued until a legal node is found. The second check is performed if the method used to generate a new individual is the full method and the number of levels of the tree still to be developed is greater than one. The aim of this check is to determine whether a function node of the correct type for each of the arguments of the chosen node exists. If not, the node is discarded and the process of selection is repeated until a suitable node is found.

The following section describes the internal representation language.

---

<sup>15</sup>A generic operator is an operator that is defined to be of type "generic". This means that during initial population generation an instance of this operator can be instantiated to be of type Integer, Real, String, Character, Boolean, or Output.

The language is comprised of functions and terminals that are needed to represent the introductory programming concepts outlined in section 2 of **Chapter 1**.

**6. Primitives**

The terminal set consists of variables representing the inputs to the problem and constants described in the programming problem specification. The terminal set used differs for each problem.

The elements of the function set utilized by the genetic programming system have been chosen so as to facilitate the induction of solutions to procedural programming problems covering introductory programming concepts. From the discussion presented in section 2.3.4 of **Chapter 3** it is evident that the inclusion of extraneous elements in the function set results in system degradation and can possibly cause the system to be unsuccessful. Thus, a subset of these elements is used to solve a particular problem. Each category of operators is described below.

**6.1 Arithmetic Operators**

**Table 5.6.1.1** lists the arithmetic operators catered for by the genetic programming system and their corresponding arities, argument types and return types.

| Operator | Arity | Argument Types                   | Return Type     |
|----------|-------|----------------------------------|-----------------|
| +        | 2     | Real or Integer, Real or Integer | Real or Integer |
| -        | 2     | Real or Integer, Real or Integer | Real or Integer |
| *        | 2     | Real or Integer, Real or Integer | Real or Integer |
| /        | 2     | Real or Integer, Real or Integer | Real            |
| sqrt     | 1     | Real or Integer                  | Real            |
| %        | 2     | Real or Integer, Real or Integer | Real or Integer |
| neg      | 1     | Real or Integer                  | Real or Integer |
| sq       | 1     | Real or Integer                  | Real or Integer |
| cube     | 1     | Real or Integer                  | Real or Integer |
| pow      | 2     | Real or Integer, Real or Integer | Real or Integer |
| trunc    | 1     | Real                             | Integer         |
| round    | 1     | Real                             | Integer         |
| ceil     | 1     | Real                             | Integer         |
| abs      | 1     | Real or Integer                  | Real or Integer |

**Table 5.6.1.1:** Arithmetic Operators



The *neg* operator negates its argument. The *pow* operator calculates  $x^y$  where  $x$  is its first argument and  $y$  its second. The *neg*, *trunc*, *sq*, *cube*, *round*, *ceil* and *abs* operators have an arity of one and the other arithmetic operators have an arity of two. The return type of the */* and *sqrt* operators is Real. The *trunc*, *ceil*, and *round* operators have an Integer return type. The return types of all the other arithmetic operators can be Integer or Real depending on the types of the arguments of these operators. The arguments of all the operators except the *trunc*, *ceil* and *round* operators can be either Integer or Real. The argument of the *trunc*, *ceil*, and *round* operators must be Real.

According to Koza [KOZA92], it is essential that the function set satisfies the closure property. Koza [KOZA92] and Langdon [LANG98] propose two methods of ensuring that the closure property is met, namely, protected functions and strong-typing. Protected functions return either a value of one or an error message if one or more of the operands of the operator is illegal. The system presented in this thesis uses protected arithmetic functions which perform the following checks to ensure that the closure property is satisfied:

- If the divisor of the division operator is zero an error message, denoted by *#error*, is returned.
- If the operand of the square root operator is negative an error message is returned.
- If the operand/s of any of the operators is infinity or negative infinity the operand is replaced with the largest positive number permitted or the smallest negative number allowed respectively.
- If the power operator does not evaluate to a valid real number an error message is returned.
- If the operands passed to any operator are error messages, an error message is returned.

If an operator attempts to perform an illegal operation an error message is propagated through to the method that calculates the fitness measure of the individual and the individual is penalized. This is discussed in more detail in section 7.

## 6.2 String Manipulation Operators

**Table 5.6.2.1** lists the string manipulation operators included in the internal representation language. The *concat* operator takes two arguments of the type String and returns a concatenation of these arguments which is also of the type String. The *concatc* operator returns a copy of its first argument with the character specified by its second argument added on to the end. The *length* operator returns the length of its String argument and thus has an Integer return type. The *copy* argument returns a copy of its String argument and hence its return type is also String. The *insert* operator inserts its third argument, a substring of the type String, at the position specified by its second Integer argument, in its first String argument. The *del* operator returns a copy of its first argument with the character at the position specified by its second argument deleted. Finally, the *delete* operator deletes the substring which begins at the position specified by its second Integer argument and terminates at the position specified by its third Integer argument, from its first String argument. The *charat* operator returns the character at the position specified by its second argument in its first argument. If the argument passed to the *upcase* operator is a capital letter, this operator returns this argument otherwise it converts the argument to a capital letter before returning it.

| Operator | Arity | Argument Types           | Return Type |
|----------|-------|--------------------------|-------------|
| length   | 1     | String                   | Integer     |
| concat   | 2     | String, String           | String      |
| concatc  | 2     | String, Char             | String      |
| del      | 2     | String, Integer          | String      |
| delete   | 3     | String, Integer, Integer | String      |
| insert   | 3     | String, Integer, String  | String      |
| copy     | 1     | String                   | String      |
| equal    | 2     | String, String           | Boolean     |
| charat   | 2     | String, Integer          | Char        |
| upcase   | 1     | Char                     | Char        |

**Table 5.6.2.1:** String Manipulation Operators

Once again, in order to ensure that the closure property is met by the function set, the following checks are performed. If the insertion point passed to the *insert* operator is invalid, the error message is returned as the result of the operation. If any of the deletion indexes are invalid for the *delete* or *del* operator or an error message is received as an operand by any of the operators an error message is returned as the result of the operator. If the integer operands are Infinity or -Infinity, the operand value is converted to the maximum integer value permitted and minimum integer value permitted respectively.

### 6.3 Logical Operators

Conditional structures such as the *if* operator as well certain iterative structures such as the *while* and the *dowhile* loops execute conditional checks. Logical operators are usually needed to perform such checks. Thus, the function set used by the genetic programming system in this study contains the logical operators listed in **Table 5.6.3.1**. These logical operators are only added to the function set if the subtree to be generated has to be Boolean.

All of these operators except the *not* operator has an arity of two and return type of Boolean. The *cequal* operator takes two Char arguments and returns a value of true if these values are equal and false otherwise while the *cnoteq* operator returns a value of true if its two Char arguments are not equal else it returns a value of false. Similarly, the *bequal* operator returns a value of true if both its Boolean arguments are equal and value of false otherwise while the *bnoteq* operator returns a value of true only if both its Boolean arguments are not equal.

In order to ensure the closure property is satisfied, the following checks are performed:

- If any of the operands received by the operators are error messages an error message is returned.
- If any of the integer operands is Infinity it is replaced by the maximum integer value permitted.

- If any of the integer operands is -Infinity it is replaced by the minimum integer value permitted.

| Operator | Arity | Argument Types                   | Return Type |
|----------|-------|----------------------------------|-------------|
| = =      | 2     | Real or Integer, Real or Integer | Boolean     |
| !=       | 2     | Real or Integer, Real or Integer | Boolean     |
| <=       | 2     | Real or Integer, Real or Integer | Boolean     |
| >=       | 2     | Real or Integer, Real or Integer | Boolean     |
| <        | 2     | Real or Integer, Real or Integer | Boolean     |
| >        | 2     | Real or Integer, Real or Integer | Boolean     |
| cequal   | 2     | Character, Character             | Boolean     |
| cnoteq   | 2     | Character, Character             | Boolean     |
| bequal   | 2     | Boolean, Boolean                 | Boolean     |
| bneq     | 2     | Boolean, Boolean                 | Boolean     |
| not      | 1     | Boolean                          | Boolean     |
| and      | 2     | Boolean, Boolean                 | Boolean     |
| or       | 2     | Boolean, Boolean                 | Boolean     |

**Table 5.6.3.1:** Logical Operators

## 6.4 Memory Manipulation Operators

Section 3.1.1 of **Chapter 3** describes the different types of memory that have been used by genetic programming systems. The proposed genetic programming system uses memory for the following purposes:

- To allow a program access to memory for internal variables. Named memory is used for this purpose. A program can access memory using the *write* and *change* operators defined below as well as the variable name assigned to the memory location.
- If the source of an input variable is defined to be “memory” in a problem specification, a named memory location is instantiated in each individual’s memory structure and the corresponding variable name is added to the terminal set. The *write* and *change* operators and the variable name are used to access each memory location. During evaluation of each program the fitness case values corresponding to the input variable are written to the corresponding memory location.
- Associated with each iterative control structure instance are two variables, namely, an iteration variable and a counter variable. Both these variables are named memory locations and are automatically updated on each iteration of the iterative control structure instance. These memory locations are accessed via the name of the memory location.



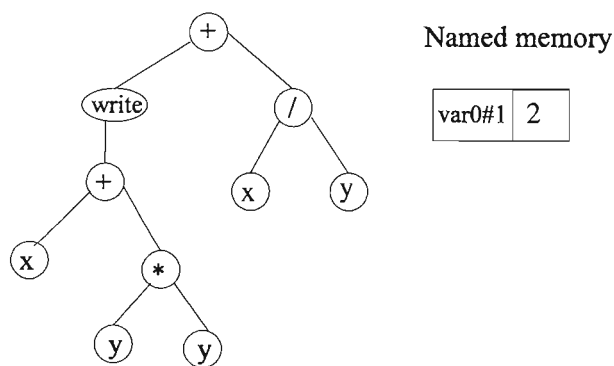
- In some problems the value of the input variable for each fitness case may be a list of values instead of a single value. This is usually the case with problems that test a student’s understanding of the iterative control structures, e.g. summing a set of numbers. In such cases the genetic programming system uses indexed memory to store the list of values. The *aread* operator is used to access each memory location of the indexed memory. During evaluation of each program, the list of values for the for each fitness case corresponding to the input variable, is written to indexed memory.

Section 6.4.1 discusses the operators used to access named memory, while section 6.4.2 describes the indexed memory operators.

6.4.1 Named Memory Operators

Named memory is the equivalent representation of a variable in an imperative program. The operators, *write* and *change* are available to access a named memory location. When the *write* operator is added as a node to a parse tree during the process of initial population generation, a memory location of the specified type, i.e. Boolean, Char, Integer, Real or String, is instantiated in the individual’s memory structure. A variable name corresponding to the location is added to the terminal set of the individual currently being created.

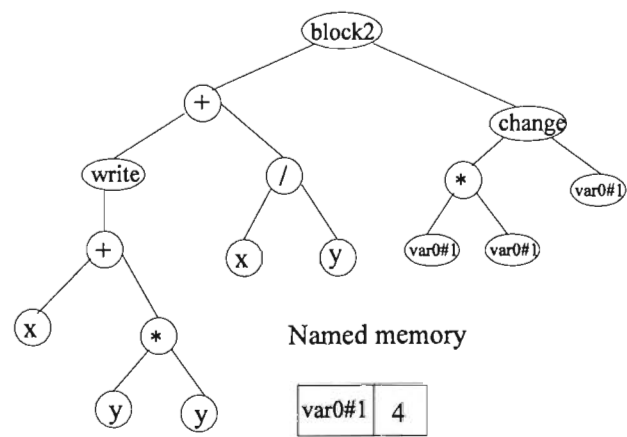
The *write* operator takes a single argument. During the execution of the parse tree, the result of evaluating the argument of the *write* operator is written to the memory location referenced by the variable linked to the particular instance of the *write* operator. The *write* operator also returns this value. **Figure 5.6.4.1.1** illustrates an individual containing an instance of the *write* operator and its corresponding memory structure. The type of an instance of the *write* operator is only instantiated during the process of initial population generation when the subtree representing its argument is generated, i.e. the *write* operator is generic.



**Figure 5.6.4.1.1:** Using the *write* operator. Assume that x=1 and y=1.

The *change* operator takes two arguments, namely, a first argument of any type, and a second argument which is a variable name of an allocated memory location. The *change* operator is only added as a node to a parse tree if memory locations of the same type as its first argument already exist, i.e. the individual in question already contains a *write* node. During execution of the parse tree the value stored at the memory location specified by the second argument of the *change* operator is replaced by the result of evaluating its first operator.

The *change* operator also returns the original value stored. The type of an instance of the *change* operator is only instantiated during the process of initial population and is the same as that of its first argument. **Figure 5.6.4.1.1** illustrates the effect of adding a branch containing an instance of the *change* operator to the tree in **Figure 5.6.4.1.2**. Note that in this case the value returned by the *change* operator will be 2.



**Figure 5.6.4.1.2:** Using the *change* operator

Checks are performed during crossover and mutation to ensure that the second child of the *change* operator is always a variable.

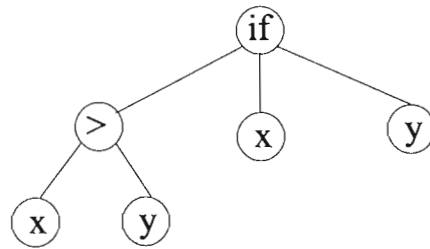
**6.4.2 Indexed Memory Operators**

The *aread* operator is used to access indexed memory. *Aread* takes one argument, an integer index to the indexed memory and returns the value stored at this index. In order to ensure that the closure property is met by the function a value of one is returned if the memory location referenced by *aread* operator is not accessible. If an input variable in the program specification requires the use of indexed memory the function *alen* is added to the terminal set. *Alen* evaluates to the number of elements in the input list. During the evaluation process, prior to the execution of a parse tree on a particular fitness case, the indexed memory of the individual is initialized to store the values of the corresponding input variable specified in the fitness case.

**6.5 Conditional Control Structures**

Three conditional control operators, namely, *if*, *switchi*, and *swtichc* are provided by the genetic programming system.

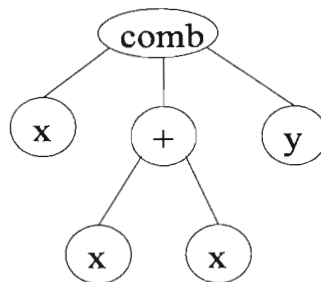
The *if* operator performs the function of an if-else statement and thus takes three arguments. The first argument represents a condition and is of the type Boolean. The second and third arguments can be of any type. The type of these arguments are instantiated during the process of creating the initial population. Both these arguments will be instantiated to the same type. If the first argument of the *if* operator evaluates to true the result of evaluating its second argument is returned otherwise the result of evaluating its third argument is returned. **Figure 5.6.5.1** illustrates a parse tree that uses an instance of the *if* operator.



**Figure 5.6.5.1:** Parse tree that calculates the maximum of two numbers

The *switchi* and *switchc* operators perform the same function as a switch statement for integer and character values respectively. These operators perform the same function as the CASE statement proposed by Pringle [PRING95] (section 5 of Chapter 3).

The first argument of the *switchi* operator represents the value to be tested against  $n$  options and is of type integer. This argument cannot be a constant. The next  $n$  arguments represent these options. Each option is represented as list of integer values with the *comb* connector combining the list. The arity of the *comb* connector is randomly selected in the range of 1 to the maximum value specified. In a *switchi* operator instance the *comb* operator is an integer while in a *switchc* operator instance it is a character. **Figure 5.6.5.2** illustrates an example subtree.

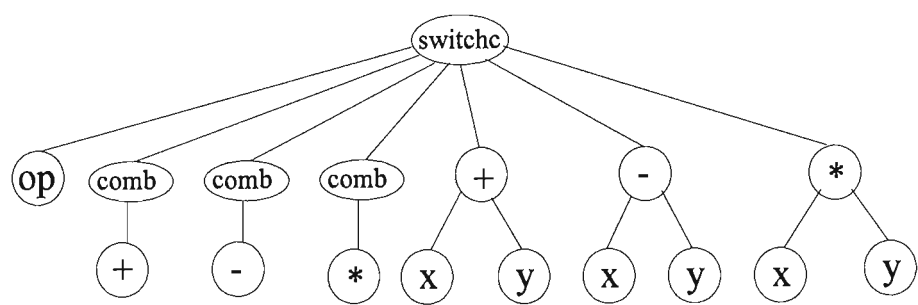


**Figure 5.6.5.2:** Represents a list of options in a *switchi* instance

The number of options,  $n$ , tested by a particular instance of this operator is chosen randomly, where  $3 \leq n \leq 7$ . It was felt that an upper bound of 7 will be sufficient for novice procedural programming problems. The GP system also randomly chooses whether a particular instance of the *switchi* or *switchc* operator should have a default option. Hence, the arity of each *switchi* operator will be  $n*2+1$  or  $n*2+2$  (if a default option exists).

The last  $n$  (or  $n+1$ ) arguments of the *switchi* operator represent the corresponding actions that will be performed if at least one value in the corresponding option list is equal to the first argument. The types of these arguments are instantiated during the initial population generation process. These arguments are all of the same type. The *switchc* operator performs the same function as that performed by the *switchi* operator for character values.

In order to ensure that the closure property is met, if none of the cases are satisfied and a default option does not exist in a particular instance, the *switchi* operator returns a one and the *switchc* operator returns the empty character. If a value in more than one option list is equal to the first argument of the *switchi* or *switchc* operator an error message is returned. **Figure 5.6.5.3** displays an instance of the *switchc* operator.



**Figure 5.6.5.3:** An instance of the *switchc* operator

The maximum arity of the *switchi* and *switchc* operators, denoted by *switch\_max*, and the maximum arity of the *comb* connector, *comb\_max*, are problem dependant and thus added to the set of GP parameters.

If one of the subtrees chosen during crossover has the *comb* connector as the root and the other does not, the operation is abandoned and new crossover points are chosen. Similarly, if the subtree chosen to be replaced during mutation has the *comb* connector as a root, the newly generated subtree also has a *comb* connector, possibly of a different arity than that being replaced, as a root.

**6.6 Iterative Control Structures**

Section 3.1.3 of **Chapter 3** provides an account of the iterative structures that have been used in genetic programming systems. This section describes the iterative operators that form part of the internal representation language. In order to facilitate the translation of evolved algorithms to a procedural programming language the syntax and functioning of these operators have been chosen to be similar to that of iterative operators commonly used by procedural programming languages. The iterative control structures provided by the genetic programming system are the *for*, *while*, and *dowhile* operators which implement the for, while and dowhile loops respectively.

The *for* operator takes three arguments. The first argument represents the initial loop value and the second argument represents the final loop value. Thus, both values are of the type Integer. The third argument can be of any type and is evaluated on each iteration of the loop. The type of this argument, and hence the particular instance of the *for* operator, is instantiated during the process of initial population generation. The number of iterations performed by each *for* operator instance is equal to the difference between its first and second argument plus one.

The *while* and *dowhile* operators take two arguments, the first of which represents a condition and is thus of the type Boolean. The second argument represents the parse tree which will be executed on each iteration. The *while* operator only performs an iteration if its first argument evaluates to true.

The *dowhile* operator performs one iteration after which an iteration is only performed if its first argument evaluates to true.

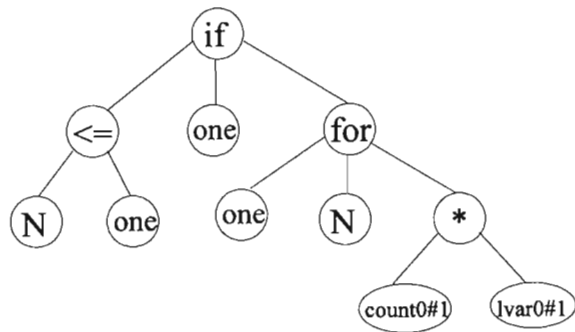
Two variables are maintained for each of the iterative control structures. The first is a counter variable which is incremented on each iteration. This variable serves the same purpose as the INDEX terminal used by Koza et al. [KOZA99a] in their implementation of automatically defined iterations and loops. In the case of the *for* operator this variable can be incremented or decremented and is given an initial value equal to the first argument of the *for* operator. In the case of the *while* and *dowhile* operators the counter variable is initialized to one. The second variable maintained by the system stores the result of each iteration and will be referred to as the iteration variable. The initial value given to this variable is context-sensitive and is denoted by the "@" character. For example, suppose this variable is represented by *lvar* and the body of the loop is  $lvar + N$  then *lvar* will be given an initial value of zero. However, if the body of the loop was  $lvar * N$ , *lvar* will be given an initial value of one. Thus, the initial value given to this variable is dependent on the operator applied to it. When the algorithm generated by the GP system is converted into a particular programming language, a programming statement initializing this variable will be placed before the iterative structure. The counter and iteration variables are automatically updated on each iteration of a loop instance. If the type of the loop instance is Output, an iteration variable is not maintained for the instance.

When an instance of the *for* operator is instantiated, both these variables are added to the terminal set when creating the third argument of the *for* operator. Similarly, when the *while* or *dowhile* operator is added as a node to a parse tree the counter variable is added to the terminal set when creating the parse tree representing the first argument and both variables are added to the terminal set when generating the second argument of the *while* or *dowhile* operators. If the *for* operator exists in the function list in a problem specification, the constant one is added to the terminal set.

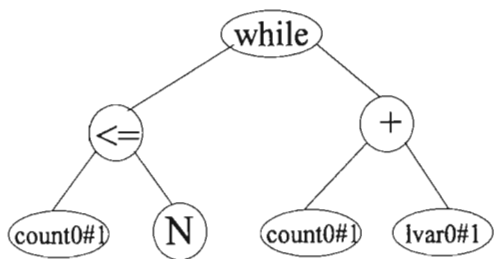
One of the difficulties associated with incorporating the use of iteration into a genetic programming system is the problem of overcoming the halting problem [BANZ98]. The literature proposes a number of methods of dealing with this problem. These methods are described in section 3.1.2 of **Chapter 3**. For the purposes of the study presented in this thesis, a similar approach to that taken by Koza [KOZA92] has been employed to prevent infinite and time-consuming iterations. An upper bound on the number of iterations, *max\_it*, that can be performed by each individual is set. This value is problem dependant and treated as a GP parameter.

In some instances the loop represented by the *for* operator and the *while* operator may not be executed. In order to satisfy the closure property of the function set it is essential that a value is still returned by the *for* and *while* operators in these instances. The GP system returns an error message if the loop is not executed. If either of the first two arguments passed to an instance of the *for* operator is the error message an error message is returned. Similarly, if either of these values is Infinity or -Infinity an error message is also returned.

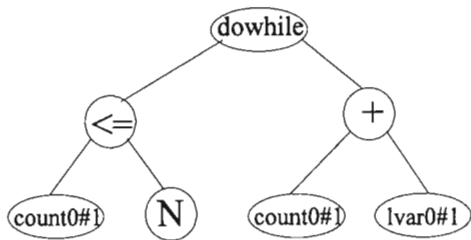
**Figure 5.6.6.1** illustrates the use of an instance of the *for* operator in an algorithm that calculates the factorial of an integer value N. **Figure 5.6.6.2** and **Figure 5.6.6.3** depict the use of the *while* and *dowhile* operators respectively in an algorithm which returns the sum of numbers from one to the given integer input N. The 0 in *count0#1* specifies the position of the individual in the population, in this case the first individual, and the 1 specifies the counter variable number. The notation used in the iteration variable *lvar0#1* is defined in the same way.



**Figure 5.6.6.1:** The use of the *for* operator in calculating the factorial of the integer N



**Figure 5.6.6.2:** An instance of the *while* operator



**Figure 5.6.6.3:** An instance of the *dowhile* operator

**6.7 Input and Output Operators**

From the survey of introductory programming courses conducted, it is evident that novice procedural programming problems have one or more inputs and one or more outputs. Thus, each algorithm derived by the genetic programming system will have one or more inputs and one or more outputs. These inputs and outputs are defined as part of the program specification. The code required to obtain these input and output values are fairly standard. Thus, it was decided that the genetic programming system will not be required to generate the code needed to obtain the input values and output the values computed by the algorithm. These standard code components, e.g. reading inputs values from a file, will be added to the algorithm derived by GP system. A further advantage of this approach is a reduction in the computational effort involved in inducing the algorithms using genetic programming.

In order to facilitate this approach the definition of the input and output variables in the problem specification must include a description of the source/s of the input variables and the destination/s of the outputs of the algorithm. Valid sources and destinations include the keyboard/screen, a file, a function and memory.

ASCII graphic problems are usually used to teach novices simple output and iteration ([DEIT01]). These problems require characters to be written to specific positions on the screen.

To cater for this category of problems the genetic programming system implemented uses a multidimensional array to represent the screen. The system keeps track of two screen variables, *currentx* and *currenty*. These variables store the x and y-coordinates of the current screen position, i.e. the position that will be written to next. These variables are updated each time the screen is written to.

The programming problem specification for an ASCII graphics problem must be defined as a set of x- and y-coordinate pairs and the character that must appear at the intersection of both coordinates, for each pair, on the screen. The dimensions of the multidimensional array representing the screen correspond to the maximum x- coordinate and y-coordinate specified in the fitness cases. The “~” character is used to represent a blank screen position. Two constants are added to the terminal set, namely, the maximum value in the x-coordinate and y-coordinate ranges listed in the problem specification. If the terminal set does not contain a constant value of one, one is also added to the terminal set.

The *place* operator is included in the function set to display characters at certain positions on the screen. The *place* operator takes three arguments: a x-coordinate value, a y-coordinate value and the character to be written to the screen. The first two arguments are of the type Integer while the third argument can be of any type. The type of the third argument of the *place* operator is randomly chosen from the existing types during the process of initial population generation. The type of the place operator is Output. Executing an instance of the *place* operator results in its third argument being written to the point ( i.e array position) specified by its first and second arguments. Both the screen variables, *currentx* and *currenty* are updated accordingly.

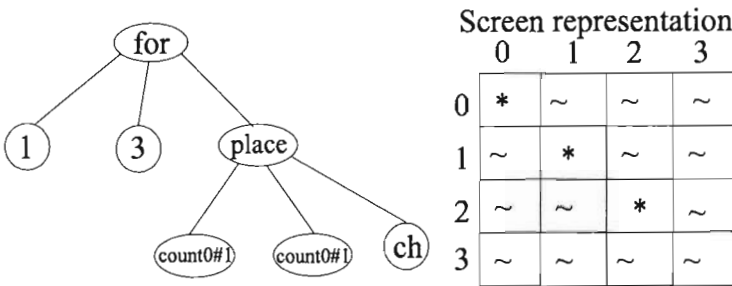
If either the x- coordinate value or the y- coordinate is less than one, the invalid coordinate is replaced by a one and the error character, represented by “!”, is written to the corresponding position on the screen. Similarly, if the x-coordinate or y-coordinate is greater than the maximum value permitted it is replaced by the maximum value and the error character is written to this screen position. Furthermore, if the *place* operator attempts to write a character to a screen position that has already been written to, the error character is written to this position. The number of times a screen location has been written to is also recorded. If the error message is passed to the place operator as either a first or second operand the operand is replaced with maximum value permitted and the error character is written to the corresponding screen position.

In order to cater for those languages that may not contain a command that writes output to specific locations on the screen, the *toscreen* and *newline* operators are defined as part of the internal representation language. Both these operators are of type Output. The *toscreen* operator takes a single argument which it writes to the screen. This argument can be of any type and is instantiated during initial population generation. The character is written to the screen position specified by *currentx* and *currenty*. Both the screen variables are updated accordingly: if *currentx* or *currenty* is greater than the maximum x-coordinate or y-coordinate respectively, the variable is decreased by one and the error character is written to the screen. Execution of the *newline* operator results in *currentx* being incremented and *currenty* being set to zero. Section 7 discusses how the fitness of an individual is calculated for these problems.

**Table 5.6.7.1** displays the problem specification for an ASCII graphics problem and **Figure 5.6.7.1** illustrates an example of an individual using the *place* operator and the corresponding screen representation maintained by the GP system.

|                            |                              |
|----------------------------|------------------------------|
| x-coordinates              | 1, 2, 2, 3, 3, 3, 4, 4, 4, 4 |
| y-coordinates              | 1, 1, 2, 1, 2, 3, 1, 2, 3, 4 |
| Character at each position | *                            |
| Constant                   | *                            |
| Constant type              | Char                         |
| Function set               | block3, block2, place, for   |

**Table 5.6.7.1:** Programming Problem Specification for an ASCII Graphics Problem



**Figure 5.6.7.1:** An individual containing the *place* operator and the corresponding screen representation maintained by the GP system

**6.8 Recursive Operators**

Section 3.1.4 of **Chapter 3** describes studies that have been conducted to evolve recursive algorithms using genetic programming. The approach taken by Brave [BRAV96] is similar to that commonly employed by imperative programming languages. The system developed by Brave [BRAV96] induces recursive algorithms by allowing a parse tree representing an ADF to call itself. Novice recursive programming problems usually require the student to write a recursive method or function. The *recur* operator will be used to represent a function call made by a parse tree to itself.

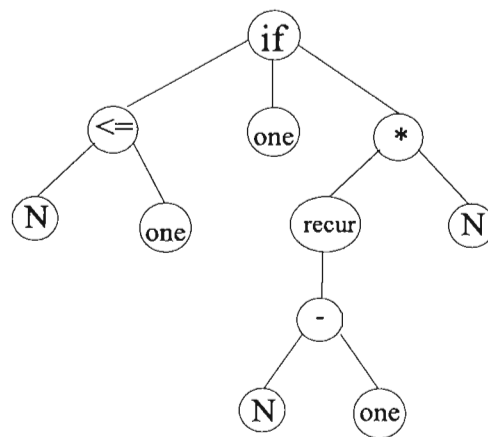
The *recur* operator allows a parse tree to call itself recursively. The arity of this operator is equal to the number of input variables for the problem. Each argument represents the corresponding input variable when a recursive call is made to the algorithm represented by the parse tree. The type of each argument of the *recur* operator is chosen to be the same as that of the input variable the argument represents. The type of the *recur* operator is set to be the same as that of the root node of the parse tree in which it appears. Thus, the type of each instance of this operator is instantiated during the process of initial population generation. The *recur* operator is prohibited from being inserted at the root of the tree.



Interpreting an instance of the *recur* operator results in the tree containing the *recur* instance being re-executed with the terminals represented by each argument of the *recur* operator being instantiated to the result of executing the subtrees representing these arguments.

As with iteration the incorporation of recursion into programs can lead to infinite and time-consuming executions. Thus, an upper-bound, denoted by *recur\_max*, is set on the number of recursive calls that can be performed per individual. This upper-bound is added to the set of GP parameters.

In order to ensure that the closure property is met, an instance of the *recur* that is not executed due to exceeding the maximum number of recursive calls permitted, still needs to return a value. If the *recur* operator instance is of type Integer or Real a value zero is returned, if it is of typed Boolean a value of false is returned, while if it is of type String or Char the null string or null character is returned respectively. If the arguments of the *recur* operator are -Infinity or Infinity an error message is returned. **Figure 5.6.8.1** illustrates the use of an instance of the *recur* operator in calculating the factorial of the integer N.

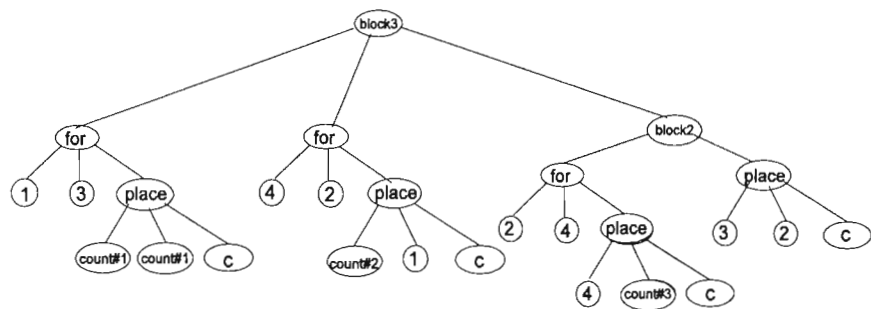


**Figure 5.6.8.1:** A parse tree representation of the algorithm calculating the factorial of the integer N using recursion

## 6.9 Multiple Statements: Blocks

In the standard genetic programming system proposed by Koza [KOZA92], each derived solution was essentially a function and thus consisted of just one programming statement. However, algorithms consist of more than one statement and thus the genetic programming system must facilitate this. The system implemented in this study employs a methodology similar to that proposed by Pringle [PRIN95] to cater for solution algorithms containing multiple statements. A number of *block* operators are included in the function set for this purpose. For example, *block2* takes two arguments of any type and returns the result of evaluating its last argument. Thus, *blockn* will take *n*, where *n* = 1...5, arguments and will return the result of evaluating its *n*th argument. The types of the arguments of each instance of a *block* operator and hence the type of the *block* operator instance is instantiated during the process of initial population generation. The type of the *block* operator is instantiated to be the same as that of its last argument. The types of its other arguments can be of any type.

**Figure 5.6.9.1** illustrates an example of an individual using *block* operators. The figure represents a solution to an ASCII graphics programming problem.



**Figure 5.6.9.1:** An individual using instances of *block* operators

6.10 Errors

The system needs to deal with two types of errors. The first error type is *#error*, which represents the error message returned by an operator if it receives an illegal operand or was unable to perform its function. This error is propagated through to the procedure calculating the raw fitness of the individual and the individual is penalised accordingly. Details of this are provided in section 7.

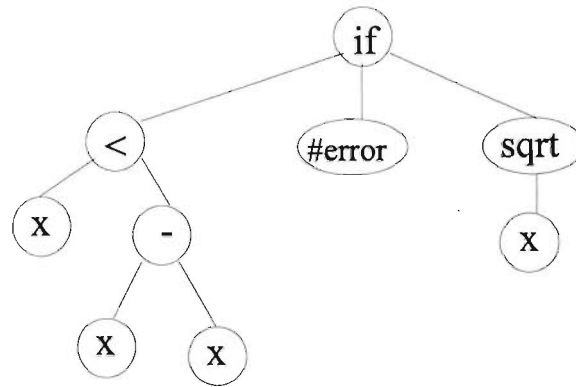
The second type of error that the system must cater for are user-defined errors. Novice procedural programming problems often require the programmer to perform some type of error checking and output an error message if invalid data has been entered. These errors need to be defined in the problem specification by using the keyword *error*. If the solution algorithm has to perform error checking for more than one input variable there will be more than one error type and each error type will be numbered in the problem specification, e.g. *error1*. **Table 5.6.10.1** depicts a problem specification for a program that calculates the square root of an integer value. If the value is negative the program must return an error message.

|                             |  |
|-----------------------------|--|
| Input variable              | x  |
| Types of input variable     | Integer                                  |
| Source of input variable    | Screen                                   |
| Input values for N          | -2, -1, 4, 9, 25, 36, 2, -10, 0          |
| Target values               | sroot                                    |
| Type of target value        | Real                                     |
| Destination of target value | Screen                                   |
| Target values for sroot     | error, error, 2, 3, 5, 6, 1.41, error, 0 |
| Function set                | +, -, *, /, sqrt, if                     |

**Table 5.6.10.1:** Problem specification for the square root problem

If the system detects an “error” keyword in the fitness cases, an error of the same type as the input variable is added to the terminal set, e.g. *#error* or *#error1* if there is more than one error type.

A possible solution algorithm for the problem specification in **Table 5.6.10.1** is illustrated in **Figure 5.6.10.1**.



**Figure 5.6.10.1:** Solution algorithm for the square root problem

## 7. Population Evaluation

The process of calculating a fitness measure for each member of the genetic programming population essentially involves executing each individual on the set of fitness cases included in the program specification. The GP system implemented in this study generally uses the error fitness function (described in section 2.6.2.1 of **Chapter 3**) defined by Koza [KOZA98a] for calculating the raw fitness of each individual in cases where the target output is a real number and a smaller fitness measure indicates a better fitness. This function is employed by summing the absolute value of the differences between the target value/s of the fitness case and the result/s obtained by executing the individual on the given input values specified in the fitness case.

In addition to calculating the raw fitness of each individual, the GP system also calculates the number of hits, known as the hits ratio (see [KOZA92]), that each individual has obtained as part of the evaluation process. The function defined by Koza [KOZA92] is used for this purpose. If the fitness cases specify a single target value, the number of hits obtained by an individual is the number of fitness cases for which the value calculated by the parse tree is within a specified bound of the target value. This bound is a GP parameter. If the number of output values per fitness case is greater than one, the number of hits obtained by an individual is the equal to the number of the fitness cases for which the differences between each value computed by the individual and the corresponding target values is less than or equal to the bound for all outputs. The hits value is used to test whether the termination criterion has been met. If an individual with a hits value equal to the number of fitness cases is found, the genetic programming run is terminated and the individual is reported as the solution to the problem.

For those problems where a higher fitness score is indicative of a fitter individual both the raw fitness and the hits obtained by the individual is calculated to be equal to the number of fitness cases for which the result produced by the individual is the same as the target value specified in the fitness case.

If the error message, *#error*, is propagated through to the evaluation method and the error formula is used to calculate the raw fitness, then if the target value is numerical a penalty equal to the corresponding target value for the fitness case plus an *error\_offset* is added to the raw fitness. However, if the target value is an error type, e.g. *error1*, and the output of the individual is not the corresponding error message, the *error\_offset* is added to the raw fitness. If the raw fitness is equivalent to the hits ratio for the particular problem, i.e. a higher raw fitness means a fitter individual, then in both cases the raw fitness is not incremented.

A different evaluation methodology is employed for assessing solutions to ASCII graphics problems. The system maintains two screen representations. The first is the target screen, which is created using the information specified in the problem specification. The second screen representation, referred to as the current screen, reflects the effect of executing an individual. The raw fitness of an individual is calculated by performing the following comparisons between the target and current screen:

- The sum of the number of characters that appear on the target screen and not on the current screen is multiplied by two and added to the raw fitness.
- If a character, different to that at the same position on the target screen, appears on the current screen the raw fitness is incremented.
- If a character occurs on the current screen in a location which is blank on the target screen the raw fitness is incremented.
- If a screen position is written to more than once, this position is flagged and the number of times it has been written to is stored. This value is multiplied by two and added to the raw fitness.
- If a position beyond the boundaries of the screen is written to by a program, the nearest valid position on the screen is flagged. A value of two is added to the raw fitness for each screen violation.

The second value calculated is the hits ratio which is used to determine whether a solution has been found or not. This value is incremented when the correct character appears at the same position on the current screen as on the target screen. The hits ratio is decremented whenever a location is flagged as being written to more than once or when a character appears at a position at which a blank should occur. If the hits ratio is equal to the number of characters on the target screen, the individual is flagged as a solution.

## 8. Selection Methods

It is evident from **Chapter 3** that the selection methods commonly used by genetic programming systems are fitness proportionate selection and the tournament selection methods [KOZA92]. Studies conducted by Koza [KOZA92] have revealed that systems using the tournament selection method produced better results than those implementing fitness proportionate selection. Furthermore, Langdon [LANG98b] states that the tournament selection method is usually employed instead of fitness proportionate selection due to fact that it does not carry an additional overhead of computing the fitness statistics for the entire population. Hence, the tournament selection method is used by the genetic programming system in this study for selection purposes when applying the mutation and crossover operators.

The larger the tournament size used the greater the selection pressure exerted by the tournament selection method. Thus, too high a tournament size can lead to premature convergence of the genetic programming algorithm. Hence, the tournament size must be carefully chosen. Soule [SOUL98] describes a common range for the tournament size to be two to five. The tournament size is problem dependant and thus a GP parameter.

## **9. Genetic Operators**

Koza [KOZA92] describes mutation, crossover and reproduction as the primary operators used by GP systems. The genetic programming system in the study presented in this thesis uses the mutation and crossover operators to create the next generation at each stage of the genetic programming algorithm. Langdon [LANG98a] is of the opinion that the reproduction operator should not be applied in order to promote genetic diversity and so prevent premature convergence of the genetic programming algorithm. Furthermore, according to Langdon [LANG96c] and Soule [SOUL96] the crossover operator is susceptible to the problem of cloning. Langdon [LANG96c] describes two types of crossover that result in the offspring being just copies of their parents, namely, crossover which swaps terminals and crossover which replaces whole trees. The mutation operator also tends to produce clones. Thus, the reproduction operator is not applied in this study.

In order to reduce bloating a maximum offspring size, defined in terms of the maximum number of nodes an individual is composed of, is added to the list of GP parameters.

### **9.1 Mutation**

The mutation operator is applied by firstly randomly selecting a gene, i.e. one of the trees, in an individual chosen using the tournament selection method. A mutation point is then randomly selected in the gene. The subtree rooted at this point is removed and replaced with a newly created subtree. A maximum depth, different from that specified for the purposes of initial population generation, is specified for the creation of these subtrees. The maximum depth is problem dependent and hence a GP parameter. The function set used for the purposes of creating this subtree is the same as that used during the process of initial population generation. If the root of the subtree to be replaced is Boolean valued, the logical operators are added to the function set.

The terminal set used is a combination of the terminal set used during the process of initial population generation and the set of all counter, named and iteration variables. The genetic programming system implemented in this study is strongly-typed, thus the newly created subtree must be of the same type as that being replaced.

The memory of the offspring is updated to include any named memory, counter or iteration variables contained in the newly created subtree and remove any memory locations that are no longer referenced.

If the offspring created by the mutation operator exceeds the maximum size permitted, the mutation operation is performed again until an offspring of the correct size is generated. However, if a solution, which is larger than maximum size permitted, is generated during a mutation operation it is returned as an offspring.

## 9.2 Crossover

The crossover operator implemented in this system returns one offspring, namely, the fitter of the two offspring produced. The crossover operator uses the tournament selection method to choose two parents which have genes that have common point types. A crossover point is randomly selected in a randomly chosen gene from the first parent. The crossover point in the same gene in the second parent is randomly chosen from a set points that have the same type as the crossover point chosen in the first parent.

The subtrees rooted at each crossover point in each gene are swapped to create two offspring. The other genes in genome remain unchanged. Furthermore, if a memory location exists in one of the crossover fragments and the memory of the parent into which the fragment will be inserted does not cater for it, the operation is aborted and new crossover points are selected.

If any of the offspring exceeds the maximum size permitted, the crossover operator is applied again until an offspring of the correct size is generated. However, if the oversize offspring is a solution it is returned as the result of the operation.

## 10. Escaping Local Optima

It is evident from the discussion presented in section 4.2 of **Chapter 3** that one of the limitations of the genetic programming algorithm is its susceptibility to converge to one or more local optima. Lack of genetic diversity, selection variance, the existence of alpha individuals and the destructive effects of genetic operators, especially crossover, were identified as the main causes of premature convergence. The following mechanisms have been built into the system to prevent premature convergence:

- Genetic diversity - In order to prevent cloning and so promote genetic diversity, the reproduction operator is not used. In addition to this, duplicates are not permitted in the initial population. It was decided to allow duplicates in successive generations as a removal of duplicates in later generations may result in the genetic programming algorithm not converging. As the genetic programming algorithm converges to a particular area of the search space, i.e a particular structure for the best individual of each generation, individuals become more and more similar as the algorithm closes in on a solution locally. High mutation rates and a removal of duplicates during later generations will disrupt the local search and prevent the algorithm from converging. Generally, the genetic programming algorithm needs to converge to find a solution. If it does not converge a solution may still be found by random chance; however, the chances of this are very small.
- Selection variance - In order to deal with selection variance, the system performs a number of iterations per seed, in the hope that a different area of the search space will be visited on each iteration. If upon applying the system to the generation of solution algorithms to the problems in **Appendix A** it is found that different areas are not visited during each iteration, a similarity index, similar to that employed by Mawhinney [MAWH00] will be devised and applied to ensure a new area is searched on each iteration or run.

- The existence of alpha individuals - With an exception of Bersano-Bergey [BERS97], there have not been any further studies citing alpha individuals as a cause of premature convergence. Hence, the cause reported by Bersano-Bergey [BERS97] may have been specific to the particular problem in that study. Thus, at this stage mechanisms have not been built into the system to prevent alpha individuals until this is studied in the context of the evolution of the solution algorithms to the problems listed in **Appendix A**.
- Destructive genetic operators - The system also provides a non-destructive mutation operator and a non-destructive crossover operator. These operators are an extension of the operators presented in section 9. The non-destructive mutation operator produces an offspring that is fitter or just as fit as its parent. The non-destructive crossover operator produces an offspring that is fitter or just as fit as both its parents. Due to the fact that non-destructive operators are computationally expensive, these operators are only applied when a solution cannot be found with the standard genetic operators.

## 11. GP Parameters

Section 2 of **Chapter 4** has described the standard genetic programming parameters that will be varied in an attempt to find a solution for each problem. This chapter has defined a number of additional GP parameters. A summary of the GP parameters that will be used by the system and their corresponding initial values are listed in **Table 5.11.1**. These parameters will be varied until a solution is found.

## 12. Summary

This chapter proposed a genetic programming system for the derivation of novice procedural solution algorithms. The result of applying this system to the problems in **Appendix A** is presented in the following chapter.

|  |                      |
|--|----------------------|
| <b>Control Model</b>   | Generational         |
| <b>Method of initial population generation</b>   | Ramped Half-and-Half |
| <b>Population Size</b>   | 500                  |
| <b>Maximum tree size</b>   | 20                   |
| <b>Initial tree depth limit</b>  | 3                    |
| <b>Mutation tree depth limit</b>   | 2                    |
| <b>Tournament size</b>   | 4                    |
| <b>Error Fitness Function</b>  | true                 |
| <b>Bound</b>   | 0.01                 |
| <b>Error Offset</b>  | 10                   |
| <b>Type of Genetic Operators</b>   | Standard             |
| <b>Crossover application rate</b>  | 50%                  |
| <b>Mutation application rate</b>   | 50%                  |
| <b>Maximum number of generations</b>   | 50                   |
| <b>Maximum number of runs per seed</b>   | 10                   |
| <b>Maximum number of iterations per individual<br/>( <i>max_it</i> )</b>                           | 60                   |
| <b>Maximum number of recursions per individual<br/>( <i>recur_max</i> )</b>                        | 50                   |
| <b>Maximum arity for the <i>switchi</i> and <i>switchc</i> operators<br/>( <i>switch_max</i> )</b> | 5                    |
| <b>Maximum arity for the <i>comb</i> connector</b>   | 5                    |

Table 5.11.1: GP Parameters for the Proposed Genetic Programming System



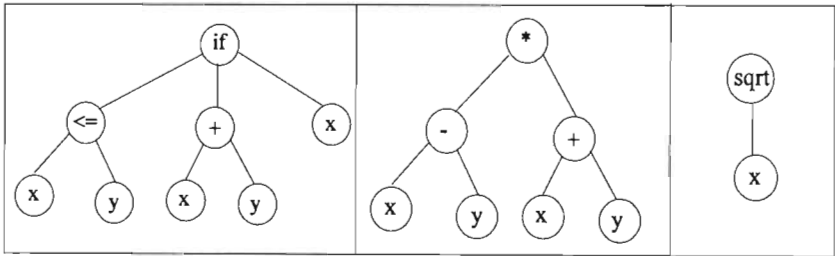
## Chapter 6 - Results and Discussion

This chapter discusses the results of applying the proposed genetic programming system in **Chapter 5** to the set of problems presented in **Appendix A**. The problem description, problem specification, and solutions evolved by the system are listed for sequential, conditional, iterative, ASCII graphics and recursive problems in **Appendix D** to **Appendix H** respectively. **Appendix C** lists the duration of each successful run for each problem. A description of the refinements made to the proposed system in cases where the system was not successful, as well as discussion on the solutions induced by the system is provided in this chapter.

Section 1 explains the changes that were made to the structure used to represent each individual. The methods used by the proposed system to escape local optima were not successful in all cases and an iterative structure-based algorithm (ISBA) was developed for this purpose. Details of the problems experienced and the overall algorithm is presented in section 2. Section 3 discusses certain observations made regarding the solutions evolved by the genetic programming algorithm. A summary of the chapter is provided in section 4.

### 1. Changes to the Program Representation

In the proposed genetic programming system each individual was represented as a multi-tree chromosome together with its corresponding memory structure. Each gene in the chromosome represents a corresponding target output. This is illustrated in **Figure 6.1.1**.

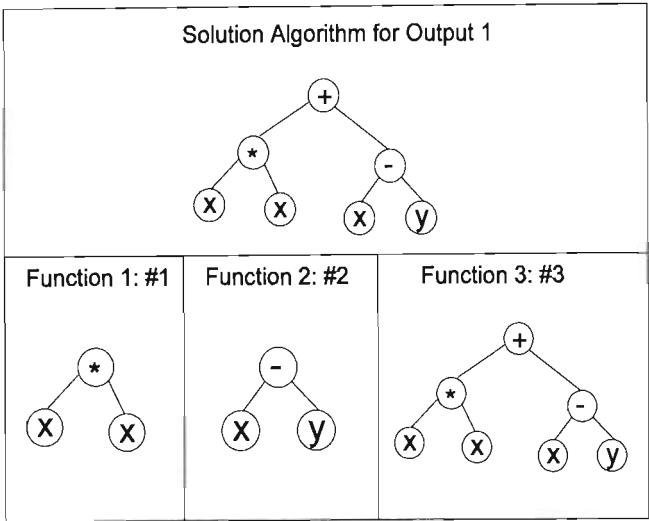


**Figure 6.1.1:** Multi-tree representation of a single individual with three outputs

Thus, if a program was required to produce more than one output, the solution algorithms for each of these outputs were evolved simultaneously. The representation proved to be sufficient for the induction of simple problems such as **Problem 1** of the sequential group of problems. However, for more complicated problems, such as sequential **Problem 3**, the system was unable to simultaneously induce solutions for all outputs. This is consistent with the difficulties reported by Bruce [BRUC95] and Langdon [LANG98c] with respect to studies conducted to apply GP to the simultaneous induction of methods for the stack, queue and list data structures.

As a result of this, the program structure used to represent each program was changed. Each individual is represented as a single parse tree with its corresponding memory structure. For those cases which require the system to evolve more than one output for a problem, a separate iteration (or set of iterations) is performed for each output.

In novice procedural programming problems it is often the case that the solution algorithm or part of the algorithm for one output forms part of the solution algorithm for one or more of the other outputs. To cater for this a function node, of arity zero, is created for each subtree of a solution algorithm evolved for an output. This set of function nodes is added to the terminal set when inducing the solution algorithms for the other outputs for the problem. **Figure 6.1.2** illustrates an example of this.



**Figure 6.1.2:** Example of a solution algorithm for an output and the corresponding set of functions

**Table D.1.3** and **Table D.3.3** in **Appendix D** and **Table E.4.3** in **Appendix E** lists solution algorithms to problems with multiple output. Future extensions of the study presented in this thesis will include extending this form of program representation to induce modular programs.

2. Escaping Local Optima

Genetic programming systems are susceptible to converging prematurely to a local optimum and thus may not find a solution in some cases. Section 10 of **Chapter 5** describes the mechanisms built into the proposed genetic programming system to escape local optima. From the results presented in **Appendix D** through to **Appendix H** it is evident that selection noise was one of the main causes of premature convergence of the proposed system. The use of multiple runs per seed proved to successfully escape local optima caused by selection variance. In some cases, namely, sequential **Problem 8** and conditional **Problem 3** and **Problem 4**, non-destructive operators were needed in order to evolve solutions. The mechanisms built into the system to escape local optima were not successful for all problems, e.g. sequential **Problem 4** and a number of the iterative and recursive problems. **Table 6.2.1** indicates that the system was able to evolve solutions to 33 of the 45

problems. For 12 of the problems the proposed GP system converged prematurely. Section 2.1 discusses why the methods built into the system to escape local optima were not successful for these examples. An algorithm for escaping local optima in such cases is presented in section 2.2.

|                         | Problems for which solutions were evolved | Problems for which the system was unable to evolve solutions due to premature convergence |
|-------------------------|---|---|
| Sequential Problems     | 1 - 3, 5 - 10                             | 4   |
| Conditional Problems    | 1 - 10                                    | None  |
| Iterative Problems      | 1, 4, 9                                   | 2, 3, 5, 6, 7, 8, 10  |
| ASCII Graphics Problems | 1 - 10                                    | None  |
| Recursive Problems      | 4   | 1, 2, 3, 5  |

**Table 6.2.1:** Performance of the Proposed Genetic Programming System

**2.1 Problems with Methods Used in the Proposed System**

The proposed genetic programming system was unable to evolve solutions to some of the problems. The evolutionary process was studied for each of these problems. Fitness function biases against certain primitives or combinations of primitives were identified as the cause of premature convergence in each of these cases. The fitness function and the set of fitness cases used, define the fitness landscape for a genetic programming run. Given a particular landscape, individuals containing certain primitives or combinations of primitives will always have a poor fitness and are thus eliminated early during the evolutionary process. If the eliminated primitives or structures are essential components of a solution algorithm, this results in the system converging to a local optimum during a run. For example, if the fitness cases contain an input variable with high positive integer values greater than one and the error fitness function is used, then individuals containing a combination of the multiplication operator, the power operator and the input variable are quickly removed from the population.

This was the cause of premature convergence of the system for **Problem 4** of the sequential problems. **Table 6.2.1.1** lists the best individual obtained for this problem for a run of the GP system for each seed. Notice that none of the best individuals contain the component *\*A pow*, which is an essential component of the solution function. Furthermore, none of the programs contain a combination of the multiplication and power operators.

A similar problem was experienced with some of the iterative and recursive problems. In these cases, individuals containing the iterative or recursive operator had a poorer fitness than those individuals not containing the operator. **Table 6.2.1.2** lists the best individual evolved for each seed for the recursive factorial problem. The recursive operator does not exist in any of these trees.

The methods employed by the proposed genetic programming system to escape local optima, namely, maintaining diversity, multiple runs and the use of non-destructive operators, have not been successful at escaping local optima caused by fitness function biases. Section **2.1.1** to **2.1.3** provides an analysis of why these methods were unsuccessful.

| Seed          | Best individual in prefix notation |
|---------------|------------------------------------|
| 1034007003030 | [ +, C, +, A, *, R, *, P, C ]      |
| 1008351890570 | [ +, C, +, A, +, C, *, P, P ]      |
| 1026110679330 | [ +, +, A, C, *, C, *, R, P ]      |
| 1027029949760 | [ +, /, *, *, A, P, R, C, A ]      |
| 1023430196170 | [ +, /, *, *, P, A, R, C, A ]      |
| 1025834715630 | [ +, +, C, *, *, R, C, P, A ]      |
| 1026110735850 | [ +, A, /, A, +, /, C, P, C ]      |
| 1023290284680 | [ +, +, A, *, P, *, R, C, C ]      |
| 1017985849240 | [ +, C, +, A, *, P, *, R, C ]      |
| 1018055297140 | [ +, +, +, C, *, P, P, A, C ]      |

**Table 6.2.1.1:** Best individuals for each seed for sequential **Problem 4**

| Seed          | Best individual prefix notation                      |
|---------------|--|
| 1034007003030 | [if, <=, n, one, one, *, -, -, *, n, n, one, one, n] |
| 1008351890570 | [if, >, n, one, -, *, n, -, *, n, n, one, n, one]    |
| 1026110679330 | [-, -, -, -, *, *, n, n, n, n, n, n, -, n, one]      |
| 1027029949760 | [-, -, -, -, one, n, n, n, -, n, *, n, *, n, n]      |
| 1023430196170 | [-, -, -, *, n, -, *, n, n, one, -, n, one, n, n]    |
| 1025834715630 | [-, -, -, -, *, *, n, n, n, n, n, n, -, n, one]      |
| 1026110735850 | [*, -, -, -, *, n, n, one, one, one, n]              |
| 1023290284680 | [-, -, *, n, -, -, *, n, n, one, one, -, n, one, n]  |
| 1017985849240 | [-, -, -, -, *, n, *, n, n, -, n, one, n, n, n]      |
| 1018055297140 | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]      |

**Table 6.2.1.2:** Best individuals for each seed for recursive **Problem 1**

**2.1.1 Maintaining Diversity**

Section **4.2.2.1** of **Chapter 4** lists higher mutation rates as one of the methods for maintaining diversity and preventing premature convergence. However, if the fitness function has a bias against certain substructures, increasing the mutation rate will not prevent premature convergence.

Although higher mutation rates may reintroduce these substructures into the population, these components will remain in the population for at most one generation unless adjustments are made to the fitness of individuals containing these substructures to ensure that they remain in the population.

Similarly, an increase in the population size is ineffective in preventing premature convergence caused by fitness function biases. While a larger population may introduce the necessary building blocks into the population, these remain in the population for at most one or two generations due to the fitness function bias against these structures.

Disabling the production of clones will not ensure that the elements of the population are dissimilar with respect to phenotype or genotype or that building blocks necessary for the derivation of solution algorithms exist and will remain in successive populations.

### 2.1.2 Multiple Runs

It is evident from the simulations performed that the use of multiple runs is successful at escaping local optima caused by selection variance or noise. However, if the fitness function has a bias against certain primitives or combinations of primitives these components are eliminated early during each run and thus the areas of the search space visited on each run will not lead to a solution being found.

### 2.1.3 Non-Destructive Operators

Non-destructive genetic operators are used to produce offspring that are fitter or just as fit as their parents. The use of these operators result in quicker convergence of the GP algorithm. Due to the elitist nature of these operators, if the fitness function has a bias against certain building blocks the use of non-destructive operators assist in the removal of these components from the population very early during the evolutionary process.

## 2.2 The Iterative Structure-Based Algorithm (ISBA)

This section describes the Iterative Structure-Based Algorithm (ISBA) which was developed as part of the study presented in this thesis to escape local optima caused by fitness function biases. Section 2.2.1 presents the algorithm. An explanation of the similarity indexes used by the algorithms is provided in section 2.2.2. A summary of the local optima parameters that must be specified when using the ISBA is tabulated in section 2.2.3. Section 2.2.4 discusses the result of applying this algorithm to those problems that the proposed genetic programming system was unable to evolve solutions to. Section 2.2.5 provides a brief overview of the chapter.

### 2.2.1 An Overview of the ISBA

**Table 6.2.2.1.1** illustrates the effect of performing multiple runs for a seed for sequential **Problem 4**. Notice that due to the fitness function bias against the components that form an essential part of the solution algorithm, each run has visited the similar, if not the same, areas of the search space. Thus, the best individual for each iteration or run basically has the some structure.

| Run No. | Best individual in prefix notation |
|---------|------------------------------------|
| 1       | [+, C, +, A, *, R, *, P, C]        |
| 2       | [+, /, *, *, A, P, R, C, A]        |
| 3       | [+, /, A, /, C, +, R, R, A]        |
| 4       | [+, +, *, *, R, C, P, C, A]        |
| 5       | [+, +, *, P, P, +, C, A, C]        |
| 6       | [+, A, +, *, *, R, C, P, C]        |
| 7       | [+, A, +, C, *, P, *, R, C]        |
| 8       | [+, A, /, *, P, *, R, A, C]        |
| 9       | [+, +, A, *, C, *, P, R, C]        |
| 10      | [+, /, *, P, A, /, C, R, A]        |

**Table 6.2.2.1.1:** Best individuals for seed 1034007003030 for sequential **Problem 4**

The ISBA uses similarity indexes, similar to those implemented by Keller et al. [KELL96] and Mawhinney [MAWH00], to prevent the GP algorithm from visiting areas that are structurally similar to those already explored. In order to determine how to apply this restriction, studies were conducted to determine how the GP algorithm converges to a particular structure. The studies revealed that the convergence process is comprised of two phases. During the first phase the GP algorithm appears to search globally until the best individual of each generation has more or less the same structure from the root to some depth  $d$ , where  $d$  is less than the depth of the tree. The second phase performs a local search, with the best individual of every generation fixed from the root to depth  $d$ . This search is localised due to the fact that each individual in the population is fixed from the root to some depth  $d$ . The rest of the tree seems to evolve during the run. This process continues until the GP algorithm converges to an overall structure with the best individual being basically the same from one generation to the next.

**Table 6.2.2.1.2** lists the best individual obtained for each generation of a run performed for recursive **Problem 1**. The best individual of each generation for generations 31 through to 50 have more or less the same structure as that of best individual for generations 27 through to 30, with the nodes at depths 0 through to 2 being exactly the same. Thus, they are not displayed. Notice that from generation 4 onwards the best individuals contain exactly the same component from depth 0 to 1. In this case the algorithm has converged to the global area - \*  $n$ . The global area is indicated in red. From generation 10 through to generation 50, with an exception of generation 26, the nodes at depth 2 of each best individual are *if* and \*. Hence, the local area converged to is *if* \* and it is indicated in blue. There are minor changes to the structure at lower levels of the best individual from generation 10 through to 50. Furthermore, there is no change in the raw fitness of the best individual from generation 10 through 50 which indicates that the algorithm has converged by generation 10. Based on this study of GP convergence, the algorithm presented in this section performs both a global level and local level search. Two similarity indexes, namely, *gsim* and *lsim* are used for the global and local level searches respectively. If a run converges to a local optimum, the algorithm performs  $n$  runs which search locally for a solution.

| Generation | Best Individual                                       | Raw Fitness |
|------------|---|-------------|
| 0          | [*, *, -, n, one, n, n]                               | 60.0        |
| 1          | [*, *, -, n, one, n, n]                               | 60.0        |
| 2          | [*, *, -, n, one, n, n]                               | 60.0        |
| 3          | [*, if, !=, one, n, -, n, one, one, *, n, n]          | 59.0        |
| 4          | [-, *, n, *, -, n, one, n, n]                         | 57.0        |
| 5          | [-, *, n, *, -, n, one, n, n]                         | 57.0        |
| 6          | [-, *, n, *, -, n, one, n, n]                         | 57.0        |
| 7          | [-, *, if, !=, one, n, -, n, one, n, *, n, n, n]      | 56.0        |
| 8          | [-, *, n, *, -, n, one, n, *, -, n, one, *, one, one] | 57.0        |
| 9          | [-, *, n, *, -, n, one, n, n]                         | 57.0        |
| 10         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 11         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 12         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 13         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 14         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 15         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 16         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 17         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 18         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 19         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 20         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 21         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 22         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 23         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 24         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 25         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 26         | [-, *, n, *, if, >, n, one, -, n, one, n, n, n]       | 56.0        |
| 27         | [-, *, if, >, n, one, -, n, one, one, *, n, n, n]     | 56.0        |
| 28         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 29         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |
| 30         | [-, *, if, >, n, one, -, n, one, n, *, n, n, n]       | 56.0        |

Table 6.2.2.1.2: Best individuals for each generation of a single run for recursive Problem 1



On each of these  $n$  runs, the root through to depth  $d$  of each individual is fixed to be the same as that of the current local optimum found as part of the global level search. If a solution is found before the  $n$ th run is reached, the GP process is terminated and the solution is reported. If a solution is not found, the best individual is stored as a local optimum for the local search and the processes of initial population generation, mutation, crossover and selection, of the rest of the  $n$  runs discard individuals that are similar (the fixed component is not counted) to the local optima detected on previous runs of the local search. However, if the similar individual is a solution it is not discarded. An individual is similar to the local optima of the local level search if its *lsim* value exceeds a given threshold (*lthresh*). Alternatively, similarity checks may not be performed during the  $n$  local runs. This has the same effect as performing multiple runs to escape local optima caused by selection noise.

If a solution is not found during the  $n$  runs of the local level search, the global level search is continued for  $m$  runs. A set of local optima is maintained for the global level search, and regions that have already been visited at the global level are not revisited. This is achieved by discarding individuals that are similar to the global level optima during the processes of initial population generation, mutation, crossover and selection. However, if the similar individual is a solution it is not discarded. An individual is similar to the global level optima if its *gsim* index evaluates to true. For each local optimum of the global level search a local level search of  $n$  runs is performed before the entire structure of the local optimum is discarded.

**Figure 6.2.2.1.1** provides a summary of the algorithm. The next section defines the similarity indexes used.

## 2.2.2 Similarity Indexes

This section defines the similarity indexes, namely, *gsim* and *lsim* used by the ISBA.

### 2.2.2.1 Calculating the *gsim* Similarity Index

The *gsim* index is used to identify similarities between elements of the population and local optima during global level searches. This index performs two checks to determine whether an individual is similar to the local optima representing global areas already visited. The first check compares the root of the individual to the root of each local optimum. The comparison can be a *Node* comparison or a *Type* comparison. In the case of a *Node* comparison the individual is reported to be similar to the local optima if the number of local optima which have exactly the same root as the individual exceeds the threshold *rthresh*. The *Type* comparison identifies an individual as being similar to the local optima if the root of the individual is of the same type of more than *rthresh* local optima. The different node types are listed in **Table 6.2.2.1.1**. The local optima parameter *rcomp* specifies the type of root comparison that must be performed.

If the first check indicates that the individual is dissimilar to the local optima a second check is performed. This check compares the nodes comprising the component of the individual from the root to a depth  $d$ , to the same component in each of the local optima. As with the first check, the comparison can be a *Node* comparison or a *Type* comparison. The local optima parameter *gcomp* specifies the type of comparison to perform. If the number of nodes that are the same (or of the same type in the case of a *Type* comparison) exceeds the threshold *gthresh* the individual is reported as being similar to the local optima.



**While a solution has not been found and the number of global areas explored is less than  $m$**

**Begin**

**Step 1:**

- Perform a run.
- If this is not the first global run, ensure that this run does not visit the same areas as the previous global runs.
- This is achieved by ensuring that each newly created individual, selected individual and offspring produced by mutation and crossover is not similar to the local optima representing the areas explored on the previous global runs. The *gsim* similarity index is used for this purpose.

**Step 2:**

- If the individual converged to is not a solution, record the best individual of the last generation as a local optimum for global areas.
- Identify the fixed component of the local optimum, consisting of the subtree comprising the nodes from the root to depth  $d$ , that will form the first  $d$  levels of each element of the all the populations of the local level search.

**Step 3:**

- *While a solution has not been found and the number of local areas visited is less than  $n$*

*Begin*

*Step 3.1:*

- Perform a run.
- The elements of all the populations are fixed from the root to level  $d$ , to be the same as the first  $d$  levels of the local optimum representing the current global area.
- If this is not the first local area visited and similarity checking must be performed, ensure that this run does not visit the same areas as the previous local runs.
- This is achieved by ensuring that each newly created individual, selected individual and offspring produced by mutation and crossover is not similar to the local optima representing the areas explored on the previous local runs. The *lsim* similarity index is used for this purpose.

*Step 3.2:*

- If the run does not converge to a solution and similarity checking must be performed, store the best individual of the last generation as a local optimum for local areas.

*End*

**End**

**Figure 6.2.2.1.1.:** Iterative Structure-Based Algorithm (ISBA) for escaping local optima caused by fitness function biases

| Type | Primitives   |
|------|--|
| 0    | Terminals specified in the problem specification           |
| 1    | Arithmetic operators                                       |
| 2    | Conditional operators                                      |
| 3    | Arithmetic Logical operators                               |
| 4    | Recursive operators  |
| 5    | Iterative operators  |
| 6    | Counter variables  |
| 7    | Iteration variables  |
| 8    | Output operators   |
| 9    | Character operators  |
| 10   | String manipulation operators                              |
| 11   | System defined arithmetic constants                        |
| 12   | Zero arity functions                                       |
| 13   | Memory manipulation operators                              |
| 14   | System defined boolean constants                           |
| 15   | Logical operators: <i>and</i> , <i>or</i> , and <i>not</i> |
| 16   | Character comparison operators                             |

**Table 6.2.2.2.1.1: Primitive Types**

### 2.2.2.2 Calculating the *lsim* Similarity Index

The *lsim* index is used to identify similar individuals during local level searches if similarity checking must be performed for the local level search. The index compares individuals by counting the number of relations that exist in both individuals. A relation is defined to be a tuple consisting of a function node and the primitives, i.e. operator or terminal, representing each of its children. If the number of relations exceeds a preset threshold *lthresh* the individuals are regarded as similar.

### 2.2.3 ISBA Parameters

If the ISBA algorithm is used a number of local optima parameters must be specified in addition to the standard GP parameters. **Table 6.2.2.3.1** provides a summary of these local optima parameters.

| Local Optima Parameter | Definition   |
|------------------------|--|
| $m$                    | Number of global level runs.   |
| $n$                    | Number of local level runs.  |
| $d$                    | Depth until which a component is fixed.  |
| $rcomp$                | Type of root comparison for <i>gsim</i> .  |
| $rthresh$              | Maximum number of local optima which an individual can have the same (or same type of) root as.  |
| $gcomp$                | Type of global area comparison for <i>gsim</i> .   |
| $gthresh$              | Maximum number of nodes, from the root to depth $d$ , which an individual can have that is the same (or the same type) in the same position in one of the local optima |
| $lthresh$              | The maximum number of relations that an individual can have in common with one of the local optima.  |

**Table 6.2.2.3.1:** ISBA Parameters

The next section discusses the result of applying the ISBA to the 12 problems the proposed genetic programming system was unable to evolve solutions to.

#### 2.2.4 Application of the ISBA

The ISBA algorithm was applied to the 12 problems, listed in **Table 6.2.1**, which the proposed GP system was unable to find solutions to. The algorithm was able to evolve solutions to all 12 of the problems. Similarity checking at the local level searches was not needed for all of these problems with an exception of recursive **Problem 3**. The local optima parameters used, as well as the problem specification and the GP parameters for these problems are listed in **Appendix D** (sequential **Problem 4**), **Appendix F** (iterative problems) and **Appendix H** (recursive problems). A value of 2 for the depth  $d$  was sufficient for all the problems and hence  $d$  has not been included as a local optima parameter.

**Appendix C** lists the duration of each successful run for all the problems. The runtime of the ISBA varies, depending on the problem, between a minimum of 2 minutes and a maximum of 4 hours and 20 minutes. The runtime of the algorithm can be improved by implementing the system as a distributed system that performs local level searches for each global area simultaneously. This will be investigated as part of future work. Furthermore, each run evolves  $g$  generations. However, the genetic programming algorithm may converge before generation  $g$  in which case additional generations are being evolved unnecessarily. Thus, instead of terminating a run after  $g$  generations, the run should be terminated as soon as the system detects that the algorithm has converged to a particular structure. It is evident from the run presented in **Table 6.2.2.1.2** that there are essentially two indications that system has converged, namely, there is no or very little change in the raw fitness of the best individual from one generation to the next, and the structure of the best individual of successive generations is similar.

A similarity index (*sim*) can be used to test the latter. If the number of generations for which the best individuals are structurally similar exceeds a specified threshold (*gen\_thresh*) or the number of generations for which the best individuals are phenotypically the same exceeds a given threshold (*phen\_thresh*) the algorithm has converged and the run will be terminated. **Table 6.2.3.4.1** defines the similarity indexes and the parameters that must be included as part of the local optima parameters needed to detect system convergence.

| Parameter/<br>Similarity Index | Definition  |
|--------------------------------|---|
| <i>sim</i>                     | This similarity index is used to determine the structural similarity of the best individual from one generation to the next when detecting system convergence. For example, if we want to determine how similar <i>tree1</i> is to <i>tree2</i> , <i>sim</i> is calculated to be the sum of the successive nodes that are the same in a depth-first traversal of both trees divided by the size of <i>tree1</i> . |
| <i>geno_thresh</i>             | The maximum number of successive generations for which the best individuals can be structurally similar.  |
| <i>pheno_thresh</i>            | The maximum number of successive generations for which there is little or no change in the raw fitness of the best individual.  |

**Table 6.2.3.4.1:** Indexes and parameters to detect system convergence

Future extensions will be made to the system to detect system convergence.

### 3. Observations

This section discusses a number of observations made with respect to the solutions evolved by the genetic programming systems.

Consistent with the literature discussed in section 4.1 of **Chapter 3** it was found that a number of the solutions evolved by the genetic programming system contain redundant code. This is evident from the solutions displayed in tables **E.8.3**, **E.9.3** and **E.10.3** which contain solutions to problems requiring the use of the *switchi* or *swtichc* operators. Some of the solutions evolved for iterative **Problem 6** contain the intron *aread(10)* as part of product. This intron evaluates to one as a result of the *aread* operator being protected and returning a value of one if a location beyond its size is accessed. Similarly, the code contained within blue brackets in the solution evolved for seed 1018055297140 in **Table F.4.3** is redundant code. The iteration variable *lvar826788#2* evaluates to the context sensitive value @. The division operator replaces any of its arguments that is an @ with its other argument, in this case one. Thus, the code within blue brackets evaluates to one. This solution is not efficient as the body of the loop within blue brackets is a constant, and hence iterations are being performed unnecessarily. The solution evolved for seed 1034007003030 displayed in **Table F.10.4** is another example of the existence of redundant code in the solution resulting in the solution being inefficient. The body of the *for* loop contained within the blue parentheses is a constant. There is much contention as to whether the existence of introns are a blessing or a curse.

While some researchers feel that they result in degradation of the genetic programming system, others state that the existence of introns is imperative to the success of the GP system. Thus, an investigation into the removal of introns causing inefficiency, without affecting the success of the GP system, will be conducted as part of future work on the project.

The approach taken to problem solving in some of the solutions is very different from that usually employed by human programmers. For example, the solutions evolved to problems requiring the use of the *switchi* and *switchc* operators (see tables E.8.3, E.9.3 and E.10.3) contain a number of nested *switchi* and *switchc*. Such nesting is not characteristic of programs written by humans. A number of the solutions listed in Table F.4.3 to the factorial problem use the *if* operator to determine the starting or final value of the *for* loop calculating the factorial, whereas human solutions to this problem usually contain this *for* loop in the *else* section of an overall *if* statement. Similarly, the solution to the average problem for seed 1017985849240 in Table F.5.4 calculates the sum of the quotient of the each number and number of numbers, instead of dividing the sum of the numbers by the number of numbers. The methodology employed by the generic architecture for comparing the solutions derived by the GP system to student solutions needs to take this difference into consideration. Alternatively, syntax rules can be built into the system to restrict the syntax of solutions evolved.

Although the induction of efficient solutions is beyond to the scope of this thesis (see section 3 of Chapter 1), it is interesting to note that there is a need to incorporate a measure, similar to that used by Koza [KOZA92], into the fitness function that penalises individuals for inefficiency for certain novice procedural problems. A number of the solutions evolved for recursive Problem 4 listed in Table H.4.3 perform unnecessary recursive calls. For example, the solution evolved for seed 1023430196170 merely swaps  $m$  and  $n$  if  $m$  is greater than  $n$  instead of performing a recursive call with  $m-n$  and  $m$ .

One of the reasons that genetic programming was considered as a means of inducing solutions to novice procedural algorithms is its ability to induce more than one solution, each taking a different problem solving approach to the same problem. As mentioned in section 3 of Chapter 1 the evolution of multiple solutions for problems was not considered as part of the study presented in this thesis and will be examined as part of future work. Thus, mechanisms to evolve multiple solutions, each taking a different problem solving approach, was not built into the system. However, despite this, the solutions evolved for each seed for some of the problems vary in the problem solving approach taken, e.g sequential Problem 4 and iterative Problem 5.

#### 4. Summary

This chapter has discussed the results of applying the proposed genetic programming system to the novice procedural programming problems listed in Appendix A. Two refinements had to be made to the proposed genetic programming system. The first involved changing the structure used to represent programs. The second was the incorporation of the ISBA into the system to escape local optima caused by fitness function biases. This chapter also identified further studies that must be conducted into the structure and efficiency of the programs evolved by the genetic programming system.

## Chapter 7 - Conclusions and Future Work

The main aim of the study presented in this thesis was to test the hypothesis that genetic programming can be used to induce novice procedural programming solution algorithms.

**Chapter 1** lists the following objectives that the study must attain:

- A problem specification must be defined - The genetic programming system will need a description of the function of the program to be induced and the application domain as input. A programming problem specification will be defined for this purpose.
- An internal representation language must be defined - In order to ensure the reusability of the expert module, the solutions generated by the genetic programming system must be language independent. Thus, solution algorithms will be expressed in an internal representation language. Defining this language will involve identifying the primitives needed by the GP system to successfully induce solution algorithms to novice procedural programming problems.
- The standard genetic programming features needed to derive novice procedural programming algorithms must be identified - This entails identifying the following: control model/s, program representation, initial population generation method/s, genetic operators, selection methods, and the fitness measure.
- Any advanced features needed by the genetic programming system to induce novice procedural programming algorithms must be identified - Examples of such features include the use of memory, strongly-typed GP, cultural learning and architecture-altering operations.
- In some cases genetic programming systems are unable to evolve solutions to a problem. The cause of this is premature convergence of the genetic programming algorithm. Methodologies for escaping local optima in the domain of novice procedural programming problems must be identified.

Section 1 describes how these objectives were met. Future investigations that will be conducted as a result of this study are outlined in section 2. Section 3 provides a summary of the study.

### 1. Conclusions

This section describes conclusions that can be made with respect to the objectives listed above.

- Problem specification - The problem specification defined in section 1 of **Chapter 5** was found to sufficiently represent novice procedural programming problems and provided adequate input to the genetic programming system.
- Internal representation language - This study has revealed that the set of primitives described in section 6 of **Chapter 5** represents all the control structures necessary to induce solutions to the novice procedural programming problems. Further studies examining more complex procedural programming domains may result in extensions being made to the language. Some of the operators, e.g. the *aread* operator, can be redefined so as to reduce the amount of redundant code contained in solution algorithms. This will be examined as part of future work on the removal of introns.

- Standard genetic programming features:
  - ▶ The generational control model was used in the induction of all solution algorithms.
  - ▶ Program representation - It is evident from the discussion presented in section 1 of **Chapter 6** that a parse tree together with a memory structure proved to sufficiently represent novice procedural programs. Furthermore, solutions to problems requiring more than one output per program are evolved by performing one run per output. Primitives representing each subtree of the solution algorithm for one output are added to the terminal set when inducing the solution algorithm of the other outputs.
  - ▶ Initial population generation - The most suitable method to use for initial population generation to ensure success of the GP system is problem dependant. For example, while the full method was needed for sequential **Problem 8**, the ramped half-and-half method was sufficient for sequential **Problem 2** and the grow method for sequential **Problem 4**.
  - ▶ Genetic operators - Two genetic operators, namely, mutation and crossover were used to successfully induce solutions to the novice procedural programming problems. The reproduction operator was not used in order to prevent cloning. A mutation application rate of 0.5 and a crossover application rate of 0.5 was sufficient to induce at least one solution to each of the problems listed in **Appendix A** and thus different operator application rates were not investigated.
  - ▶ Selection methods - The tournament selection method was used for all problems. While a tournament size of 4 was used for most of the problems, this study revealed that the most suitable tournament size is problem dependant.
  - ▶ Fitness measure - The error fitness function was used to calculate the raw fitness of the population for problems with real or integer target values. In all other cases, except ASCII graphics problems, the number of fitness cases for which the output of the program was the same as the target value was the fitness measure. This value, referred to as the hits ratio, was also used to determine whether a solution has been found for all the problems (with an exception of ASCII graphics problems). If the number of fitness cases for which the program produces the same output as the target value is equal to the number of fitness cases, the program is flagged as a solution.

In the case of ASCII graphics problems the raw fitness and the hits ratio are calculated to be a function of the number of missing characters, number of screen locations containing the correct character, the number of screen locations at which the incorrect character has been written, the number of locations which display a character when the location should be blank, the number of screen locations written to more than once and the number of times the program attempted to access a screen location beyond the boundaries of the screen. Details regarding the calculation of both these values for ASCII graphics problems can be found in section 7 of **Chapter 5**.

- Advanced genetic programming features:
  - All procedural programming languages are typed, thus strong-typing was necessary for the induction of novice imperative programming solution algorithms. Each function, function argument, terminal, and constant was defined to be of a particular type. The types catered for by the system include Integer, Real, String, Char, Boolean and Output.
  - Named memory was used to represent memory locations that can be written to, and the iteration and counter variable for each instance of the *for* operator.
  - For some problems, e.g. iterative **Problem 1**, each value for the input variable is a list of values instead of a single value. In this case indexed memory is used to store each list of values and the *aread* operator is used to access these values.
- Methods to Escape Local Optima - The prevention of clones, multiple runs and non-destructive operators were used to prevent premature convergence. Selection variance appeared to be the main cause of premature convergence in this study and the use of multiple runs enable the system to escape local optima in these cases. The use of non-destructive operators was needed to find solutions to sequential **Problem 8** and conditional **Problem 3** and **Problem 4**. However, there were 12 problems for which these methods were unsuccessful at escaping local optima. The cause of premature convergence in these 12 cases, was fitness function biases against certain primitives or combination of primitives. An algorithm, namely, the iterative structure-based algorithm (ISBA) was developed to escape local optima caused by fitness function biases. The algorithm was successful at inducing solutions to all 12 problems.

The study presented in this thesis has identified a number of further investigations that need to be conducted. These are discussed in the following section.

## 2. Future Research

Future extensions to the study presented in this thesis will include the following:

- Redundant Code

Methods for removing redundant code and preventing the growth of introns will be investigated. The effect of these methods on the success of the evolutionary process will also be examined.

- The Evolution of Efficient Programs

The importance of efficient programs is emphasised in novice procedural programming courses. The study presented in this thesis has revealed that the genetic programming system may produce inefficient programs. Mechanisms need to be built into the system to ensure that solution algorithms evolved are efficient. Koza [KOZA92] achieves this by including a measure in the fitness function that penalises inefficient solutions. A Pareto fitness measure [LANG98c] could also be used to cater for this.



- The Evolution of Multiple Solutions

The study presented in this thesis forms part of a larger initiative aimed at reducing the developmental costs associated with building intelligent programming tutors. One of the overheads contributing to this cost, is that the developer has to derive more than one solution to a problem, each taking a different problem solving approach, to cater for all possible solutions that could be presented by a student. For example, developers of SIPLeS II [XU99] stored 4 to 5 different solutions for each of the problems. One of the reasons that genetic programming was chosen to induce solution algorithms in the generic architecture, is its ability to produce multiple solutions, each taking a different problem solving approach, to the same problem. The genetic programming system developed in this study will be extended to produce multiple solutions.

- The Use of Syntax Rules

The study presented in this thesis has revealed that some of the solutions produced by the genetic programming system are very different to those written by human programmers. Hence, it may be necessary to apply set of syntax rules during the processes of initial population generation, mutation and crossover to ensure that solutions evolved adhere to good programming practices. This matter will be investigated further as part of future research.

- The Evolution of Modular Programs

The topic of modularization is covered by most, if not all, procedural programming courses. The system presented in this thesis will be extended to produce modular programs. Automatically defined functions [KOZA94] and module acquisition will be evaluated as means of producing modular solutions to novice procedural programming problems. If these mechanisms are not effective new methodologies will be developed for this purpose.

- Further Applications of the ISBA

Premature convergence is a common problem encountered with genetic programming systems. Future research will include studies that apply the ISBA to domains which genetic programming has not been able to evolve solutions for. Future studies will also include the application of the ISBA to more rugged fitness landscapes than those associated with the set of novice procedural programming problems in **Appendix A**, in order to identify possible limitations of the ISBA and necessary extensions.

- Deriving Object-Oriented Program Solutions

The research presented in this thesis has contributed to the study being conducted on the automatic generation of solutions to procedural programming solution algorithms. However, the generic architecture must also cater for the induction of solution algorithms to object-oriented programming problems. Future studies will be conducted into the automatic derivation of object-oriented solutions algorithms.

- Generating Solutions to Different Types of Novice Programming Problems

Section **4.5** of **Chapter 2** describes the different types of programming problems that must be catered for by the generic architecture. This thesis has concentrated on the induction of novice procedural solution algorithms which is just one type of programming problem. Methodologies must be identified for the automatic generation of solutions to the other types of problems.

- Improving the Runtime of the ISBA

The runtime of the ISBA can be possibly improved by detecting system convergence during a run and performing the local level searches for each global level simultaneously. Further investigations into this will be conducted.

### 3. Summary

The research presented in this thesis was aimed at determining whether genetic programming is able to induce solutions to novice sequential, conditional, iterative, ASCII graphics and recursive procedural programming problems. This study has identified a problem specification, internal representation language and the standard and advanced GP features that enable genetic programming to successfully induce solutions to these novice procedural programming problems. Furthermore, this study has also determined the causes of premature convergence in this domain and corresponding methodologies for escaping from local optima in these instances. As part of this investigation into premature convergence, the ISBA (Iterative Structure-Based Algorithm) was developed for escaping local optima caused by fitness function biases. Genetic programming was successfully applied to the generation of solutions to the 45 novice procedural programming problems listed in **Appendix A**. A number of the solutions induced by the genetic programming system contained redundant code. In addition to this some solution algorithms evolved were not efficient while others did not adhere to the standard practices of good programming. Further studies will be conducted to investigate these issues.

## References

1. [AHLU96] - Ahluwalia M. , Fogarty T.C., Co-Evolving Hierarchical Programs Using Genetic Programming, in Genetic Programming, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 419, MIT Press, 1996.
2. [ALPE95] - Alpert S. R., Singley M. K., Carroll J. M., Multiple Multimodal Mentors, Delivering Computer-Based Instruction via Specialized Anthropomorphic Advisors, in Behaviour and Information Technology, Vol. 14, No. 2, pp. 69-79, 1995.
3. [ALPE99] - Alpert S. R., Singley M. K., Carroll J. M., Multiple Instructional Agents in an Intelligent Tutoring System, in Proceedings of the AIED '99 Workshop on Animated and Personified Pedagogical Agents, July 1999, Le Mans, France, 1999.
4. [ALTE96] - Altenberg L., Emergent Phenomena in Genetic Programming, in Evolutionary Programming, Proceedings of the Third Annual Conference, editors A. V. Sebald and L. J. Fogel,, pp. 233-241, World Scientific Publishing 1996, <http://dynamics.org/~altenber/PAPERS/EPIGP/>.
5. [ANDE85] - J. R. Anderson, Reiser B.J., "The Lisp Tutor", *BYTE*, April 1985, pp.159 -175, 1985.
6. [ANDE90] - Anderson J.R., Boyle C.B., Corbettm A.T., Lewis M.W., Cognitive Modelling and Intelligent Tutoring, in Artificial Intelligence, Clancy (Ed.), pp. 7- 49, Elsevier, 1990.
7. [ANDE96] - Anderson J.R. and Skwarecki E., The Automated Tutoring of Introductory Computer Programming, Communications of the ACM, Volume 29, September 1996, pp. 842-849, ACM Press, 1996.
8. [ANDR96] - Andre D., Teller A., A Study in the Response and the Negative Effects of Introns in Genetic Programming, in the proceedings of the First Conference on Genetic Programming, editor J.R. Koza, , pp. 12 - 20, AAAI Press, 1996.
9. [ANGE96] - Angeline P.J. , An Investigation into the Sensitivity of Genetic Programming to the Frequency of Leaf Selection During Subtree Crossover, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, , pp. 21 -29, MIT Press, 1996.

10. [ANGE97] - Angeline P. J., Subtree Crossover: Building Block Engine or Macromutation, Genetic Programming 1997: Proceedings of the Second Annual Conference, editors J. R. Koza, K. Deb, M. Dorigo, D. B. Fogel, M. Garzon, H. Iba and R. L. Riolo, Morgan Kaufmann, 1997, <http://www.natural-selection.com/people/pja/docs/gp97a.zip>.
11. [ARRU96] - Arruarte A., Elorriaga J.A., Fernandez-Castro I., Ferrero B., Knowledge Reusability: Some Experiences in Intelligent Tutoring Systems, Position Paper at the ITS '96 Workshop on Architectures and Methods for Designing Cost Effective and Reusable ITSs, Montreal, June 1996.
12. [BANZ98] - Banzhaf W., Nordin P., Keller R.E., Francone F.D., Genetic Programming - An Introduction - On the Automatic Evolution of Computer Programs and its Applications, Morgan Kaufmann Publishers, Inc., 1998.
13. [BARR99] - M. Barr, S. Holden, D. Phillips, T. Greening, "An Exploration of Novice Programming Errors in an Object-Oriented Environment", *SIGCSE Bulletin*, ACM Press, pp. 42-46. Vol. 31, No. 4, ACM Press, December 1999.
14. [BART95] - Barton M., Position Paper for Workshop on "Authoring Shells for Intelligent Tutoring Systems", 1995  
<http://www.pitt.edu/~al/aied/barton.html>.
15. [BEAS93] - Beasley D., Bull D. R., Martin R.R, A Sequential Niche Technique for Multimodal Function Optimization, *Evolutionary Computation*, Vol, 1, No, 2, pp. 101-125, 1993,  
<http://citeseer.nj.nec.com/beasley93sequential.html>.
16. [BECK96] - Beck J., Stern M., Haugsjaa E., Applications of AI in Education, 1996,  
<http://www.acm.org/crossroads/xrds3-1/aied.html>.
17. [BECK99] - Beck J., Stern M.K., Bringing Back the AI to AI&ED, in *Artificial Intelligence in Education: Open Learning Environments: New Computational Technologies to Support Learning, Exploration and Collaboration*, editors S. P. Lajoie, M. Vivet, pp. 233-240, IOS Press, 1999.
18. [BENN96] - Bennett III F.H., Automatic Creation of an Efficient Multi-Agent Architecture Using Genetic Programming with Architecture-Altering Operations, in *Genetic Programming 1996, Proceedings of the First Annual Conference*, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 30 -38, MIT Press, 1996.
19. [BERS97] - Bersano-Bergey T.F., Controlling Exploration, Diversity and Escaping Local Optima in GP: Adapting Weights of Training Sets to Model Resource Consumption, in *Genetic Programming 1997: Late-Breaking Papers of the 2<sup>nd</sup> Annual Conference*, J.R. Koza (ed.), Cambridge, pp. 7-11, MIT Press, 1997.

20. [BISH87]- Bishop J., Pascal Precisely, Addison-Wesley Publishing Company, 1987.
21. [BRAV96] - Brave S., Evolving Recursive Programs for Tree Search, in Advances of Genetic Programming 2, editors Angeline P., and Kinnear Jr. K.E., Chapter 10, pp. 203 - 219, MIT Press, Cambridge, 1996.
22. [BRUC95] - Bruce W. S., The Application of Genetic Programming to the Automatic Generation of Object-Oriented Programs, Phd Dissertation, School of Computer and Information Sciences, Nova Southeastern University, 1995.
23. [BRUC96]- Bruce W.S., Automatic Generation of Object-Oriented Programs Using Genetic Programming, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 267-272, MIT Press, 1996.
24. [CALL97] - Calloni B. A, Bagert D. J., Iconic Programming Proves Effective for Teaching the First Year Programming Sequence, in Proceedings of SICSE 1997 Conference, pp. 262-266, ACM Press, 1997.
25. [CHAP95] - Chappell A. R., Addressing the Trained Novice/Expert Performance Gap in Complex Dynamic Systems: A Case-Based Intelligent Tutoring System,  
[http://www.isye.gatech.edu/chmsr/Alan\\_Chappell/proposal/proposal.html](http://www.isye.gatech.edu/chmsr/Alan_Chappell/proposal/proposal.html).
26. [CHEE97] - Chee Y.S., Xu S., SIPLeS: Supporting Intermediate Smalltalk Programming through Goal-Based Learning Scenarios, in Artificial Intelligence, editors B. du Boulay and R. Mizoguchi, pp. 95 -102, IOS Press, 1997.
27. [CHEI95] - Cheikes B.A., GIA: An Agent-Based Architecture for Intelligent Tutoring Systems, in Proceedings of CIKM '95 Workshop on Intelligent Information Agents, 1995, <http://citeseer.nj.nec.com/cheikes95gia.html>
28. [CLAN87] - Clancey W.J., Knowledge Based Tutoring: The GUIDON Program, MIT Press, 1987.
29. [DALE97] - Dale N. Howard R.A., Kaczmarczyk L., Springstel F., Learning Styles and Computer Science Education (Panel), Proceedings of the 28<sup>th</sup> SIGCSE Technical Symposium on Computer Science Education (SIGCSE-97), SIGCSE Bulletin, Vol. 29, pp. 383, ACM Press, February 27 - March 1, 1997.
30. [DALE98] - Dale N., Two Threads from the Empirical Studies of Programmers, SIGCSE Bulletin, pg. 16a -17a, Vol. 30, No. 4, ACM Press, 1998.
31. [DECL96] - DeClue T., Object-Orientation and the Principles of Learning Theory: A New Look at Problems and Benefits, in Proceedings of SIGCSE '96, pp. 232 - 236, ACM Press, 1996.

32. [DEIT01] - Deitel H.M., Deitel P.J., C How to Program, 3<sup>rd</sup> Edition, Addison-Wesley Publishing, 2001.
33. [DUBO88] - B. du Boulay , “Intelligent Systems for Teaching Programming”, in Artificial Intelligence Tools in Education, IFIP Working Conference on Artificial Tools in Education, Frasacti, Italy, May 1988, pp. 5-15, 1988.
34. [EREN96] - Ehrenburg H., Improved Directed Acyclic Graph Evaluation and the Combine Operator, in Genetic Programming, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 285-290, MIT Press, 1996,.
35. [FERN99] - Fernandez J.J., Lemaerts T., The Genetic Programming Tutorial Notebook, February 1999,  
<http://cair.kaist.ac.kr/gp/Tutorial/tutorial.html>,.
36. [FREE00] - Freedman R., Ali S.S., McRoy S., “What is an Intelligent Tutoring System”, in International Journal of Artificial Intelligence in Education, pp. 15 -16, Vol. 11, No. 3, Fall 2000.
37. [GAON00] - Gaona A. L., The Relevance of Design in CS1, in SIGCSE Bulletin-Inroads, pp. 53 - 55, Vol. 32, No. 2, ACM Press, 2000.
38. [GATH96] - Gathercole C., Ross P., An Adverse Interaction Between Crossover and Restricted Tree Depth in Genetic Programming, in Genetic Programming, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 291-296, MIT Press, 1996.
39. [GATH98] - Gathercole C., An Investigation of Supervised Learning in Genetic Programming, Phd Thesis, University of Edinburgh, 1998,  
<ftp://ftp.dai.ed.ac.uk/pub/daidb/papers/pt9810.ps.gz>
40. [GINA00] - Ginat D., Colourful Examples for Elaborating Exploration of Regularities in High-School CS1, SIGCSE Bulletin, Conference Proceedings of the 5<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2000, Vol. 32, No. 3, pp. 81 -84 ACM Press, 2000.
41. [GOOL00] - Goold A., Rimmer R., Factors Affecting Performance in First-year Computing, in SIGCSE Bulletin, pp. 39 - 43, Vol. 32, No. 2, 2000,.
42. [GRAN96] - Grandbastien M., Why and How to Share Common Knowledge Bases for Building Intelligent Learning Environments, ITS'96 Workshop on Architectures and Methods for Designing Cost-Effective and Reusable ITSs, Montreal, June 10 th 1996.
43. [GRAN00] - Grant M. S., An Investigation into the Suitability of Genetic Programming for Computing Visibility Areas for Sensor Planning, Phd Thesis, Heriot-Watt University, May 2000.

44. [GROV98] - Grove R. F., Using the Personal Software Process to Motivate Good Programming Practices, in SIGCSE Bulletin, Conference Proceedings of the 6<sup>th</sup> Annual Conference on the Teaching of Computing, 1998, pp. 98 - 101, Vol. 30, No. 3, ACM Press, 1998.
45. [HAGA98] - Hagan D., Sheard J., The Value of Discussion Classes for Teaching Introductory Programming, in SIGCSE Bulletin, Conference Proceedings of the 6<sup>th</sup> Annual Conference on the Teaching of Computing, pp. 108 - 111, Vol. 30, No. 3, ACM Press, 1998.
46. [HAYN96] - Haynes T.D., Schoenfield D.A., Wainwright R.L. , Type Inheritance in Strongly Typed Genetic Programming, in Advances of Genetic Programming 2, editors Angeline P., and Kinnear Jr. K.E., Chapter 10, pp. 203 - 219, MIT Press, Cambridge, 1996.
47. [HAYN97] - Haynes T.D., Collective Adaptation: The Sharing of Building Blocks, Phd Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, 1998, <http://www.cs.twsu.edu/~haynes/thesis.ps>.
48. [HOLM95] - Holmes P., The Odin Genetic Programming System, Technical Report RR-95-3, Computer Studies Department, Napier University, December 1995.
49. [HOOP96] - Hooper D.C., Flann N. S. , Improving the Accuracy and Robustness of Genetic Programming through Expression Simplification, in Genetic Programming, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 428, MIT Press, 1996.
50. [HSIE96] - Hsieh P.Y., Workshop: Authoring Shells for Intelligent Tutoring Systems, presented at the Workshop on Authoring Shells for Intelligent Tutoring Systems, 1996.
51. [JERI97] - Jerinic L., OBOA Model of Explanation Module in Intelligent Tutoring Shell, in proceedings of ItiCSE '97, Uppsala, Sweden, pp. 133 - 135, 1997, ACM Press.
52. [JOHA98] - Johansson P., Programming by Example: An Instructional Approach Allowing Introductory Students to Quickly Grasp the Power and Excitement of Programming, in SIGCSE Bulletin, Conference Proceedings of the 6<sup>th</sup> Annual Conference on the Teaching of Computing, pp. 284, Vol. 30, No. 3, ACM Press, 1998.
53. [JOHNa] - Johnson C., What is Research in Computing Science, Department of Computer Science, Glasgow University, [http://www.dcs.gla.ac.uk/~johnson/teaching/research\\_skills/reserach.html](http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/reserach.html)
54. [JOHNb] - Johnson C., Basic Research Skills in Computer Science, Department of Computer Science, Glasgow University, [http://www.dcs.gla.ac.uk/~johnson/teaching/research\\_skills/basics.html](http://www.dcs.gla.ac.uk/~johnson/teaching/research_skills/basics.html).

- 55. [JOHN87] - Johnson W. L., Soloway E., PROUST: An Automatic Debugger for Pascal Programs, in Artificial Intelligence and Instruction: Application and Methods, edited by G.P. Kearsley, Addison-Wesley Publishing, 1987.
- 56. [JOHN90] - Johnson W. L., "Understanding and Debugging Novice Programs", in Artificial Intelligence and Learning Environments, edited by W.J. Clancy, E. Soloway, pp. 51-97, MIT Press, 1990.
- 57. [KELL96] - Keller R.E., Banzaf W., Genetic Programming Using Genotype-Phenotype Mapping from Linear Genomes to Linear Phenotypes, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 1160-122, MIT Press, 1996.
- 58. [KENT97] - Kent S., Genetic Programming, 1997  
<http://http2.brunel.ac.uk:8080/~cspgssk/documents/nca97/node2.html>.
- 59. [KIMB82] - Kimball R., A Self-Improving Tutor for Symbolic Integration, in Intelligent Tutoring Systems, editors Sleeman D. H., Brown S. J., Computers and People Series, Academic Press Inc., London, 1982.
- 60. [KNOX98] - Knox L., Repository Growth a Near Future, SIGCSE Bulletin, pp. 21a - 22a, Vol. 30, No. 4, ACM Press, 1998.
- 61. [KOFF99] - Koffman E., Wolz U., CS1 Using Java Language Features Gently, in SIGCSE Bulletin, Conference Proceedings of the 4<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education ITiCSE '99, pp. 40-43, Vol. 31, No. 3, ACM Press, 1999.
- 62. [KOLL96] - Kolling M., Rosenberg J., Blue: A Language for Teaching Object-Oriented Programming, in SIGCSE Bulletin, Vol. 28, No. 1, 1996  
<http://citeseer.nj.nec.com/michael96blue.html>.
- 63. [KOPE92] - Kopec D., Thompson R.B., Artificial Intelligence and Intelligent Tutoring Systems - Knowledge-based Systems for Teaching and Learning, Ellis Horwood, 1992.
- 64. [KOZA92] - Koza J. R., Genetic Programming I : On the Programming of Computers by Means of Natural Selection - John R. Koza, MIT Press, 1992.
- 65. [KOZA94] - Koza J.R., Genetic Programming II, Automatic Discovery of Reusable Programs, MIT Press, 1994.
- 66. [KOZA98a] - Koza J.R., Genetic Programming, in Encyclopaedia of Computer Science and Technology, editors J.G. Williams and A. Kent, Marcel-Dekker, pp. 29 - 43, Vol. 39, 1998,  
<http://www.genetic-programming.com/KENTWILLIAMS98.ps>.



67. [KOZA98b] - Koza J.R, Using Biology to Solve a Problem in Automated Machine Learning, in Models of Action: Mechanisms for Adaptive Behaviour, editors C.Wynne, J.Staddon, Lawrence Erlbaum Associates, Inc Publishers, Hillsdale, NJ,USA, Ch. 5, pp. 157 - 199, 1998, <http://www.genetic-programming.com/WYNNE.ps>.
68. [KOZA99] - Koza J.R.,Bennett III F.H.,Andre D., Keane M.A., Genetic Programming III, Darwinian Invention and Problem Solving, Morgan Kaufmann Publishers, 1999.
69. [KUMA00] - Kumar V.S., Greer J.E., McCalla G. I., Pedagogy in Peer-Supported Help Desks, in the proceedings of the KBCS-2000: International Conference on Knowledge-Based Systems, Mumbai, India, Sasikumar M., Durgesh D. R., Prakash P.R. (editors), pp. 205 - 216, Allied Publishers, New Delhi, December 2000.
70. [LANG94] - Langdon W.B., Quick Intro to simple-gp.c, 1994 [http://www.cs.bham.ac.uk/~wbl/WBL\\_papers.html](http://www.cs.bham.ac.uk/~wbl/WBL_papers.html).
71. [LANG95a] - Langdon W.B. , Pareto, Population Partitioning, Price's Theorem and Genetic Programming, 1995, [http://www.cs.bham.ac.uk/~wbl/WBL\\_papers.html](http://www.cs.bham.ac.uk/~wbl/WBL_papers.html).
72. [LANG95b] - Langdon W.B., Evolving Data Structures with Genetic Programming, in the Proceedings of ICGA-1995, 1995.
73. [LANG96a]- Langdon W.B., Directed Crossover within Genetic Programming, Technical Report RN/95/71, Department of Computer Science, University College London, April 1996.
74. [LANG96b] - Langdon W.B., Using Data Structures withing Genetic Programming, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, MIT Press, 1996,
75. [LANG96c] - Langdon W.B., Evolution of Genetic Programming Populations, Technical Report, RN/96/125, University College, London, 1996, <ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/papers/WBL.ecj.pric e.125.ps.gz>
76. [LANG97a] - Langdon W.B., Poli R., Fitness Causes Bloat: Mutation, in Late-Breaking Papers Proceedings of the Genetic Programming Conference 1997, editor J.R. Koza,, pp. 132-140, Stanford Bookstore, <ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-16.ps.gz>, 1997.
77. [LANG97b] - Langdon W.B, Poli R., Fitness Causes Bloat, Technical Report CSRP-97-09, School of Computer Science, University of Birmingham, 1997, <ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-09.ps.gz>.

78. [LANG97c] - Langdon W.B., Fitness Causes Bloat in Variable Size Representations, Position Paper, Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97, 20 July 1997, USA, <ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-14.ps.gz>.
79. [LANG98a] - Langdon W.B., Genetic Programming and Data Structures, January 1998, [http://www.cs.bham.ac.uk/~wbl/GP+DS\\_AP/conclusions.html](http://www.cs.bham.ac.uk/~wbl/GP+DS_AP/conclusions.html).
80. [LANG98b] - Langdon W.B., Genetic Programming and Data Structures, Genetic Programming + Data Structures = Automatic Programming!, Kluwer Academic Publishers, 1998.
81. [LIDT98] - Lidtke D.K., Zhou H. H., A Top-Down, Collaborative Teaching Approach to Introductory Courses in Computer Sciences, in SIGCSE Bulletin, Conference Proceedings of the 6<sup>th</sup> Annual Conference on the Teaching of Computing, pp. 291, Vol. 30, No. 3, ACM Press, 1998.
82. [MANJ97] - Manjunath B.S., Winkeler J., Genetic Programming, September 1997 <http://vivaldi.ece.ucsb.edu/projects/GP/aboutgp.html>.
83. [MARI99] - Marion W., What Should We Be Teaching, in SIGCSE Bulletin - Inroads, pp. 35 - 38, Vol. 31, No. 4, December 1999.
84. [MAWH00] - Mawhinney D., Preventing Early Convergence in Genetic Programming ByReplacing Similar Programs, 2000, <http://citeseer.nj.nec.com/mawhinney00preventing.html>.
85. [MAUF95] - Mahfoud S. W., Niching Methods for Genetic Algorithms, IlliGal Report No. 95001, University of Illinois at Urbana-Champaign, Urbana, IL, May 1995, <http://citeseer.nj.nec.com/mahfoud95niching.html>.
86. [MITR99] - Mitrovic A., Ohlsson S., Evaluation of a Constraint-Based Tutor for a Database Language, in International Journal of Artificial Intelligence in Education, pp. 238 - 256, Vol. 10, Nos. 3- 4, IOS Press, 1999.
87. [MIZO97] - Mizoguchi R., Ikeda M., Sinita K., Roles of Shared Ontology in AI-ED Research, Intelligence, Conceptualization, Standardization and Reusability, in Artificial Intelligence in Education: Knowledge and Media in Learning Systems, editors B. du Boulay, R. Mizoguchi, pp. 537-544, IOS Press, 1997.
88. [MIZO00] - Mizoguchi R., Bourdeau J., Using Ontological Engineering to Overcome Common AI-ED Problems, in International Journal of Artificial Intelligence in Education, pp. 107 - 121, Vol. 11, No. 2, IOS Press, 2000.
89. [MOOR97] - Moore F.W., Garcia O.N., A New Methodology for Reducing Brittleness in Genetic programming, in Proceedings of the National Aerospace and Electronics 1997 Conference (NAECON-97), editor E. Pohl, IEEE Press, <ftp://ftp.cs.bham.ac.uk/pub/authors/W.B.Langdon/papers/moore/moore.naecon97.ps.gz>.

90. [MURR96] - T. Murray, "From Story Boards to Knowledge Bases: The First Paradigm Shift in Making CAI Intelligent", Position Paper at the ITS '96 Workshop on Architectures and Methods for Designing Cost Effective and Reusable *ITSs*, Montreal, June 1996.
91. [MURR97] - Murray T., Expanding the Knowledge Acquisition Bottleneck for Intelligent Tutoring Systems, in International Journal of Artificial Intelligence in Education, pp. 222-232, Vol. 8., Nos 3- 4, IOS Press, 1997.
92. [MURR99] - Murray T., Authoring Intelligent Tutoring Systems: An Analysis of the State of the Art, in International Journal of Artificial Intelligence in Education, pp. 98 - 129, Vol.10, No. 1, IOS Press, 1999.
93. [NAEM98] - Naemura T., Hashiyama T., Okuma S., Module Generation for Genetic Programming and Its Incremental Evolution, in the proceedings of the Second Asia-Pacific Conference on Simulated Evolution and Learning 1998, editor C. Newton, 1998,  
<http://www.okuma.nuee.nagoya-u.ac.jp/~naemura/research/SEAL98.ps>.
94. [ODEK00] - Odekirk E., Analysing Student Programs, SIGCSE Bulletin, Conference Proceedings of the 5<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2000, pp. 191, Vol. 32, No. 3, IOS Press, 2000.
95. [OSHE82] - A Self-Improving Quadratic Tutor, in Intelligent Tutoring Systems, editors Sleeman D. H., Brown S. J., Computers and People Series, pp. 309 - 329, Academic Press Inc., London, 1982.
96. [OREI94] - O'Reilly U., Oppacher F., Program Search with a Hierarchical Variable Length Representation: Genetic Programming, Simulated Annealing and Hill Climbing, Technical Report 94-04-021, Santa Fe Institute, Mexixco, USA, 1994.
97. [PATE96] - Patel A., Kinshuk, Knowledge Characteristics: Reconsidering the Design of Intelligent Tutoring Systems, July 1996,  
<http://zeus.gmd.de/~kinshuk/papers/kt96.html>.
98. [PERK94] - Perkis T., Stack-Based Genetic Programming, in the proceedings of the 1994 IEEE World Congress on Computational Intelligence, IEEE Press, Vol.1, pp. 148-153, 1994, <ftp://ftp.mad-scientist.com/pub/genetic-programming/papers/stack-gp.ps.gz>.
99. [PILL00] - Pillay N., A Generic Architecture for the Development of Intelligent Programming Tutors, in International Journal of Continuing Engineering Education and Life-Long Learning, Vol. 10, Nos. 1-4, pp. 275 - 285, 2000.

100. [PILL01] - Pillay N., A Genetic Programming System for the Induction of Novice Programming Problems in Intelligent Programming Tutors, in Proceedings of Intech' 2001: 2nd International Conference on Intelligent Technologies, ed. T. Thitipong, Assumption University, Bangkok, pp. 180-192, November 2001.
101. [PILL02] - Pillay N., Using Genetic Programming for the Induction of Novice Procedural Programming Solution Algorithms, in ACM Proceedings of the 2002 Symposium on Applied Computing (SAC2002), Madrid, Spain, pp. 578-584, ACM Press, March 2002.
102. [PILL03a] - Pillay N., Evolving Solutions to ASCII Graphics Programming Problems in Intelligent Programming Tutors, in the Proceedings of the International Conference on Applied Artificial Intelligence, Kolhapur, India, pp. 236 - 243, TMRF, 2003.
103. [PILL03b] - Pillay N., Developing Intelligent Programming Tutors for Novice Programmers, in inroads- the SIGCSE Bulletin, Vol. 35, No. 2, pp. 78 - 82, ACM Press, June 2003.
104. [POLI97] - Poli R., Langdon W.B., Genetic Programming with One-Point Crossover and Point Mutation, Technical Report CSRP-97-13, School of Computer Science, University of Birmingham, 1997,  
<ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1997/CSRP-97-13.ps.gz>.
105. [POLI98] - Poli R., Langdon W.B., On the Ability to Search the Space of Programs of Standard, One-Point and Uniform Crossover in Genetic Programming, Technical Report CSRP-98-7, School of Computer Science, University of Birmingham, 1998,  
<ftp://ftp.cs.bham.ac.uk/pub/tech-reports/1998/CSRP-98-07.ps.gz>.
106. [PRIN95] - Pringle W.R., ESP: Evolutionary Structured Programming, 1995,  
<http://www.gv.psu.edu/personal/wrp103/wpr/esp.ps>.
107. [PROU00] - Proulx V. K., Programming Patters and Design Patterns in the Introductory Computer Science Course, in SIGCSE Bulletin, Conference Proceedings of the Thirty First SIGCSE Symposium on Computer Science Education 2000, pg. 80-84, Vol. 32, No. 1, ACM Press, 2000.
108. [REIS88] - Reiser B.R., Friedmann P., Gevins J., Kimberg D. Y., Ranney M., Romero A., A Graphical Programming Language Interface for an Intelligent Lisp Tutor, in Conference Proceedings of CHI '88: Human Factors in Computing Systems, special issue of the SIGCHI Bulletin, pp. 39-44 , ACM Press, 1988.

109. [ROBL00] - Robling G., Schler M., Freisleben B., The ANIMAL Algorithm Animation Tool, SIGCSE Bulletin, Conference Proceedings of the 5<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2000, 2000, pp. 37 - 40, Vol. 32, No. 3, ACM Press, 2000.
110. [ROSE97] - Rosenberg J. Kolling M., Testing Object-Oriented Programs: Making it Simple, in the Proceedings of SIGCSE '97, pp. 77 - 81, ACM Press, 1997.
111. [RYAN96] - Ryan C., Reducing Premature Convergence in Evolutionary Algorithms, Phd Thesis, Computer Science Department, University College, Cork, <http://tiger.ees.kyushu-u.ac.jp/~hu/Reference/Thesis/Conor.ps>, 1996.
112. [RYAN98] - Ryan C., Grammatical Evolution: A Steady State Approach, in the proceedings of the First European Workshop on Genetic Programming 1998, editor W. Banzhaf, R. Poli, M. Schoenauer and T. C. Fogarty", Vol. 1391, pg. 83-95, Springer-Verlag, <http://shine.csis.ul.ie/papers/fea98/index.html>, 1998.
113. [SING90] - Singley M.K., Carroll J.M., Alpert S.R., Psychological Design Rationale for an Intelligent Tutoring System for Smalltalk, in Empirical Studies of Programmers, Koenemann-Beliveau, Moher T.G., Robertson S.P. (editors), pp. 196 -209, Norwood, NJ: Ablex, 1991.
114. [SPEC96] - Spector L., Luke S., Cultural Transmission of Information in Genetic Programming, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 209-214, MIT Press, 1996.
115. [SOUL96] - Soule T., Foster J.A., Dickinson J., Code Growth in Genetic Programming, in Genetic Programming 1996, Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 215-223, MIT Press, 1996.
116. [SOUL98] - Soule T., Code Growth in Genetic Programming, Phd Dissertation, University of Idaho, May 1998.
117. [SUTH96] - D. Suthers, "Introduction: Architectures and Methods for Designing Cost Effective and Reusable ITS's", ITS'96 Workshop on Architectures and Methods for Designing Cost Effective and Reusable ITSS, Montreal, June 1996.
118. [TELL93] - Teller A. , Learning Mental Models, in the proceedings of the 1993 International Simulation Technologies Conference, OmniPress, 1993.
119. [TELL94a] - Teller A., Genetic Programming, Indexed Memory, The Halting Problem and Other Curiosities, in proceedings of the 7<sup>th</sup> Annual Florida AI Research Symposium, pp. 270 - 274, IEEE, 1994.

120. [TELL94b] - Teller A., Turing Completeness in the Language of Genetic Programming with Indexed Memory, in the proceedings of the 1994 First International World Congress on Computational Intelligence, pp. 136 - 146, IEEE Press, 1994.
121. [TELL94c] - Teller A. , The Evolution of Mental Models, in Advances in Genetic Programming, edited by K. Kinnear, MIT Press, 1994.
122. [TELL96a] - Teller A., Veloso M., PADO: A New Learning Architecture for Object Recognition, in Symbolic Visual Learning 1996, pp. 81 - 116, Oxford University Press, 1996.
123. [TELL96b]- Teller A., Evolving Programmers: The Co-evolution of Intelligent Recombination Operators, in Advances in Genetic Programming II 1996, edited by P. Angeline and K. Kinnear, MIT Press, 1996.
124. [TENN94] - Tennison J., EXPLAIN: On Implementing More Effective Tutoring, August 1995, <http://www.psyc.nott.ac.uk/jft/>
125. [UENO96] - Ueno H., Concepts and Methodologies for Knowledge-Based Program Understander ALPHUS, Proceedings of Psychology of Programming Interest Group '96(PPIG8), pp. 43-59, 1996.
126. [UENO00] - Ueno H., A Generalized Knowledge-Based Approach to Comprehend Pascal and C Programs, IEICE Transactions on Information Systems, Vol. E81-D, No.12, pp.1323-1329, IOS Press, 2000.
127. [WALK98] - Walker H. M., The Balance Between Programming and Other Assignments, SIGCSE Bulletin, pp. 23a -25a, Vol. 30, No. 4, ACM Press, 1998.
128. [WEBE00] - Weber-Wulff D., Combating the Code Warrior: A Different Sort of Programming Instruction, SIGCSE Bulletin, Conference Proceedings of the 5<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2000, pg. 85 -89, Vol. 32, No. 3, ACM Press, 2000.
129. [WHIG96] - Whigham P.A., Search Bias, Language Bias and Genetic Programming, Genetic Programming 1996: Proceedings of the First Annual Conference, editors J.R. Koza, D.E. Goldberg, D.B. Fogel, R.L. Riolo, pp. 230-237, MIT Press, 1996,  
<ftp://dwr-ftp.adl.dwr.csiro.au/pub/gp96/whighamgp96.ps>.
130. [WONG98] - Wong L.H., Quek C., Looi C.K., TAP-2: A Framework for an Inquiry Dialogue Based Tutoring System, in International Journal of Artificial Intelligence in Education, pp. 88 -110, Vol. 9, Nos. 1-2, IOS Press, 1998.

131. [WOOD95] - P.J. Wood, J.R. Warren, Rapid Prototyping of an Intelligent Tutorial System, 1995,  
<http://www.ascilite.org.au/conference/melbourne95/smtu/papers/woods.pdf>.
132. [WOOL87] - Woolf B. P., Theoretical Frontiers in Building a Machine Tutor, in Artificial Intelligence and Instruction, Applications and Methods, pp. 229 - 265, Addison-Wesley Publishing, 1987.
133. [XU99] - S. Xu, Y.S. Chee, "SIPLES-II: An Automatic Program Diagnoses System for Programming Learning Environments", in Artificial Intelligence in Education- Open Learning Environments: New Technologies to Support Learning, Exploration, and Collaboration, edited by S.P. Lajoie and M. Vivet, Ohmsha, July 1999, pp. 397 - 404, IOS Press, 1999.
134. [ZELL00] - Zeller A., Making Students Read and Review Code, SIGCSE Bulletin, Conference Proceedings of the 5<sup>th</sup> Annual SIGCSE/SIGUE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2000, pp. 89 -92, Vol. 32, No. 3, ACM Press, 2000.
135. [ZHAN95]- Zhang Byoung-Tak, Muhlenbein H., Balancing Accuracy and Parsimony, in Genetic Programming, in Evolutionary Computing, Vol. 3, No. 1, pp. 17 -38, 1995, [ftp://borneo.gmd.de/pub/as/ga/gmd\\_as\\_ga-94\\_09.ps](ftp://borneo.gmd.de/pub/as/ga/gmd_as_ga-94_09.ps).
136. [ZIEG99]- Ziegler U., Crews T., An Integrated Program Development Tool for Teaching and Learning How to Program, in SIGCSE Bulletin, Conference Proceedings of the Thirtieth SIGCSE Symposium on Computer Science Education, pp. 276 - 280, Vol. 31, No.1, ACM Press, 1999.

## Appendix A - Novice Programming Problems

This appendix lists the set of arbitrarily chosen problems that the genetic programming system was applied to. There are five categories of problems, namely, sequential, conditional, iterative, ASCII graphics and recursive programming problems.

### 1. Sequential Problems

1. Given the radius of a circle, write a program which prints out the circumference and area of the circle. Note that  $PI = 3.14159265$ , circumference =  $2 \times PI \times \text{radius}$ , Area =  $PI \times \text{radius} \times \text{radius}$ .
2. At the beginning of a journey the reading on a car's odometer is S kilometres and the fuel tank is full. After the journey the reading is F kilometres and it takes L litres to fill the tank. Write a program which reads in the values of S, F, and L and outputs the rate of the fuel consumption.
3. In the "old days" in Britain measurements used to be expressed in yards, feet and inches (12 inches = 1 foot, 3 feet = 1 yard). Show how to convert from inches to yards, feet and inches.
4. If an amount of money A earns R% interest over a period of P years then at the end of that time the sum will be  $T = A \times ((100 + R) / 100)^P$ . Write a program which inputs A, R and P and outputs T.
5. An object falls to the ground from a height h in time t given by  $t = (2h / g)^{0.5}$  where g is the gravitational constant (= 9.81 metres/sec<sup>2</sup>). Write a program which computes the time t.
6. Calculate the difference in volume of two three-dimensional boxes.
7. Write a program that calculates the function  $x^6 - 2x^4 + x^2$  given values of x as input.
8. Write a program to calculate the distance between two points (x1 , y1) and (x2, y2) using the formula:  $\text{sqrt}((x2 - x1)^2 + (y2 - y1)^2)$ .
9. Write a program which takes in a two digit positive integer, swaps the digits and returns the new number, e.g. an input of 23 should output 32.
10. Write a program which swaps two strings.

### 2. Conditional Problems

1. A bus company has the following charges for a tour. If a person buys less than 5 tickets they cost \$1 each, otherwise they cost 0.75. Write a program which calculates a customers bill given a number of tickets as well as data.
2. Write program that takes as input a positive integer and outputs a message indicating whether the number is even or odd.
3. Write a program which takes as input real values a, b, and c for a quadratic  $ax^2 + bx + c$  and returns the message "real roots" or "no real roots".
4. Write a program which takes a real value as input and outputs the inverse, square root and square of the value. If the value entered is zero an error message must be displayed when calculating the inverse. Similarly, if the value entered is negative, an error message must be returned when calculating the square root.
5. A certain city classifies a pollution index less than 35 as "pleasant", 35 through 60 as "unpleasant", and above 60 as "hazardous". Write a method that returns the appropriate classification for a pollution index.



6. Write a program that takes in three marks and outputs a message indicating whether the student has passed or not. A student has passed if he/she has obtained at least 50% for all three marks otherwise the student has failed.
7. Write a program that takes two positive integer numbers as input and displays a message indicating whether the first number is a multiple of the second number or not.
8. Write a program which takes an integer representing a day of the week as input and outputs the day of the week, e.g. an input of 1 results in Monday being displayed.
9. Write a program which takes a single character as input and outputs a message indicating whether the character is a vowel or consonant.
10. A mail order house sells five different products whose retail prices are shown in the following table:

| Product Number | Retail Price |
|----------------|--------------|
| 1              | \$2.98       |
| 2              | \$4.50       |
| 3              | \$9.98       |
| 4              | \$4.49       |
| 5              | \$6.87       |

Write a program that reads a pair of numbers: product number and quantity sold. Your program should use a switch statement to help determine the retail price for each product. Your program should calculate and display the total sales for a particular product.

### 3. Iterative Problems

1. Write a program that calculates and outputs the sum of a list of numbers entered by the user.
2. Find the largest number in a sequence of integers.
3. Write a program that takes in two integers  $m$  and  $n$  with  $m \leq n$  and outputs the sum of the square of the numbers from  $m$  to  $n$ .
4. Write a program which calculates the factorial of some nonnegative integer  $N$ .
5. Given a list of numbers, calculate the average of the numbers.
6. Write a program that inputs a number of single digits and combines them into an integer value. The user will enter a negative number to indicate the end of the digits. For example, inputting

2  
3  
4  
-1

should result in the program outputting 234.

7. Write a program to calculate and display the sum of the sequence  
$$\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N}.$$
8. A piece of text is terminated by the special character #. Write a program to count the number of spaces which occur in the text.
9. Write a program that reads in a string in and converts the string lower case.
10. Given an integer number  $x$  and a nonnegative integer  $n$ , use a loop to calculate  $x^n$  and output the answer.

#### 4. ASCII Graphics Problems

1. **(Language caters for “gotoxy-type” commands):** Write program to print the following two-toned square:

```
+---  
++--  
+++--  
+++++
```

2. **(Language caters for “gotoxy-type” commands):** Write a program that takes a nonzero, positive integer, *size*, and displays a square of asterisks of the specified size on the screen. For example if the user enters 3 for the size and the following square should be displayed on the screen:

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

3. **(Language caters for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer, indicating the size of the triangle, as input. The program must display an inverted right-angled triangle of asterisks of the specified size on the screen. For example, if the user enters a size of 3 the following triangle must be displayed:

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

4. **(Language caters for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer value as input and displays a right-angled number triangle on the screen. For example, if the user enters a value of 3 the following triangle must be displayed on the screen.

```
1  
2 2  
3 3 3
```

5. **(Language caters for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer value as input and displays a right-angled number triangle on the screen. For example, if the user enters a value of 3 the following triangle must be displayed on the screen.

```

1
2 2
3 3 3

```

6. **(Language that does not cater for “gotoxy-type” commands):** Write program to print the following two-toned square:

```

+---
++--
+++
++++

```

7. **(Language that does not cater for “gotoxy-type” commands):** Write a program that takes a nonzero, positive integer, *size*, and displays a square of asterisks of the specified size on the screen. For example if the user enters 3 for the size and the following square should be displayed on the screen:

```

***
***
***

```

8. **(Language that does not cater for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer, indicating the size of the triangle, as input. The program must display an inverted right-angled triangle of asterisks of the specified size on the screen. For example, if the user enters a size of 3 the following triangle must be displayed:

```

***
**
*

```

9. **(Language that does not cater for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer value as input and displays a right-angled number triangle on the screen. For example, if the user enters a value of 3 the following triangle must be displayed on the screen.

```

1
2 2
3 3 3

```

10. **(Language that does not cater for “gotoxy-type” commands):** Write a program that takes a non-zero, positive integer value as input and displays a right-angled number triangle on the screen. For example, if the user enters a value of 3 the following triangle must be displayed on the screen.

```
  1
 2 2
3 3 3
```

## 5. Recursive Problems

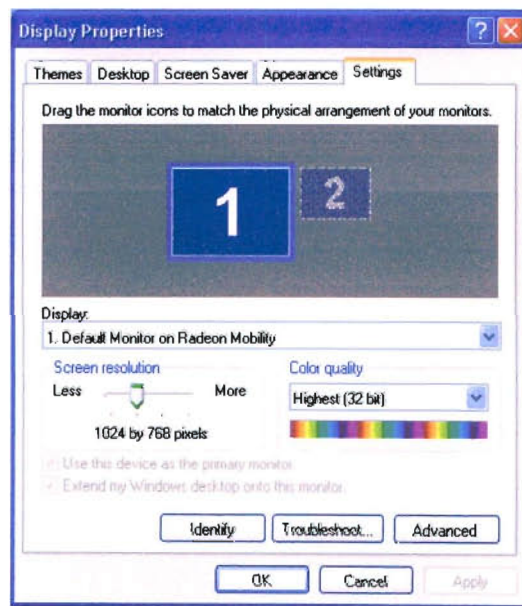
1. Write a recursive function to calculate the factorial of a non-negative number.
2. Write a recursive function to calculate the length of a given string. Each string is terminated by a hash. The string may also contain spaces. A space is represented by a tilde.
3. Write a recursive function that outputs the Nth element of the Fibonacci sequence given N.
4. The highest common factor HCF of two non-negative integers M and N can be defined as follows: If M is equal to N then the highest common factor is M or N. If M is less than N then the highest common factor is the highest common factor of M and N minus M. However, if N is less than M then the highest common factor is the highest common factor of N and M minus N. Write a recursive function to calculate the highest common factor of two non-negative integers.
5. Write a recursive function which calculates the sum of the numbers in the range one to the given positive integer N. For example, if N is 4 the function must output 10.

## Appendix B - Running Simulations

This appendix provides details on how to run simulations. Consult **Appendix C** to get an approximation of how long each simulation will take to find a solution.


### 1. System Requirements

- Windows 2000 or Windows XP
- The **D:** drive is the **CD drive**.
- Screen resolution: 1024 by 768 pixels (see **Figure B.1.1**)
- Color quality: Highest (32 bit) (see **Figure B.1.1**)



**Figure B.1.1:** Screen Resolution and Colour

### 2. Running the Program

- Double click on the batch file **D:\gp.bat** on the CD. The file is denoted by the icon:  
gp
- The following panel illustrated in **Figure B.2.1** will be displayed.

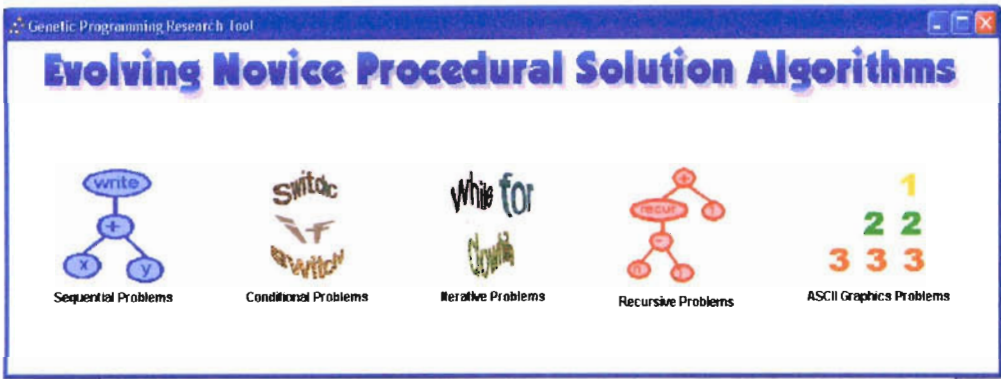


Figure B.2.1: GP Panel

3. Running Simulations

Step 1

Click on the icon in the GP Panel (see Figure B.2.1) representing the category of the problem you wish to run a simulation for. For example, if you wish to run a simulation for an iterative problem click :



Step 2

A dialog box, similar to that displayed in Figure B.3.1, will be displayed for the particular problem category.

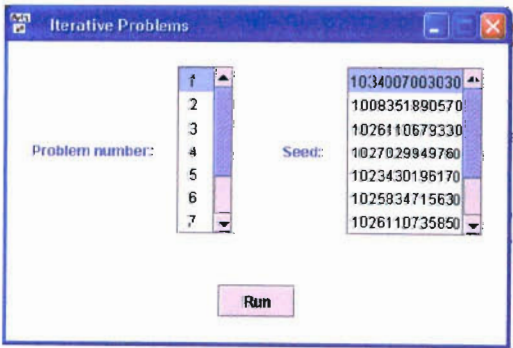

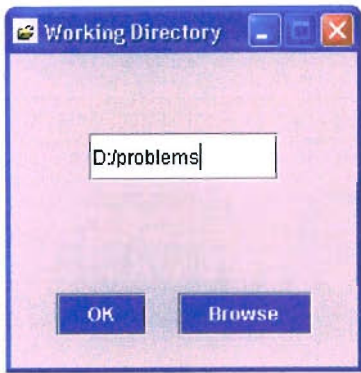


Figure B.3.1: Problem Category Dialog Box


- Click on the number of the problem you wish to run a simulation for (see Appendix A).
- Click on the seed that you wish to run a simulation for.
- Click the **Run** button  .


**Step 3**

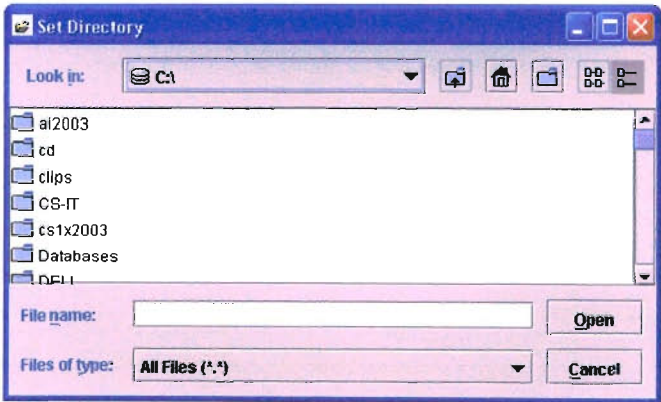
The **Working Directory** box in **Figure B.3.2** will be displayed.



**Figure B.3.2:** Working Directory Dialog Box

This box displays the directory in which the problem specification, genetic programming parameters and local optima parameters for each problem are stored. To run simulations off the CD this should be **D:/problems**. If the correct directory is displayed click the **OK** button  .

If the incorrect directory is displayed, type in the correct directory and click on the **OK** button. Alternatively, you can click on the **Browse** button  and choose the correct directory from the **Directory** dialog box illustrated in **Figure B.3.3**.



**Figure B.3.3:** Directory Dialog Box

**Step 4**

The **Simulation** box in **Figure B.3.4** will be displayed on the screen.

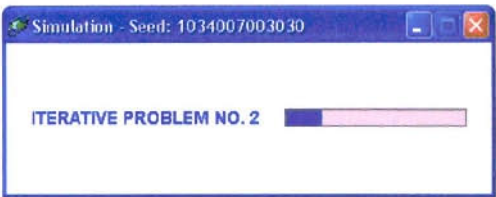



Figure B.3.4: Simulation Dialog Box

This box indicates the progress of the simulation and will remain on the screen until the end of the simulation. If you wish to stop the simulation at any point click on the **Close** button  of the window. Please note that in some cases it may take a minute or two to for the simulation to stop as the current operation (i.e. crossover, mutation, or evaluation of an individual)being performed by the system will be completed prior to the halting of the simulation.

Step 5

Once the simulation is complete a **Solution** dialog box will be displayed. If a solution has been found a box similar to that illustrated in **Figure B.3.5** will be displayed.

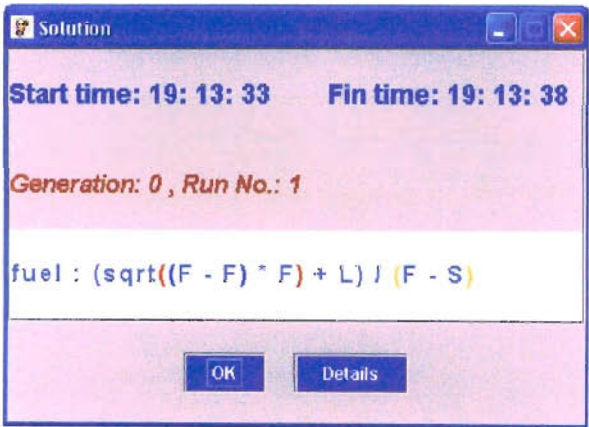
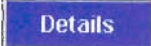

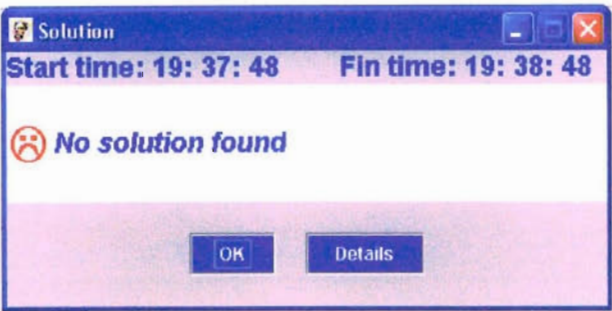


Figure B.3.5: Solution Dialog Box

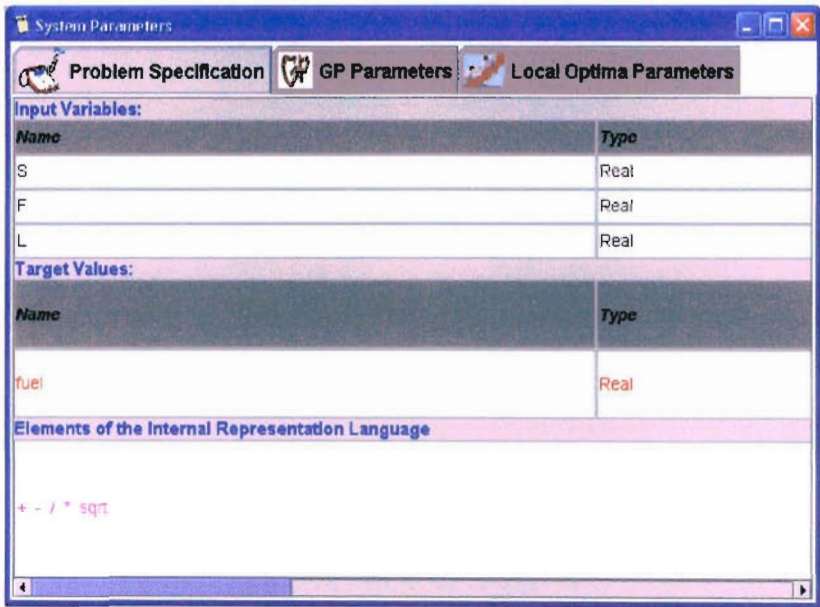
If a solution is not found a box similar to that illustrated in **Figure B.3.6** will be displayed. If you wish to view the genetic programming parameters and local optima parameters used and the problem specification click on the **Details** button .

The **System Parameters** box similar to that depicted in **Figure B.3.7** will be displayed. Click on the tab of the parameters that you wish to view. To close the **System Parameters** box click on the **Close** button .

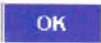




**Figure B.3.6:** Solution Dialog Box displayed if a solution is not found



**Figure B.3.7:** System Parameters Dialog Box

If you do not wish to view the system parameters click on the **OK** button  of the **Solution** box.

### Appendix C - Simulation Durations

This appendix tabulates the approximate duration of each successful simulation for each problem. The simulation time for each seed is listed. Grey columns indicate that the ISBA was used to evolve a solution. Please note that these times are approximations and may differ if the simulation is run on another machine.

| Problems Nos. |        |         |         |                |         |                  |         |                |                |         |
|---------------|--------|---------|---------|----------------|---------|------------------|---------|----------------|----------------|---------|
| Seed          | 1      | 2       | 3       | 4              | 5       | 6                | 7       | 8              | 9              | 10      |
| 1034007003030 | 4 secs | 11 secs | 36 secs | 3 hrs, 10 mins | 15 secs | 4 mins , 45 secs | 3 mins  | 11 mins        | 36 secs        | 1 sec   |
| 1008351890570 | 4 secs | 4 secs  | —       | —              | 7 secs  | 54 secs          | 13 mins | 6 hrs          | 2 mins 19 secs | 1 sec   |
| 1026110679330 | 5 secs | 5 secs  | 42 secs | —              | 14 secs | 17 mins          | 34 secs | 1hr , 30 mins  | 20 mins        | 1 sec   |
| 1027029949760 | 4 secs | 2 secs  | 36 secs | 4 hrs          | 8 secs  | 10 mins          | 5 mins  | 2 hrs, 13 mins | 6 mins         | < 1 sec |
| 1023430196170 | 4 secs | 8 secs  | —       | 4 hrs          | 12 secs | 14 mins          | 2 mins  | 1 hr, 50 mins  | 30 mins        | 1 sec   |
| 1025834715630 | 3 secs | 5 secs  | —       | —              | 5 secs  | 24 mins          | 4 mins  | 2 hrs 23 mins  | 6 mins         | 1 sec   |
| 1026110735850 | 3 secs | 5 secs  | —       | 4 hrs, 20 mins | 12 secs | 1 min , 4 secs   | 20 mins | 39 mins        | 4 mins         | < 1 sec |

| Problems Nos. |        |         |        |                   |         |         |         |                   |         |         |
|---------------|--------|---------|--------|-------------------|---------|---------|---------|-------------------|---------|---------|
| Seed          | 1      | 2       | 3      | 4                 | 5       | 6       | 7       | 8                 | 9       | 10      |
| 1023290284680 | 3 secs | 2 secs  | –      | 4 hrs             | 5 secs  | 10 mins | 13 mins | 2 hrs,<br>26 mins | 10 mins | 1 sec   |
| 1017985849240 | 4 secs | 7 secs  | 3 mins | –                 | 17 secs | 3 mins  | 2 mins  | 1 hr,<br>25 mins  | 6 mins  | 1 sec   |
| 1018055297140 | 3 secs | 21 secs |        | 3 hrs,<br>20 mins | 12 secs | 9 mins  | 4 mins  |                   | 16 mins | < 1 sec |

**Table C.1:** Approximate Duration of Simulations for **Sequential Problems**

| Problems Nos. |         |                    |                  |                     |         |         |        |         |                    |         |
|---------------|---------|--------------------|------------------|---------------------|---------|---------|--------|---------|--------------------|---------|
| Seed          | 1       | 2                  | 3                | 4                   | 5       | 6       | 7      | 8       | 9                  | 10      |
| 1034007003030 | 12 secs | 1 min,<br>41 secs  | 41 mins          | 1 min               | 4 secs  | 2 mins  | 2 secs | 29 secs | 15 secs            | 50 mins |
| 1008351890570 | 7 secs  | 5 secs             | 2hrs,<br>51 mins | 5 mins,<br>10 secs  | 11 secs | 11 mins | 2 secs | 24 secs | 20 secs            | –       |
| 1026110679330 | 15 secs | 2 mins,<br>13 secs | 2 hrs            | -                   | 9 secs  | 1 min   | 4 secs | 28 secs | 23 secs            | 12 mins |
| 1027029949760 | 5 secs  | 4 secs             | –                | 23 mins             | 9 secs  | 2 mins  | 9 secs | 31 secs | 19 secs            | 27 mins |
| 1023430196170 | 27 secs | 2 secs             | 3 mins           | 19 mins             | 2 secs  | 2 mins  | 2 secs | 22 secs | 18 secs            | 3 mins  |
| 1025834715630 | 8 secs  | 2 mins,<br>4 secs  | 41 mins          | 1 min,<br>13 secs   | 11 secs | 37 secs | 4 secs | 21 secs | 5 secs             | 27 mins |
| 1026110735850 | 2 secs  | 5 secs             | –                | 19 mins,<br>31 secs | 4 secs  | 1 min   | 2 secs | 29 secs | 26 secs            | 30 mins |
| 1023290284680 | 3 secs  | 2 secs             | –                | 2 mins              | 7 secs  | 28 secs | 1 sec  | 25 secs | 2 mins,<br>14 secs | 39 mins |
| 1017985849240 | 17 secs | 2 mins,<br>7 secs  | –                | 2 mins,<br>35 secs  | 6 secs  | 2 mins  | 2 mins | 29 secs | 15 secs            | 55 mins |
| 1018055297140 | 11 secs | 4 secs             | 1 hr,<br>40 mins | –                   | 9 secs  | 1 min   | 1 sec  | 24 secs | 18 secs            | 4 mins  |

**Table C.2:** Approximate Duration of Simulations for **Conditional Problems**

| Problems Nos. |         |                   |         |         |                  |                   |                    |                           |        |         |
|---------------|---------|-------------------|---------|---------|------------------|-------------------|--------------------|---------------------------|--------|---------|
| Seed          | 1       | 2                 | 3       | 4       | 5                | 6                 | 7                  | 8                         | 9      | 10      |
| 1034007003030 | 14 secs | 1 hr,<br>30 mins  | 29 mins | 4 mins  | –                | 1 hr,<br>18 mins  | –                  | –                         | 6 mins | 20 mins |
| 1008351890570 | 1 min   | –                 | –       | 5 mins  | –                | 2 hrs,<br>23 mins | –                  | –                         | –      | 12 mins |
| 1026110679330 | 2 mins  | 2 hrs,<br>34 mins | 23 mins | 11 secs | –                | 3 hrs             | –                  | –                         | –      | 13 mins |
| 1027029949760 | 1 min   | 2 hrs             | 17 mins | 8 mins  | 1 hr,<br>24 mins | 1 hr,<br>18 mins  | –                  | –                         | –      | 21 mins |
| 1023430196170 | 1 min   | –                 | –       | 5 mins  | –                | 47 mins           | 4 mins,<br>12 secs | –                         | –      | 20 mins |
| 1025834715630 | 1 sec   | –                 | 41 mins | 2 secs  | 1 hr,<br>38 mins | 1 hr,<br>39 mins  | –                  | –                         | –      | 28 mins |
| 1026110735850 | 1 sec   | 2 hrs             | 17 mins | 9 mins  | –                | 2 hrs,<br>13 mins | –                  | –                         | –      | –       |
| 1023290284680 | 6 secs  | 2 hrs,<br>28 mins | 17 mins | 28 secs | –                | 1 hr,<br>51 mins  | –                  | –                         | –      | 23 mins |
| 1017985849240 | 1 min   | –                 | 2 mins  | 38 secs | 1 hr,<br>19 mins | –                 | –                  | –                         | –      | –       |
| 1018055297140 | 2 mins  | –                 | 34 mins | 17 mins | 1 hr,<br>28 mins | 2 hrs,<br>14 mins | –                  | 1hr,<br>2 mins,<br>7 secs | –      | –       |

**Table C.3:** Approximate Duration of Simulations for **Iterative Problems**

| Problems Nos. |                    |                    |                   |                    |                    |                     |                   |                    |                    |                   |
|---------------|--------------------|--------------------|-------------------|--------------------|--------------------|---------------------|-------------------|--------------------|--------------------|-------------------|
| Seed          | 1                  | 2                  | 3                 | 4                  | 5                  | 6                   | 7                 | 8                  | 9                  | 10                |
| 1034007003030 | 26 secs            | 2 mins,<br>19 secs | 10 mins           | 7 secs             | 26 secs            | 6 mins              | 7 secs            | 19 secs            | 4 mins             | 5 mins            |
| 1008351890570 | 1 min              | 2 mins,<br>8 secs  | –                 | 15 secs            | 14 mins            | 5 mins,<br>5 secs   | 10 secs           | 16 mins            | 3 mins,<br>9 secs  | 35 mins           |
| 1026110679330 | 4 mins,<br>37 secs | 8 mins             | 14 mins           | 17 mins,<br>7 secs | 15 mins,<br>4 secs | 17 mins,<br>16 secs | 1 min,<br>20 secs | 1 min              | 5 mins,<br>13 secs | –                 |
| 1027029949760 | 3 mins             | 7 mins             | –                 | 7 mins             | 1 min              | 4 mins              | 30 secs           | 22 secs            | –                  | –                 |
| 1023430196170 | 2 mins             | 6 mins,<br>23 secs | –                 | 2 mins             | 6 mins             | 9 mins              | 4 secs            | 28 secs            | –                  | –                 |
| 1025834715630 | No.<br>Soln.       | 2 mins,<br>12 secs | 1 min,<br>14 secs | 2 mins             | 4 mins,<br>35 secs | 2 mins              | 8 secs            | 3 mins,<br>22 secs | –                  | –                 |
| 1026110735850 | 15 mins            | 11 mins.<br>4 secs | 5 secs            | 7 mins             | 2 mins             | 9 mins              | 4 mins            | 4 mins,<br>17 secs | –                  | 2 mins,<br>9 secs |
| 1023290284680 | 41 secs            | 9 mins             | –                 | 15 secs            | 5 mins             | 5 mins,<br>40 secs  | 10 secs           | 3 mins             | –                  | –                 |
| 1017985849240 | 25 secs            | 4 mins,<br>27 secs | –                 | 2 mins             | 4 mins,<br>29 secs | 26 secs             | 7 secs            | 3 mins             | –                  | –                 |
| 1018055297140 | 3 mins             | 3 secs             | –                 | 4 mins             | 6 mins,<br>19 secs | 14 secs             | 4 mins            | 40 secs            | –                  | 25 mins           |

**Table C.4:** Approximate Duration of Simulations for ASCII Graphics Problems



| Problems Nos. |                   |                          |         |                    |                    |
|---------------|-------------------|--------------------------|---------|--------------------|--------------------|
| Seed          | 1                 | 2                        | 3       | 4                  | 5                  |
| 1034007003030 | 4 hrs,<br>1 min   | 10<br>mins               | 32 mins | 33 secs            | 3 mins,<br>25 secs |
| 1008351890570 | 36 mins           | 5 mins                   | –       | 4 mins             | –                  |
| 1026110679330 | –                 | –                        | –       | 32 secs            | 12 mins            |
| 1027029949760 | –                 | –                        | –       | 4 mins,<br>35 secs | –                  |
| 1023430196170 | 4 hrs             | –                        | –       | 53 secs            | –                  |
| 1025834715630 | 2 hrs,<br>11 mins | –                        | –       | 34 secs            | –                  |
| 1026110735850 | 2 hrs             | 8<br>mins,<br>43<br>secs | –       | 4 mins             | –                  |
| 1023290284680 | 3 hrs,<br>13 mins | 5 mins                   | –       | 13 mins            | –                  |
| 1017985849240 | 2 hrs,<br>20 mins | –                        | –       | 4 mins,<br>3 secs  | –                  |
| 1018055297140 | –                 | –                        | –       | 3 mins             | –                  |

**Table C.5:** Approximate Duration of Simulations for **Recursive Problems**