UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Genetic Algorithm with Temporal Logic for Automated Markers

by

Francois Jacobus Redelinghuys

Submitted in partial fulfillment of the requirements for the degree
Master of Science (Computer Science)
in the Faculty of Engineering, Built Environment and Information Technology
University of Pretoria, Pretoria

July 2025

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Publication data:

Francois Jacobus Redelinghuys. Genetic Algorithm with Temporal Logic for Automated Markers. Master's dissertation, University of Pretoria, Department of Computer Science, Pretoria, South Africa, December 2024.

Electronic, hyperlinked versions of this dissertation are available online, as Adobe PDF files, at:

http://upetd.up.ac.za/UPeTD.htm

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Genetic Algorithm with Temporal Logic for Automated Markers

by

Francois Jacobus Redelinghuys
E-mail: u19089296@tuks.co.za

Abstract

Automated assessment is essential for large-scale programming courses, as it addresses the challenges of evaluating correctness and providing feedback efficiently. Traditional methods of automated assessment rely on hand-crafted test cases, which are time-intensive and lack scalability. Automated test generation methods, such as random input generation or genetic algorithms combined with code coverage metrics, offer alternatives, but often fail to capture the complexity required for educational assessments. To address this, a novel approach that integrates genetic algorithms with linear temporal logic (LTL) formulae is proposed. LTL properties, commonly used in model checking, formalise correctness criteria for programming assignments. The proposed method evolves test cases designed to maximise violations of the LTL properties in student submissions, determining correctness and generating personalised feedback by providing examples of failing test cases. Experiments were conducted to evaluate the approach in terms of sensitivity to hyperparameter settings, and effectiveness relative to the established techniques, such as random input generation and code coverage-based genetic algorithms. Results demonstrate that combining genetic algorithms with LTL properties enhances automated assessment accuracy and feedback quality, offering a solution to the problems faced by automated assessment in large educational settings for both students and lecturers.

Keywords : Genetic Algorithm, Temporal Logic, Automated Marking
Supervisors : Dr. A. S. Bosman, Prof. L. Marshall, Dr. N. Timm
Department : Department of Computer Science
Degree : Master of Science

Contents

List of Figures vii

List of Algorithms xi

List of Tables xii

1 Introduction 1

1.1 Motivation . . . 3

1.2 Objective and Research Questions . . . 4

1.3 Contributions . . . 5

1.4 Dissertation Outline . . . 6

2 Genetic Algorithms 10

2.1 Preliminaries . . . 10

2.2 Main Algorithm . . . 12

2.3 Initial Population Generation . . . 12

2.4 Termination Criterion . . . 13

2.5 Selection Method . . . 14

2.5.1 Fitness Function . . . 14

2.5.2 Different Selection Methods . . . 14

2.6 Genetic Operators . . . 16

2.6.1 Reproduction . . . 16

2.6.2 Crossover . . . 16

2.6.3 Mutation . . . 17

2.7 Summary . . . 18

3 Software Testing Methods **20**
3.1 Manual Software Testing Methods 20
3.1.1 White Box Testing 21
3.1.2 Black Box Testing 24
3.2 Automated Test Generation Methods 27
3.2.1 Model-Based Test Generation 27
3.2.2 Property-Based Testing 28
3.2.3 Requirement-Based Test Generation 28
3.2.4 Static and Dynamic Automated Test Generation 29
3.2.5 Bounded-Exhaustive Test Generation 30
3.2.6 Symbolic Execution-Based Testing 30
3.2.7 Random-Based Test Generation 30
3.3 Genetic Algorithms for Software Testing 31
3.4 Temporal Logic Model Checking 33
3.4.1 Temporal Logic Formulae 33
3.4.2 Variations of Temporal Logic Formulae 34
3.4.3 Model Checking Approach 35
3.4.4 Temporal Logic Formulae in Artificial Intelligence 35
3.5 Summary 36

4 Theory of Automated Assessment **37**
4.1 Background 38
4.2 Feedback 39
4.3 Benefits of Automated Assessment Strategies 40
4.4 Drawbacks of Automated Assessment Strategies 42
4.5 Automated Assessment Considerations 43
4.6 Types of Automated Assessment 46
4.6.1 Static Automated Assessment 47
4.6.2 Dynamic Automated Assessment 47
4.6.3 Advantages and Disadvantages of Static and Dynamic Assessment 47
4.6.4 Hybrid Automated Assessment 48
4.7 Aspects Used to Compare Automated Assessment 49

4.7.1 Common Aspects . . . 49
4.7.2 Extended Aspects . . . 52
4.8 Summary . . . 53

5 Automated Assessment in Practice **54**
5.1 Static Assessment Strategies . . . 54
5.2 Dynamic Automated Assessment Strategies . . . 57
5.2.1 Common Strategies . . . 57
5.2.2 Distinctive Strategies . . . 67
5.3 Summary . . . 69

6 Proposed Assessment Strategy **71**
6.1 GATLAM Overview . . . 71
6.2 GATLAM Internal Details . . . 73
6.2.1 LTL Properties . . . 73
6.2.2 Genetic Algorithm . . . 80
6.2.3 Solutions . . . 85
6.2.4 Interpreter . . . 86
6.2.5 Running modes . . . 87
6.3 Benefits, Drawbacks and Concerns . . . 90
6.3.1 Benefits . . . 90
6.3.2 Drawbacks . . . 94
6.3.3 Aspects Used to Compare Automated Assessment . . . 95
6.3.4 Additional Concerns . . . 97
6.4 Summary . . . 99

7 GATLAM Experimental Setup **100**
7.1 Error Pools . . . 101
7.1.1 Concurrency Errors . . . 101
7.1.2 Infinite Loop Errors . . . 102
7.1.3 Logic Errors . . . 104
7.1.4 Memory Errors . . . 108

7.2 Hyperparameter Tuning Experiments . . . 110
7.2.1 Number of Generations Experiments . . . 112
7.2.2 GA-specific Experiments . . . 112
7.2.3 Omega (ω) Weights Experiments . . . 113
7.3 Benchmarking Automated Software Testing Tools Experiments . . . 114
7.3.1 RNG (Fuzzing) . . . 115
7.3.2 CCGA . . . 115
7.3.3 Comparison Experiments . . . 116
7.4 Student Assignment Experiments . . . 117
7.5 Summary . . . 119

8 Hyperparameter Sensitivity Results 120
8.1 Metrics . . . 120
8.1.1 Average Fitness . . . 121
8.1.2 Best Fitness . . . 121
8.1.3 Population Diversity . . . 121
8.1.4 Percentage of Errors Found . . . 122
8.2 Number of Generations Experiment Results . . . 123
8.3 GA Specific Hyperparameter Experiment Results . . . 124
8.3.1 Crossover Probability . . . 125
8.3.2 Crossover Type . . . 126
8.3.3 Mutation Probability . . . 131
8.3.4 Mutation Type . . . 134
8.3.5 Population Size . . . 137
8.3.6 Reproduction Probability . . . 142
8.3.7 Selection Method . . . 147
8.3.8 Tournament Size . . . 152
8.3.9 Summary and Discussion . . . 157
8.4 Summary . . . 159

9 Omega (ω) Weights Experimental Results 163
9.1 Average Fitness Value . . . 163

9.2 Best Fitness . . . 166
9.3 Population Diversity . . . 166
9.4 Errors Found . . . 168
9.5 Discussion . . . 170

10 Benchmarking and Real-World Experimental Results 173
10.1 Benchmarking Experimental Results . . . 174
10.1.1 Average Percentage of Errors Found . . . 174
10.1.2 Execution Duration . . . 176
10.1.3 Summary . . . 177
10.2 Mass Single Run Real-World Experimental Results . . . 178
10.2.1 Errors Found Per Generation . . . 179
10.2.2 Percentage of the Output that Matched Before First Error Occurred 182
10.2.3 Types of Errors Found . . . 184
10.2.4 Final Generation Error Types Found . . . 187
10.2.5 Example Feedback . . . 188
10.2.6 Summary . . . 189
10.3 Selected Multiple Run Experimental Results . . . 189
10.3.1 Errors Found . . . 190
10.3.2 Types of Errors Found . . . 192
10.3.3 Percentage of the Output that Matched Before First Error Occurred 194
10.3.4 Summary . . . 194
10.4 Summary . . . 196

11 Conclusion 197
11.1 Attainment of Research Objective . . . 197
11.1.1 Utilising a GA for Automated Assessment . . . 198
11.1.2 LTL Properties as Part of GATLAM's Fitness Function . . . 199
11.1.3 Memory Component as Part of GATLAM's Fitness Function . . . 200
11.1.4 Benchmarking GATLAM . . . 201
11.2 Contributions . . . 202
11.3 Future Work . . . 203

Bibliography **205**

A LTL Negations **217**
A.1 General Notation: . . . 217
A.2 Safety: . . . 218
A.3 Proper Termination: . . . 219
A.4 Segmentation faults: . . . 219
A.5 Exceptions: . . . 220
A.6 Execution time: . . . 220
A.7 Illegal output: . . . 221
A.8 Expected output: . . . 222
A.8.1 Exact match: . . . 222
A.8.2 Contains: . . . 223

B Real Assignment **224**

C Supporting Results **237**
C.1 GA Specific Experiments . . . 237
C.1.1 Crossover Probability . . . 237
C.1.2 Crossover Type . . . 240
C.1.3 Mutation Probability . . . 244
C.1.4 Mutation Type . . . 247
C.1.5 Population Size . . . 250
C.1.6 Reproduction Probability . . . 253
C.1.7 Selection Method . . . 257
C.1.8 Tournament Size . . . 261
C.2 Omega (ω)-weight Experiments . . . 264
C.2.1 Average Fitness . . . 265
C.2.2 Best Fitness . . . 265
C.2.3 Population Diversity . . . 265
C.2.4 Percentage of Errors Found . . . 267

D Example Feedback **272**

List of Figures

1.1 Diagram depicting the flow and dependencies of the chapters within the dissertation. 7

2.1 Simplified generic genetic algorithm. 13
2.2 One point crossover example where the shaded area shows the portion of the chromosome which is swapped. 16
2.3 N point crossover, $N = 2$, where the shaded area shows the portion of the chromosome which is swapped. 17
2.4 Uniform Crossover, where the shaded area shows the portion of the chromosome which is swapped. 17
2.5 Single bit inversion, where the purple gene is flipped from 0 to 1. 18
2.6 Bitwise inversion 18
2.7 Random selection mutation. 18

3.1 Example for CFG Analysis 23
3.2 Cause-Effect Graph example from Srivastava et al. [98] 26

6.1 Abstracted view of GATLAM, with the starting and end point being the GA step. 72
6.2 Example of the data flow of GATLAM when configured in SS mode. . . 88
6.3 Example of the data flow of GATLAM when configured in ISS mode. . 89

7.1 Diagram illustrating how the hyperarameter tuning experiments will be conducted. 111

8.1 Experimental results for the number of generations. 124

8.2 Average fitness of the different crossover probability hyperparameter values for different error pools. 126
8.3 Best fitness for the different crossover probability values for different error pools. 127
8.4 Population diversity for the different crossover probability values for different error pools. 128
8.5 Percentage of errors found by the best chromosome for the different crossover probability values using different error pools. 129
8.6 Average fitness for the different crossover types for different error pools. 130
8.7 Best fitness for the different crossover types for different error pools. . . 131
8.8 Diversity of the population for the different crossover types for different error pools. 132
8.9 Percentage of errors found by the best chromosome for the different crossover types for different error pools. 133
8.10 Average fitness for the different mutation probabilities for different error pools. 135
8.11 Best fitness for the different mutation probabilities for different error pools. 136
8.12 Diversity of the population for the different mutation probabilities for different error pools. 137
8.13 Percentage of errors found by the best chromosome for the different mutation probabilities for different error pools. 138
8.14 Average fitness for the different mutation types for different error pools. 139
8.15 Best fitness for the different mutation types for different error pools. . . 140
8.16 Diversity of the population for the different mutation types for different error pools. 141
8.17 Percentage of errors found by the best chromosome for the different mutation type for different error pools. 142
8.18 Average fitness for the different population sizes for different error pools. 143
8.19 Best fitness for the different population sizes for different error pools. . 144

8.20 Diversity of the population for the different population sizes for different error pools. 145
8.21 Percentage of errors found by the best chromosome for the different population sizes for different error pools. 146
8.22 Average fitness for the different reproduction probability for different error pools. 147
8.23 Best fitness for the different reproduction probability for different error pools. 148
8.24 Diversity of the population for the different reproduction probabilities for different error pools. 149
8.25 Percentage of errors found by the best chromosome for the different reproduction probability for different error pools. 150
8.26 Average fitness for the different selection methods for different error pools. 152
8.27 Best fitness for the different selection methods for different error pools. 153
8.28 Diversity of the population for the different selection methods for different error pools. 154
8.29 Percentage of errors found by the best chromosome for the different selection method for different error pools. 155
8.30 Average duration (minutes) per generation for the different selection methods for different error pools. 156
8.31 Average fitness for the different tournament sizes for different error pools. 158
8.32 Best fitness for the different tournament sizes for different error pools. . 159
8.33 Diversity of the population for the different tournament sizes for different error pools. 160
8.34 Percentage of errors found by the best chromosome for the different tournament sizes for different error pools. 161
8.35 Average duration per generation for the different tournament sizes for different error pools. 162

9.1 Average fitness value with GATLAM configured to use different ω-weights for different error pools. 165

9.2 Best fitness value per generation with GATLAM configured to use different ω-weights for different error pools. 167
9.3 Diversity in the population with GATLAM configured to use different ω-weights for different error pools. 169
9.4 Average percentage of errors found by GATLAM when configured to use different ω-weights for different error pools. 172

10.1 Different automated software test generation methods' average percentage of errors found for different error pools. Best, Worst, and Default refer to the different configurations of GATLAM. 175
10.2 Different automated software test generation methods' average execution duration for different error pools. 177
10.3 Errors found per student per generation. Red: error found, blue: no error found, grey: compilation failure. 181
10.4 Percentage of the output matched between the student and the instructor solutions before the first error occurred for different automated assessment strategies. 183
10.5 Count of each type of error that different automated assessment strategies were able to find in different student code submissions. 185
10.6 Total number of error types that different automated assessment strategies were able to find. 187
10.7 Number of types of errors that different automated assessment strategies were able to find in the final generation. 188
10.8 Number of errors different automated assessment strategies were able to find in the final generation with the standard deviation overlaid as a line. 191
10.9 Types of errors that the different automated assessment strategies were able to find in the final generation. 193
10.10 Execution distance before an error was found by different automated assessment strategies. 195

List of Algorithms

3.1 Example used for Decision/Condition Coverage 21
3.2 Example used for Loop Testing 22
3.3 Example for Boundary Value Analysis 24
3.4 Example for Equivalence Partitioning 25
6.1 Example code for *safety* property. 76
6.2 Example code for *proper termination* property. 77
6.3 Example code for *segmentation fault* property. 77
6.4 Example code for *exception* property. 78
6.5 Example code for *execution time* property. 78
6.6 Example code for *illegal output* property 79
6.7 Example code for *expected output - exact match* property. 79
6.8 Example code for *expected output - contains* property. 80
7.1 Infinite Loop Error 1's code. 103
7.2 Infinite Loop Error 2's code. 103
7.3 Infinite Loop Error 3's code. 104
7.4 Logic Error 1's code. 105
7.5 Logic Error 2's code. 106
7.6 Logic Error 3's code 107
7.7 Logic Error 4's code. 107
7.8 Logic Error 5's code. 108
7.9 Memory Error 2's code. 109

List of Tables

3.1 Prime and Simple paths for Figure 3.1. 23
3.2 Analysis of possible boundary values. 25

4.1 Advantages and disadvantages of static and dynamic automated assessment strategies. 48

5.1 Comparison of common dynamic automated assessment strategies with a set of common aspects. 65
5.2 Comparison of common dynamic automated assessment strategies with a set of extended aspects. 66
5.3 Comparison of different common assessment strategies in terms of how each complies with the benefits and drawbacks of automated assessment. 67
5.4 Comparison of distinctive dynamic automated assessment strategies with a set of common aspects. 70
5.5 Comparison of distinctive dynamic automated assessment strategies with a set of extended aspects. 70
5.6 Comparison of different distinctive assessment strategies in terms of benefits and drawbacks. 70

6.1 LTL properties, defined by name, LTL formula, and the negation of the LTL formula. 74
6.2 GATLAM's GA Parameters. 82
6.3 Theoretical execution time differences for different configurations of GATLAM. 92

7.1 Default GA parameter values for the number of generation experiments. 112

7.2 Parameter names, the values/types for each parameter that was tried, and the default values/type for each iteration of the GA-specific hyperparameter tuning. 113
7.3 Grid search table for the ω-weight experiments. 114

8.1 Average values for the duration (minutes) for each error pool using different values for the selection method hyperparameter. 153
8.2 Final values for the duration (minutes) for each error pool using different values for the selection method hyperparameter. 154
8.3 Average values for the duration (minutes) for each error pool using different values for the tournament size hyperparameter. 157
8.4 Final values for the duration (minutes) for each error pool using different values for the tournament size hyperparameter. 157
8.5 Summary of the best and worst hyperparameters for the GA specific hyperparameters of GATLAM. 161

C.1 Average values for the average fitness for each error pool using different values for the crossover probability hyperparameter. 238
C.2 Standard deviation for the average fitness for each error pool using different values for the crossover probability hyperparameter. 238
C.3 Final values for the average fitness for each error pool using different values for the crossover probability hyperparameter. 238
C.4 Average values for the best fitness for each error pool using different values for the crossover probability hyperparameter. 238
C.5 Standard deviation for the best fitness for each error pool using different values for the crossover probability hyperparameter. 239
C.6 Final values for the best fitness for each error pool using different values for the crossover probability hyperparameter. 239
C.7 Average values for the diversity for each error pool using different values for the crossover probability hyperparameter. 239
C.8 Standard deviation for the diversity for each error pool using different values for the crossover probability hyperparameter. 239

C.9 Final values for the diversity for each error pool using different values for the crossover probability hyperparameter. 240
C.10 Average values for the percentage of errors for each error pool using different values for the crossover probability hyperparameter. 240
C.11 Standard deviation for the percentage of errors for each error pool using different values for the crossover probability hyperparameter. 240
C.12 Final values for the percentage of errors for each error pool using different values for the crossover probability hyperparameter. 240
C.13 Average values for the average fitness for each error pool using different values for the crossover type hyperparameter. 241
C.14 Standard deviation for the average fitness for each error pool using different values for the crossover type hyperparameter. 241
C.15 Final values for the average fitness for each error pool using different values for the crossover type hyperparameter. 241
C.16 Average values for the best fitness for each error pool using different values for the crossover type hyperparameter. 242
C.17 Standard deviation for the best fitness for each error pool using different values for the crossover type hyperparameter. 242
C.18 Final values for the best fitness for each error pool using different values for the crossover type hyperparameter. 242
C.19 Average values for the diversity for each error pool using different values for the crossover type hyperparameter. 242
C.20 Standard deviation for the diversity for each error pool using different values for the crossover type hyperparameter. 243
C.21 Final values for the diversity for each error pool using different values for the crossover type hyperparameter. 243
C.22 Average values for the percentage of errors for each error pool using different values for the crossover type hyperparameter. 243
C.23 Standard deviation for the percentage of errors for each error pool using different values for the crossover type hyperparameter. 243

C.24 Final values for the percentage of errors for each error pool using different values for the crossover type hyperparameter. 244
C.25 Average values for the average fitness for each error pool using different values for the mutation probability hyperparameter. 244
C.26 Standard deviation for the average fitness for each error pool using different values for the mutation probability hyperparameter. 244
C.27 Final values for the average fitness for each error pool using different values for the mutation probability hyperparameter. 245
C.28 Average values for the best fitness for each error pool using different values for the mutation probability hyperparameter. 245
C.29 Standard deviation for the best fitness for each error pool using different values for the mutation probability hyperparameter. 245
C.30 Final values for the best fitness for each error pool using different values for the mutation probability hyperparameter. 246
C.31 Average values for the diversity for each error pool using different values for the mutation probability hyperparameter. 246
C.32 Standard deviation for the diversity for each error pool using different values for the mutation probability hyperparameter. 246
C.33 Final values for the diversity for each error pool using different values for the mutation probability hyperparameter. 246
C.34 Average values for the percentage of errors for each error pool using different values for the mutation probability hyperparameter. 247
C.35 Standard deviation for the percentage of errors for each error pool using different values for the mutation probability hyperparameter. 247
C.36 Final values for the percentage of errors for each error pool using different values for the mutation probability hyperparameter. 247
C.37 Average values for the average fitness for each error pool using different values for the mutation type hyperparameter. 248
C.38 Standard deviation for the average fitness for each error pool using different values for the mutation type hyperparameter. 248

C.39 Final values for the average fitness for each error pool using different values for the mutation type hyperparameter. 248
C.40 Average values for the best fitness for each error pool using different values for the mutation type hyperparameter. 248
C.41 Standard deviation for the best fitness for each error pool using different values for the mutation type hyperparameter. 249
C.42 Final values for the best fitness for each error pool using different values for the mutation type hyperparameter. 249
C.43 Average values for the diversity for each error pool using different values for the mutation type hyperparameter. 249
C.44 Standard deviation for the diversity for each error pool using different values for the mutation type hyperparameter. 249
C.45 Final values for the diversity for each error pool using different values for the mutation type hyperparameter. 249
C.46 Average values for the percentage of errors for each error pool using different values for the mutation type hyperparameter. 250
C.47 Standard deviation for the percentage of errors for each error pool using different values for the mutation type hyperparameter. 250
C.48 Final values for the percentage of errors for each error pool using different values for the mutation type hyperparameter. 250
C.49 Average values for the average fitness for each error pool using different values for the population size hyperparameter. 251
C.50 Standard deviation for the average fitness for each error pool using different values for the population size hyperparameter. 251
C.51 Final values for the average fitness for each error pool using different values for the population size hyperparameter. 251
C.52 Average values for the best fitness for each error pool using different values for the population size hyperparameter. 252
C.53 Standard deviation for the best fitness for each error pool using different values for the population size hyperparameter. 252

C.54 Final values for the best fitness for each error pool using different values for the population size hyperparameter. 252
C.55 Average values for the diversity for each error pool using different values for the population size hyperparameter. 253
C.56 Standard deviation for the diversity for each error pool using different values for the population size hyperparameter. 253
C.57 Final values for the diversity for each error pool using different values for the population size hyperparameter. 253
C.58 Average values for the percentage of errors for each error pool using different values for the population size hyperparameter. 254
C.59 Standard deviation for the percentage of errors for each error pool using different values for the population size hyperparameter. 254
C.60 Final values for the percentage of errors for each error pool using different values for the population size hyperparameter. 254
C.61 Average values for the average fitness for each error pool using different values for the reproduction probability hyperparameter. 255
C.62 Standard deviation for the average fitness for each error pool using different values for the reproduction probability hyperparameter. 255
C.63 Final values for the average fitness for each error pool using different values for the reproduction probability hyperparameter. 255
C.64 Average values for the best fitness for each error pool using different values for the reproduction probability hyperparameter. 256
C.65 Standard deviation for the best fitness for each error pool using different values for the reproduction probability hyperparameter. 256
C.66 Final values for the best fitness for each error pool using different values for the reproduction probability hyperparameter. 256
C.67 Average values for the diversity for each error pool using different values for the reproduction probability hyperparameter. 256
C.68 Standard deviation for the diversity for each error pool using different values for the reproduction probability hyperparameter. 257

C.69 Final values for the diversity for each error pool using different values for the reproduction probability hyperparameter. 257
C.70 Average values for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter. 257
C.71 Standard deviation for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter. 257
C.72 Final values for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter. 258
C.73 Average values for the average fitness for each error pool using different values for the selection method hyperparameter. 258
C.74 Standard deviation for the average fitness for each error pool using different values for the selection method hyperparameter. 258
C.75 Final values for the average fitness for each error pool using different values for the selection method hyperparameter. 259
C.76 Average values for the best fitness for each error pool using different values for the selection method hyperparameter. 259
C.77 Standard deviation for the best fitness for each error pool using different values for the selection method hyperparameter. 259
C.78 Final values for the best fitness for each error pool using different values for the selection method hyperparameter. 259
C.79 Average values for the diversity for each error pool using different values for the selection method hyperparameter. 260
C.80 Standard deviation for the diversity for each error pool using different values for the selection method hyperparameter. 260
C.81 Final values for the diversity for each error pool using different values for the selection method hyperparameter. 260
C.82 Average values for the percentage of errors for each error pool using different values for the selection method hyperparameter. 260
C.83 Standard deviation for the percentage of errors for each error pool using different values for the selection method hyperparameter. 261

C.84 Final values for the percentage of errors for each error pool using different values for the selection method hyperparameter. 261
C.85 Average values for the average fitness for each error pool using different values for the tournament size hyperparameter. 261
C.86 Standard deviation for the average fitness for each error pool using different values for the tournament size hyperparameter. 262
C.87 Final values for the average fitness for each error pool using different values for the tournament size hyperparameter. 262
C.88 Average values for the best fitness for each error pool using different values for the tournament size hyperparameter. 262
C.89 Standard deviation for the best fitness for each error pool using different values for the tournament size hyperparameter. 262
C.90 Final values for the best fitness for each error pool using different values for the tournament size hyperparameter. 263
C.91 Average values for the diversity for each error pool using different values for the tournament size hyperparameter. 263
C.92 Standard deviation for the diversity for each error pool using different values for the tournament size hyperparameter. 263
C.93 Final values for the diversity for each error pool using different values for the tournament size hyperparameter. 263
C.94 Average values for the percentage of errors for each error pool using different values for the tournament size hyperparameter. 264
C.95 Standard deviation for the percentage of errors for each error pool using different values for the tournament size hyperparameter. 264
C.96 Final values for the percentage of errors for each error pool using different values for the tournament size hyperparameter. 264
C.97 Average values for the average fitness for each error pool using different weight configurations for the fitness function. 265
C.98 Standard deviation for the average fitness for each error pool using different values for the Fitness Function hyperparameter. 266

C.99 Final values for the average fitness for each error pool using different weight configurations for the fitness function. 266
C.100 Average values for the best fitness for each error pool using different weight configurations for the fitness function. 267
C.101 Standard deviation for the best fitness for each error pool using different values for the Fitness Function hyperparameter. 267
C.102 Final values for the best fitness for each error pool using different weight configurations for the fitness function. 268
C.103 Average values for the diversity for each error pool using different weight configurations for the fitness function. 268
C.104 Standard deviation for the diversity for each error pool using different values for the Fitness Function hyperparameter. 269
C.105 Final values for the diversity for each error pool using different weight configurations for the fitness function. 269
C.106 Average values for the percentage of errors for each error pool using different weight configurations for the fitness function. 270
C.107 Standard deviation for the percentage of errors for each error pool using different values for the Fitness Function hyperparameter. 270
C.108 Final values for the percentage of errors for each error pool using different weight configurations for the fitness function. 271

Chapter 1

Introduction

> *In a University dormitory late at night, a first-year student is struggling to understand why their apparently flawless code is failing on the automated marker their programming module is utilising to mark assignments. The student is feeling demotivated as the feedback that is being provided is vague, and they do not have access to the test cases used to evaluate their assignment.*

In large programming modules where the student body is over 200 students, it is standard practice to make use of an automated assessment tool to mark the programming assessments of students [85]. Automated assessment tools refer to software that implements an automated assessment strategy. Some examples of traditional automated assessment tools are FitchFork and BOSS[1], which utilise software testing methods to assess a student's submitted programming assignment for correctness [85].

Software testing methods determine if a piece of software is correct by considering the system's behaviour, or output, based on a set of inputs, compared to the expected behaviour of the system described in the system specification [5, 32]. Following on from this idea, it is common for automated assessment strategies to determine correctness by executing the student's assignment with some form of input test data, and then comparing the output of the student's assignment to the expected output. The process

[1] FitchFork is an automated assessment tool developed by the University of Pretoria's Computer Science department, while BOSS serves a similar purpose to FitchFork, except it was developed at the University of Warwick. Both strategies are discussed in greater detail in Chapter 5.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

of creating test input data manually is time-consuming, and as such, a need arises to automate not only the assessment portion of the automated assessment, but also the creation of test input data. The created test input data are often static and do not change during the evaluation of the student's submitted code [10, 41], which means that if an instructor did not design a test case for a specific task, that task will not be evaluated. As such, the test cases are only as good as the human expert (i.e., the lecturer) who makes them. Allowing the test cases to change based on the performance of the submitted assignment allows the assignment to be assessed in greater depth.

Apart from the process of generating test input data, another major problem with automated assessment tools is that the feedback given to students is often regarded by students as being misleading, ambiguous, or vague [85]. This vagueness can be attributed to the automated assessment strategy not knowing exactly where the fault is originating from, or which test input data is causing the error. The feedback provided to the student is often too general to be useful.

This dissertation introduces a new automated assessment strategy, namely **G**enetic **A**lgorithm with **T**emporal **L**ogic for **A**utomated **M**arking (GATLAM), that incorporates an evolutionary algorithm, specifically a genetic algorithm (GA), to evolve the test input data in such a way that errors in the student's code become exposed. GAs have been used in software testing [97], especially combined with code coverage metrics, in order to measure the quality of the test input data. Code coverage metrics are used to determine how many of the possible execution paths in a system were executed while testing the system [50]. A problem with the code coverage metric is that the metric can mislead the GA into creating input data that produces longer execution paths rather than shorter execution paths, which may contain the same errors. In other words, the metric encourages the GA to produce longer paths. The code coverage metric causes the GA to generate non-error finding test input data.

To combat this, this study proposes an alternative way to evaluate the "quality" of generated test cases. This alternative strategy relies on a formalism to describe the behaviour of the student's code with different test input data, which is accomplished by linear temporal logic (LTL) properties, describing the desired behaviour of a system over time [23]. A violation of the desired behaviour will be seen as a violation of correctness.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

LTL formulae have been used as part of fitness functions for other use cases, such as using LTL as part of the reward function for reinforcement learning [74], and using LTL as part of a genetic programming fitness function [58, 86]. To the best of the author's knowledge[2], this is the first integration of LTL formulae into the GA-based automated assessment algorithm.

A key automated assessment benefit is timely feedback. Therefore, GATLAM utilises a memory component to speed up the evolution of the test input data. The memory component aims to place preference on portions of test input data that regularly cause the student's assignment to be incorrect. The goal is to evolve the test input data such that these commonly occurring portions of test input data are used more frequently, and as such increase the possibility of finding errors in the student's coding assignment. Using the solution that the GA converged on, the test input data can be returned to a student such that they can see a use case for which their code is failing.

To evaluate GATLAM, two types of experiments were conducted. The first experiment focuses on the influence of different hyperparameter values and types on the performance of GATLAM. The second experiment compared GATLAM to a code coverage-based approach, as well as a random number generator baseline . The experiment considered a controlled and uncontrolled set of programs. The controlled set of programs contains artificially injected errors, while the uncontrolled set of programs is volunteer student submissions for a coding practical completed in 2022.

Motivation, objectives and contributions of adding GATLAM to an already large pool of automated assessment strategies are discussed in Section 1.1, Section 1.2, and Section 1.3 respectively.

1.1 Motivation

Currently, in universities around the world, a large number of automated assessment strategies are employed [96]. Why is there a need for another automated assessment strategy, specifically one that involves the execution of the student's code, if there already exists so many? The motivation for this study is to solve two closely connected problems

[2]This fact is determined on 31^{st} of November 2024.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

in automated assessment. The first problem lies in the fact that timely feedback given to students can be misleading or vague, which leads to students losing motivation and becoming frustrated. This feedback is often general, such that the exact test input data is not revealed to students. Revealing the exact test cases to students will enable them to reverse engineer the solution and hard-code the assignment to only work for the specific test cases. The second problem is static test input data, which cannot be made available to students before the deadline, and forms a part of generation of the memorandum solution, which is usually the expected output of the assignment given the test cases. The automated assessment strategy proposed in this dissertation attempts to address the above-mentioned problems. Solving these two problems will aid the student's understanding of where and why their submission failed, and give the student an example of test input data that was used to assess their submission without compromising or revealing the memorandum.

1.2 Objective and Research Questions

The main objective of this study is to investigate whether a GA can be used to enhance automated marking. To address this objective, the following research questions and sub-questions are formulated:

1. Can a GA be utilised as part of an automated assessment strategy?
   (a) Does a GA have the capability of converging on a set of values that, when used as test input data, will cause errors in code samples?
   (b) How do the hyperparameter values affect the convergence of the GA?
   (c) Can an optimised set of hyperparameters yield the same performance across multiple runs for different code samples?
2. Can a GA be combined with LTL formulae as part of its fitness function in order to produce test input data which can expose errors in a student's submitted code assignment?

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

   (a) Which LTL formulae can be used to describe the desired behaviour of a student's assignment?
   (b) How can the LTL formulae be incorporated into the fitness function?
   (c) How does the weight associated with the LTL component of the fitness function affect the performance of the GA?

3. Can some type of memory component be incorporated into the fitness function of the GA?
   (a) How can such a memory component be defined?
   (b) How does the weight of the memory component affect the performance of the GA?

4. How does the newly proposed automated assessment strategy compare to other automated assessment strategies (or automated software test generation methods) used in the real world, such as random number generators and code coverage-based GAs, in the following case studies:
   (a) Theoretically
   (b) Empirically, using hand-crafted error pools, with artificially injected errors.
   (c) Empirically, using a real-world assignment with student-submitted code.

1.3 Contributions

This dissertation makes the following novel contributions:

1. Utilising a GA as part of an automated assessment strategy.
2. Utilising LTL formulae as part of a fitness function for a GA.
3. Incorporating a memory component as part of the fitness function for a GA.
4. Introduced a novel approach to generate constructive feedback personalised to each student's submission by providing user-specific test case examples of where the assignment fails.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The combination of all of the above contributions comprises GATLAM.

1.4 Dissertation Outline

This section discusses the chapters that make up this dissertation, with Figure 1.1 summarising the flow of the chapters. The black solid arrows in Figure 1.1 indicate the order in which chapters are presented. The dashed black arrows represent "dependencies" for each chapter, and the dashed red arrows show alignment between the chapters and the appendices. The remaining chapters of the dissertation are briefly discussed below.

- **Chapter 2** discusses the main principles and components of a GA. This chapter also lists and describes the hyperparameters of a GA.
- **Chapter 3** covers different methodologies and techniques used for software testing. This includes methods from static to dynamic testing, and black-box to white-box techniques. This chapter also introduces and discusses LTL formulae.
- **Chapter 4** discusses automated assessment strategies. The advantages and disadvantages of automated assessment strategies are covered, and the common aspects that are used to compare them are discussed.
- **Chapter 5** compares different examples of real-world strategies that range from static automated assessment to dynamic automated assessment. Dynamic automated assessment strategies are then further broken down into two subcategories, namely common and distinct. All of the strategies are compared and discussed.
- **Chapter 6** introduces and discusses GATLAM. The hypothetical advantages and disadvantages of GATLAM are discussed.
- **Chapter 7** outlines all the experiments conducted. These experiments include hyperparameter optimisation, a mock assignment, and a real-world practical assignment. The aim of the experiments is to evaluate GATLAM's ability to converge on error-prone test input data.

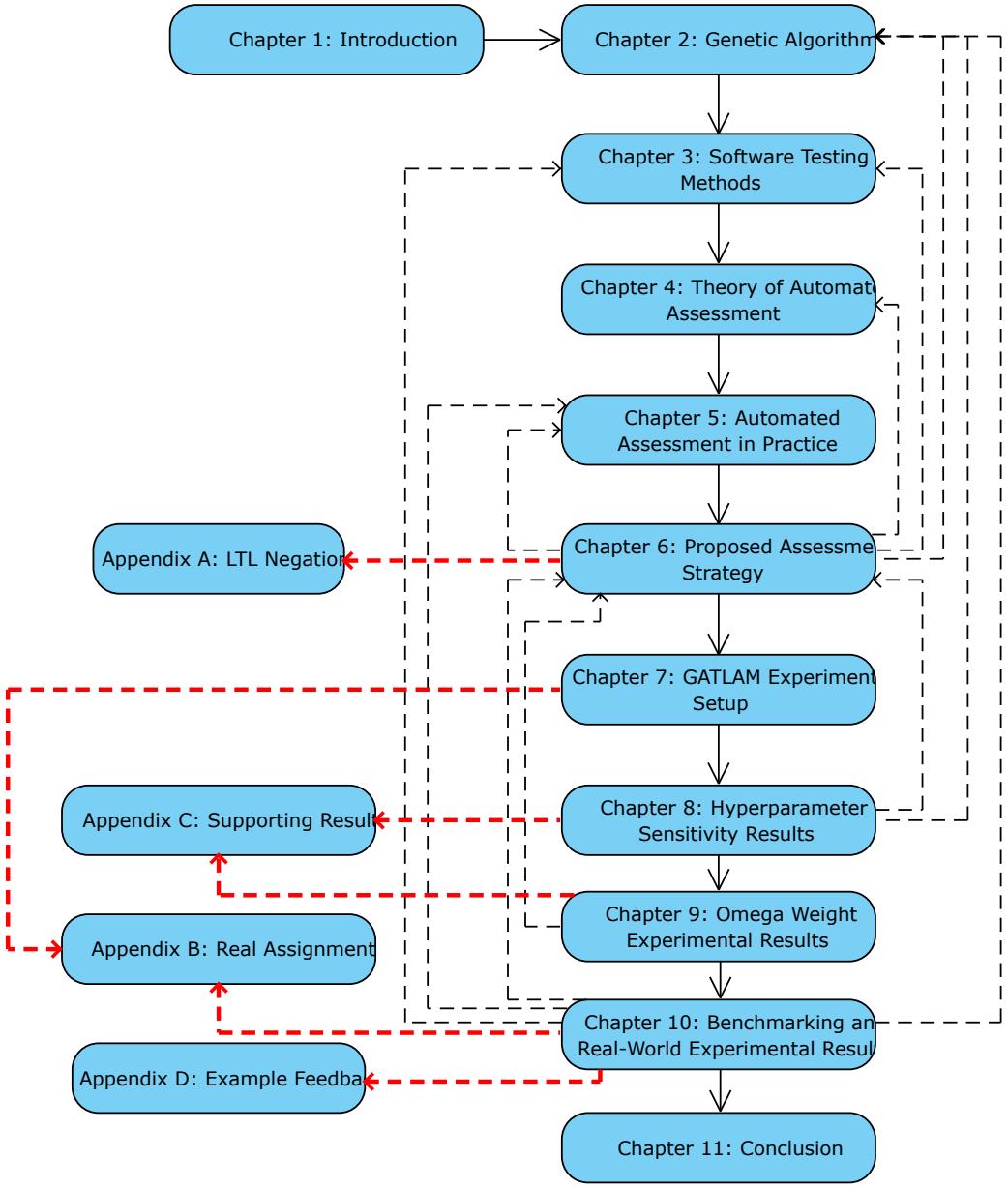


Figure 1.1: Diagram depicting the flow and dependencies of the chapters within the dissertation.

- **Chapter 8** discusses the results of the GA-specific hyperparameter tuning experiments that were performed. Different error pools were utilised to optimise the GA-specific hyperparameters on different types of errors.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Chapter 9** discusses the results of the fitness function-specific hyperparameter tuning experiments that were performed. The same error pools that were used in Chapter 8 were used.
- **Chapter 10** discusses the results of the comparative experiments in which GATLAM was compared to two alternative software test generation methods, namely, a random number generator and a code coverage-based GA. The three methods were compared in two types of experiments, which utilised a controlled and an uncontrolled set of code samples. The controlled experiments used the same error pools as in Chapter 8, and the uncontrolled experiments used a real-world programming practical completed by first-year students.
- **Chapter 11** concludes the dissertation and lists possible directions for future work.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Appendices of the dissertation are briefly discussed below:

- **Appendix A** covers the negations of the LTL formulae described in Chapter 6.
- **Appendix B** provides the specification given as a first-year coding practical assignment that is used as the base for the real-world experiments discussed in Chapter 7.
- **Appendix C** provides detailed result graphs and tables that are not explicitly shown in Chapters 8 and 9.
- **Appendix D** provides an example of the feedback that can be returned by GATLAM to the student, for the experiments discussed in Chapter 10.

Chapter 2

Genetic Algorithms

The genetic algorithm (GA) is one of the earliest population-based stochastic optimisation algorithms [80]. The idea behind the GA comes from Darwin's theory of evolution [64] and the theory of survival of the fittest applied to optimisation problems. A GA is built on the process of searching the predefined domain of input variables (i.e., the search space) for the fittest (best) candidate solution. In this chapter, the key components of a GA are discussed. Section 2.1 discusses the core terminology used to describe GAs. The main algorithm is discussed in Section 2.2, and Section 2.3 discusses how the GA population is generated. Section 2.4 discusses GA termination criteria. The different selection methods together with the fitness function are described in Section 2.5. Section 2.6 discusses the different operators that can be used to modify the GA population. Lastly, Section 2.7 summarises the chapter.

2.1 Preliminaries

The following terminology is used throughout this chapter. Each definition builds on the previous definitions. Definitions are adapted from Mathew [75]:

- **Candidate solution**: A possible solution to the optimisation problem.
- **Gene**: A single parameter value representing a certain property of the solution.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Chromosome**: A collection of genes that form a candidate solution for the optimisation problem. Chromosomes are usually encoded as binary strings.
- **Population**: A collection of chromosomes that form a set of candidate solutions for the optimisation problem.
- **Fitness**: A score that describes to what degree the candidate solution solves the optimisation problem.
- **Selection Methods**: These are methods that are used to determine which chromosomes are "fitter" and which chromosomes can be replaced. This concept is expanded upon in Section 2.5.
- **Genetic Operators**: These are operators that either change a single chromosome or a pair of chromosomes with the goal of creating fitter chromosomes. Genetic operators are formally introduced in Section 2.6.
- **Mating Pool**: These are the chromosomes that were selected, using a selection method, to be used to create new chromosomes using the genetic operators.
- **Offspring**: The resulting chromosome(s) that is/are formed when a genetic operator is applied to chromosomes in the mating pool.
- **Encoding**: The process where the chromosome is converted to a specialised format for evaluation, for example, the process of converting integer values to binary strings.
- **Decoding**: The process where the chromosome is converted back to the original values from the specialised format needed for evaluation, for example, the process of converting a binary string back to integer values.
- **Generation**: A generation is a single iteration of the evolutionary loop whereby chromosomes are evaluated, selected to form the mating pool, and the new offspring is added to the population.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

2.2 Main Algorithm

The GA consists of a set of chromosomes (where a chromosome is an encoding of a candidate solution) known as the population, genetic operators, selection methods, fitness function, and termination criteria [113]. The main algorithm is generic for all implementations of a GA. Figure 2.1 presents a flowchart diagram of a simplified general form of a GA algorithm [69]. In Figure 2.1, step one creates an initial population of a predefined size. This population then undergoes a systematic change over a set of generations until a termination criterion has been satisfied. This systematic change is defined by selecting the best chromosomes in the population using a selection method, which is discussed in Section 2.5, and having the best chromosomes "mate" and generate offspring. The mating and generation of offspring are discussed in greater detail in Section 2.6. The expectation is that the population will converge on an approximate optimal solution. Convergence is defined as a state in which each chromosome in the entire population starts to have the same fitness value or chromosome structure. Solutions can be seen as approximate, as the chromosome representation may not lead to an actual optimal solution but a solution that is relatively similar to the optimal solution.

2.3 Initial Population Generation

Initial population generation refers to the process where the initial set of chromosomes is created. According to Hill [52], the following two methods are used:

- **Random Generation**
  This consists of populating each gene in the chromosome with a random value. This method has the negative side effect of producing chromosomes that are invalid or contain faulty genes. Hill [52] suggests that a fitness function penalising chromosomes that are invalid or faulty will be able to overcome this side effect.

- **Generation Based on Problem Structure**
  Using information about the structure of the chromosome and the problem space, the generated values can be limited or restricted such that a higher rate of valid chromosomes is created.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

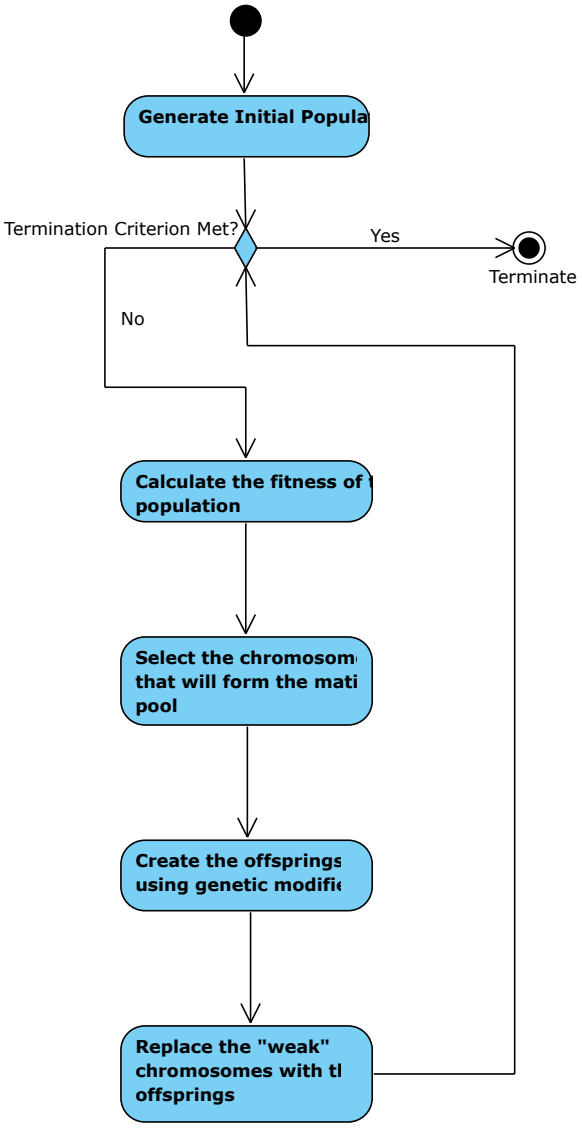


Figure 2.1: Simplified generic genetic algorithm.

2.4 Termination Criterion

A termination criterion is a set of criteria that, when met, will halt the execution of the algorithm. Safe et al. discussed [93] the following two main categories of termination criteria:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Number of Iterations**
  This termination criterion halts the algorithm when a predefined number of iterations has been reached. This predefined number is a hyperparameter that can be optimised through experimentation.

- **Problem-Specific Criteria**
  This termination criterion halts the algorithm when a predefined set of criteria is met, for example, a certain fitness value or chromosome structure has been created.

Usually, a combination of the two main categories of termination criteria is used to halt a GA, but they can be used separately as well.

2.5 Selection Method

The selection method is used to determine which chromosomes are added to the mating pool such that offspring can be created by the genetic operators. This section is laid out as follows: The fitness function is discussed first in Section 2.5.1. Then, Section 2.5.2 discusses different selection methods.

2.5.1 Fitness Function

The fitness function is used by the selection method to determine how "fit" or "good" a chromosome is at solving the problem at hand. The GA tries to maximise the fitness values of the population until the population converges. A fitness function is usually problem-specific, as the function needs to be capable of evaluating a specific format for the chromosome, where the format is evaluated in relation to the problem [113]. Fitness functions can also include domain-specific information to evaluate the chromosome, such as some type of software testing metric or LTL properties.

2.5.2 Different Selection Methods

A set of different possible selection methods as described by Jebari and Madiafi [64] are discussed below:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Roulette Wheel Selection**
  Roulette wheel selection uses the following function to assign a selection probability $p(i)$ to each chromosome i in the population:

$$p(i) = \frac{f(i)}{\sum_{j=1}^{n} f(j)}$$

  where $p(i)$ represents the selection probability assigned to each chromosome in the population, $f(i)$ represents the fitness of the i^{th} chromosome, and n is the size of the population. Based on the assigned selection probability, a chromosome has a chance to be selected to go into the mating pool.

- **Stochastic Universal Sampling**
  Stochastic universal sampling (SUS) selects chromosomes from the population using a single randomly chosen value. Before sampling, the population is typically sorted based on fitness. The random value determines the starting point, and chromosomes are then selected at evenly spaced intervals across the cumulative fitness distribution [14].

- **Exponential Rank Selection**
  Exponential rank selection assigns a rank to each chromosome in the population. Each chromosome is ranked based on its fitness, and the selection probability decays exponentially with the chromosome's rank. Exponential rank selection uses the following equation:

$$p(i) = 1.0 \times \exp\left(\frac{-rank(i)}{(n \times 2 \times (n-1))/(6 \times (n-1) + n)}\right)$$

  where $exp(x)$ refers to e^x.

- **Tournament Selection**
  Tournament selection is the process of randomly selecting k chromosomes, and finding the best-performing chromosome amongst the selected chromosomes. The best-performing chromosome is then added to the mating pool. This process is repeated until the entire mating pool has been filled.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Truncation Selection**
  Truncation selection consists of calculating the fitness value for each chromosome in the population and adding all the chromosomes with a fitness value equal to or greater than the set fitness threshold value to the mating pool.

2.6 Genetic Operators

Genetic operators are used to create new offspring from the mating pool in an attempt to "evolve" a better solution. This section will focus on three genetic operators: reproduction (Section 2.6.1), crossover (Section 2.6.2) and mutation (Section 2.6.3).

2.6.1 Reproduction

Reproduction consists of creating a new chromosome identical to the selected parent chromosome. In other words, an identical clone of the parent chromosome is created as the offspring [17].

2.6.2 Crossover

Crossover consists of swapping a subset of genes from one parent in the mating pool with another parent from the mating pool, such that two new offspring are created [17]. There are a few sub-types of crossover operators, namely [17]:

- **One-Point Crossover:**
  This crossover operator selects the same single random point in both chromosomes. The same side of the point is chosen from both chromosomes and swapped with each other. The process is illustrated in Figure 2.2.

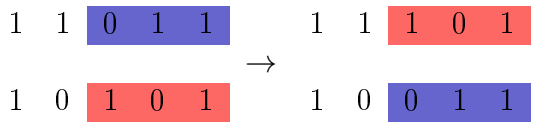


Figure 2.2: One point crossover example where the shaded area shows the portion of the chromosome which is swapped.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **N-Point Crossover:**
  This crossover operator is similar to the one-point crossover operator, except that the same N random points are selected in both chromosomes, and swapped to form two new chromosomes. Figure 2.3 illustrates a 2-point crossover, where the two crossover points were between the first and second gene, and the fourth and fifth gene.

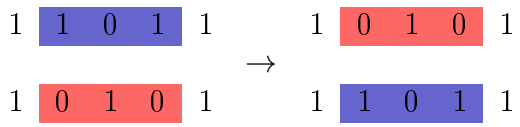


Figure 2.3: N point crossover, $N = 2$, where the shaded area shows the portion of the chromosome which is swapped.

- **Segmented Crossover:**
  This crossover operator is similar to the N-point crossover operator, with the difference being that N is not a fixed value but rather chosen at random for each generation.

- **Uniform Crossover:**
  Each gene in the chromosome has a probability sampled from a uniform distribution to undergo crossover. An example of uniform crossover is shown in Figure 2.4.

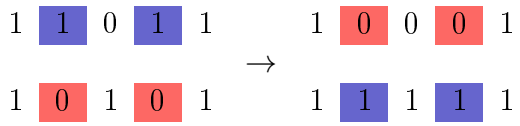


Figure 2.4: Uniform Crossover, where the shaded area shows the portion of the chromosome which is swapped.

Whether a crossover operator will be applied to a pair of chromosomes is determined by a crossover probability, which can be statically defined or dynamically determined.

2.6.3 Mutation

This genetic operator introduces randomness into the population. Randomness is added such that new information can be introduced into the population, which can assist the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

discovery of fitter chromosomes. The mutation probability is used to determine if a chromosome will undergo mutation or not. The mutation probability can either be statically defined or dynamically determined. Some mutation techniques that assume the chromosome is a binary string are listed below [17]:

- **Inversion of a Single Bit:**
  This technique consists of inverting a single randomly chosen bit in a bit string, illustrated in Figure 2.5.

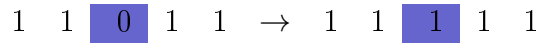


Figure 2.5: Single bit inversion, where the purple gene is flipped from 0 to 1.

- **Bit-Wise Inversion:**
  This technique is similar to the inversion of a single bit, with the difference being that the whole chromosome is changed by inverting every bit in the chromosome, as shown in Figure 2.6.

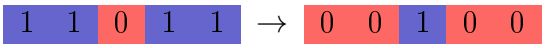


Figure 2.6: Bitwise inversion

- **Random Selection:**
  This technique consists of replacing a random section of the chromosome with a randomly created section. Figure 2.7 illustrates an example of a random selection mutation where the blue bits are replaced with a random new set of bits (the purple section).

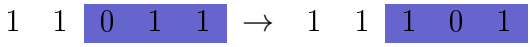


Figure 2.7: Random selection mutation.

2.7 Summary

Chapter 2 reviewed genetic algorithms . A GA is an evolutionary algorithm that, over a series of generations, evolves a set of candidate solutions until an approximate optimal

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

solution is found. The candidate solutions are possible solutions that attempt to solve the problem the GA is currently trying to solve. In order to determine how well the candidate solution solves the problem, a fitness function is used. The different GA components and operators used to evolve the candidate solutions were discussed. These components are experimented with in later chapters to determine the effect that each of the components has on the automated assessment strategy proposed in this dissertation. Chapter 3 introduces software testing techniques, and discusses how the software testing techniques are employed to create software tests.

Chapter 3

Software Testing Methods

From the early history of software, software testing has been a crucial part of the software development process. Formally, software testing is the process of determining if a software system is correctly implemented when compared to the design specification of the software system [95]. Similarly, to evaluate if a student's programming assignment is correct with respect to the provided specification, their programming assignment (i.e., code) needs to be tested, and even though software testing in industry has different goals compared to automated assessment, software testing methods can be used to ensure the assignment is correctly implemented. Software testing methods can be either manual (i.e., human-generated) or automated (i.e., computer-generated). Section 3.1 covers manual software testing methods. Section 3.2 looks at automated software testing strategies. Section 3.3 discusses how GAs have been applied to software testing. Section 3.4 discusses model checking. Lastly, this chapter concludes with a summary in Section 3.5.

3.1 Manual Software Testing Methods

The traditional way of creating software tests is for a software developer to manually determine all possible test cases, and create tests for each identified test case. Two main software testing methods adapted by the software development industry at large are white-box testing (Section 3.1.1) and black-box testing (Section 3.1.2).

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

3.1.1 White Box Testing

White box testing requires knowing the internal structure of the software that will be tested. The primary purpose of white box testing is to find internal logic-related errors, as every program flow within the system being tested can be individually tested for correctness [83]. The following white box methods are described in the 1999 testing of Light Combat Aircraft (LCA) procedure for the United States Department of Defence (USDOD) [89]:

Decision/Condition Coverage

Decision/Condition Coverage consists of evaluating each decision branch in the system being tested. An example of a decision coverage is illustrated in Algorithm 3.1. The tester will need to call `func` with an integer that is odd and with an integer that is even to ensure that the function correctly returns 1 and 0 with the correct inputs.

```
int func(int n){
     //Function returns 0 if the passed in number is even and 1 if
     //it is odd.
     if(n % 2 == 0)
         return 0;
     else
         return 1;
 }
```

Algorithm 3.1: Example used for Decision/Condition Coverage

Loop Testing

Loop testing consists of evaluating each loop to ensure that the correct behaviour is achieved by the loop, as well as that the correct termination criteria of the loop are achieved. The code segment shown in Algorithm 3.2 is used to illustrate loop testing. As can be seen in Algorithm 3.2, the loop will iterate until `curr` is equal to `n`. An example of where the code will not return the correct result is when `n < 0`, as `curr` will never become equal to `n`.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
int sumFunc(int n){
//Function will return the summation of all the integer values
//between 0 and n
    int sum = 0;
    int curr = 0;
    while(curr != n){
        sum += curr;
        curr++;
    }
    return sum;
}
```

Algorithm 3.2: Example used for Loop Testing

Mutation Analysis

This manual testing method involves creating mutants(i.e., small variations of the source code) and trying to create software tests that will reveal such mutants. When mutants are revealed, it is referred to as "killing" the mutant. Two ways to kill the mutants are to "strongly kill" them, by which the mutant causes the program to produce a different outcome, which then reveals which program contains the mutant. The other alternative is to "weakly kill" the mutant, where the mutant causes the program to go into an erroneous state, but still produce the correct output, revealing which program contains the mutant. Mutation analysis is an excellent way to test compilers and the effectiveness of the test suite used to test the code [2, 9].

Control Flow Graph Analysis

Control flow graphs (CFGs) are graphical representations of the source code. An example of a CFG is illustrated by Figure 3.1. Decision nodes represent different possible paths that can be taken within the graph, and sequential code statements are compressed into single nodes in the graph. Using CFG, a set of different testing criteria can be applied. Specifically, the coverage of simple and prime execution paths can be analysed. Simple paths are defined as paths in the CFG where nodes in the path may not repeat, apart from the first and last nodes. Simple paths produce excessive numbers of possible short execution paths, due to simple paths including every possible path in the CFG. Thus, simple paths include paths that are of length 0 up to and including the longest paths

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

in the CFG. Prime paths were introduced in an attempt to reduce the number of paths generated by simple paths. Prime paths are defined as simple paths that are not a subset of any other simple path. Using the paths produced by the prime paths, the tester can define an execution path through the CFG, and thus produce possible test cases that maximise the execution path coverage [6, 9, 46]. Table 3.1 contains all the simple and prime paths for the CFG illustrated in Figure 3.1. It can be seen that the number of simple paths is more than the number of prime paths; any of the simple paths forms as a subpath of one of the prime paths. The paths, {SABD}, {SACD}, and {SAD} are prime paths, as they are not a subpart of any other simple paths.

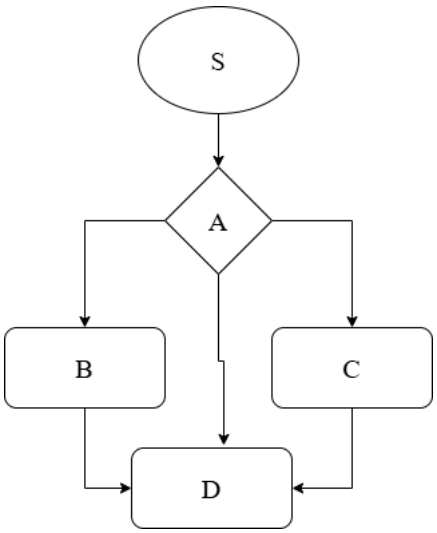


Figure 3.1: Example for CFG Analysis

Simple paths of length 0	Simple paths of length 1	Simple paths of length 2	Simple paths of length 3	Prime paths
{S}, {A}, {B}, {C}, {D}	{SA}, {AB}, {AC}, {BD}, {CD}, {AD}	{SAB}, {SAC}, {ABD}, {ACD} {SAD}	{SABD}, {SACD}	{SABD} {SACD} {SAD}

Table 3.1: Prime and Simple paths for Figure 3.1.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

3.1.2 Black Box Testing

Black box testing consists of testing the system with no prior knowledge of the internal workings of the system [68]. The main purpose of black box testing is to test the behaviour of the system. Violation of the behaviour of the system occurs when the system does not meet the requirements specifications for the system [83]. The following black-box methods are described in the USDOD's 1999 testing of LCAs [89] and are relevant to the research proposed in this dissertation:

Boundary Value Analysis

Boundary value analysis consists of dividing the possible inputs into partitions. These partitions consist of inputs and outputs from which clear boundaries can be identified and tested [87]. The input and output values are obtained by inspecting the specifications and requirements of the component being tested. In other words, boundary value analysis consists of testing boundary values to ensure that the system will be able to work correctly for edge cases.

```
boolean isNegativeNumber(String str){
    int val = Integer.parseInt(str);
    if(val < 0)
        return true;
    else
        return false;
}
```

Algorithm 3.3: Example for Boundary Value Analysis

Considering Algorithm 3.1, Table 3.2 lists the test cases (e.g., the input values) and the corresponding expected output. From Table 3.2, boundary values can be identified and tested.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Argument	Range	Input	Output
String	Alphanumeric including special characters	≥ 0	false
		< 0	true
		Alphabetic	false
		Alphanumeric	false

Table 3.2: Analysis of possible boundary values.

Equivalence Partitioning

This method of software testing consists of splitting the test cases into equivalent partitions [11]. A partition is seen as equivalent if there exists an equivalence relation between all of the test cases in the partition.In other words, the system being tested will have the same outcome for all the test cases in a partition. To illustrate how equivalence partitioning works the example from [90] is provided below. Assume the system to be tested is described by Algorithm 3.4. Assume the parameters `Xin` and `Yin`, where `Xin` $\in [-4; 4]$ and `Xin` $\in \mathbb{R}$, `Yin` $\in [0; 6]$ and `Yin` $\in \mathbb{R}$, and the input domain is divided into four equivalence partitions, namely EP1, EP2, EP3, and EP4. Assuming a one-to-one mapping for values is used, the equivalence partitions are formed as follows:

$$\begin{aligned}
\text{EP1:} \quad & -4 \leq \texttt{Xin} \leq -3 \quad \bigwedge \quad 4 < \texttt{Yin} \leq 6 \\
\text{EP2:} \quad & -4 \leq \texttt{Xin} \leq 4 \quad \bigwedge \quad 0 \leq \texttt{Yin} < 3 \\
\text{EP3:} \quad & -1 < \texttt{Xin} < 1 \quad \bigwedge \quad 3 \leq \texttt{Yin} \leq 4 \\
\text{EP4:} \quad & -2 \leq \texttt{Xin} \leq -1 \quad \bigwedge \quad 3 \leq \texttt{Yin} \leq 4
\end{aligned}$$

```
void system(float Xin, float Yin){
    if(Yin > 4) P();
    else if(Yin < 3) Q();
    else if(abs(Xin) < 1) S();
    else R();
}
```

Algorithm 3.4: Example for Equivalence Partitioning

All the values for `Xin` and `Yin` in EP1 will cause the `P` function to be invoked. EP2's values for `Xin` and `Yin` will cause `Q` to be invoked. `Xin` and `Yin`'s values in EP3 will cause `S` to be invoked, and lastly the `Xin` and `Yin` values for EP4 will cause `R` to be invoked.

A predefined number of random values is chosen from each partition and tested. If all the tests pass, it is assumed that all the values in the equivalence partitions will work.

Cause-Effect Graphing

Cause-effect graphing consists of creating a combinational logic network which can be translated to a decision table. The test cases can then be derived from the decision table. The combinational logic network consists of nodes which can be referred to as "causes and effects". The nodes are connected to each other with edges consisting of Boolean operators and constraints [98]. Srivastava et al. [98] provide an example of Cause-Effect graphs in their paper titled, "Cause Effect Graph to Decision Table Generation", which is illustrated in Figure 3.2.

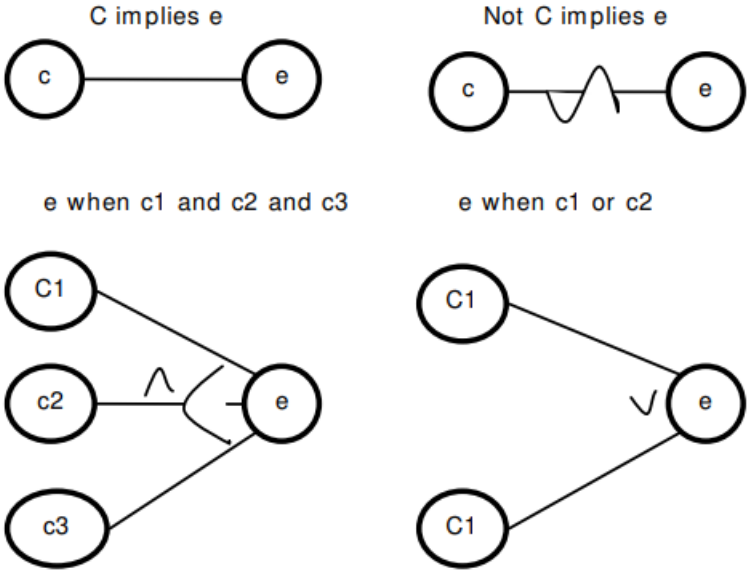


Figure 3.2: Cause-Effect Graph example from Srivastava et al. [98]

Error Guessing

Error guessing is the process where the software developer uses their experience and past knowledge to guess possible test cases which will cause an error to occur [65]. This

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

method is relevant to the proposed software testing method, which will initially "guess" some input values until the algorithm has enough information to start converging on an optimal solution.

As can be seen, the above-mentioned techniques require the software developer that is responsible for testing the code to spend a lot of time and effort analysing the system being tested, and then handcrafting test cases. This approach wastes developer resources, and does not guarantee that all system failures will be discovered. Thus, a way to automate software test generation is needed.

3.2 Automated Test Generation Methods

Automated software test generation leverages the "good at tasks that repeat" property of computers. Tests are created automatically by software, instead of a software developer manually handcrafting software tests [62]. The sections below list some of the automated software test generation techniques. These techniques can be used to create either static, dynamic or a combination of both. These paradigms are defined later in the section. The core idea/strategy of all the methods that are discussed is to reduce the search space of possible input values such that input values that will cause the system to fail can be identified quickly. A popular method of determining the quality of the generated test cases is to use path coverage or code coverage. Path coverage, or code coverage, is a metric that determines how many of the execution paths in the software system were executed [77].

3.2.1 Model-Based Test Generation

Model-based test generation is the process where a formal model(which can be handcrafted or automatically generated) of the system being tested and a set of test generation directives are used to guide a tool/algorithm in generating possible tests for the model [49]. According to Hartman [49], some examples of tools that can automatically

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

generate models are: TVEC, JCONTACT, UCBT-SALT, and MULSAW[1]. The tests for the model are then converted or translated to code/input for the system to be tested. The model-based test generation method also has the ability to generate correct output for each test, based on the provided formal model of the system. The downside to model-based test generation is the model itself. If the model is too simplistic or does not fully represent the entire system, the software tests generated for the model may miss or not fully test all edge cases.

3.2.2 Property-Based Testing

Different types of model-based testing exist, for example, property testing. Property testing was introduced by Fink and Bishop [43], and uses a similar idea to GATLAM, in the sense that property-based testing uses properties, specified in TASPEC, to generate tests. These tests are then executed, evaluated, and, if needed, expanded until all of the properties have been satisfied. One of the main differences between property-based testing and GATLAM is that property-based testing uses properties specified in TASPEC, whereas GATLAM uses LTL properties. The other difference is that property-based testing uses static analysis to generate the test cases while GATLAM uses a GA to generate abstract test cases, and an interpreter (formally introduced in Chapter 6) to generate concrete test cases.

3.2.3 Requirement-Based Test Generation

Requirement-based test generation is one method that tries to solve the problem of handcrafting a model of the system that can be used in a model-based test generator. The problem is solved by having software create a system model, taking system requirements as input. The generated system model is then used to generate software tests [102], as described in Section 3.2.1. Similar to model-based test generation, the downside to requirement-based test generation is the system requirements, which dictate how the system model will be generated, as well as the process used to generate the model. If the system requirements are not correctly defined and do not fully encompass the

[1]This list is not exhaustive and Hartman [49] mentions more of these tools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

system's expected behaviour, the model that is then generated might not fully represent the system. Any software test cases that are automatically generated with the non-represented model might not fully cover all use and edge cases.

3.2.4 Static and Dynamic Automated Test Generation

Automated test generation can be static or dynamic. Both paradigms are discussed below.

Static Test Generation

Static test generation is performed by running a static analysis of the source code, without the code ever being executed. Static analysis performs an analysis of possible execution paths within the system being tested. Software tests are then created which will satisfy some testing criteria on the detected execution paths [115]. An example of a testing criterion is predicate coverage, which requires that for every predicate[2] inside of the source code, two software tests are created. The two software tests must follow an execution path where one results in the predicate being false and the other in the predicate being true [9].

Dynamic Test Generation

This approach to automated test generation consists of actually executing the code being tested, monitoring the behaviour of the software under test, and then generating tests based on this monitored behaviour [115]. The goal is typically to generate test cases that yield the most diverse range of behaviour in the system under test. Behaviour is measured using a range of different metrics or criteria. An example of such a metric is code coverage, which is the percentage of the system executed by the software test [50].

[2] A predicate is a combination of conditions that can be found in a conditional statement, like an `if` statement.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

3.2.5 Bounded-Exhaustive Test Generation

Bounded-exhaustive test generation is the process of testing all possible input values within a bounded set, to determine if any of the input values within this range cause the system to fail [100]. This technique of bounding the "search space" of possible input values is also used by the automated marker proposed in this dissertation.

3.2.6 Symbolic Execution-Based Testing

Symbolic execution-based software tests consist of trying to find all possible values that will execute the highest number of possible execution paths through the system being tested. This approach differs from static testing, as the code under test is executed to determine which section of code was executed and which was not, while in static testing the code was not executed rather inspected. The idea is that the higher the number of execution paths traversed, the greater the probability that errors in the system will be found. Instead of actual input data, symbolic values are used. Symbolic values are a collection of symbolic expressions, which can then be used in execution paths to determine the values of the program variables. Thus, the resulting outputs are symbolic expressions for which software tests can then be automatically determined. Each expression represents a possible execution path in the system [25].

A problem with symbolic execution-based testing is that the amount of test cases needed to be formulated to cover every possible execution path becomes infeasible as there can be a possible infinite number of execution paths in a software system. Determining the test cases for each symbolic expression can become complex depending on the expression, which reduces the scalability of symbolic execution-based testing.

3.2.7 Random-Based Test Generation

Random-based software test generation (also known as fuzzing), as the name suggests, consists of randomly generating input data for the system being tested [26]. Random-based generation produces random input values that can help uncover edge cases, but are not necessarily representative of how the system will be used or how input for the system will be provided.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Random-based test generation is easy to implement and does not rely on any metric to help improve the generated software tests, but that means that random-based test generation can also produce mediocre or inadequate test input data, which never exposes faults in the system.

3.3 Genetic Algorithms for Software Testing

GAs have been applied to software testing in a variety of ways. In this section, only approaches that involved a GA are considered. Srivastava and Kim [97] used a GA to generate test input data for a control flow graph (CFG) to find the most critical path. Srivastava and Kim's approach first assigns different weights to each edge in the CFG depending on the critical level of the edge. Srivastava and Kim do suggest an approach to calculate the critical level of an edge, but it is dependant on the input of the test engineer. Predicate, loop, and branch nodes are given a higher critical level than other CFG nodes, for example. Using the generated test input data for the CFG, the fitness function sums all of the weights of traversed edges to calculate the fitness value. This approach relies on a CFG of the system being created and then weights being assigned to different edges based on the critical level of the nodes. Based on the granularity of the CFG, nodes containing errors might be omitted and as such could negatively impact the assignment of weights, which impacts the fitness function and ultimately the performance of the GA. The method proposed in this dissertation rather relies on a set of formalised system properties that, when violated, contribute positively to the fitness function. The other difference between Srivastava and Kim's approach and the method proposed in this dissertation is that the method proposed in this dissertation does not rely on a CFG representation of the system.

Esnaashari and Damia [40] use a GA with a reinforcement learning (RL) model as a memetic algorithm to encode a set of test cases that will maximise the number of traversed execution paths in the system under test. Memetic algorithms form part of the evolutionary algorithm family with the distinction that another algorithm is used to perform a local search instead of a global search [38]. Thus, a the GA performed a global search while the RL algorithm performed a local search. RL is a machine

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

learning paradigm that does not form part of the evolutionary algorithm family, and is based on the idea of finding a set of steps that will maximise the reward awarded to the model [101].

Berndt et al. [15] propose the use of *fossil records* to aid the GA with searching through the possible input values for possible software test cases. Fossil records are defined as the output from previous tests that are stored in a data warehouse. The fitness function uses the fossil records to calculate three factors to describe a chromosome, namely, novelty, proximity, and severity. Berndt et al. define novelty as how unique the chromosome is, proximity as the distance between the current chromosome and the fossil records that also resulted in system failures, and severity as the seriousness or danger level of the failure that the current chromosome caused. fossil records are similar to the memory component that the proposed method uses. The difference is that the proposed method calculates the average number of times a specific value for a specific gene was present when the system failed for any type of failure, which forms a component of the overall fitness function.

Michael et al. [77] expanded on previous work where the goal of generating dynamic software test cases was represented as minimising a function. Michael et al. proposed the use of GA as an alternative to standard function minimisation algorithms such as gradient descent. This led to the development of the Genetic Algorithm Data Generation Tool (GADGET). GADGET is an automated software test generation tool for C/C++ programs. GADGET tries to find test input data that results in an execution path that will satisfy the condition-decision coverage (C/DC) test criterion. A condition is a logical condition in a branching statement, and a decision is a collection of conditions that determine the execution path of the program. C/DC is defined by three criteria: every exit point(termination point in a program) needs to be executed at least once, every possible execution path created by a condition is executed at least once, and every possible decision is executed at least once. GADGET thus tries to find test input data for the system being tested such that each condition and decision is executed at least once.

Goschen et al. [47] utilise a genetic programming (GP) algorithm to automatically generate programs that can be used to generate input data for the system being tested.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The main difference between GP and GA is that GP uses a tree-like structure to represent the chromosomes, while GA uses a linear structure to represent the chromosomes [71]. Goschen et al. utilised path coverage to calculate the fitness of the evolved input generation program. Thehe approach was able to evolve programs that generated input that could traverse the majority of the possible execution paths in the system investigated by the Goshcen et al.

From the discussion in this section, it can be seen that there is a high level of reliance on path coverage metrics to determine the quality of generated software tests. However, path coverage metrics are not perfect metrics, and could lead to situations where the metric will cause the GA to prefer longer execution paths over shorter execution paths. Satisfying coverage metrics can also become infeasible for large, complex systems.

Hemmati [50] investigated the effectiveness of different code coverage metrics, and discovered that the tools that calculate the coverage metrics can be unreliable or error-prone in complex systems. The use of a single code coverage metric is often not good enough, and a combination of a few metrics is needed to successfully detect a large number of errors [50]. As such, an alternative method to describe the quality of software tests is needed. This dissertation proposes the use of linear temporal properties expressed as linear temporal logic formulae, which can describe the behaviour of the system being tested given , input data.

3.4 Temporal Logic Model Checking

This section briefly covers temporal logic model checking. Section 3.4.1 describes temporal logic formulae Section 3.4.2 discusses examples of variations of Temporal Logic Formulae, and Section 3.4.3 gives a brief overview of the model checking approach. Lastly, Section 3.4.4 discusses the use of temporal logic formulae as part of the fitness function of a GA.

3.4.1 Temporal Logic Formulae

Standard logic is the fundamental core of many branches of mathematics and computational theories, as it describes the deductive steps and reasoning used in both of the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

aforementioned fields [20]. Standard logic formulae do have one significant drawback: The neglect of time [91]. This means that standard logic cannot describe any system where a property might hold for some time frame, but will not hold for another time frame. For example, standard logic cannot model a traffic light where the light will always be green until the light turns yellow. Time is an essential part of everyday life and is used to chronologically order the sequence of events. Temporal logic formulae account for time in the logic expression. Temporal logic formulae can be used to describe the behaviour of a system during execution.

3.4.2 Variations of Temporal Logic Formulae

As there are many different types of temporal logics, only a few are considered for this dissertation, and are discussed below.

- *Linear Temporal Logic Formulae*
  Linear temporal logic (LTL) formulae can be evaluated on infinite linear sequences of states. LTL formulae are used in model checking to evaluate if a given model satisfies a requirement specified in LTL. One of the tools for validating and verifying LTL formulae is the model-checking tool Spin [55]. An example of requirements that can be specified in LTL is safety (nothing bad will ever happen) and liveness (progress will be made infinitely often).

- *Real-Time Temporal Logic Formulae*
  Real-time temporal logic formulae are used to specify requirements for real-time systems, such that it can be determined if the system is satisfactory and reliable in a real-time setting. Real-time temporal logic uses *first-order explicit clock logic*. Real-time temporal logic formulae are bounded by a finite amount of time in the past and an infinite amount of time in the future. Due to the explicit clock logic nature of real-time temporal logic, real-time temporal logic formulae may use a time variable t. One of the tools used to validate and verify real-time temporal logic formulae is UPPAAL [70]. An example of a property that these temporal logic formulae can represent is that the system successfully terminates within a certain amount of time.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- *Propositional Quality Extension for Linear Temporal Logic Formulae*
  Propositional quality extension for LTL tries to solve the problem of quantifying how well the requirement specified by the temporal logic formula was satisfied. In the work of Almagor et al [7], the concept of propositional quality extension for LTL formulae was introduced. These types of formulae are useful as they provide a method of obtaining a real value between 0 and 1, which represents a quantitative evaluation of how well the formula was satisfied by a given system.

3.4.3 Model Checking Approach

Automated software verification is a form of "exhaustive" testing which can be performed by LTL model checking the system under consideration. The LTL model checking approach is based on creating two Büchi automata[3] and checking whether the intersection of the languages of the two automata is empty. The Büchi automata are formed from a model (the Kripke structure) that represents the system and from correctness requirements, which are expressed in temporal logic [28]. A Kripke structure is a graph-based model that is used to model the behaviour of a system [21]. The nodes in the graph represent the states that the system can be in, and the edges that connect the nodes describe how the system changes between states [72]. The model checking algorithm will either prove that no possible execution path in the Kripke structure will violate the correctness requirements, or it will generate an example execution path that violates the correctness requirements [27]. This idea of providing an example of a failing execution path is used by GATLAM to generate feedback for the student. There exist many specialisations and alternative techniques for model checking. These deviations include bounded model checking [16], and multi-agent model checking [114].

3.4.4 Temporal Logic Formulae in Artificial Intelligence

Temporal logic formulae have been used as part of fitness functions for artificial intelligence models in the past, for example, to quantify reward in reinforcement learning [74].

[3] Büchi automata are finite state machines that can accept words of infinite length [23].

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Temporal logic formulae were also used as part of a grammar-based genetic programming model for mining road traffic rules [86].

Indri et al. [58] proposed using a genetic programming technique whereby temporal logic formulae were learned by a variation of the GP algorithm. The temporal logic formulae were used for anomaly detection in cyber-physical systems [58]. As opposed to the work of Indri et al. [58], this study uses predefined temporal logic formulae.

To the best of the author's knowledge, this dissertation is the first application of LTL as part of a fitness function in the context of automated assessment and automated software test generation.

3.5 Summary

This chapter covered both manual and automated testing methods with a focus on automated testing.The two main categories of automated testing were introduced, namely static and dynamic testing, and a series of examples of each were discussed. Some of the examples that were discussed, like model-based generation, can be categorized as being either static or dynamic. The use of GA to dynamically create software tests was discussed. The chapter also introduced temporal logic formulae to capture behavioural aspects of the system as part of the fitness functions. Chapter 4 will discuss the theory of automated assessment, which gives context for the use of the GA and temporal logic formulae combination to generate test input data and evaluate the student programming assignment submissions.

Chapter 4

Theory of Automated Assessment

Assessment is a core part of any student's educational career, and students rely on feedback from the assessment to learn from their mistakes [110]. Inadequate assessment feedback can be detrimental to student success, which is especially true for coding assessments. For this dissertation, only assessment strategies that are related to coding assignments are considered. Automated assessment strategies with respect to coding assignments can be defined as assessment tools where the assignment is evaluated using some sort of script that automatically tests the submitted code and determines the grade and/or feedback [39].

This chapter is laid out as follows: Section 4.1 discusses some key concepts that form the background of automated assessment strategies. Section 4.2 contains a small discussion on feedback and how it relates to automated assessment strategies. Section 4.3 and Section 4.4 look at the benefits and drawbacks of automated assessment strategies, respectively. Section 4.5 discusses the considerations that need to be undertaken when making use of an automated strategy. Section 4.6 differentiates between the different types of automated assessment strategies. Section 4.7 looks at the aspects that are used to compare different automated assessment strategies, and Section 4.8 contains a summary of the chapter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.1 Background

Assessment with respect to education, according to the Cambridge Dictionary [33], is defined as follows: *"The process of testing, and making a judgement about, someone's knowledge, ability, skills, etc."*. Thus, assessment with respect to education is used to determine to what extent the student currently being evaluated has learnt the work they were taught throughout the course/module. These evaluations are usually linked with learning activities and learning objectives where the activities are used to determine if the student successfully completed the learning objective [112]. For the remainder of the dissertation, when referring to "assessment", the dissertation refers to the process of evaluating the knowledge or skill about a course's content, which a student obtained during the course.

Two predominant types of assessment exist, namely formative and summative. Formative assessments are defined as low-stakes assessments where the student's progress is measured, and are designed to aid the student. Summative assessments are defined as high-stakes assessments where the student is tested, and it is determined to what extent the student has learnt the content of the material being taught [36].

Automation is defined in the Cambridge Dictionary as follows: *"The use of machines and computers that can operate without needing human control"* [34]. Thus, automation is the process of getting a computer/machine to do something that would normally be manually completed by a human.

Using the above two definitions as inspiration, the term *automated assessment* can be defined as follows: *"The use of a computer or machine to assess a student's answers/solutions to determine to what degree the student understood the coursework"*. In 1960, Jack Hollingworth published the first work on using an automated assessment tool to evaluate programming assignments of students [53].

Software testing techniques are used during the automated assessment process to assess a programming assignment using a set of predefined criteria to determine to what degree the submitted assignment followed the specification [84]. Software testing, or just testing, can refer to many things, including a form of assessment. The main concept of testing includes some form of test. Again, according to the Cambridge Dictionary,

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

testing is defined as follows: *"The act or practice of giving tests to measure someone's knowledge or ability"* [35], which sounds similar to assessment as defined above. Another definition listed under the definition of testing in the Cambridge Dictionary reads as follows: *"The process of using or trying something to see if it works, is suitable, obeys the rules, etc."* [35]. Using this definition of testing, an informal definition of software testing can be derived: *"Software testing is the process of trying something with a software program to see if it works and if it obeys the specification"*. A formal definition of software testing was provided in Chapter 3. For the rest of this dissertation, when referring to the term "testing", the dissertation refers to the concept of software testing, as informally defined above, to evaluate if the student's practical programming assignment conforms to the specifications they were provided with. With testing comes the definition of correctness. A program is deemed to be correct when it fulfils the specification that defines the program [32].

This chapter covers the assessment strategies, and it should be noted that an automated assessment tool is the implementation of the assessment strategy. Usually, the tool shares the same name as the strategy, and as such, for the rest of the dissertation, the term "automated assessment tool" is seen as a realisation of the automated assessment strategy, and as such is seen as a subset of the term "automated assessment strategy". Although many of the strategies and the tools that implement these strategies are similar, there is enough of a difference between the strategies/tools that each tool can be seen as the realisation of the strategy.

4.2 Feedback

A crucial part of any assessment process is the feedback provided to the student, such that they can learn from their mistakes [10]. Feedback can have many different forms, ranging from information describing the final mark for an assignment to the support provided to students while they complete the assignment [41]. This dissertation groups feedback into two main categories, namely automated feedback and manually created feedback.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Automated feedback is defined as *"Feedback that is generated by a computer or machine based on a set of criteria"*, which was derived from the definitions presented in Section 4.1. Manual feedback refers to the feedback the instructor or lecturer handcrafts. Feedback associated with automated assessment can vary from a single word, such as "Correct" or "Incorrect", to a detailed description of the edge cases tested including the exact location where the error was found in the submitted assignment..

4.3 Benefits of Automated Assessment Strategies

Before discussing different automated assessment strategies, it should first be noted what the benefits are of automated assessment strategies over manual assessment strategies. Pieterse [85] lists the following benefits:

1. **Speed**
   Speed is the predominant advantage of an automated marker over a manual marker. Speed here refers to the time it takes to assess correctness. Automated assessment strategies determine if a program is correct at a quicker pace and more accurately than manual marking [53, 66]. This is due to the automation of the marking process, and reduced reliance on the speed limitations of the human brain.

2. **Availability**
   Automated assessment tools are machines. They do not have working hours or sick/leave days. This enables them to be available 24/7 to assess the students' code. Students have the ability to submit whenever they want before the deadline, and do not have to wait until working hours to have their code assessed by a human marker [39]. This is beneficial as it allows students to work according to their own pace and not be restricted by specific hours.

3. **Consistency/Objectivity**
   As stated above, an automated assessment tool is a machine. Assuming the implementation of the tool is deterministic, the tool will produce the same mark for the same program time after time. Thus, the student will not be penalised due to their ethnicity/gender/past experience with the marker/tool. The automated

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

assessment tool is thus objective and consistent, as each student is treated equally and fairly [39].

4. **Immediate Feedback**
   This benefit is a side product of the first two benefits mentioned above, namely speed and availability. Automated assessment has the ability to provide near-instant feedback to the student upon assignment submission. Thus, the student will know what they need to improve on before possibly resubmitting the assignment. With manual marking, the student may need to wait until all the feedback has been collected for all the students before they will receive feedback on their assignment [99].

5. **Multiple Attempts**
   An automated assessment tool usually allows multiple submissions, whereas manual marking usually only has a single attempt. The reason manual marking usually only allows for a single attempt is that the human resources needed to mark the assignments are too few to allow for multiple submissions. In contrast, an automated assessment tool has the resources to enable students to submit multiple times. This can aid student performance, as the students have the opportunity to improve on previous failures [99].

6. **Can Act as Both Formative and Summative Assessment**
   Automated assessment strategies are beneficial as they have the ability to accommodate both formative and summative assessments. Formative assessment can allow the student to have an unlimited number of submissions for an optional practical assignment. For summative assignments, the number of submissions is limited, and feedback is usually limited. These assignments usually count towards the final grade of the student [39].

Ideally, any new automated assessment strategy should try and contain all the benefits listed above.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.4 Drawbacks of Automated Assessment Strategies

Despite their obvious attractiveness, automated assessment strategies do have some drawbacks. The following list was composed from studentand instructor feedback listed in the literature [22, 30, 39, 66, 85, 94, 108]:

1. **Testing Data:**
   Testing data refers to the input data used to create the test cases that is used to evaluate the student's solution for correctness. Often, this test data cannot be revealed to students before the deadline, as students might hard-code the solution for the test data. This is a drawback as students struggle to reproduce the test data as they are solely reliant on the feedback provided to them after the solution is assessed.

2. **Lack of Adequate Feedback:**
   As the system is automated, it is not always possible to provide in-depth feedback as to what error caused the student to lose marks, which causes students to get easily frustrated and demotivated. These two effects can negatively influence the learning experience for the student, which is undesired. The ALOHA grading tool tries to overcome the problem of inadequate automated feedback. The actual marking is done manually [3], but based on the marks obtained by a student for each question, ALOHA will automatically generate the feedback. Another problem with solely automated feedback is that the feedback can be misleading. For example, erroneous output for a previous section can cause the incorrect feedback to be returned for another, possibly correct, section of output. This can cause the student to spend time and effort debugging a piece of code that could be logically perfectly correct, but the output may be flawed.

3. **Strictness:**
   Character-for-character automated assessment strategies are strategies that compare each individual character in the student output to the expected output to determine correctness. These automated assessment strategies do not have the capability to determine if the code is logically correct or if the output does not match

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

the expected output. Thus, for example, in a very strict assessment strategy, a single extra unwanted whitespace could cause the student to fail the assignment. This over-strictness can add to the student's frustration and demotivation. This over-strictness can be alleviated to some degree by removing all white space before comparing or using regular expressions[1] to perform the comparison.

4. **Configuration Time and Effort:**
   Automated assessment strategies require more effort and time to correctly configure as opposed to the setup required for manual marking. For bigger classes, this disadvantage is reduced as compared to smaller classes (e.g. with 20 students).

5. **Misuse by Students:**
   From personal experience, students tend to use the automated assessment tool as a black box that can validate their code for them without having the need to actually test their own code. As this drawback is directly related to the mindset of the student and not the automated assessment tool, it is omitted from the comparisons performed in Chapter 5.

Ideally, any new assessment strategy should try and minimise the disadvantages listed.

4.5 Automated Assessment Considerations

Due to the strictness of automated assessment strategies, some considerations need to be kept in mind by the instructor when opting to use an automated assessment strategy. The following considerations were identified by Pieterse [85]:

1. **Clarity of Specification:**
   To determine if a programming assignment is correct, a detailed specification is needed. If the specification contains any ambiguity or uncertainty, it will lead to students interpreting the purpose of sections of the assignment incorrectly, or misunderstanding the format in which the automated assessment expects output.

[1] Regular expressions are sequences of characters that define search patterns for text processing, commonly used for matching, extracting, and replacing strings.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

If the specification provided to students is not clean and concise, the grades may be negatively impacted, thus contributing to the student's frustration/demotivation as time may be wasted on unwanted, and possibly incorrect, functionality.

2. **What Software Testing Technique(s) are Used to Create the Test Cases:** Chapter 3 described many of different test generation techniques. These techniques can be used to generate different test cases that will be used to determine if the student followed the specification and implemented the functionality of the assignment correctly. An example is to use input space partitioning (Section 3.1.2) to determine the possible ordering of an array for a sorting algorithm to sort. Different test creation techniques result in different test cases, and some may overlap. If the wrong testing technique is used or not used correctly, edge cases may be missed, and the student's program may not be fully correct with respect to the provided specification. Testing techniques used should also ensure that a balance is found between edge cases and general use cases.

3. **Release of Testing Data/Cases:**
In many of the surveys performed [22, 30, 39, 66, 85, 94, 108], students felt that they would have liked to have access to the test cases that the automated assessment strategy employed to grade their project, such that they could determine exactly why it failed with those cases before resubmitting. Test cases or test input data can be defined as any data that is given to a program and used during program execution. Test input data is a specialisation of input data, as it is used to determine if the program behaves as described in the specification of the program. Test input data can have multiple forms, which include but are not limited to command line arguments, terminal input, text file, external server requests and main functions. With this test input data, a student's assignment can be tested. The tests can then be used as part of an assessment to assess the student's knowledge/skill in the topic of the assignment. Usually, in summative assignments, test cases cannot be given to the students before the deadline, as that would render the practical pointless, since the students will know exactly how their assignment will be tested, and will therefore only cater to those edge cases. Some of the assessment tools

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

release the test cases after the deadline had passed, while others do not, such that the assignment could be reused in the future. Test case generators, like GATLAM, can overcome this consideration.

4. **Feedback:**
   Just as the clarity of the specification needs to be considered, what is returned to the student as feedback should also be considered. This point is an extension of the previous point. If students do not receive correct feedback, or received vague feedback, they might be unsure as to what caused them to receive a bad mark. On the other hand, if students receive feedback that is too detailed, it might give the test case away, and the student will be able to reverse engineer the test case and hard-code the solution.

5. **Functionality:**
   Automated assessment tools should have the following functionality [37]:

   (a) Ability to pass the test input data to the student's assignment:
   This is needed such that the student's code can be tested without the student knowing what specific values the assignment will be tested with.

   (b) Ability to record marks:
   The assessment strategy will need to record how each student performed on the assignment to determine to what extent the student understood the content taught in the module. This functionality is outside of the scope of this dissertation and can be expanded on in future work.

   (c) Timing the execution of the assignment:
   Loops are a key part of programming, as loops have the ability to repeatedly execute a section of code. Loops need a stopping condition that will govern when the loop should halt execution and move on to the next section of code. If the student does not properly define the stopping condition, it may lead to the code running into an infinite loop[2]. Timing the execution of the assignment can be used to halt the execution of the assignment to avoid an

[2] An infinite loop is a loop that will never halt/terminate.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

infinite loop and provide feedback to students. It can also be used as a metric to determine the efficiency of the student's code. Other valuable resources like memory and disk usage can also be monitored, but this is outside the scope of the dissertation.

(d) Security:
An automated assessment tool should ensure that any unsafe or malicious code does not affect the assessment of the current student or any other students. The assignments should thus be run in isolated containers that can be destroyed if malicious code or program behaviour is detected. This functionality, however, is outside of the scope of this dissertation and can be expanded on in future work.

The above three functionalities should be considered when creating an automatically assessed assignment, as they will dictate how the assignment will be tested, how big the possible input values for algorithms could be, such that it reduces the risk of a false infinite loop, and how marks will be stored for the assignment. A false infinite loop is when a student's code executes for a long time, and as such, triggers the halting of execution of the assignment prematurely. This results in the student possibly losing marks when their code would terminate successfully, given more time.

4.6 Types of Automated Assessment

There are many different ways to classify or group automated assessments. Since automated assessment of code is related to automated software testing, software testing classification can be used, namely, static and dynamic [37, 84]. Section 4.6.1 discusses what is meant by a static automated assessment, Section 4.6.2 discusses dynamic automated assessment, and Section 4.6.3 discusses the advantages and disadvantages of both. Section 4.6.4 discusses what a hybrid automated assessment type entails.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.6.1 Static Automated Assessment

Static automated assessment follows the idea of static testing, where the submitted assignment is not executed, but rather the source code is automatically inspected and assessed. These techniques include creating a CFG , applying assessment criteria to the graph, and analysing the readability of the code and comments[3] in the code [84], which all process the source code. Some non-processing techniques include looking at the number of lines of code, the size of the files, or the size of the executable once compiled.

4.6.2 Dynamic Automated Assessment

Dynamic automated assessment is inspired by dynamic testing, where the submitted assignment is executed, and the results/behaviour of the executed program are assessed [4]. Assessment techniques include matching the expected output [85] and evaluating generated/returned objects[4] and their properties [66].

4.6.3 Advantages and Disadvantages of Static and Dynamic Assessment

Table 4.1 highlights the advantages and disadvantages of static (as discussed in Section 4.6.1) and dynamic (as discussed in Section 4.6.2) testing as listed in the work of Zougari et al. [116].

Something to note is that although static analysis can provide the exact location for errors, it becomes infeasible for large-scale assignments. In contrast, dynamic assessment strategies can evaluate bigger assignments, as only the output is used to determine correctness. The corresponding disadvantage is the output dependency of the dynamic assessment strategies: dynamic assessment strategies can only give feedback to students based on the expected output and deviation therefrom. Dynamic strategies also only need the executable, thus decoupling the source code from what is being assessed. This

[3] Comments are meta-data that can be added to source code to describe/document the code

[4] An object is an instance of a class in the Object-Oriented Programming paradigm

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

	Static Analysis	**Dynamic Analysis**
Advantages	Finds precise point of the error in the code. Grades code on all possible execution paths. Grading is not compilation dependent, which means code that does not fully compile can be assessed to some degree.	Implementation-wise easier to write tests. Evaluates based on the expected output, which broadens application possibilities, due to the wide type of assignment that can be assessed. Only needs an executable and not the actual source code.
Disadvantages	The high number of possible solutions limits structural analysis capabilities. Not feasible for extremely complex programs. Can produce test cases that are not possible during runtime.	Security precautions for malicious code. Compilation-dependent for grading. Feedback is dependent on the difference between student output and expected output. Increased difficulties to ensure correct algorithm and implementation is followed by the student.

Table 4.1: Advantages and disadvantages of static and dynamic automated assessment strategies.

allows for different standards of the same programming language[5] to be evaluated. With static strategies, the same version of the programming language has to be used such that the syntax rules remain the same.

4.6.4 Hybrid Automated Assessment

Hybrid automated assessments are automated assessment strategies that consist of both static and dynamic automated assessments [8]. These strategies usually attempt to assess the student on more criteria than static or dynamic can by themselves, for example, the automated assessment strategy can assess the student on their coding style[6] (static) and if the assignment followed the specification (dynamic) as part of a single strategy.

[5] For example, a student can submit their code in C++98 standard or in C++11 standard

[6] Coding style refers to the variable/function naming standard, indentation, program structure, and comment structure followed by the programmer [24]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.7 Aspects Used to Compare Automated Assessment

This section discusses the different aspects that are used to describe automated assessment. The aspects are grouped into common and extended aspects. Common aspects are commonly used to compare automated assessment strategies (Section 4.7.1), extended aspects are introduced in this dissertation (Section 4.7.2).

4.7.1 Common Aspects

In the literature, many aspects are used to compare and classify different automated assessment tools and strategies.

1. **Automated Assessment Generation:**
   Douce et al. [37] group automated assessment strategies by generations. In 2005, when the paper was written, only three generations were clearly defined, as described below:

   (a) Generation 1:
   Automated assessment strategies in Generation 1 are known as "Early Assessment Systems" and were developed during the 1960s. Initially, students submitted their code using punch cards. Later, they transitioned to submitting source code electronically. After the deadline, an automated assessment system evaluated the submissions. Generation 1 assessment strategies compared the student's output with a stored expected output. These strategies also incorporated the ability to provide detailed feedback to students.

   (b) Generation 2:
   Generation 1 strategies did not use any external tools to assess students on different or even extended criteria. Generation 2 was introduced and called the "Tool-based" generation. Douche et al. provide an example of this expansion of criteria in the form of a strategy that can assess how well the submission follows the specification, as well as the readability of the code. Assessing the readability of the code is not a standard function of an operating system, and as such, a specialised tool is needed. Douche et al. also mention that the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

tools used in generation 2 can be grouped into two categories, namely *tutor orientated*, where the tool is designed to aid a tutor or instructor in assisting the student or providing feedback to the student, and *student orientated*, where the tool is aimed at providing assistance directly to the student.

(c) Generation 3:
Generation 3 introduced web technologies to aid the automated assessment strategy. These technologies include the ability to submit the assignment via a website and have the feedback from the assignment displayed on a webpage. Thus, Douche et al. refer to generation 3 as the *Web-Orientated Systems*. Generation 3 also includes having programming questions/challenges displayed on a webpage, with the student required to answer/complete the question on the webpage.

The generations described above can be used to classify and compare different automated assessment strategies. Generations of automated assessment, however do not determine to what degree the student understood the work.

2. **Assessment Criteria:**
The University of Newcastle defines assessment criteria as: *"The criteria that will be used to determine to what extent the student understood the work"* [106]. For automated assessment with respect to programming, the assessment criteria are closely related to the type of automated assessment that will be used. If the automated assessment criteria, for example, include both readability and correctness, then the type of automated assessment will be a hybrid automated assessment strategy, and will evaluate the execution and the source code of the submitted assignment. Thus, different criteria define how the submitted assignment is tested, i.e., assessed.

3. **Feedback:**
This aspect looks at how feedback is given to the student. Will the feedback be generated automatically, semi-automatically, or manually after the deadline? Each of the types of feedback is discussed below:

- Automatically Generated (A):
  Here, feedback is automatically created based on the performance of the student. The strategy is able to automatically tell the student where they failed, without the instructor pre-loading feedback [96].
- Semi-automatically Generated (SA):
  Here, feedback is loaded into the strategy, and when needed, the appropriate feedback is combined and returned to the student. The automatic part comes in when the instructor's feedback for the relevant parts of the assignment is automatically combined to create a custom feedback report for the student [96].
- Manually Generated (M):
  Here, feedback is hand-crafted individually by an instructor or teaching assistant for each assignment. This requires each student's assignment to be manually evaluated, and time and effort are required to create feedback for all the students [96].

4. **Need for Additional Support:**
   This aspect looks at how much additional human resources are needed to support the automated assessment strategy. These resources include but are not limited to: lecturers, assistant lecturers, tutors and teaching assistants. Additional support can informally be defined as: *"Any help or time that a student needs from the instructors to successfully process the feedback from the automated assessment strategy, or assistance needed to complete the assignment"*. Additional support also includes the time and effort needed to create manually generated feedback for the students' assignments.

5. **How Correctness is Determined:**
   The automated marking strategy needs to determine if the student's project is correct or to what degree it is correct. This is needed so that the student's code can be assessed and graded [4].

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4.7.2 Extended Aspects

The additional aspects that are used in this dissertation to compare the assessment strategies are:

1. **Input Data Used to Test:**
   This aspect looks at what type of data can be given to the student's assignment for the purpose of assessment. Examples of the type of data are: text files, terminal import or command line input. This aspect is important as it can limit the type of assignments to which each assessment strategy can be applied.

2. **Generation of Test Input Data:**
   This aspect looks at how test input data is created for each assessment strategy. This is important as it aids in the understanding of what is required of the instructor to set up the automated assessment strategy.

3. **Language Restrictions:**
   The language restriction aspect looks at what programming languages the strategy can be used for. This aspect is important as it is an indication of how adaptable the assessment strategy is. Many different programming languages have different key features such as the lack of a garbage collector in C++[7], and Python being an interpreted language[8]. Thus, if the assessment strategy is able to assess projects written in multiple programming languages, the assessment strategy can be applied to multiple courses. If the assessment strategy can only be applied to one programming language, the assessment strategy will only be useful to modules using that specific programming language.

4. **Types of Topics That Can Be Assessed:**
   This aspect refers to the type of programming assignments to which each assess-

[7] A garbage collector in a programming language refers to the runtime system of the language automatically handling the allocating and freeing of memory of the program [48].

[8] An interpreted language is a programming language where each instruction in the program is compiled only when it is executed. In contrast to a compiled programming language (e.g., C++) the entire program is compiled before execution [73].

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

ment strategy can be applied. This aspect closely links to the previous aspect (language restrictions) in the sense that it can determine the usage of the assessment strategy. An assessment strategy that can only assess trivial introductory assignments can only be used in introductory programming courses. In comparison, an assessment strategy that is able to evaluate networking (for example, interaction with an application programming interface to ensure correct data is sent and received), algorithms, introductory assignments, and object-oriented programming assignments, for example, can be used in more modules.

5. **Correct Output Generation:**
   This aspect deals with the method that is used to generate the correct output that will be used to determine if the student's project is correct or not (using the definition of correctness from Section 4.1).

4.8 Summary

Chapter 4 discussed the theory of automated assessment from the need for automated assessment to the different criteria that are used to compare automated assessment strategies. This chapter identified the benefits, drawbacks and concerns that are associated with automated assessment. Chapter 5 discusses examples of existing automated assessment strategies.

Chapter 5

Automated Assessment in Practice

Using the concepts discussed in Chapter 4, this chapter discusses and compares examples of different types of automated assessment strategies. Focus is placed on dynamic strategies over static automated assessment strategies due to their relevance to the proposed automated assessment strategy, and as such, this chapter is laid out as follows. Due to the broad nature of different automated assessment strategies, this dissertation places a higher focus on automated assessment strategies that were developed for an educational setting, although a single example of an automated assessment strategy used in programming competitions is discussed. Section 5.1 discusses different examples of static automated assessment strategies. Section 5.2 compares different examples of dynamic automated assessment strategies. Lastly, Section 5.3 summarises the chapter.

5.1 Static Assessment Strategies

In this section, strategies that use static analysis of source code to evaluate and assess student's code are discussed.

- **AssignSim:**
  Naude et al. [81] suggested a semi-automated marking strategy to assess student's code. This strategy consists of converting the student code to graphs. The instructors then mark a representative set of student graphs. After these graphs have been graded, they are inputted to the AssignSim software together with the unmarked

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

graphs. The strategy then compares the graded graphs and the ungraded graphs, and grades according to the similarity. Thus, as explained in, if a student's graph is 72% similar to one of the graphs marked as correct, the student will receive a mark of 72%.

- **eGrader:**
  eGrader is introduced by AlShamsi and Elnagar [8] and makes use of both static and dynamic strategies to assess student's code. For this section, only the static strategy is discussed. Program dependence graph (PDGs) are a graphical representation of the system and are derived from, which was introduced in Chapter 3. Control dependence graphs (CDGs) are an extension of PDG where only the control mechanisms are modelled [42]. Method call dependencies graphs (MCDs) are graphs of methods in source code, where each MCD represents a single method in the source code [111]. eGrader uses two graph-based representations to assess student's code, namely: CDG and MCD. eGrader then applies different software metrics to the created MCD and CDG graphs. The results of the metrics are then encoded, together with the positions of control structures in both graphs, into a numerical representation. This numerical representation is referred to as an "Identification Pattern", and compared to the instructor-provided Identification Pattern. The grade is the difference between the instructor-provided identification pattern and the student's identification pattern.

- **CFG Similarity Strategy**[1]
  CFG similarity strategy is introduced by Zougari et al. [116] and, just as eGrader, also makes use of both static and dynamic strategies. Only the static assessment is discussed here. This strategy is similar to the above-mentioned two strategies in the sense that the similarity between the graphical representations of student code and instructor code is used as a grading criterion. This strategy differs from the eGrader strategy with regard to the graphical representation of the source code: eGrader uses CDG and MCD, whereas this strategy uses CFGs.

[1] Zougari et al. [116] do not give a name to their strategy, so the strategy is referred to as CFG similarity strategy.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **AutoLEP:**
  AutoLEP is introduced by Tiantian et al. [104] and also makes use of both static and dynamic assessment strategies. AutoLEP proposes taking into account not only the structural similarities to the model answer, but also the semantic similarities to the model answer. AutoLEP is confined to C-based languages. AutoLEP also performs normalisation to remove variations and account for coding style differences. AutoLEP transforms the student's code to a system dependency graph before normalisation. AutoLEP also compares the graphical similarities of the student's graphical representation to the model solution's graphical representation. AutoLEP provides immediate feedback to the student in the form of semantic analysis, syntactical/structural breakdown, and the results of the dynamic assessment.

- **Hairball:**
  Hairball is introduced by Tiantian et al. [18] and performs static analysis on the Scratch programming language. Scratch is a programming language that uses a visual interface which allows the user to drag and drop blocks representing instructions. Scratch was developed for people of all ages and backgrounds to quickly and easily introduce them to programming [92]. The designers of Hairball stated the following two main goals when they designed Hairball:

  – Hairball should automatically assess Scratch projects,

  – Hairball should be easily expandable as a Scratch plugin.

  Hairball uses a plugin that takes in Python[2] code that can then perform static analysis. In the work of Tiantian et al. [18], an example is given where the static analysis counts the number of a specific type of instruction blocks used in the Scratch file. The student is then graded based on the results of these Python static analyses, that were performed on the submitted Scratch program.

The above examples illustrate that syntactical assessments, although powerful, use the same broad technique to evaluate the student, just as dynamic analysis uses the output

[2]Python is an interpreted programming language

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

produced by the student's code as part of the assessment process. The examples also illustrate the importance of graphs and graphical representations of the source code.

5.2 Dynamic Automated Assessment Strategies

The main difference between dynamic and static automated assessment is that dynamic automated assessment executes the code to determine its correctness. This section is laid out as follows: Section 5.2.1 compares common dynamic automated assessment strategies, and Section 5.2.2, outlines the distinctive dynamic automated assessment strategies. Strategies are identified as distinctive if the strategy utilises a unique method of assessing the student's programming assignment.

5.2.1 Common Strategies

For this study, a dynamic automated assessment strategy is seen as a common strategy if its assessment of a student's code uses some form of matching of the resulting artefact with an expected artefact, whereby the tests/template are created by the instructor. An example of a common strategy is one that matches the output of a student's assignment with that of the expected output.

Existing Common Strategies

Pieterse [85] lists a set of popular automated testing strategies developed specifically for programming assignments. These strategies are labelled as common using the above definition. The existing strategies are listed below, and key factors relating to the proposed new automated assessment technique are discussed.

- **QuizPACK (Quizzes for Parameterised Assessment of C Knowledge)** QuizPACK is an automated assessment software, introduced by Brusilovsky [22] and developed by the University of Pittsburgh. QuizPACK is a software tool where a set of parameterised questions is given to a student, where the student either needs to predict what the outcome of the given program will be, or write a small program to solve a problem. The reason for the parameterised questions

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

is that random values can be substituted for each student to avoid cheating. To determine if the answer the student provided is correct for the fill-in question, the memo code is compiled with their answer and checked for correctness. The same process is followed for the code generation questions. QuizPACK can determine if the student's code is correct by running an instructor-provided solution with the input and comparing the student output to the instructor output to determine correctness. A significant drawback of QuizPACK is that at the time of publication, it is designed to handle only the C programming language. Thus, QuizPACK does not have the ability to evaluate students in other programming languages. The format of QuizPACK's questions is suited for small practical tasks.

- **BOSS Online Submission and Assessment System**
  BOSS is a Java automated assessment system that allows students to submit their code and be graded according to a set of categories [66]. The categories are:
  - **Correctness**
    Correctness is used to determine if the program does what it is supposed to do. BOSS has the capability to determine to what extent the student's assignment conforms to the provided specification. This can be done both in a text input/output (I/O) manner or with JUnit[3], for example.
  - **Style**
    Style refers to the student's coding style, such as the comments and variable names they use in the program. Assessing coding style has the benefit of forcing students to provide high-quality comments and structures within their code, such that their coding standard is improved. This assessment aspect is static in nature and does not need the code to be executed.
  - **Authenticity**
    Authenticity is a common problem in take-home practical assignments, as students tend to plagiarise from each other [85]. This is why BOSS incorporates a plagiarism detection system as part of its assessment categories.

[3]JUnit is a testing framework for Java.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Authenticity assessment is beyond the scope of the dissertation and will not be discussed in greater detail.

BOSS can use the output from the assignment being assessed to determine correctness. The following grading criteria are used by BOSS:

- **Correctness of Code**
  Determines if the student correctly implemented the specification.
- **Code Efficiency**
  Ensures that solutions do not execute longer than expected and, as such, waste computer resources.
- **Use of External Libraries**
  Ensures that students do not use libraries when they may not.

According to the survey done on student satisfaction with BOSS, most students were pleased with the system, although some did feel that the system was very strict [66].

- **EasyAccept:**
  EasyAccept is a tool that uses acceptance tests to test software, particularly student assignments [94]. Acceptance tests are functional tests that test if the system as a whole is behaving as expected [78]. EasyAccept takes acceptance tests (created using user stories[4]) that contain business rules and output to determine if the provided system conforms to the specification. These tests are also run as automated regression tests. Regression tests refer to old test cases for a previous version of the system run against the new system to ensure that old errors in the system are not reintroduced during upgrades. Due to this re-running of tests, regression testing takes more time to determine if the code passes all tests [102].

- **Peach:**
  Peach is an automated assessment strategy that is mainly used for programming competitions/challenges and not programming courses. Peach groups assessment

[4] A user story is a way to formalise software requirements [56]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

tasks into "Task Packages". Each package contains the assessment and testing framework that is needed to determine if the student passed the challenge or not. These packages can then be swapped out to other programming courses or programming competitions as required. The following modules are of interest to the dissertation:

- **Input generator**
  The input generator is developed by the instructor and is used to generate the input for the challenge, such that the student's code can be evaluated.
- **Input validator**
  The input validator ensures that the input is valid before being presented to the student's code.
- **Expected output generator**
  The expected output generator determines what the correct output should be, such that the student's code can be judged on correctness.
- **Output checker**
  The output checker determines, based on the expected output, if the student's code is correct or not.

Although the area of automated assessment strategies used in programming competitions was not extensively investigated in this dissertation, a comparison between GATLAM and examples of these strategies can be performed in future.

- **Kattis:**
  Kattis is designed to provide both formative and summative assessment. Kattis incorporats an instructor demo component for each assignment in conjunction with the automated assessment tool. The automated assessment tool allows students to submit their assignment to the system several times to ensure that the assignment is correct according to the specification. Then, during the demo part, which is conducted in a style which is based on the selected teaching style, the student does not need to worry whether their assignment is correct or not. They can just explain

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

how they implemented it and answer questions about the implementation. Kattis is able to mark the following assignments:

- Algorithmic assignments, where the student needs to implement some algorithm.
- Cryptographic assignments.
- Specification-based assignments, where students are given a specification, with a problem that needs to be solved using either a specific technique or the students' own creativity.
- Pseudo-code based assignments, where students are given pseudo-code to implement. In other words, the student needs to do the translation from pseudo-code to a specific programming language.

Kattis uses efficiency and correctness assessment criteria to evaluate the student's code, and unit tests to test function implementations at an atomic level [39].

- **FitchFork:**
  FitchFork is an automated assessment tool developed by the Computer Science department at the University of Pretoria [85], which has been actively used and continuously improved since its inception. FitchFork has the ability to assess practical assignments in a variety of different languages. FitchFork has the following components:
  - **Results.json**
    This is a JSON file that contains the standard output (which contains all the output of the student's program), the standard error (which contains information about any compilation or runtime errors or warnings), and the return code with which the program exited. If a return code of 0 is received, it means the program successfully halted in a controlled manner.
  - **Overwrites**
    The overwrites are all the files required by a task to assess the student. These include files the students are not allowed to modify, a main file that will

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

be used to test the student assignment, and a makefile that will be used to compile the code with all the required compilation flags. If the assignment has multiple tasks, each task will have its own overwrites archive.

– **Marking script**
The marking script is a Python 3 script that receives the `results.json` file . Here, the instructor has the opportunity to define how the student will be graded based on the output. The marking script is also responsible for generating the feedback provided to the student. The traditional method of checking for correctness employed by many instructors is to have the marking script check line by line if the student's output matches the expected output hardcoded into the marking script, and assign a feedback string and mark to each line. If the line does not match the expected output, then the feedback string is added to the overall feedback, but if the line does match, "Correct" is added instead. If the line matches, the mark assigned to that line is awarded to the student. The marking script can be configured to use regular expressions instead of a verbatim line by line matching. The marking script also has the ability to inspect the student's code. This is usually done to ensure that no illegal libraries were used or that the student followed the prescribed restrictions.

FitchFork has been used successfully with classes of well over 600 students in first-year and second-year modules [85]. FitchFork can be used in both introductory programming courses as well as advanced programming courses, such as data structures and algorithms. FitchFork has been used to evaluate concurrent programs and assembly language programs, since it is not language-specific.

Comparison of Existing Common Strategies

The existing dynamic assessment strategies discussed here have some similarities as well as some unique features. Thus, two tables are used to compare the strategies. Table 5.1 uses the common aspects and Table 5.2 uses the expanded aspects both of which were discussed in Section 4.7.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

As Table 5.1 and 5.2 show, most of the strategies discussed rely on the instructor to create the test cases, with only one (QuizPACK) using a random value generator. The feedback generation (FDBK) is split between Automatic (A) and Manual (M), with only FitchFork using a semi-automatic (SA) approach. The strategies are also split evenly between having programming language restrictions and being programming language independent. Apart from QuizPACK, all the other strategies use input files that are either directly used for testing or describe how the test should be executed. It can also be seen that most of the assessment strategies compared in the table have the ability to at least assess and test introductory-level coding assignments. The term "de-sync" refers to a misalignment that occurs between the student output and the memorandum output. This causes all remaining output to be seen as incorrect when, in fact, it could be correct. Desync could occur if the student has an unexpected additional character or newline in their output.

Benefits and Drawbacks

Using the benefits listed in Section 4.3, Table 5.3 provides a comparison between the strategies. Table 5.3 also compares the drawbacks listed in Section 4.4 for each of the strategies. Note that "✓" in the table represents that the strategy has this benefit/drawback, "P" represents a partial benefit/drawback, and "X" indicates that the strategy does not have the benefit/drawback.

Table 5.3 indicates that all the strategies discussed in Section 5.2.1 adhere at least partially to the benefits of an automatic assessment strategy as highlighted in Section 4.3. For the "formative and summative" benefit, only assessment for Kattis explicitly states that it can be used for both formative and summative assessment [39], thus all the other strategies are labelled as "partly" (P), as the strategies predominantly cater for summative assessments. Easy Accept scored a partial for the "Speed" and "Immediate Feedback" benefits, as the regression tests may affect the speed at which assessments are evaluated and the rate at which feedback can be given to the student.

Regression testing involves running all previous tests, and as such may require an extended amount of time to execute all the tests [102]. BOSS and Easy Accept both have the option for the instructor to manually compile a feedback report, thus delaying

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

feedback, and therefore were awarded a "partly" classification. Fitchfork was classified as "partly" for Consistency/Objectivity, as from firsthand experience, especially with concurrent programming assignments, it was possible to upload the exact solution for the same assignment and receive different results. This was partly due to the thread interleaving that occurred during the testing of the solution. Fitchfork was also given a combination of check (✓) and "partly" rating for the "Multiple attempts", as it is possible to give students a single submission or multiple submissions depending on the type of assignment.

For the drawbacks listed in Table 5.3, QuizPACK does not require more time or effort to set up compared to normal questions, as a template of the question is used, and QuizPACK generates the correct output. This differs from the other strategies, as all the others require multiple solutions and more complex test cases and projects. Thus, QuizPack scored an "X" while all the other strategies scored a check (✓). BOSS and EasyAccept have a partial drawback for the "Lack of adequate feedback" drawback as the instructor has the option to send a detailed feedback report to the student. This is only a partial drawback, as it remains the instructor's choice. FitchFork, Peach and Kattis all suffer from the drawbacks listed in Section 4.4.

	Generation	Assessment Criteria	Feed-back	Additional Support	How is correctness determined
Quiz PACK	3	Correctness	A	Does not require any additional support but may utilise it if necessary.	Comparing the output the students' answers generate to the expected output.
BOSS	3	Correctness and efficiency	M	Option for additional support by creating the feedback report if needed.	Using the student's JUnit object and matching their data to the expected data.
Easy Accept	2	Correctness	A	Does not require any additional support but may utilise it if necessary.	Using regression testing to ensure that the specification is followed and the solution is accepted.
Peach	2	Correctness	M	Option for additional support by creating the feedback report if needed.	Utilising testing packages to test for correctness.
Kattis	3	Correctness and efficiency	A	Students still need to demo to instructors after submitting to Kattis. Thus, a high volume of additional support is needed.	Unit tests to test for correctness.
Fitch Fork	3	Correctness, memory management, and efficiency	SA	Tutors are still needed to clarify the feedback, especially if the feedback "de-syncs" and incorrect feedback is displayed. Tutors also provide information on runtime errors.	Matching output with expected output, using sets of different test input data.

Table 5.1: Comparison of common dynamic automated assessment strategies with a set of common aspects.

	Test Data (TD)	Test Data Creation	Language Restriction	Assessment Types	Correct Output Generation
Quiz PACK	Question templates where values can be inserted at random to generate new questions	Random value generator	C	Introductory programming courses	Running an instructor provided solution with question values to ensure student's answer is correct
BOSS	Input files containing input data or Java Objects	Instructor created	Java	Java assignments that can use JUnit	Manually created by either using asserts in the JUnit tests or in the expected output file
Easy Accept	Input is given within a set of acceptance tests	Instructor created	Java	User Story based assignments	Manually created
Peach	Input files created for each test package	Instructor created	None	Any programming assignment	Running an instructor provided solution to determine correct output
Kattis	Input files	Instructor created	None	Introductory Programming, Algorithms, Data Structures and Networking assignments	Running an instructor provided solution to determine correct output
Fitch Fork	Command line arguments, text file or main function	Instructor created	None	Introductory Programming, Data Structures and Algorithms, Computer Architecture and Concurrent Programming assignments	Manually created

Table 5.2: Comparison of common dynamic automated assessment strategies with a set of extended aspects.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

	Benefits						Drawbacks			
	Speed	AVAIL	Consis-tency	Immediate FDBK	Multiple Attempts	Formative and Summative	Testing data	Bad FDBK	Strict-ness	Time Effort
Quiz Pack	✓	✓	✓	✓	✓	P	✓	✓	✓	X
BOSS	✓	✓	✓	P	✓	P	✓	P	✓	✓
Easy Accept	P	✓	✓	P	✓	P	✓	P	✓	✓
Peach	✓	✓	✓	✓	✓	P	✓	✓	✓	✓
Kattis	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fitch Fork	✓	✓	P	✓	✓/P	✓	✓	✓	✓	✓

Table 5.3: Comparison of different common assessment strategies in terms of how each complies with the benefits and drawbacks of automated assessment.

5.2.2 Distinctive Strategies

Any strategy that differs from the status quo of common dynamic automated assignment strategies is labelled as a distinctive strategy. Examples are discussed and compared below.

Examples of Distinctive Strategies

The following dynamic automated assessment strategies serve as examples of distinctive strategies:

- **Edgar:**
  Edgar was introduced by Mekterovic [76] and uses a graph[5] structure to generate tests, where questions are grouped into nodes, and new questions are formulated based on the nodes. Thus, Edgar has the ability to generate any possible number of questions based on this grouping of nodes. Edgar, at the time of publication is the only known automated assessment strategy that can accommodate both, as the authors state, "standalone" and "managed" programming languages. Standalone programming languages are defined as any programming language that can be interpreted and compiled by its own compiler/interpreter, for example, C++ or

[5] A graph data structure is a data structure that is comprised of vertices and edges. Edges connect the vertices with each other. One vertex may have multiple edges to create a network/web-like structure.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Java. Managed programming languages, on the other hand, are programming languages that are used by other systems to accomplish a special task, for example, SQL[6] or CSS[7]. Although Edgar can use the output from executing the code, the reason why it is labelled as distinct is due to the graph theory-based generation of programming questions.

- **Web-CAT:**
  Aaltonen et al. [1] deviate from common strategies in the sense that what is assessed is not the student's ability to implement the assignment, but rather the student's ability to write tests for their own code . This is done using a metric known as code coverage. This metric is complemented by another metric known as mutation analysis. Code coverage is a metric that determines the percentage or portions of the code that was not executed. This is used to determine areas of code that may still contain bugs and remain untested [105]. Mutation testing was discussed in Chapter 3. Web-CAT uses a combination of the two aforementioned metrics to assess the student's ability to create tests for their own assignments.

- **CLP strategy:**
  Dewey et al. [31] proposed using constraint logic programming (CLP) to create a test suite to assess students' assignments. CLP assumes that all parts of a problem are stated as constraints, which are combined by a set of rules. CLP aims to solve the constraints that were derived from the problem [60]. The CLP strategy uses CLP to formulate a testing package that can be used to assess student's code, which, according to the author, could find more errors in the student's code as compared to the hand-crafted testing packages. The CLP strategy thus differs from the common strategies in the sense that the tests are automatically created by a computer using CLP. CLP can also be used to evaluate the effectiveness of the tests created by the instructor. LTL formulae can be expressed as part of annotated CLP

[6]Structured Query Language (SQL) is a query language that is used to manipulate data in a relation database.

[7]Cascading Style Sheets (CSS) is used to add styling to HTML pages.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

where the LTL formulae can be expressed as a constraint annotated with temporal information [44]. However, the CLP strategy did not utilise annotated CLP.

Comparison of Existing Distinctive Strategies

Tables 5.4 and 5.5 compare the strategies listed. Table 5.4 compares the common aspects, and shows that only Edgar has no restriction on the programming language, whereas Web-CAT and CLP Strategy make use of Java. Edgar uses question-based specifications for students to implement, whereas Web-CAT and the CLP Strategy can perform assessment on any assignment type. Table 5.5 shows that the listed strategies generate feedback for students automatically. Edgar and CLP Strategy use correctness as part of the assessment criteria.

Benefits and Drawbacks

Table 5.6 lists the benefits and drawbacks of the distinct dynamic strategies listed in Section 5.2.2. As can be seen, the distinct strategies also suffer from the drawbacks as listed and described in Section 4.4. Edgar scored an "unknown" (U) for the multiple attempts benefit, as the report describing Edgar did not specify the number of attempts for different sections. Web-CAT and CLP Strategy received a check ($\checkmark$) for the Testing data drawback, as the reports state that the generated test data can be released back to the student.

5.3 Summary

In this chapter, different existing automated assessment strategies were discussed. It was discovered that the use of instructor provided solutions to generate expected output is an established practice in the automated assessment of programming assignments. It is also important to note that different automated assessment strategies create test input data differently, and as such, it can be concluded that there is no standard way to create test input data for automated assessment. The next chapter introduces a novel automated assessment strategy where a genetic algorithm is employed for test input data generation.

	Test Data (TD)	TD Create	Lang Rest	Assessment Types
Edgar	Test questions/Instructions	Autogenerated based off graph theory	Any	Questions and simple assignments
Web-CAT	Mutation Analysis	Automatic	Java	Any assignments
CLP Strategy	Tests created by CLP	Automatically created by CLP	Java	Any assignments

Table 5.4: Comparison of distinctive dynamic automated assessment strategies with a set of common aspects.

	Generation	Assessment Criteria	FDBK	Additional Support
Edgar	3	Correctness of answers	A	Questions are automatically generated but need a question bank to formulate the graph for question generation.
Web-CAT	3	Code coverage and mutation analysis	A	Student's testing is evaluated, and as such will be able to give precise feedback on where they can improve testing.
CLP Strategy	3	Correctness using CLI	A	Does not specify explicitly.

Table 5.5: Comparison of distinctive dynamic automated assessment strategies with a set of extended aspects.

	Benefits						Drawbacks			
	Speed	AVAIL	Consis-tency	Immediate FDBK	Multiple Attempts	Formative and Summative	Testing data	Bad FDBK	Strict-ness	Time Effort
Edgar	✓	✓	✓	✓	U	P	✓	✓	✓	✓
Web-CAT	✓	✓	✓	✓	✓	✓	P	P	✓	✓
CLP Strategy	✓	✓	✓	✓	P	P	✓	P	✓	✓

Table 5.6: Comparison of different distinctive assessment strategies in terms of benefits and drawbacks.

Chapter 6

Proposed Assessment Strategy

AI has been applied to the area of software testing, as discussed in Chapter 3, however, to the best of the author's knowledge (up until September 2024), an area that has not yet incorporated AI is dynamic automated assessment of programming assignments. The assessment strategy proposed in this dissertation aims to fill this gap. In this chapter, the new automated assessment strategy is formally introduced and is referred to as **G**enetic **A**lgorithm with **T**emporal **L**ogic for **A**utomated **M**arkers (GATLAM). Briefly, GATLAM is an automated assessment strategy that aims to assess programming assignments using a GA to create input values with the hopes that LTL properties will be violated.

The remainder of the chapter is structured as follows: Section 6.1 provides a brief overview of GATLAM, as well as an abstract set of steps which GATLAM follows during the evaluation of a programming assessment. Section 6.2 discusses the internal finer details of GATLAM. Section 6.3 discusses GATLAM in terms of the benefits, drawbacks, and aspects listed in Chapter 4, and discusses some concerns associated with GATLAM. Finally, Section 6.4 provides a summary of the chapter.

6.1 GATLAM Overview

An abstract overview of the steps that GATLAM follows during the evaluation of a student is illustrated in Figure 6.1. GATLAM comprises four main components, namely:

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

a GA (to generate abstract test data), interpreter (to convert the abstract test data to concrete test data), and solutions (which are represented by the instructor-provided solution and the student-provided solution).

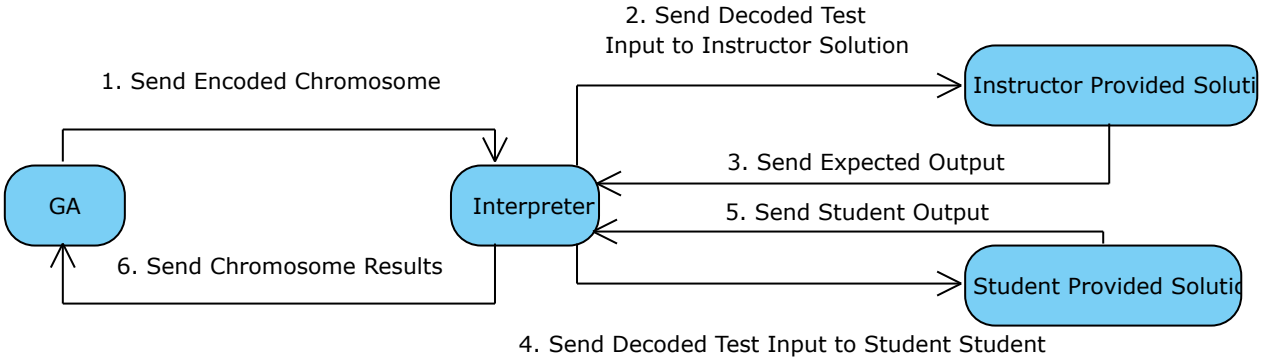


Figure 6.1: Abstracted view of GATLAM, with the starting and end point being the GA step.

The steps shown in Figure 6.1 are described below.

1. **Send Encoded Chromosome**
   This step involves selecting a chromosome and sending the chromosome to be converted to test input data for the solutions. The chromosome is sent from the GA to the interpreter as an array of integer values, which are easier to interpret than an array of bytes.

2. **Send Decoded Test Input to Instructor Solution**
   This step involves the translation of the encoded chromosome to test input data that is then passed to the instructor-provided solution.

3. **Send Expected Output**
   The output from the instructor-provided solution is returned to the interpreter and is stored as the expected output in the interpreter.

4. **Send Decoded Test Input to Student Solution**
   This step involves the translation of the encoded chromosome to test input data, which is then passed to the student-provided solution.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5. **Send Student Output**
   This step is similar to the expected output from Step 3. The only difference is that the output from the student's solution is stored in the interpreter as the student's solution.

6. **Send Chromosome Results**
   This step consists of returning the results from the interpreter to the GA. These results are then used to assign fitness to the chromosome, and to evolve the GA's population.

The above steps are then repeated for every chromosome that is evaluated. Thus, both of the solutions can be repeatedly tested until the maximum number of generations for the GA has been reahced.

6.2 GATLAM Internal Details

This section explores the internal details of GATLAM, but before each component is discussed, Section 6.2.1 provides an in-depth overview of the LTL properties used by GATLAM. These properties are used by the GA to evaluate the fitness of a chromosome, by describing the behaviour of the assignment being assessed. Section 6.2.2 discusses how the GA is utilised by GATLAM. Section 6.2.3 discusses the two solutions that are used during the assessment of a student's programming assignment. Section 6.2.4 discusses how the GA and solution components are combined via the interpreter. Finally, Section 6.2.5 describes the two different running modes in which GATLAM can be configured to execute.

6.2.1 LTL Properties

To evaluate if the assignment under assessment behaved as expected, a set of predefined temporal logic formulae are used. The reason for using predefined temporal logic formulae is to be able to describe how the assignment under assessment behaved. Using a configuration file, the temporal logic formulae are either enabled or disabled, as well as configured to allow for different errors to be detected based on the type of assignment

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

being assessed. As part of the configuration of the temporal logic formulae, each formula can be enabled or disabled, which allows the instructor to configure the strictness of the strategy. It should be noted that the term "LTL property" refers to the name of the formula. Table 6.1 lists the name, the LTL formula, and its negation of the LTL formula for each property.

Property name	**LTL Formula**	**Negation of LTL Formula**
Safety	$G(\neg(\text{unsafe}))$	$F(\text{unsafe})$
Proper Termination	$G(\text{ter} \Rightarrow \text{VRC})$	$F(\text{ter} \wedge \neg\text{VRC})$
Segmentation Fault	$G(\neg(\text{segfault}))$	$F(\text{segfault})$
Exceptions	$G(\neg(\text{exception}))$	$F(\text{exception})$
Execution Time	$G(\text{ter} \Rightarrow (r \leq e))$	$F(\text{ter} \wedge (r > e))$
Illegal output	$G(\text{ter} \Rightarrow (\bigwedge_{o \in \text{Output}} \bigwedge_{x \in X} (x \neq o)))$	$F(\text{ter} \wedge (\bigvee_{o \in \text{Output}} \bigvee_{x \in X} (x = o)))$
Expected output - Exact match	$G\left(\text{ter} \Rightarrow (\bigwedge_{i=1}^{\text{length}(Y)} (y_i = s_i))\right)$	$F(\text{ter} \wedge (\bigvee_{i=1}^{\text{length}(Y)} (y_i \neq s_i)))$
Expected output - Contains	$G(\text{ter} \Rightarrow \left(\bigwedge_{i=1}^{\text{length}(Y)} \bigwedge_{j=1}^{\text{length}(S)} (y_i = s_j)\right))$	$F(\text{ter} \wedge \left(\bigvee_{i=1}^{\text{length}(Y)} \bigvee_{j=1}^{\text{length}(S)} (y_i \neq s_j)\right))$

Table 6.1: LTL properties, defined by name, LTL formula, and the negation of the LTL formula.

The *Expected output - Exact match* and *Expected output - Contains* are sub-properties of the *Expected output* property. The following general notation was used to formulate the LTL formulae as listed in Table 6.1:

- The G operator is defined as *always* or *globally*. Thus, something that always needs to happen [13].
- The F operator is defined as eventually. Thus, something that will *eventually* happen [13].
- The $unsafe$ label refers to states/program points along the execution path that result in unsafe[1] behaviour, for example:
  - Erroneous output
  - Non-terminating program
  - Unhandled exceptions
  - Memory errors

[1]Safe behaviour is defined as nothing unwanted ever happening during execution [61]. Thus, unsafe behaviour is when something unwanted occurred.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- The *exception* label refers to states/program points along the execution path that result in an uncaught exception being thrown.
- The *segfault* label refers to states/program points along the execution path that result in a segmentation fault[2] occurring.
- The *VRC* (Valid Return Code) label refers to states/program points along the execution path that result in an exit code of 0 returned.
- The *ter* label refers to states/program points along the execution path that result in the program correctly terminating execution.
- The $r \leq e$ atomic proposition refers to the real execution time r of the assignment being less than or equal to the expected execution time e.
- The *Output* set refers to the output produced by the assignment being assessed, where o is a single element of the set *Ouput*.
- The X set refers to illegal output that may not be present in the output produced by the assignments, where x is a single element of illegal output in the set X.
- The Y label refers to the ordered set of expected output, and y are elements of Y.
- The S label refers to the ordered set of output produced by the assignment being assessed, and s are elements of S.

The reason why Y and S are ordered sets is that the order in which a program produces output is also important, as it ensures that the correct execution path was followed, as defined by the specification which the program was designed.

The decision to negate each LTL formula and use the negation to implement the assignment properties stems from the concept of existential model checking [109]. This approach is based on the principle that determining whether a property holds for all possible execution paths is equivalent to determining whether the negation of the property holds for at least one execution path [19]. By applying this idea, GATLAM only needs

[2] A segmentation fault is when a program tries to access a block of memory that is not assigned to that program.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

to find a single execution path where the negation of the LTL property holds to conclude that the property does not hold universally. The full list of negation derivations can be found in Appendix A of this dissertation.

Concrete examples are provided below to demonstrate how the properties listed in Table 6.1 are utilised.

Safety

Algorithm 6.1 is used to illustrate how the *safety* property can be utilised. Algorithm 6.1 indicates that if the input parameter's `anotherObj` member variable is not NULL, then `obj` is erroneously deleted twice. Assuming C++ is used, this can lead to a `double free or corruption` error as a block of memory that has already been freed is freed again. If an object pointer is passed to the `deallocatePointer` whose member variable is not NULL, the code undergoes a `double free or corruption` error, which is an unsafe behaviour. It should, however, be noted that a double delete has undefined behaviour according to the C++ language specification. On some compilers, a double free will occur when performing a double delete, and in others, a segmentation fault might occur. This is due to the difference in implementation of the C++ language specifications. Thus, the safety property is violated, as eventually, the program becomes unsafe.

```
struct Object{
    Object* anotherObj;
};

void deallocatePointer(Object* obj){
    if(obj->anotherObj != NULL){
        delete obj;
    }
    delete obj;
}
```

Algorithm 6.1: Example code for *safety* property.

Proper Termination

The *proper termination* property is illustrated in Algorithm 6.2. Based on the input that is passed to the program during runtime, the execution of the program may result

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

in a non-zero return code. If any non-zero input is passed to the program shown in Algorithm 6.2, the program terminates with an exit code that is not 0, violating the proper termination property.

```
#define UNSAFE 1
#define SAFE 0

int main() {
    int input;
    cin >> input;
    if(input != 0){
        //Improper termination reached
        return UNSAFE;
    } else {
        return SAFE;
    }
}
```

Algorithm 6.2: Example code for *proper termination* property.

Segmentation Fault

The *segmentation fault* property is demonstrated using Algorithm 6.3. In the program , a segfault occurs if the input pointer is a NULL[3] pointer. If NULL is passed in as a parameter to the `printFunc` function, the *segmentation fault* property is violated.

```
void printFunc(Object* obj){
    cout << obj->getStr() << endl;
}
```

Algorithm 6.3: Example code for *segmentation fault* property.

[3]A null pointer is a pointer that has the address 0×0 and does not point to a valid memory location.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Exceptions

Algorithm 6.4 demonstrates how the *exception* property is used, and is loosely based on Java's `Integer` class. An exception[4], specifically the "NumberFormatException", is thrown if the input parameter `str` is not an integer, violating the exception property.

```
public Integer convertToInt(String str){
    return Integer.parse(str);
}
```

Algorithm 6.4: Example code for *exception* property.

Execution Time

The next property that is illustrated is the *execution time* property. Algorithm 6.5 describes a program that enters an infinite loop if the input parameter, `stop`, is passed a `true` value. If `true` is passed into the function `f`, the code results in an infinite loop error, thus exceeding the allocated execution time and violating the *execution time* property.

```
void f(bool stop){
    while(stop){
    }
}
```

Algorithm 6.5: Example code for *execution time* property.

Illegal Output

This property is used to ensure that there are no forbidden words in the output generated by the solution. As an example, let the set of forbidden words be "error", "problem", and "issue". Algorithm 6.6 violates the *illegal output* property if a non-zero value is passed as a parameter to the `successRun` function. The property is violated as element(s) in the set of forbidden words are present in the output.

[4] An exception is an error in the program that halts execution. The difference is the program can be structured in such a way that the error can be dealt with, without the program terminating.

```
void successRun(int value){
    switch(value){
        case 0:
            cout << "Ran successfully" << endl;
            break;
        case 1:
            cout << "There is a problem" << endl;
            break;
        case 2:
            cout << "An error occurred" << endl;
            break;
        default:
            //Illegal output printed
            cout << "There is an issue" << endl;
    }
}
```

Algorithm 6.6: Example code for *illegal output* property

Expected Output - Exact Match

The *expected output* property has two sub-properties, namely: *exact match* and *contains*. The *exact match* sub-property is where the entire output of the student solution needs to match the output of the instructor's solution. Algorithm 6.7 indicates that the *exact match* sub-property is violated when `a` and `b` are not the same and "invalid" is printed, where "valid" is the correct expected output.

```
void assertFunction(int a, int b){
    if(a == b){
        cout << "valid" << endl;
    } else {
        //Expected output violated
        cout << "invalid" << endl;
    }
}
```

Algorithm 6.7: Example code for *expected output - exact match* property.

Expected Output - Contains

The *contains* sub-property of the *expected output* property states that the instructor solution's output should be contained in the student's output. This is a less strict sub-

property of the *expected output* property. Assume "valid" is the instructor's output. The sub-property is violated when the word "valid" does not appear in the student's output. Using Algorithm 6.8, the sub-property is thus violated when `a` and `b` do not match, and the sub-property is not violated if `a` and `b` match. Even though the student's output does not perfectly match the instructor's output, the instructor's output is contained within the student's output in the case where `a` equals `b`.

```
void assertFunction(int a, int b){
    if(a == b){
        cout << "the␣a␣and␣b␣match␣and␣and␣the␣comparison␣is␣valid" <<
           endl;
    } else {
        //Expected output violated
        cout << "not␣correct" << endl;
    }
}
```

Algorithm 6.8: Example code for *expected output* - *contains* property.

The GA utilises the properties discussed in Table 6.1 as part of its fitness function to detect errors that are similar to the errors illustrated by the examples in this section. Section 6.2.2 discusses the GA component of GATLAM.

6.2.2 Genetic Algorithm

The GA is used to generate test input in an algorithmically "intelligent" way to explore the search space of possible input values. The GA creates a chromosome bounded by some requirements or restrictions, and passes it to the interpreter. The GA iteratively generates chromosomes, evaluates, and alters them as described in Chapter 2 until specific stopping criteria have been met. The remainder of the section discusses the different components of the GA that GATLAM utilises.

Gene and Chromosome Representation:

A gene (which represents a part of the abstract test case) in the chromosome (which represents the entire abstract test case) is represented by an integer number. This

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

number is bounded by the instructor in order to reduce the search space. The bounds of the number are based on the instructor's experience and assessment style, and the way the interpreter was configured. The author recommends that in future studies, a generic interpreter and boundaries for numbers should be investigated.

The chromosome, which consists of a set of integer numbers, is then encoded as a binary string. The length of the binary string is determined by the following functions:

$$largestMagnitude(\texttt{minVal}, \texttt{maxVal}) := max(|\texttt{minVal}|, |\texttt{maxVal}|)$$

$$length(biggestValue) := length(convertToBinary(largestMagnitude)) + 1$$

Both `minVal` and `maxVal` are parameters chosen by the instructor and represent the minimum value and the maximum value for the gene in its integer representation. The absolute value of both the minimum value and the maximum value is used to determine which value has the greatest magnitude. The conversion to binary is done by the *convertToBinary* function, which converts the number into a minimal-length binary string that contains no extra padding, using the unsigned representation of the number. The largest of the two absolute values is then used to determine the minimum number of bits to represent the value. An extra 1 bit is added to the length to account for the sign[5].

To illustrate how the encoding works, the following example is used: Assume a gene is bounded between the two decimal numbers -10 and 9. The absolute maximum value between $|-10|$ and $|9|$ is 10. The binary string representation for 10 is 1010. Thus, the length of the binary string that represents the gene is $4 + 1 = 5$.

A chromosome is thus a concatenation of all of the binary strings of each gene. To illustrate this, assume the chromosomes consist of 10 genes, each bounded between -15 and 15. Then the binary representation of an example chromosome is:

00111	10100	01110	10101	00100	00001	01001	01010	01110	01001

which can then be decoded to the following gene values in decimal:

[5]The sign bit is the leading bit of the string, and a positive bit is indicated by a sign bit of 0.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7	-4	14	-5	4	1	9	10	14	9

Genetic Algorithm Configuration

The GA is configured with a configuration file defining the parameters listed in Table 6.2.

Parameter	**Description**
Population Size	Defines the number of chromosomes in the population.
Number of Generations	Defines the number of generations the algorithm is allowed to run for, which will be the termination criterion for GATLAM.
Selection Size	Defines the number of chromosomes that are replaced by the selection method in each generation.
Reproduction Probability	The probability that a chromosome from the mating pool undergoes reproduction to produce a new offspring. This parameter must be a positive real value bounded between 0 and 1.
Crossover Probability	The probability that two chromosomes from the mating pool undergo crossover to produce two new offspring. This parameter must be a positive real value bounded between 0 and 1.
Mutation Probability	The probability that a chromosome from the mating pool undergoes mutation to produce a new offspring. This parameter must be a positive real value bounded between 0 and 1.
Genes	An array of gene configurations that describe the bounds for each gene in the chromosome, as well as any invalid values.
Type of Crossover	States what type of crossover operator can be applied to chromosomes in the mating pool during the run of the GA.
Type of Mutation	States what type of mutation operator can be applied to chromosomes in the mating pool during the run of the GA.

Table 6.2: GATLAM's GA Parameters.

Examples of invalid gene values include passing a gene with the value of 0 as a denominator in a division operation. These hyperparameters need to be set by the instructor, and allow the instructor to control the convergence and exploration behaviour of the GA.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Fitness Function

The proposed fitness function consists of three weighted terms which represent the LTL properties component, the failure component, and the memory component. This allows the fitness function to be adapted based on the instructor's assessment style and past experience. Each component is discussed in greater detail in the remainder of this section and operates on the submitted assignment being assessed. The entire fitness function is the combination of the above three terms, as defined in Equation (6.1).

$$f(x, gen) = \omega_1 \Big(LTL(x) \Big) + \omega_2 \Big(Fail(x) \Big) + \omega_3 \Big(Mem(x, gen) \Big), \tag{6.1}$$

In Equation (6.1), the following notation is used:

- x is the chromosome currently being evaluated.
- gen is the current generation that the algorithm is at.
- ω_1, ω_2, and ω_3 are the weights assigned to the LTL, failure, and memory fitness function components, respectively. These weights have the following properties:
  1. Each weight is a real value bounded between 0 and 1.
  2. The summation of the three weights needs to be equal to 1.

  The result of the fitness function will thus be bounded between a 0 and 1, given that each individual component is also bounded between 0 and 1.
- *LTL*, *Fail* and *Mem* functions are sub-fitness functions, which relate to the fitness function components, and are discussed independently below.

The *LTL* component is defined as follows, and uses the LTL properties listed in Table 6.1:

$$LTL(x) = \frac{\sum_{i=1}^{n} TL_i(x)}{n}$$

where:

- n is the number of temporal logic properties that are currently enabled.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- $TL_i(x)$ returns the resulting value of the negation of the i^{th} temporal logic property for chromosome x. If the program input associated with chromosome x results in an execution path that violates the i^{th} property, the function returns a 1, and if the property is not violated, a 0 is returned.

The reason why the number of violated LTL properties is divided by the total number of enabled LTL properties is that the return value of the *LTL* sub-fitness function must be bounded between 0 and 1.

The failure component is defined as follows:

$$Fail(x) = \frac{\sum_{i=1}^{m} M_i(x)}{m}$$

where:

- m is the number of tasks that the assignment consists of. A task is a smaller part of the assignment and deals with a specific portion of the assignment.
- $M_i(x)$ returns a 1 if any LTL property was violated by the i^{th} task using chromosome x, otherwise it returns 0.

The sum of the failed tasks is calculated to determine how many tasks failed. This is divided by the total number of tasks such that the result of the *Fail* sub-fitness function must be bounded between 0 and 1.

The last component, namely the memory component, is defined as follows:

$$Mem(x, gen) = \frac{\sum_{i=1}^{k} \frac{G_i(x)}{NC(gen)}}{k}$$

where:

- k is the number of genes in chromosome x.
- $G_i(x)$ returns the number of times the i^{th} gene of chromosome x was present in previously evaluated chromosomes from previous generations, where an LTL property was violated. Thus, $G_i(x)$ gives the fitness function a "memory" of past chromosomes.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- $NC(gen)$ function returns the total number of chromosomes that have been evaluated at the start of the gen^{th} generation.

The $G_i(x)$ is divided by the $NC(gen)$ function such that each value that is summed together is bounded between 0 and 1. The summation is then again divided by the number of genes such that the return value of the *Fail* component is also bounded between 0 and 1. The memory component allows the GA to not only utilise the information of the current chromosome, but also previous chromosomes that experienced at least one LTL violation. Thus, genes that are repeatedly present when LTL properties are violated have a higher chance of either directly causing the violation, or being part of a set of genes that cause the violation.

Each of the weights, namely, ω_1, ω_2, and ω_3, can also be configured by the instructor such that the importance of different aspects of the fitness function can be fine-tuned.

6.2.3 Solutions

Due to the fact that the assessment of the student's assignment is based on different dynamically generated test input data, the expected output cannot be determined by the instructor before submission. Therefore, lending the idea from some of the dynamic strategies listed in Chapter 5, GATLAM can make use of two solutions: the instructor solution and the student solution. Thus, GATLAM can be configured to make use of an instructor solution based on the selected running mode, which are discussed in Section 6.2.5. The Instructor and Student solutions are discussed below.

Instructor Solution

The instructor provides the solution, which serves as the model solution that generates the expected output. The instructor-provided solution is executed with the same inputs as the student-provided solution.

Student Solution

The student also provides a solution. This is the solution that is being assessed and graded. Thus, this solution is used to determine the fitness of the chromosome.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

6.2.4 Interpreter

The GA designed in this study generates integer values. As such, an "interpreter" is needed to translate these primitive values to more complex values that can be used to evaluate a more complex assignments . The interpreter concept is based on the interpreter design pattern [57], wherein a simplified version of one system has the ability to communicate with another system. The interpreter is defined for each assignment, and thus allows GATLAM to generalise to any type of assignment[6]. The interpreter also bridges the gap between the black box nature (where the implementation is unknown) of the GA and the white box nature (where the implementation is known) of the assignment that is evaluated. The interpreter needs to be supplied with the instructor's solutions such that the translation from the chromosome values to the input data for the assignment can be accomplished.

The interpreter receives the array of numbers as input, which represents the current chromosome being evaluated. In return, the interpreter produces the following: program output, execution time, and return code from both the student and instructor-provided solutions depending on the running mode used. The output contains both the standard output and the standard error output from the respective solutions. As with most automated assessment strategies, due to the possibility of never terminating solutions, the interpreter is also configured with a maximum allowed execution time. If a solution exceeds this maximum allowed execution time, the interpreter halts the execution of the solution and marks it as an infinite loop.

The complexity of the interpreter depends on the complexity of the test input data that the instructor would like to create. If, for example, the student needs to create a calculator, the instructor can simply map the first gene to an operator, and the remaining genes to inputs for the operator. For another example, assume the student needs to implement a program that calculates the average of a set of values. Then the interpreter can simply pass the abstract test data to the solutions without having to perform any additional functionality. A more complex approach would be to have the interpreter

[6] An example of an interpreter can be found at: https://github.com/Cobus-Redelinghuys/FJ-Redelinghuys-MSc-Genetic-Algorithm-with-Temporal-Logic-for-Automated-Assessment/tree/Real_Interpretor

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

create a main function where each gene is either mapped to a function call or input for a function call. The interpreter then creates a main function based on the genes. As the complexity of the interpreter is up to the instructor, GATLAM can be configured to adapt to the difficulty of the assignment.

6.2.5 Running modes

Due to how the majority of the LTL properties defined in Section 6.2.1 are set up, it is possible for GATLAM to be used in two different modes, namely:

1. Single Solution (SS) mode.
2. Instructor-Student Solution (ISS) mode.

The two above-mentioned modes are discussed in the remainder of the section, with SS being discussed first, followed by ISS.

Single Solution Mode

In the SS mode, only one solution (the student's submitted assignment) is used to gather output and feedback. This implies that the LTL property's expected output has to be kept constant for the entire execution of GATLAM. Figure 6.2 shows the data flow for GATLAM when configured in SS mode. In step 1, the raw chromosome is transferred from the GA to the interpreter, which, as indicated by step 2, converts the chromosome to test input data (TID). The appropriate portion of the TID is then sent to each task of the system under test (indicated by the Solution node), which is then executed using this TID. In the case of Figure 6.2, the TID is the entire `void testFunction`. The result of the execution is collected and sent back to the interpreter in step 3. All of the results of each task are collected and formatted by the interpreter, which then passes the formatted results back to the GA (indicated by step 4). This process is repeated for each chromosome evaluation of GATLAM running in SS mode. The major difference between Figures 6.1 and 6.2 is the lack of asecond solution. SS mode can be used if the output is not considered for correctness, and only the absence of any runtime or memory-related errors is considered for correctness. The author recommends that future

studies incorporate custom LTL properties to enhance the expressiveness of SS mode correctness.

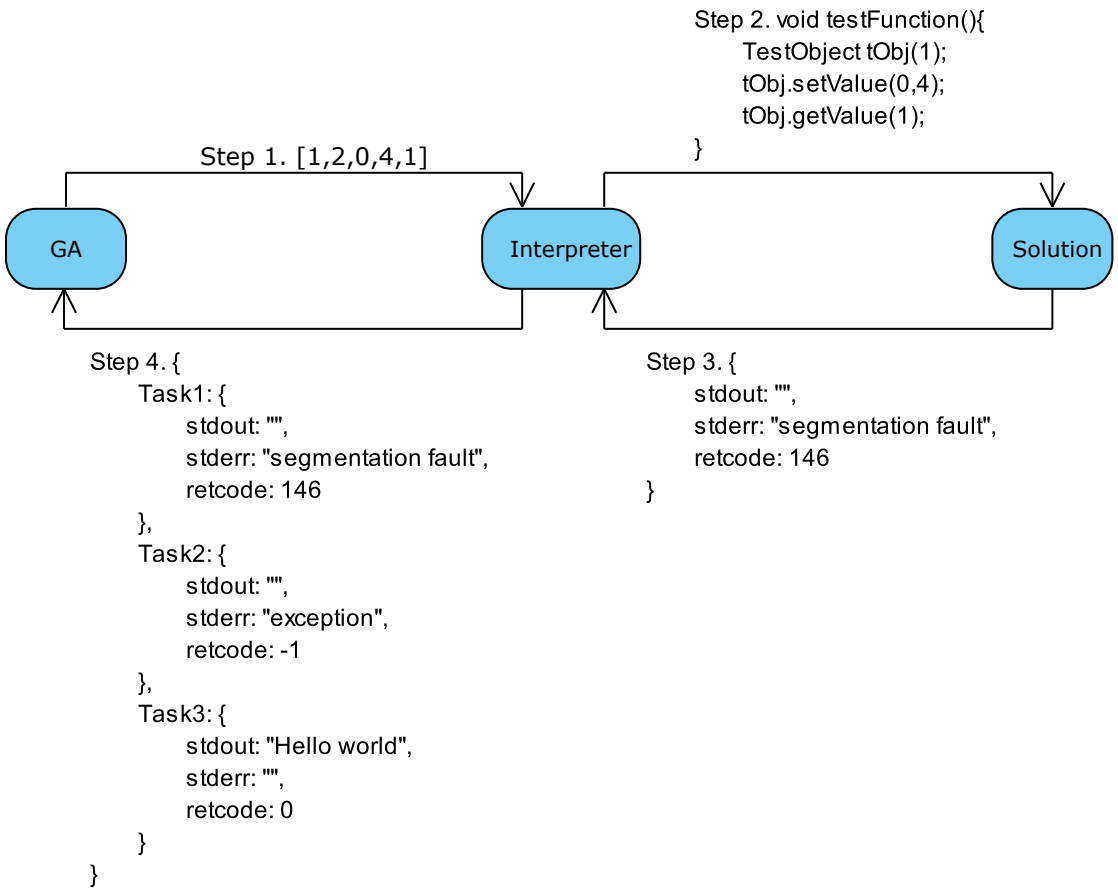


Figure 6.2: Example of the data flow of GATLAM when configured in SS mode.

Instructor-Student Solution Mode

In the ISS mode, both Instructor and Student solutions are present, and both gather output and feedback. Figure 6.3 shows the data flow for GATLAM when configured in ISS mode. The entire data flow in the ISS mode is nearly identical to the data flow in the SS mode, with the only difference being that steps 2 and 3 are repeated for the different solutions. For the instructor solution, steps 2 and 3 are represented as steps 2 (a) and 3 (a), while for the student solution, steps 2 and 3 are represented as 2 (b) and

3 (b). The reason for this representation is due to the same process for steps 2 and 3 being followed, just applied to the different solutions.

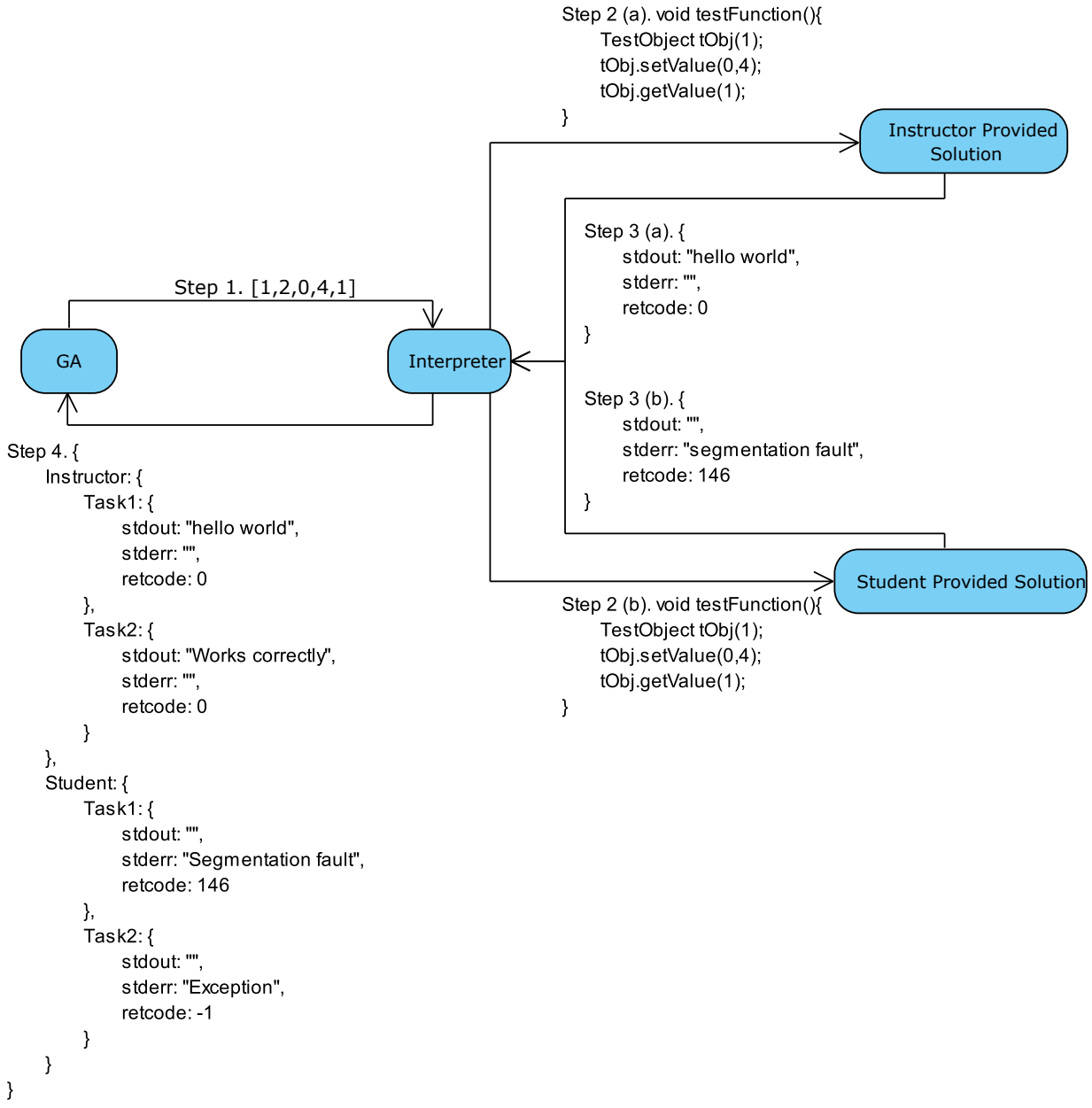


Figure 6.3: Example of the data flow of GATLAM when configured in ISS mode.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

6.3 Benefits, Drawbacks and Concerns

This section explains how GATLAM attempts to address each of the benefits, drawbacks and aspects as discussed in Chapter 4. This section also addresses some concerns that accompany GATLAM. The section is laid out as follows: Section 6.3.1 discuses the benefits, Section 6.3.2 discusses the drawbacks, Section 6.3.3 discusses the aspects of automated testing, and lastly Section 6.3.4 discusses the concerns. Concerns are aspects of GATLAM that need to be taken into account when using GATLAM to assess an assignment.

6.3.1 Benefits

This section discusses how GATLAM addresses each of the benefits listed in Section 4.3.

1. **Speed**
   GATLAM does not fully accomplish the speed benefit as the strategies in Chapter 5 demonstrated. To support this statement, consider the function, defined in Equation (6.2), that describes the time it takes GATLAM to complete its execution in a worst-case scenario. The function[7] also assumes that there is no delay between the three fitness function components, nor any delay with the creation and evolution of the chromosomes within the GA. This example utilises the tournament selection method as discussed in Chapter 2, due to only a subset of the entire population being evaluated.

$$\texttt{maxRunningTime}(g, t, c, e) = 2 \times g \times 2 \times t \times c \times e \tag{6.2}$$

   where:

   - g refers to the number of generations that the GA executes for.
   - t refers to the number of tournaments.

[7] This theoretical maximum assumes both of the solutions encounter an infinite loop.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- c refers to the number of contestants per tournament.
- e refers to the maximum execution time in seconds.

In the `maxRunningTime` function, $t \times c \times e$ refers to the selection of chromosomes for crossover, and their fitness calculations. We then multiply it by two to give $2 \times t \times c \times e$, which refers to the process of selecting both chromosomes going into the mating pool which are also the chromosomes that are replaced. Thus, the selection process is repeated twice. After that, g is multiplied to form $g \times 2 \times t \times c \times e$, which is the maximum evolution time per solution. Again, the result is multiplied by two to get $2 \times g \times 2 \times t \times c \times e$, which in turn refers to the fact that two solutions are executed. Note, the two solutions are evoled together, based on the results obtained from the student solution.

To further illustrate Equation (6.2), let us assume the following configurations for GATLAM, each trying to improve the previous configurations' efficiency..

(a) Every chromosome is re-evaluated, and the interpreter has been configured with a maximum timeout of 30 seconds. The GA runs for 50 generations using tournament selection, where the tournament size c is 5, and runs 3 times to create the mating pool. Assume the same setup is used to determine which chromosomes are replaced. This configuration assumes that both solutions are not run in parallel. This configuration is further referred to as *Configuration (a)* and utilises the `maxRunningTime` function defined in Equation (6.2).

(b) *Configuration(b)* is similar to *Configuration (a)*, apart from the fact that both of the solutions are executed in parallel. *Configuration (b)* utilises the `maxRunningTimeB` variant of the `maxRunningTime` function with the following definition:

$$\texttt{maxRunningTimeB}(g, t, c, e) = g \times 2 \times t \times c \times e$$

(c) *Configuration(c)* is similar to *Configuration (b)*, apart from the fact that each chromosome in the tournament is evaluated in parallel. *Configuration (c)* utilises the `maxRunningTimeC` variant of the `maxRunningTime` function with

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

the following definition:

$$\texttt{maxRunningTimeC}(g, t, c, e) = g \times 2 \times t \times e$$

(d) *Configuration(d)* is similar to *Configuration (c)*, apart from the fact that the maximum execution time is reduced to 10 seconds instead of 30 seconds. *Configuration (d)* utilises the `maxRunningTimeC` variant of the `maxRunningTime`.

Table 6.3 contains the runtime results of the different configurations listed above.

Configuration	maxRunningTimes	Populated	Seconds	Minutes	Hours
a	maxRunningTimeA	$(50 \times (3 \times 5(30s)) \times 2) \times 2$	90000	1500	25
b	maxRunningTimeB	$(50 \times (3 \times 5(30s)) \times 2)$	45000	750	12.5
c	maxRunningTimeC	$(50 \times (3 \times (30s)) \times 2)$	9000	150	2.5
d	maxRunningTimeC	$(50 \times (3 \times (10s)) \times 2)$	3000	50	0.83

Table 6.3: Theoretical execution time differences for different configurations of GATLAM.

As can be seen in Table 6.3, GATLAM with its most inefficient (*Configuration (a)*) setup takes extremely long to produce a final result. An attempt to decrease this execution time is to parallelise the execution of both of the solutions (*Configuration (b)*). This time consumption can be further reduced by executing the chromosomes in the tournament in parallel (*Configuration (c)*). Lastly, if the maximum execution time is reduced, the overall execution time is further reduced (*Configuration (d)*). Thus, even with all the attempts to reduce the execution times with the configurations mentioned in Table 6.3, it can be concluded that GATLAM does not provide near-instantanious results. This implies that GATLAM cannot be used as an interactive system, but it can still be used to perform automated assessment. For example, the student can submit the assignment and be notified via email once GATLAM has finished evaluating the assignment.

An alternative way to show that GATLAM lacks the speed benefit is by evaluating the time complexity of GATLAM. The worst time complexity (derived from *Configuration (a)*) of GATLAM can be expressed with the following Big-O[8] nota-

[8]Big-O notation is used to express asymptotic complexities of algorithms, in terms of time or memory [12, 29]

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

tion: $O(g,t,c,e) = g \times t \times c \times e$[9], while the best time complexity (derived from *Configuration (d)*) of GATLAM can be expressed as: $O(g,t,c,e) = g \times t \times e$. As can be seen by both the worst and best-case complexities, GATLAM's execution time is completely dependent on the hyperparameters, as well as the iterative nature of GATLAM. It should also be noted that each speedup attempt increases the implementation complexity of both the GA and the interpreter.

2. **Availability**
   This benefit depends on the implementation and computing power that is available to GATLAM. In theory, given enough computing power and internet bandwidth, GATLAM should be able to be available at any time for students to use.

3. **Consistency/Objectivity**
   If consistency is defined as each student's assignment is evaluated with the same input data, GATLAM does not share this benefit. Initially, the population should be identical for each submission received, but as the population evolves and converges, the test input data may differ from submission to submission. It should be noted, however, that assuming the same seed for the random number generator[10] is used, there is no latency between the components ((i.e., time it takes to communicate between the different components)), and if the same unaltered assignment is submitted to GATLAM, GATLAM produces the same result.

   Although GATLAM is in a grey area for consistency, it does exhibit the objectivity benefit. GATLAM evaluates all solutions equally, although with different test input data. The student's race, gender, age, or past grades do not influence how GATLAM evaluates the student.

4. **Immediate feedback**
   This benefit suffers from the same repercussions as the speed benefit. Feedback is delayed until the GA has reached the termination condition.

[9] Due to the asymptotic complexity being taken the constant values in the Big-O notation are omitted due to their insignificance the closer they are to to infinity [29].

[10] Randomness is used to select chromosomes and crossover/mutation points.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

5. **Multiple attempts**
   This benefit suffers from the same repercussions as the availability benefit. Depending on computing power and availability, multiple attempts should in theory be possible.

6. **Can Act As Both Formative and Summative Assessment**
   Theoretically, GATLAM can be used for both formative and/or summative assessment, as a large range of types of assignments can be assessed by GATLAM.

6.3.2 Drawbacks

This section discusses how GATLAM addresses each of the drawbacks of existing strategies listed in Section 4.4.

1. **Lack of Adequate Feedback**
   Instead of a feedback string that may mislead or confuse the student, GATLAM can return the chromosomes that the GA was able to converge on. Thus, GATLAM has the ability to provide the test input data that causes the student's assignment to fail, assuming failure-causing test input data was found. Thus, as the test cases evolve based on the student's submitted assignment, depending on the instructor's discretion, all the test input data that were evaluated can be given back to students. This will hopefully encourage students to debug their own assignments and thus improve their own debugging and testing skills. This is one of the main benefits of GATLAM for students.

2. **Strictness**
   GATLAM suffers from the same strictness drawback as the other dynamic automated strategies. GATLAM may even be stricter than the other dynamic automated strategies, as the expected output is generated during evaluation, and pattern matching cannot be trivially implemented. In other words, GATLAM may uncover bugs and deficiencies in student code that a static memorandum based on pattern-matching may have overlooked.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

3. **Configuration Time and Effort**
   This drawback is extremely pronounced in GATLAM, as the amount of time and effort needed to set up the interpreter and instructor solution may be considerably greater than the time and effort required by other automated assessment strategies, for example, FitchFork. It should be mentioned that if the process of creating an interpreter can be streamlined, the time and effort drawback can be reduced.

6.3.3 Aspects Used to Compare Automated Assessment

This section discusses how GATLAM addresses each of the aspects listed in Section 4.7 under the common aspects and expanded aspects subcategories.

Common Aspects

1. **Automated Assessment Generation**
   If the standard convention is followed as described in Section 4.7, this strategy would form part of Generation 2 or 3, as it can be either utilised as a command line tool or implemented to use a web-based graphical user interface (GUI).

   Two possibilities exist here. If the convention is followed where any technology that has some AI- or data science-driven component is considered part of the so-called Fourth Industrial Revolution [107], GATLAM can then be classified as part of a new automated assessment generation, namely the 4th generation. The 4th generation can thus be classified as any automated assessment strategy that incorporates some form of AI to evaluate or generate test input data for the student's assignment.

2. **Assessment Criteria**
   The assessment criteria for GATLAM are defined by the LTL properties that are activated for the assignment. This allows GATLAM to not only deal with correctness, but with other instructor-defined criteria as well, for example, the program will terminate successfully.

3. **Feedback**
   As mentioned in the *lack of adequate feedback* drawback in Section6.3.2, feedback is

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

in the form of the set of test input data, and as such, the feedback is automatically generated.

4. **Need for Additional Support**
   As the problematic test input data is given back to the student, it would reduce the amount of tutor and instructor support required, as the student is automatically provided with an example of input data that breaks their code.

5. **How Correctness is Determined**
   For the expected output properties, correctness is determined with a verbatim character matching between the memo and the student's output, as done by other dynamic automated assessment strategies. This is due to the dynamic generation of test input data and expected output. For the other LTL properties, the standard error and return code are parsed to determine if the LTL properties were violated. Further investigation could be done into additional techniques that can be used to determine correctness.

Extended Aspects

1. **Input Data Used to Test**
   This aspect is where GATLAM differentiates itself from the other assessment strategies. GATLAM automatically generates the test input data based on the performance of the previous test input data.

2. **Generation of Test Input Data**
   This aspect ties in closely with the above aspect, as GATLAM uses an interpreter to convert a set of values into test input data for the student's assignment. The raw values are generated by the GA.

3. **Language Restrictions**
   GATLAM is distinct in this regard, in the sense that GATLAM is both language[11]-dependent and language-independent at the same time. GATLAM's language independence stems from the GA. As the GA component only produces a set of values

[11]Language in this context refers to programming language.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

that can be converted to test input data, the prescribed language used by the programming assignment is irrelevant. GATLAM's language dependency stems from the interpreter. Due to the fact that an interpreter needs to convert the values from the GA to test input data, the prescribed programming language is essential. Theoretically, GATLAM can be applied to any programming language if and only if an interpreter can be constructed such that the set of input values can be converted to syntactically and semantically correct[12] test input data.

4. **Types of Topics That Can Be Assessed**
   This aspect shares the same constraint as the *language restrictions* aspect, namely that an interpreter that creates valid test input data is needed. This constraint can be formally applied to the *types of topics that can be assessed* aspect as follows: Theoretically, any type of topic can be assessed if and only if there exists an interpreter that can take as input a set of values and convert the input to syntactically and semantically correct test input data.

5. **Correct Output Generation**
   Correct output is generated using an instructor-provided solution that takes as input the same test input data as the student-provided solution. The output from the instructor-provided solution is then assumed to be correct and used as part of the evaluation process.

6.3.4 Additional Concerns

Three concerns need to be noted:

- **Efficiency**:
  It should be noted that during the convergence of the GA, the same chromosome is repeatedly evaluated, due to the recalculation of the fitness value of the chromosome. To re-evaluate the same chromosome over and over again is inefficient, as the result will not change. An easy solution is to cache the chromosome and its

[12]This constraint on the interpreter is to ensure that the test input data does not cause the student's assignment to not compile or run.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

associated fitness in an attempt to speed up execution. A possible problem that may arise is that the evaluation of the solutions may be dependent on the operating system and its memory allocation and thread interleaving. In assignments that are susceptible to the above-mentioned problem, the inefficient option of re-evaluating the chromosome will be needed.

- **Instructor Solution Failure**:
  Another concern that should be noted is what GATLAM should do if the decoded test input causes the instructor-provided solution to fail. One option would be to mark that chromosome as invalid. Another option would be to give the student the benefit of the doubt and mark the output as correct. If a failure in the provided solution is indeed discovered via student submissions, it may be necessary for the instructor to update their solution to remove the error, and subsequently reevaluate all student submissions.

- **Amount of Test Input Data To Return**:
  Adopted from the quote in William Shakespear's Hamlet play, *To give or not to give, that is the question* sums up this concern. How many test input data cases should be returned to the student? If GATLAM gives the student every test input data case, it may flood the student population with test cases and have the opposite effect, by making students lazy and unwilling to independently test their code. If only the final generation's fittest chromosome's interpreted form is given back to a student, other errors may still be present which were not discovered during evolution.

- **What Should Be Returned if the GA Did Not Converge?**
  One possible solution is to place a restriction on GATLAM that if the variety in the population is above a certain threshold at the end of the evolutionary process, the student receives full marks, as GATLAM was not able to converge.

It is the author's recommendation that the effects and possible solutions of these concerns are studied in a longitudinal study for at least five years.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

6.4 Summary

Chapter 6 is the culmination of everything previously discussed, and introduces the proposed automated assessment strategy, GATLAM. The four main components, namely, the GA with temporal logic, the interpreter, and both solutions, were introduced and discussed. The communication between the four components was also addressed. The method used by GATLAM to generate test input data and how it evaluates the correctness of the student assignment was discussed. In order to generate test input data, GATLAM evolves a population of chromosomes by means of the GA component. The evaluation of correctness is done using the fitness function that makes use of the LTL properties. Lastly, the chapter discussed how GATLAM addresses the benefits, drawbacks, aspects used to compare automated assessments, and concerns that were laid out in Chapter 4.

From this discussion, it was theoretically concluded that GATLAM does not follow speed benefits as shared by other automated assessment strategies. The trade-off for the lack of speed is an improvement in the feedback. The feedback from GATLAM is different from other automated assessment strategies in the sense that, instead of an explanation for why the submission is incorrect, GATLAM can give an example of test input data that caused the submission to violate correctness. The next chapter, Chapter 7, discusses the experiments that were performed to empirically evaluate GATLAM under different assignment setups.

Chapter 7

GATLAM Experimental Setup

This chapter discusses the experiments that were performed to evaluate GATLAM. These experiments are used to discuss GATLAM's viability as a dynamic automated assessment strategy. Each experiment set was repeated 10 times with different random seed values to obtain statistically significant results. This resulted in a large set of experiments being conducted, which included investigating the influence of the GA-specific hyperparameters, the effect of ω_1, ω_2, and ω_3 weights (as discussed in Chapter 6), comparing GATLAM to different automated software test generation methods, and evaluating GATLAM's performance on a real-world assignment that was completed by first-year students in 2022. It is important to note that the experiments were conducted offline, after the deadline, using the volunteered student submissions.

The remainder of the chapter is laid out as follows: Section 7.1 discusses four error pools which were used to tune GATLAM's hyperparameters and compare GATLAM to different automated software test generation methods. Sections 7.2, 7.3, and 7.4 discuss the hyperparameter tuning experiments, alternative software test generation experiments, and experiments with real-world volunteered student submissions, respectively. Lastly, Section 7.5 discusses the summary of this chapter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.1 Error Pools

In order to evaluate the effects of different hyperparameters, a set of four pools of errors were created[1]. Each pool consists of a series of programs which contain a single artificially injected error. The programs were created in either Java or C++, based on the type of error that they contain. The errors can also be configured, using a text file which contained the number or numbers on a new line, to increase or decrease the "difficulty" of finding the error. The process of configuring each error is discussed in greater detail later in this section. These pools were also used as a benchmark to compare GATLAM to alternative software test generation methods. The pools were created by taking inspiration from Kaczorowska [67], in which common programming errors or mistakes made by first-year programming students are discussed and grouped. Four distinct groups were created and are discussed , in alphabetical order, as follows: Section 7.1.1 discusses concurrency errors, Section 7.1.2 discusses infinite loop errors, Section 7.1.3 discusses logic errors, and lastly Section 7.1.4 discusses memory-related errors.

7.1.1 Concurrency Errors

The concurrency error pool comprises three programs with artificially injected errors, which are discussed below.

Concurrency Error 1

The program demonstrating concurrency error 1 contains a deadlock error. The program receives a command line argument that causes the program to run into a deadlock state where no progress can ever be made. Deadlock is defined as the state where $n > 1$, processes are waiting for one or more resources that the processes are never able to acquire [54].

[1]The code for the error pools can be found at https://github.com/Cobus-Redelinghuys/FJ-Redelinghuys-MSc-Genetic-Algorithm-with-Temporal-Logic-for-Automated-Assessment/tree/Error-Pools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Concurrency Error 2

The program demonstrating concurrency error 2 contains a race condition error. A race condition is observed when the synchronisation of access to a memory resource is not correctly implemented [82]. The program receives a command line argument that could cause the program to run into a race condition where no progress can ever be made. This program is configured such that it does not always fail for a range of input values, and then always fails for another range. This is done to see how GATLAM handles errors that are not solely based on input values but rather on process interleaving.

Concurrency Error 3

The program demonstrating concurrency error 3 program contains a starvation error. Starvation is defined as the program state where a process that is not in deadlock waits for a resource that it is never able to obtain [103]. This program receives a command line argument, and based on this value and the value in the configuration file, the program could cause a starvation situation where no progress can ever be made.

7.1.2 Infinite Loop Errors

The infinite loop error pool comprises three programs with artificially injected errors discussed below.

Infinite Loop Error 1

The program demonstrating infinite loop error 1 contains an infinite loop which is illustrated in Algorithm 7.1. The program reads in a value from a configuration file as well as a value that is passed as a command line argument. If `n` is smaller or equal to `v`, then the program terminates. Otherwise, if `n` is larger than `v`, the program will run into an infinite loop.

Infinite Loop Error 2

The program demonstrating infinite loop error 2 contains an infinite recursive loop error, which will terminate with a stack overflow error which is described in Algorithm 7.2. The

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
public static void main(String[] args) {
    int n = readFile(args[1]);
    int v = Integer.parseInt(args[0]);

    while(n > v){}

    System.out.println("Finished␣Loop");

}
```

Algorithm 7.1: Infinite Loop Error 1's code.

program reads a value from a configuration file as well as a value that is passed in as a command line argument. If `n` is greater than or equal to `v`, then the program will

```
public static void func(int n, int v){
    if(n < v){
        func(n-1, v);
    }
}

public static void main(String[] args) {
    int n = readFile(args[1]);
    int v = Integer.parseInt(args[0]);
    func(n,v);
}
```

Algorithm 7.2: Infinite Loop Error 2's code.

terminate without any recursive calls. Otherwise, if `n` is smaller than `v`, the program executes until `n` negative overflows[2] to a value that is bigger than `v`. This causes the program to run for an extremely long time, which is unwanted and may result in a stack overflow error.

Infinite Loop Error 3

The program demonstrating infinite loop error 3 contains an incorrect scope-based infinite loop which is described in Algorithm 7.3. Note that the commented brackets have

[2]The variable `n` will negative overflow as `n` will surpass the greatest negative number that can be stored as an int and "wrap-around" to the biggest positive number.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

been added to aid the readability of the code by indicating what the student mistakenly thought was included in the scope of the while loop. The program reads a value that is

```
public static void main(String[] args) {
    int v = Integer.parseInt(args[0]);
    int b = 5;
    int count = 0;
    while(v != b)\\{
        count++;

    if(v > b)
        v--;

    if(v < b)
        v++;
    \\}
    System.out.println(count);
}
```

Algorithm 7.3: Infinite Loop Error 3's code.

passed in as a command-line argument. Due to the incorrect scope of the two if statements, if `v` and `b` are not equal, the code will execute indefinitely, as `v` is never modified in the while loop.

7.1.3 Logic Errors

The logic error pool comprises five programs with artificially injected errors as discussed below.

Logic Error 1

The program demonstrating logic error 1 is described by the code in Algorithm 7.4. This program reads a value from a configuration file and a value from a command line argument. If `n` is equal to `v` then the object `obj` won't be initialised and an error occurs.

Logic Error 2

The program demonstrating logic error 2 is described by the code in Algorithm 7.5. This program reads two values from a configuration file and a value that is passed in as a

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
class Number{
    public:
        virtual void print() = 0;
};

class GreaterThan: public Number{
    public:
        void print(){
            cout << "GreaterThan" << endl;
        }
};

class LessThan: public Number{
    public:
        void print(){
            cout << "LessThan" << endl;
        }
};

int main(int argc, char *argv[]){
    int n = readConstant(argv[2]);
    int v = stoi(argv[1]);
    Number* obj;
    if(n > v){
        obj = new GreaterThan();
    }
    if(n < v){
        obj = new LessThan();
    }
    obj->print();
    delete obj;
    return 0;
}
```

Algorithm 7.4: Logic Error 1's code.

command line argument. The variable *obj* is not initialised if `v` is equal to `max` or if `v` is equal to `min`, and an error occurs.

Logic Error 3

The program demonstrating logic error 3 is described with the code in Algorithm 7.6. The program reads two values from a configuration file, as well as a value that is passed in as a command-line argument. From the code , *obj* is not initialised if `v` is less than or

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
class Position{
    public:
        virtual void print() = 0;
};

class Inbound: public Position{
    public:
        void print(){
            cout << "Inbetween boundaries" << endl;
        }
};

class Outbound: public Position{
    public:
        void print(){
            cout << "Not between boundaries" << endl;
        }
};

int main(int argc, char *argv[]){
    Pair<int, int> bounds = readConstant(argv[2]);
    int v = stoi(argv[1]);
    Position* obj;
    if(bounds.max < v && bounds.min > v){
        obj = new Inbound();
    }
    if(bounds.max > v || bounds.min < v){
        obj = new Outbound();
    }
    obj->print();
    delete obj;
    return 0;
}
```

Algorithm 7.5: Logic Error 2's code.

equal to *min*, or if v is greater than or equal to *max*. If *obj* is not initialised, an error occurs.

Logic Error 4

The program demonstrating logic error 4 contains a conversion error, and is shown in Algorithm 7.7. A bit string value is passed to the program as a command-line argument. The program then attempts to convert the bit string to a float number. If the bit string

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
class Position{
    public:
        virtual void print() = 0;
};

class Inbound: public Position{
    public:
        void print(){
            cout << "Inbetween boundaries" << endl;
        }
};

int main(int argc, char *argv[]){
    Pair<int, int> bounds = readConstant(argv[2]);
    int v = stoi(argv[1]);
    Position* obj;
    if(v > bounds.min && v < bounds.max){
        obj = new Inbound();
    }
    obj->print();
    delete obj;
    return 0;
}
```

Algorithm 7.6: Logic Error 3's code

is of invalid size or format, the program will throw an error. Although this is not a logic error, the assumption that any passed in value can be converted is.

```
public class Main{
    public static void main(String[] args) {
        String binaryString = args[0];
        Long v = Long.parseLong(binaryString, 2);
        System.out.println(v.intValue());
    }
}
```

Algorithm 7.7: Logic Error 4's code.

Logic Error 5

The program demonstrating logic error 5 contains a division by zero error or unchecked division error. The logic error is described with the code in Algorithm 7.8. The program

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

reads a value from a configuration file as well as a value that is passed in as a command-line argument. As can be seen , if `v` is 0, a division by zero error will occur in the

```
public static void main(String[] args) {
    Integer n = readFile(args[1]);
    int v = Integer.parseInt(args[0]);
    System.out.println(n/v);
}
```

Algorithm 7.8: Logic Error 5's code.

program. Although this error is an arithmetic error, the omission of an if statement to ensure that `v` is not 0 is an logical error.

7.1.4 Memory Errors

The memory error pool comprises two programs with artificially injected errors, which are discussed below.

Memory Error 1

The program demonstrating memory error 1 contains a memory-based error. An array with a size obtained from a configuration file is created and populated with random values. An index is then passed in as a command-line argument and printed out. This can cause an error, as the passed-in index may be out of the array's range.

Memory Error 2

The program demonstrating memory error 1 contains an incorrect parameter type error where the programmer mistakenly passes the input by reference[3] to one function and correctly passes the input by value[4] to another function. Both functions initialise and populate the object with certain values. The code described in Algorithm 7.9 contains

[3]Pass by reference is when the memory location of the original input parameter is passed to the function as a parameter [63].

[4]Pass by value is when a copy of the original input parameter is passed to the function as a parameter, and not the original input value [63].

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

the described memory error. If the `greater` function is called, the input parameter's

```
struct Object{
    bool leq;
    Object* other;
};

void less(Object& obj){//correct
    obj.other = new Object;
    obj.other->leq = true;
}

void greater(Object obj){//error-passed by value
    obj.other = new Object;
    obj.other->leq = true;
}

int main(int argc, char const *argv[])
{
    int v = std::stoi(argv[1]);
    Object obj;
    obj.other = NULL;
    if(v > 5){
        greater(obj);
    } else {
        less(obj);
    }
    std::cout << (obj.other->leq ? "less" : "greater") << std::endl;
    return 0;
}
```

Algorithm 7.9: Memory Error 2's code.

`other` member variable will be initialised. The problem is that due to the input variable being passed by value, the modification performed inside the `greater` function will not be reflected outside the function. Thus, during the printout statement, where member variables of an uninitialised member variable are called, an error will occur. This is a memory error, as the incorrect type of parameter passing is used, causing the variable at the incorrect memory location to be updated.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.2 Hyperparameter Tuning Experiments

In order to evaluate the sensitivity of GATLAM to different hyperparameters, a three-stage experimental methodology was followed, which is illustrated in Figure 7.1. This three-stage experimental methodology was followed in an attempt to reduce the number of possible hyperparameter combinations and evaluate each grouping of hyperparameters independently. It is the author's recommendation that an extensive grid search be performed such that all possible hyperparameter combinations are evaluated.

The first stage of the hyperparameter tuning experiments consists of running GATLAM on the different error pools (as discussed in Section 7.1) for a large number of generations. These experiments aim to determine the optimal number of generations GATLAM should execute to maximise exploration in the minimal amount of time. The precise details of these experiments are discussed in Section 7.2.1.

Once the optimal number of generations is found, the second stage of hyperparameter experimentation occurs, which focuses on the GA-specific hyperparameters. The precise details of this stage of experimentation are discussed in Section 7.2.2. The purpose of the GA-specific hyperparameter experiments was to determine how sensitive GATLAM is to GA-specific hyperparameters using the error pools discussed. After the GA-specific hyperparameter experiments were concluded, the average best hyperparameter combination across all the error pools was chosen for the next set of experiments.

The last stage of experiments in the hyperparameter experimentation portion of the dissertation focuses on the three ω weights that form part of the fitness function introduced in Chapter 6. The details of the experiments are discussed in Section 7.2.3. The goal of the ω weights experiments was to determine the effects of having different ω weight values on the final result of GATLAM.

All of the experiments in the hyperparameter tuning section of this dissertation used GATLAM's SS mode, as discussed in Section 6.2.5. The reason for this is that not all of the error programs in the error pools contain output statements, which would result in the instructor solution returning empty output as correct output.

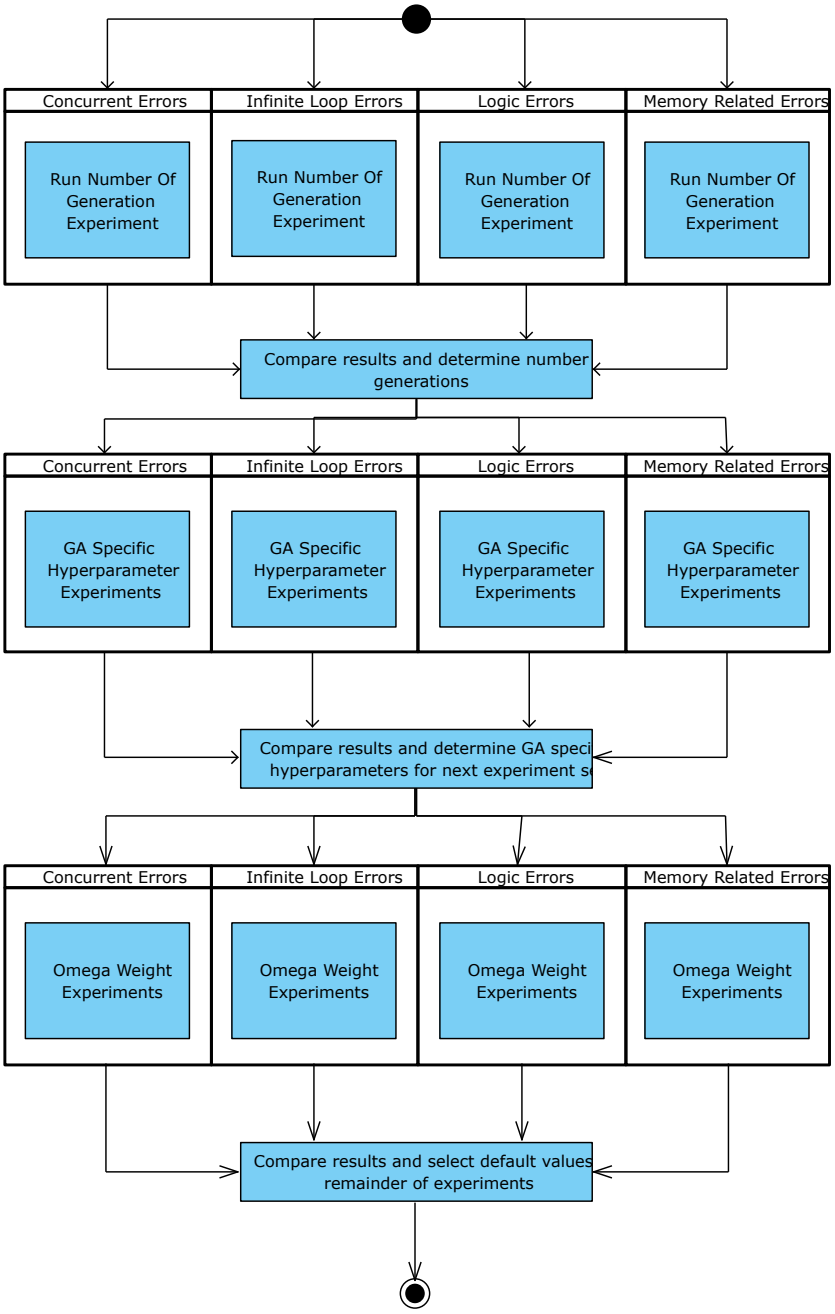


Figure 7.1: Diagram illustrating how the hyperarameter tuning experiments will be conducted.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

7.2.1 Number of Generations Experiments

The number of generation experiments is conducted using default hyperparameter settings for all of the other GA-specific hyperparameters. Table 7.1 lists the default values for the hyperparameters. For the ω weights, each weight was assigned a value of $\frac{1}{3}$ such that the ω weights sum up to 1. For each of the error pools, GATLAM was run for 1000 generations to determine at which generation GATLAM was able to converge in terms of fitness value while still being able to find the majority of errors. The baseline values for the respective probabilities were chosen to be 50% as this is the middle value in the range of possible values. Tournament selection was chosen since it does not evaluate the entire population and, as such, should yield a quicker result. A tournament size of 2 was chosen using the same reasoning as the selection method. The genetic operators were chosen due to their simplicity in implementation and operation. The results of the number of generation experiments are discussed in Section 8.2.

Parameter Name	**Hyperparameter Value**
Reproduction probability	0.5
Crossover probability	0.5
Mutation type	Inversion of single bit
Population size	50
Crossover type	One point crossover
Mutation probability	0.5
Selection type	Tournament selection
Tournament size	2

Table 7.1: Default GA parameter values for the number of generation experiments.

7.2.2 GA-specific Experiments

In this section, the GA-specific hyperparameter experiments are discussed. Table 7.2 lists the hyperparameters, values that were evaluated, and the default or baseline values that were used. For each experiment, a single value in the list of possible values for a single hyperparameter was chosen. The remaining hyperparameters were assigned the baseline values. The same configuration was then executed on each of the error pools discussed in Section 7.1.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Parameter Name	Values Tried	Baseline Values
Reproduction probability	0, 0.25, 0.5, 0.75, 1	0.5
Crossover probability	0, 0.25, 0.5, 0.75, 1	0.5
Mutation type	Inversion of single bit, Random selection, Bitwise-inversion	Inversion of single bit
Population size	25, 50, 75, 100, 125, 150	50
Crossover type	N-point crossover, Segmented crossover, Uniform crossover One point crossover	One point crossover
Mutation probability	0, 0.25, 0.5, 0.75, 1	0.5
Selection type	Roulette Wheel, Stochastic Universal Sampling, Exponential Rank Selection, Truncation Selection Tournament Size	Tournament selection
Tournament size	1, 2, 5, 10	2

Table 7.2: Parameter names, the values/types for each parameter that was tried, and the default values/type for each iteration of the GA-specific hyperparameter tuning.

Using the results obtained from the number of generation experiments, discussed in Section 7.2.1, each of the hyperparameter values in Table 7.2 was evaluated in their ability to guide GATLAM to find the maximum number of errors and fitness values in the least amount of execution time. Again, each of the ω weights was assigned a value of $\frac{1}{3}$. A gird search was not performed for the GA specific hyperparameters as the GA's hyperparameters were not the main focus of the dissertation, but in future work, this can be investigated in more depth.

7.2.3 Omega (ω) Weights Experiments

The final set of hyperparameter tuning experiments consisted of evaluating the ω weights that are present in the fitness function, as introduced in Chapter 6. The weights control the contribution of the different components that comprise the fitness function, and ensure that the fitness function is bounded between 0 and 1.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

The ω-weight experiments were conducted using a grid search. Each of the combinations of ω-weights evaluated is shown in Table 7.3.

Experiment Set	ω_1	ω_2	ω_3
1	0.0	0.0	1.0
2	0.0	0.25	0.75
3	0.0	0.5	0.5
4	0.0	0.75	0.25
5	0.0	1.0	0.0
6	0.25	0.0	0.75
7	0.25	0.25	0.5
8	0.25	0.5	0.25
9	0.25	0.75	0.0
10	0.5	0.0	0.5
11	0.5	0.25	0.25
12	0.5	0.5	0.0
13	0.75	0.0	0.25
14	0.75	0.25	0.0
15	1.0	0.0	0.0

Table 7.3: Grid search table for the ω-weight experiments.

The GA hyperparameters were determined by the experiments discussed in Section 7.2.2. As with the previous set of experiments, all of the proposed combinations of ω weights were run on the different error pools discussed. Each of the combinations of ω weights listed in Table 7.3 was evaluated in their ability to guide GATLAM to find the maximum number of errors and fitness values in the least amount of execution time, using the execution time recorded by the interpreter. The best and worst performing hyperparameter setting across all of the metrics , was used for the remaining experiments.

7.3 Benchmarking Automated Software Testing Tools Experiments

This section describes experiments conducted to compare GATLAM to different software test generation methods, namely a Random Number Generator (RNG) and a code

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

coverage-based GA (CCGA), as these methods evaluate the solutions in a similar method to GATLAM. It is the author's recommendation that in future work, an extensive comparison against other software testing methods is performed. The aim was to evaluate how GATLAM compares to competitive methods that utilise alternative approaches to generate or evaluate the chromosome. This section is laid out as follows: Section 7.3.1 discusses the RNG, Section 7.3.2 discusses CCGA, and lastly, Section 7.3.3 discusses the experimentation in more detail.

7.3.1 RNG (Fuzzing)

This section discusses one of the two main "competitors" of GATLAM, , namely RNG (which has been introduced in Chapter 3).

GATLAM produces a chromosome which is represented as a set of values, as described in Chapter 6. RNG also has the ability to generate a set of (random) numbers that can easily be configured with the same restrictions and format as GATLAM. It should be noted, however, that RNG is random, and no intelligent searching happens. This set of random numbers is passed to the interpreter in the same fashion as GATLAM passes encoded chromosomes to the interpreter. The interpreter then passes the decoded test data to the system under test (SUT), which retrieves the output and passes it back to the RNG.

7.3.2 CCGA

Path coverage, or code coverage is a popular approach to quantify the goodness of the tests that the GA produces. In this dissertation, for the CCGA, the number of lines executed is used as the coverage metric.

The CCGA utilises two tools depending on the active error program. If the SUT is written in Java, Jacoco[5] is used, and if it is written in C++, gcov[6] is used. Using the results of each tool, the average coverage percentage of each program is used as the

[5]Jacoco is a code coverage tool developed for the Java programming language that computes a range of software test metrics [59].

[6]gcov is a code coverage tool for the C/C++ programming languages that computes a range of software test metrics [45].

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

fitness value. Thus, the higher the average coverage, the higher the fitness value of the chromosome. Apart from a difference in the fitness function, the CCGA interacts with the different components in an identical manner to GATLAM. This proposed competitor for GATLAM is a combination of the methods proposed by Hermadi et al. [51] and Mishra et al. [88]. Equations (7.1) and (7.2) are used to calculate the fitness of a chromosome.

$$fc_i(x) = \begin{cases} cc_i(x) & \text{if code coverage is available} \\ 0 & otherwise \end{cases} \tag{7.1}$$

In Equation (7.1), $cc_i(x)$ is the code coverage result for SUT i using chromosome x. The reason for including the possibility that code coverage could not be calculated is due to the requirement of gcov for the program to successfully terminate for the code coverage to be calculated. Thus, if the C++ program encounters an error, such as a segfault, gcov cannot calculate the number of lines that were executed. If the code coverage cannot be calculated, it is assumed that the program did not run, and the value of 0 is returned.

Equation (7.2) returns the average code coverage across all of the currently active error programs. M represents the number of active error programs, and x is the chromosome currently being evaluated.

$$f(x) = \frac{\sum_{i=1}^{M} fc_i(x)}{M} \tag{7.2}$$

7.3.3 Comparison Experiments

The experiments to compare GATLAM to the alternative software test generation tools consisted of running each of the alternative tools on each of the error pools listed in Section 7.1 ten times with different random seed values. Based on the experiments described in Section 7.2, GATLAM with a default, average best, and average worst configuration were also run on each of the error pools ten times. The default configuration utilises the same default hyperparameters determined as discussed in Section 7.2.2. Using the number of errors detected metric, the different tools and GATLAM configuration are compared. GATLAM was configured to run in the SS mode, because RNG and the CCGA do not account for whether the output generated by the program is correct or

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

not. This was only done for the experiments that utilised the error pools, as no model solution for the error pools was created. By using SS mode, GATLAM's performance could also be evaluated for the situation where no model solution is available. CCGA used the same GA configuration as GATLAM's default configuration, and RNG evaluated 25 chromosomes in each generation of execution. 25 was selected as this is half of GATLAM's default configuration's population size, which will provide enough chromosomes to be evaluated to determine if GATLAM's fitness function can be used as a stand-alone function to evaluate the quality of abstract test input data.

7.4 Student Assignment Experiments

Although the error pools, discussed in Section 7.1, have their advantages in the sense that the number, type and location of the error are known, the same cannot be said for real-world programs. The experiments discussed in this section put GATLAM and its competitors to a real-world test.

At the University of Pretoria, the Computer Science department presents a programming module, COS110[7]. This module is aimed towards first-year students studying the following degrees: BSc[8] Computer Science, BIS[9] Multimedia, BEng[10] Computer Engineering, and some applied mathematics programs. COS110, as of March 2025, still utilises the automated marking strategy FitchFork (discussed in Chapter 5) to assess a set of weekly practicals and monthly assignments. The aim of these practicals and assignments is to demonstrate the concepts that were discussed during lectures.

In 2022, one of the practicals presented to the student body aimed to familiarise students with the Object Orientated Programming (OOP) concept of polymorphism. Polymorphism is a programming concept that allows objects of a derived class, within an inheritance hierarchy, to be treated as if the object is of the parent class' type, enabling the same interface or method to seamlessly operate on different types of objects

[7] Computer Science 110: Program Design: Introduction

[8] Bachelor of Science

[9] Bachelor of Information Science

[10] Bachelor of Engineering

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[79]. The practical entailed building a polymorphic number tester that had the ability to perform a set of tests on numbers. The formal specification for this practical can be found in Appendix B.

The real-world assignment experiments consisted of using GATLAM and its competitors to create test input data for 52 student submissions. GATLAM was configured with the worst-, default- and best-case configurations to evaluate the effect of different configurations. GATLAM and its configurations were compared to RNG and CCGA on their ability to find errors which include abnormal exits (segmentation faults or exceptions), infinite loops, or faulty output. A custom-built interpreter was used to translate the encoded chromosomes produced by GATLAM, RNG and the CCGA to test input data. In these experiments, GATLAM was configured in ISS mode, because the original assessment was designed to be graded on output alone.

The student solutions were obtained on a volunteer basis, where students were given the opportunity to volunteer their submitted practicals. For a student to be allowed to volunteer, their practical they needed to have obtained marks on FitchFork when the practical was used as a formative assessment in 2022. The practicals are stored anonymously, thus protecting the student's private information. In an attempt to reduce the maximum execution time of the solutions, the interpreter was configured to assume five seconds as the limit for an infinite loop. This was a safe assumption, as the practical was relatively simple and did not contain any complex algorithms. The solution, which was originally used to create the memorandum output in 2022, is used as the instructor solution.

During the analysis of the student assignment experiments, if the instructor's solution encountered an error while the student's solution did not, the output generated by the instructor's solution was compared to that of the student's solution to determine whether any output errors were present. It is, however, the author's recommendation that alternative methods to deal with such instances be investigated further.

The experiments were conducted as follows. For each marker (GATLAM with its different configurations, CCGA, and RNG), each student was evaluated once, and the results were compared. A selection of five interesting cases was then selected and re-evaluated 10 times with different random seeds to investigate the influence of the random

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

number seed on the different markers. As random number seed is a core feature for each of the automated assessment strategies, it is important to evaluate how big of a role the random seed plays. Having said this, it is important to note that during assessment all assignments are evaluated using the same hyperparameters (including the random seed), such that results are reproducible and consistent.

7.5 Summary

Chapter 7 discussed the experimental setup for this dissertation. The chapter started by introducing the different types of error pools, as well as each error-injected program in each of the pools. These programs contain artificially injected errors, and can thus be used to accurately measure the quality of the software tests produced. The chapter continued by explaining the three-stage process that was used to evaluate GATLAM, and select hyperparameter values used for the remainder of the experiments. The chapter also discussed how GATLAM was compared to other alternative automated software testing tools. These tools consisted of a random number generator and a code-coverage-based GA, which acts as the main competitor for GATLAM. Lastly, the chapter discussed the experiments that utilised a coding assessment completed by first-year university students, and how GATLAM was evaluated using this assessment. Chapter 8 discusses the GA-specific hyperparameter tuning experimental results, Chapter 9 discusses the ω-weight experiments, and Chapter 10 discusses the comparative experiments. The implementation files can be found at `https://github.com/Cobus-Redelinghuys/FJ-Redelinghuys-MSc-Genetic-Algorithm-with-Temporal-Logic-for-Automated-Assessment`.

Chapter 8

Hyperparameter Sensitivity Results

This chapter discusses the results of the first set of experiments, which aimed to investigate the influence of different hyperparameter values on GATLAM's performance, as discussed in Chapter 7. This chapter provides selected results of particular interest. The full set of experimental results can be found in Appendix C. The tables presented in this chapter, along with those in Appendix C, employ a blue-white-red colour scheme, with each table individually colour-scaled to display the performance of GATLAM configured with different hyperparameter values for a certain metric (discussed later in this chapter). Cells shaded in blue represent superior values, while those in red indicate inferior values; white signifies a midpoint between these extremes. Additionally, the optimal value in each column is highlighted in bold to facilitate identification. The remainder of this chapter is laid out as follows: Section 8.1 discusses the different metrics used to evaluate the results. Section 8.2 discusses the results of fine-tuning the number of generations that GATLAM is allowed to execute for. Section 8.3 discusses the results of the GA-specific hyperparameter experiments. Lastly, Section 8.4 contains a summary of the chapter.

8.1 Metrics

In this chapter, different metrics are used to compare different configurations of GATLAM. The remainder of the section is laid out as follows: Section 8.1.1 discusses the aver-

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

age fitness metric, Section 8.1.2 discusses the best fitness metric, Section 8.1.3 discusses the diversity in the population metric, and lastly Section 8.1.4 discusses the percentage of errors found. It should be noted that the results shown and discussed in this chapter are the average of the metric, over the multiple runs performed. The average and best fitness as well as the population diversity are calculated by GATLAM during execution, and the number of errors is calculated *post factum* by a specialised script which inspects the results produced by GATLAM.

8.1.1 Average Fitness

The average fitness metric calculates the average fitness value across all of the chromosomes evaluated per generation. Ideally, this metric should increase towards 1 over the generations before reaching a plateau. Furthermore, ideally, the standard deviation of the average fitness value per generation across multiple runs should be relatively small, which speaks towards the reliability of GATLAM to reproduce similar results.

8.1.2 Best Fitness

The best fitness metric records the fitness value of the best chromosome for each generation. Ideally, this metric should mimic the performance of the average fitness metric in terms of an increase in fitness value. Additionally, the best fitness value should not decrease per generation, and if it does decrease, it should return to its previous fitness value if not better as the evolution progresses.

8.1.3 Population Diversity

The population diversity metric looks at GATLAM's ability to converge onto a relatively small diversity in the population. The diversity of the population is calculated by dividing the number of unique chromosomes by the population size. The value is then min-max scaled using Equation (8.1):

$$scale(x, x_{min}, x_{max}) = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{8.1}$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

where:

- x is the diversity score.
- x_{min} is the minimum diversity score.
- x_{max} is the maximum diversity score.

Equation (8.1) is used such that the diversity lies between 0 and 1, where 0 is the minimum diversity, and 1 is the maximum diversity. Ideally, the population diversity should decrease, and reach a low, stable diversity value. However, the diversity should not converge too quickly, as this might lead to GATLAM getting stuck at a local minimum, and making no further progress. The opposite is also true: The population should not converge too slowly, as this implies that GATLAM is struggling to consistently find chromosomes which violate the LTL properties. A good balance between the different convergence rates is desired.

8.1.4 Percentage of Errors Found

This metric determines how effectively GATLAM is at finding the artificially injected errors in each of the error pools. An analysis of the results produced by the best chromosome in each generation is performed using a specifically designed program to determine how many of the errors were found. This metric is calculated by dividing the number of errors found by the total number of errors present. Ideally, this percentage should increase over the generations as GATLAM discovers more errors in the programs. GATLAM is not able to calculate this metric at runtime, as that would imply GATLAM knows how many errors there are in the system, which is not reflective of a real-world scenario.

Having both the average and best fitness values, as well as the percentage of errors found, means we can evaluate how well the proposed fitness function reflects the true performance of GATLAM. A strong correlation between the fitness value and the number of errors found will indicate that the proposed fitness function captures useful information that can effectively guide the search for errors.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

8.2 Number of Generations Experiment Results

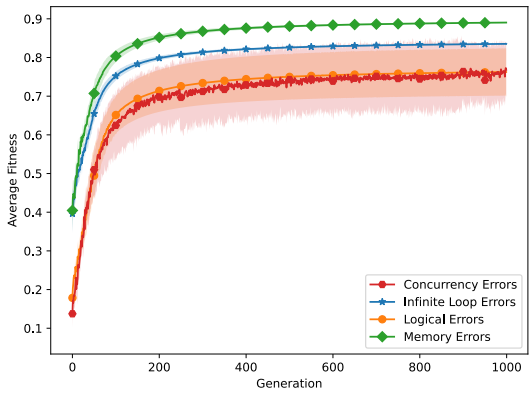
In this section, the experiments relating to the number of generations needed by GATLAM to achieve a stable fitness value are discussed. Figure 8.1 contains the results of each of the metrics discussed in Section 8.1 for all four error pools.

Based on the results in Figure 8.1 (d), it can be seen that for all 1000 generations, GATLAM with a default configuration was able to find all of the infinite loops and memory errors consistently. The maximum percentage of logic errors was found before generation 200, after which the maximum was consistently found. GATLAM was able to also find the maximum percentage of concurrency errors before generation 200, but after generation 400, the average percentage of errors found started to fluctuate.

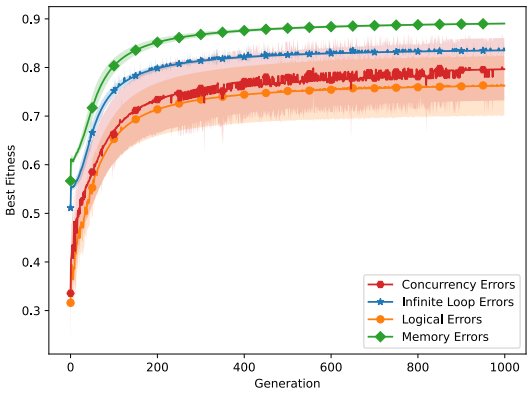
Looking at GATLAM's fitness function performance in Figure 8.1 (a), it can be seen that the average fitness per generation for each error pool increased rapidly before starting to plateau between generations 100 and 200. In both the infinite loop and memory error pools, GATLAM was able to consistently achieve the same average performance across all chromosomes evaluated for that generation. This is due to GATLAM finding all of the errors in the error pool from the initial generation. The reason GATLAM's average and best fitness (Figure 8.1 (b)) per generation continues to increase is due to the memory component of the fitness function increasing in value, as the same genes are constantly present in chromosomes where the SUT violated an LTL property.

Lastly, when comparing the diversity in the population metric shown in Figure 8.1 (c), it is clear that for every error pool, GATLAM converged onto the lowest average diversity by generation 300. For the remainder of the generations, GATLAM's population was constantly at a low level of population diversity. This is directly attributed to both the average fitness values of the population plateauing as well as no new errors being discovered after generation 200. For the concurrency error pool, GATLAM took longer to converge, which is also reflected in the fluctuations that were seen in the average percentage of errors found for this error pool. This can be attributed to the thread interleaving performed by the operating system and the current state of the operating system.

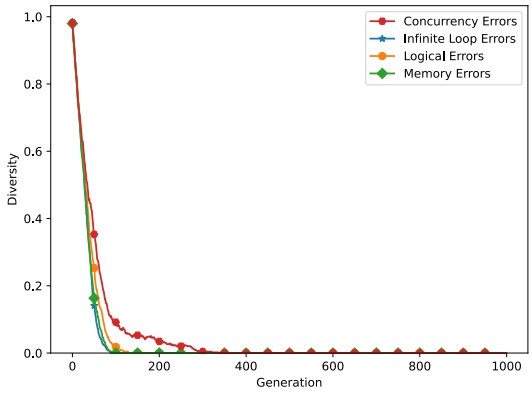
Based on the results, the remaining hyperparameter and fitness function experiments were executed for 350 generations. While GATLAM with the default configuration was able to converge before 300 generations, 350 was chosen as a safety measure to allow other hyperparameter configurations enough time to converge.



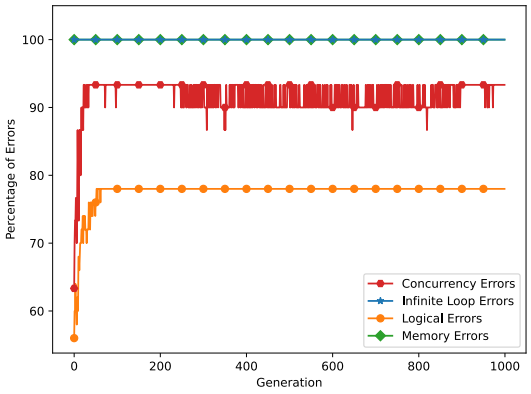
(a) Average Fitness



(b) Best Fitness



(c) Population Diversity



(d) Percentage of Errors Found

Figure 8.1: Experimental results for the number of generations.

8.3 GA Specific Hyperparameter Experiment Results

This section discusses the results pertaining to the GA-specific experiments, presented in alphabetical order. Tabulated results can be found in Appendix C.1.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

8.3.1 Crossover Probability

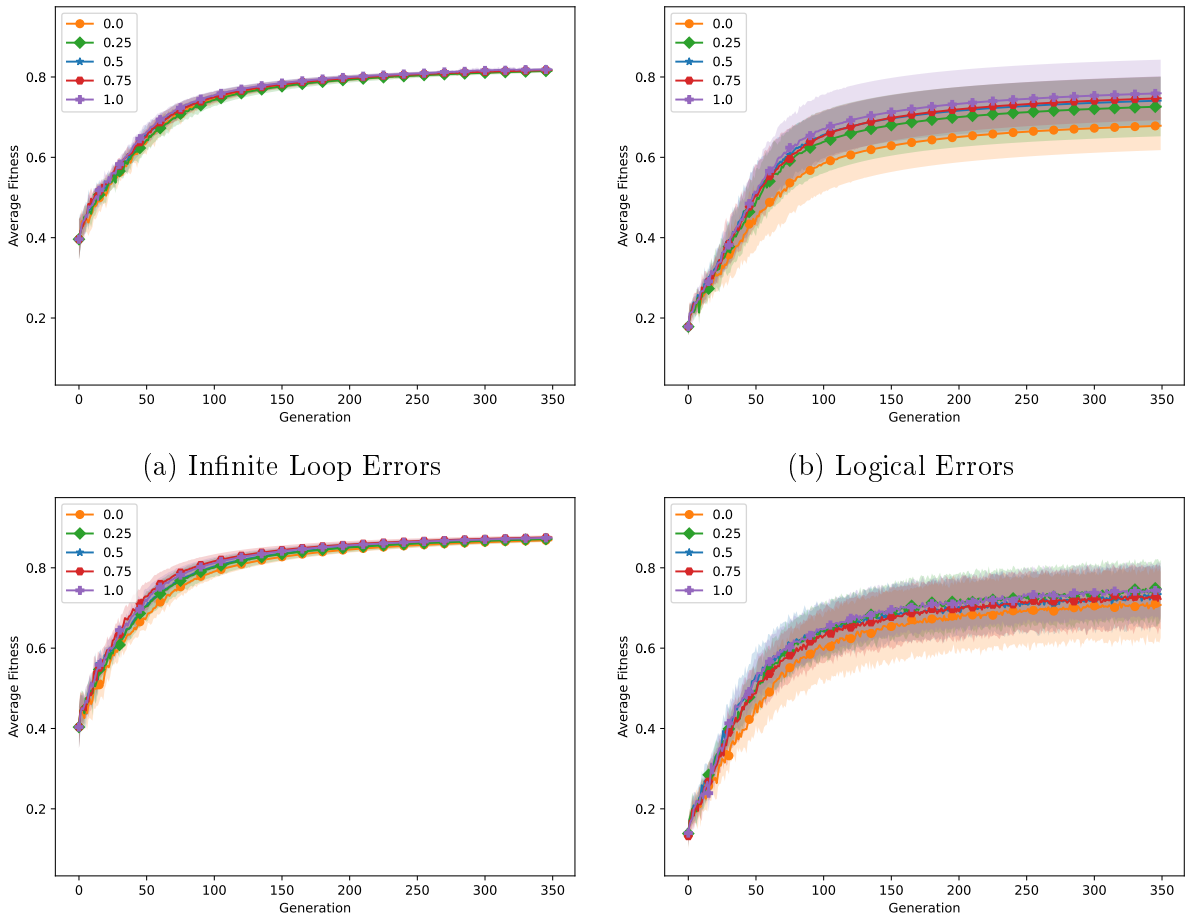
The crossover probabilities that were evaluated were 0%, 25%, 50%, 75%, and 100%, with 50% being the default parameter.

Evaluating the average and best fitness (Figures 8.2 and 8.3), it is observed that for both the infinite loop and memory error pools, all of the crossover probabilities enabled GATLAM to converge onto a relatively similar fitness value. This is due to GATLAM discovering all errors in both the infinite loop and memory error pools from the first generation, which can be attributed to the trivial nature of the errors in these pools.. In contrast, GATLAM for the logic and concurrency error pools showed a small but clear difference in the average fitness and best fitness metrics for the different crossover probability values. A crossover probability of 0% yielded the lowest performance due to no information exchanged between the chromosomes. A crossover probability of 100% was able to achieve the highest average and final fitness values.

For all of the error pools, GATLAM was able to converge onto the same final population diversity of 0 regardless of the crossover probability, as seen in Figure 8.4. A crossover probability of 0% reduced the diversity the quickest. A crossover probability of 100% caused GATLAM to converge the slowest, but convergence was still achieved onto a population diversity of 0 before 150 generations for all error pools.

Evaluating the percentage of errors found (Figure 8.5), GATLAM consistently found all of the errors in each generation for both the infinite loop and the memory error pools. For the logical error pool, a crossover probability of 100% yielded the highest percentage of errors, and a crossover probability of 0% yielded the lowest number of errors. In contrast, for the concurrency error pool, a crossover probability of 25% yielded the highest number of errors, but only for a few generations before converging on the same number of errors found as a crossover probability of 100%. As expected, a crossover probability of 0% yielded the lowest number of errors.

Based on the discussion, the best crossover probability was set to 100%, and the worst crossover probability was set to 0%. The overall percentage of errors found exhibited a strong correlation with the proposed fitness function.



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

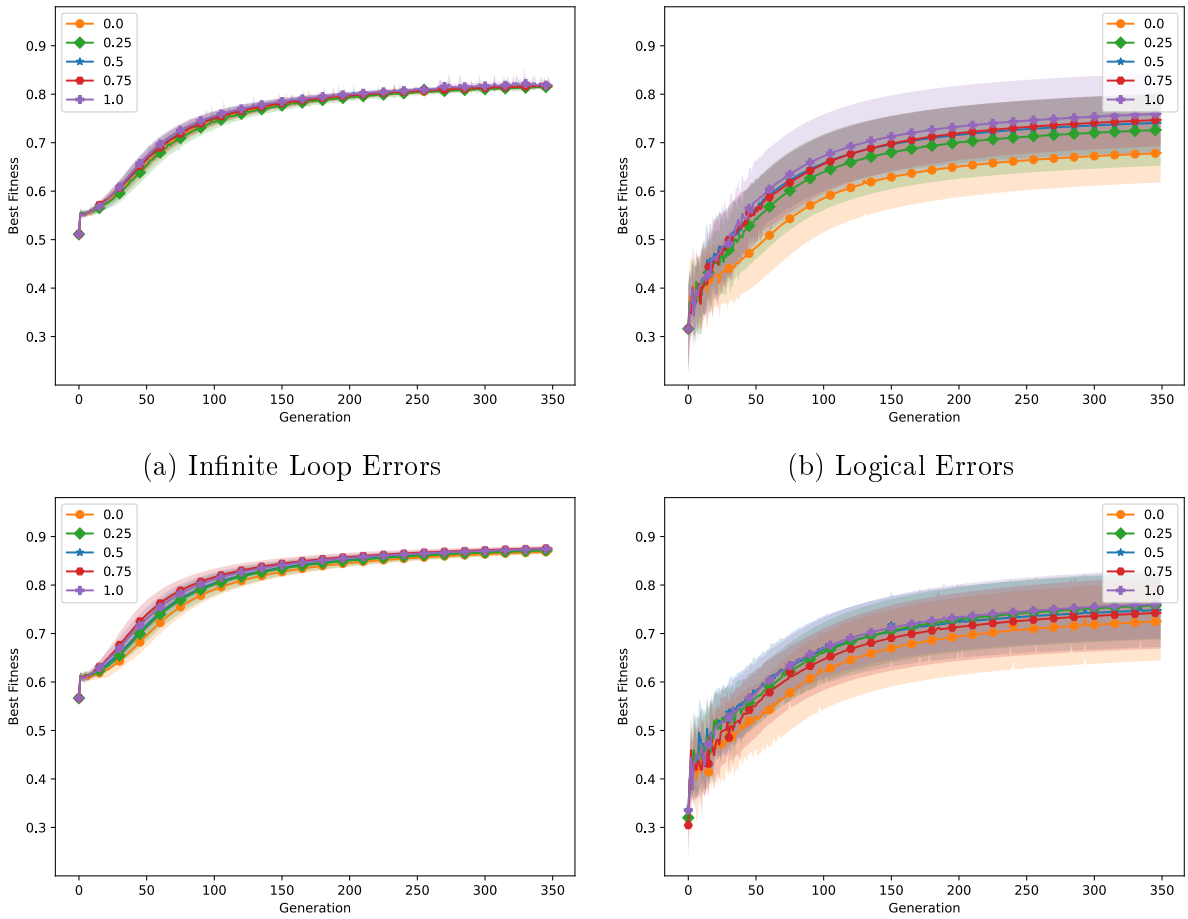
(d) Concurrency Errors

Figure 8.2: Average fitness of the different crossover probability hyperparameter values for different error pools.

8.3.2 Crossover Type

This section discusses the results for the different crossover types that were evaluated. These types include one-point crossover, n-point crossover, segmented crossover, and uniform crossover. The default crossover type was the one-point crossover.

The results of the average and best fitness metrics (Figures 8.6 and 8.7) for the infinite loop and memory error pools reflect the same observation as seen in Section 8.3.1. The reason is again that for both the infinite loop and memory error pools, GATLAM was consistently able to find all of the errors. Looking at the logical error pool's results,



(a) Infinite Loop Errors

(b) Logical Errors

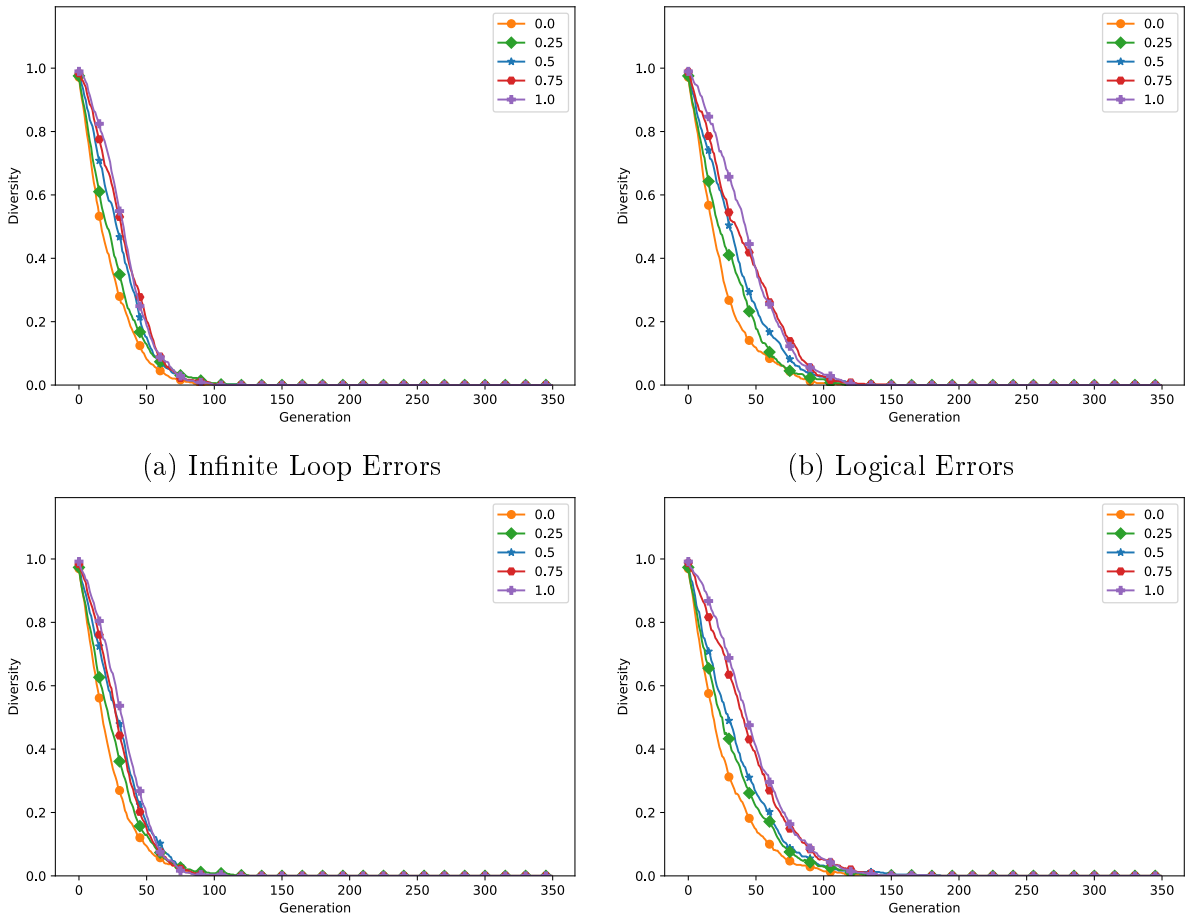
(c) Memory Errors

(d) Concurrency Errors

Figure 8.3: Best fitness for the different crossover probability values for different error pools.

GATLAM utilising segmented crossover was able to find the majority of the errors in the final stages of execution, with GATLAM utilising n-point crossover finding the second highest number of errors. In the concurrency error pool, the performance of the different crossover types was relatively similar, with GATLAM utilising uniform crossover performing the best, and GATLAM with n-point crossover performing second best. In the logic error pool, GATLAM with uniform crossover performed the worst, and GATLAM with one-point crossover performed second worst. The reason n-point crossover performs on average better than uniform and segmented crossover can be attributed to the randomness associated with determining how many crossover points to use. One point's

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

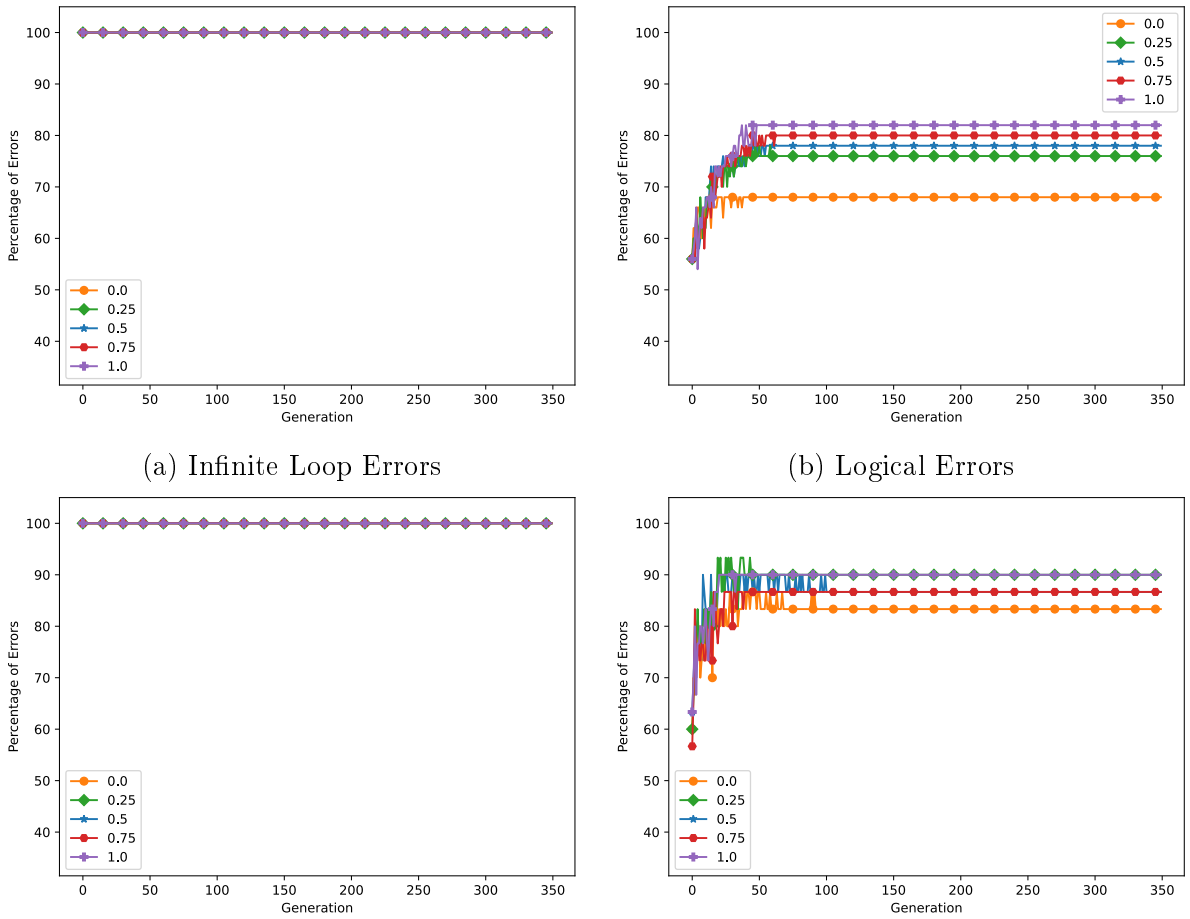
(d) Concurrency Errors

Figure 8.4: Population diversity for the different crossover probability values for different error pools.

poor performance can be attributed to the slow rate at which the chromosomes "share" information when creating the offspring.

The different crossover types resulted in GATLAM consistently converging onto a final diversity of 0 for all of the error pools, as seen in Figure 8.8. For the logic error pool, GATLAM with segmented crossover was the slowest to converge, and GATLAM with one-point crossover was the quickest to converge. In the concurrency error pool, GATLAM with uniform crossover was able to rapidly converge before diverging and then converging again. Uniform crossover was also the only crossover type in all of the error pools to not reach a population diversity of 0 before 200 generations.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

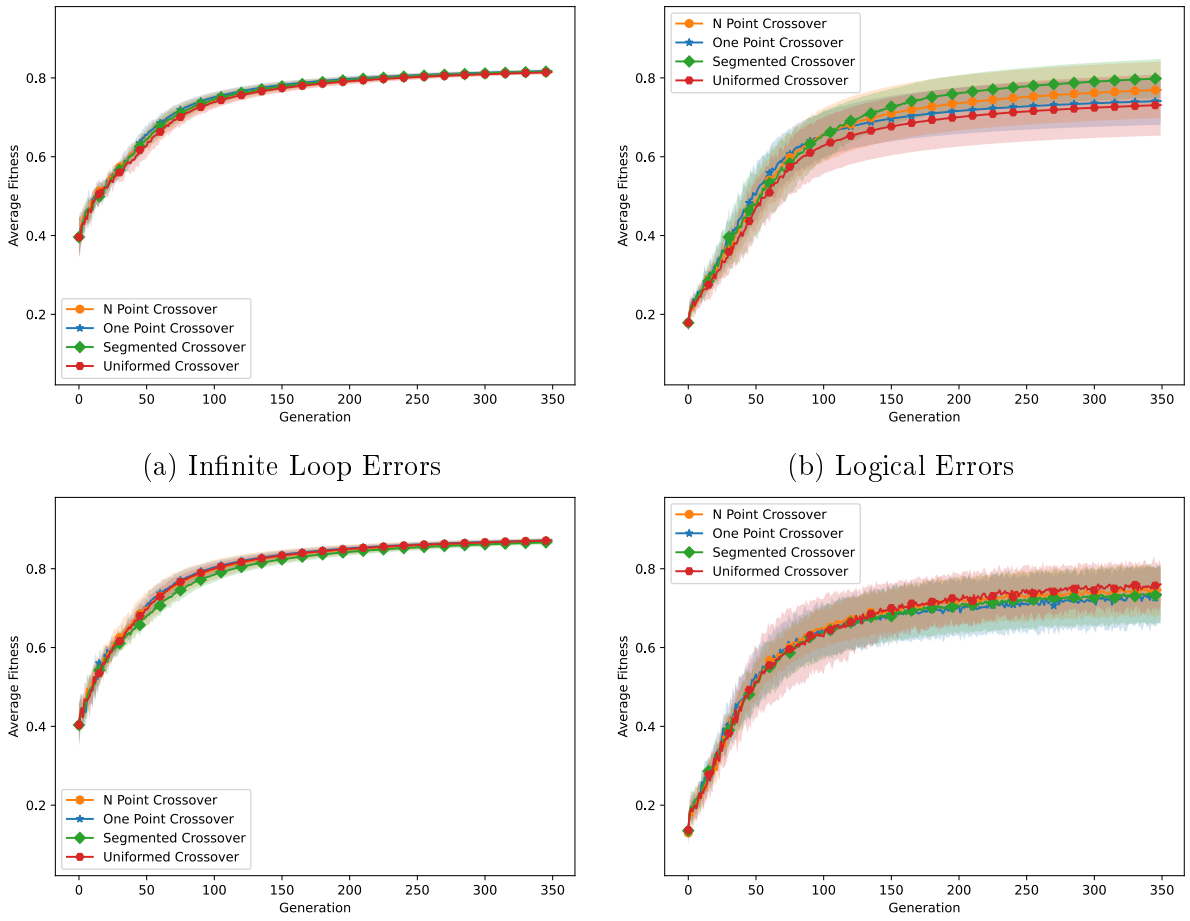
Figure 8.5: Percentage of errors found by the best chromosome for the different crossover probability values using different error pools.

The results of the percentage of errors found (Figure 8.9) indicated that for both the infinite loop and memory error pools, GATLAM was able to consistently find the errors regardless of the crossover type used. For the remaining two error pools, namely concurrency and logical error pools, GATLAM with segmented crossover performed the best, finding the majority of the errors, while GATLAM with n-point crossover came second best. Uniform crossover yielded the lowest number of logical errors, with one-point crossover yielding the second lowest number of logical errors. For concurrency error,s the reverse occurred: GATLAM with uniform crossover found the most error,s while GATLAM with segmented crossover found the lowest number of errors. GATLAM with

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

n-point crossover was still able to find the second highest number of concurrency errors, and GATLAM with one-point crossover found the second lowest number of concurrency errors.

Based on the discussion in this section, the overall best-performing crossover type is chosen to be n-point crossover, and the overall worst-performing crossover type is chosen to be one-point crossover.



(a) Infinite Loop Errors

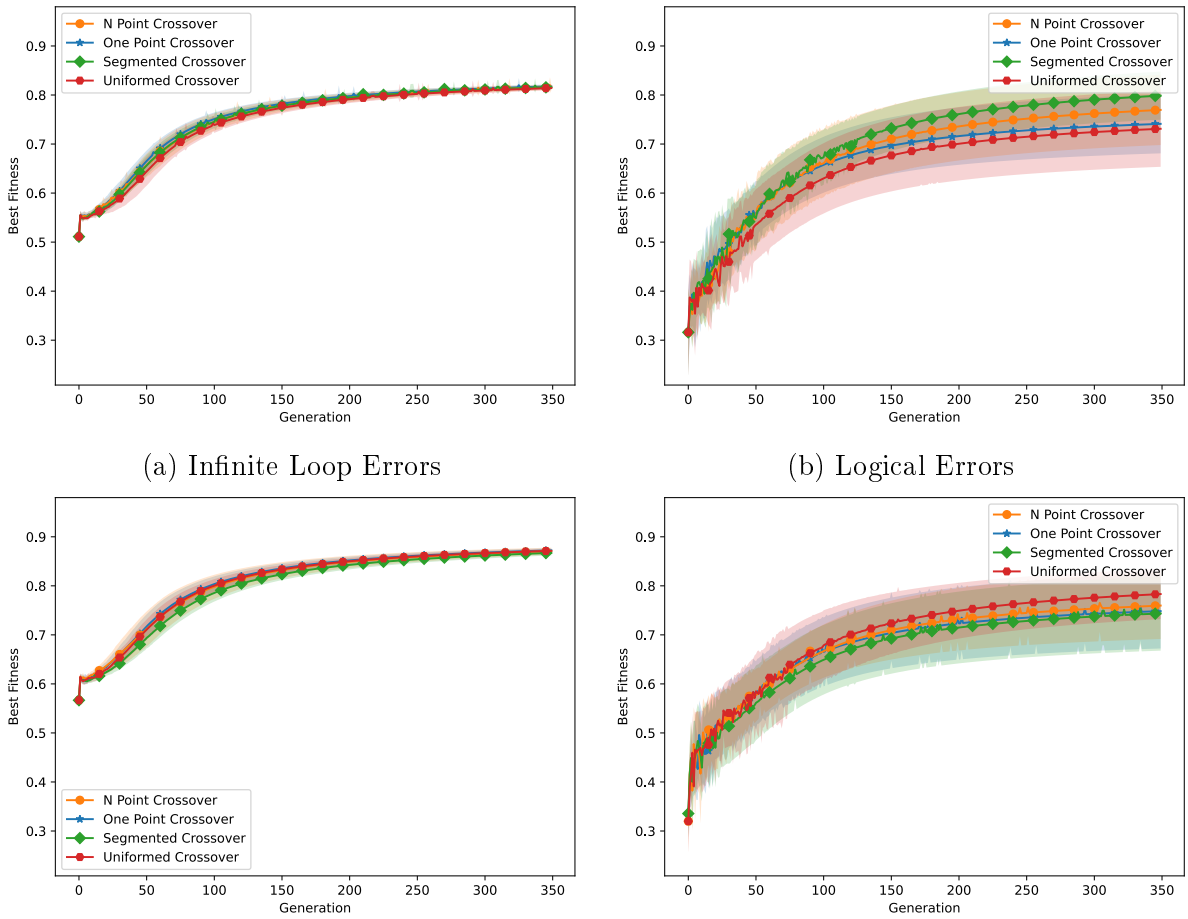
(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.6: Average fitness for the different crossover types for different error pools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

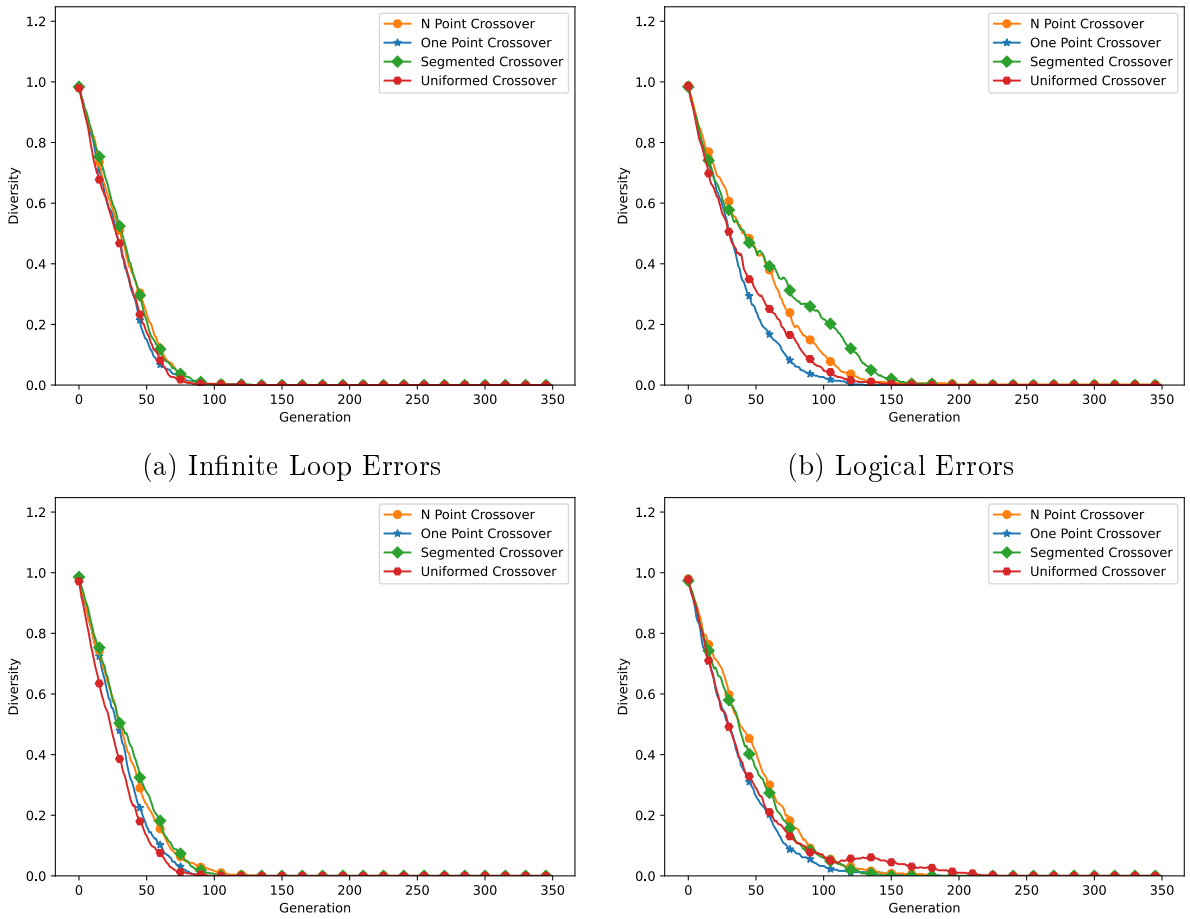
(d) Concurrency Errors

Figure 8.7: Best fitness for the different crossover types for different error pools.

8.3.3 Mutation Probability

This section discusses the results pertaining to different mutation probabilities and their effects on GATLAM's performance. The different mutation probabilities evaluated are 0%, 25%, 50%, 75%, and 100%, with 50% being the default parameter.

The average fitness and best fitness (Figures 8.10 and 8.11) of both the infinite loop and memory error pools exhibited the same behaviour as before apart from a mutation probability of 0% performing slightly worse. For the logic error pool, a mutation probability of 0% performed the best, and GATLAM with a mutation probability of 75% performed the worst. This could be due to a high mutation probability destroying too



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

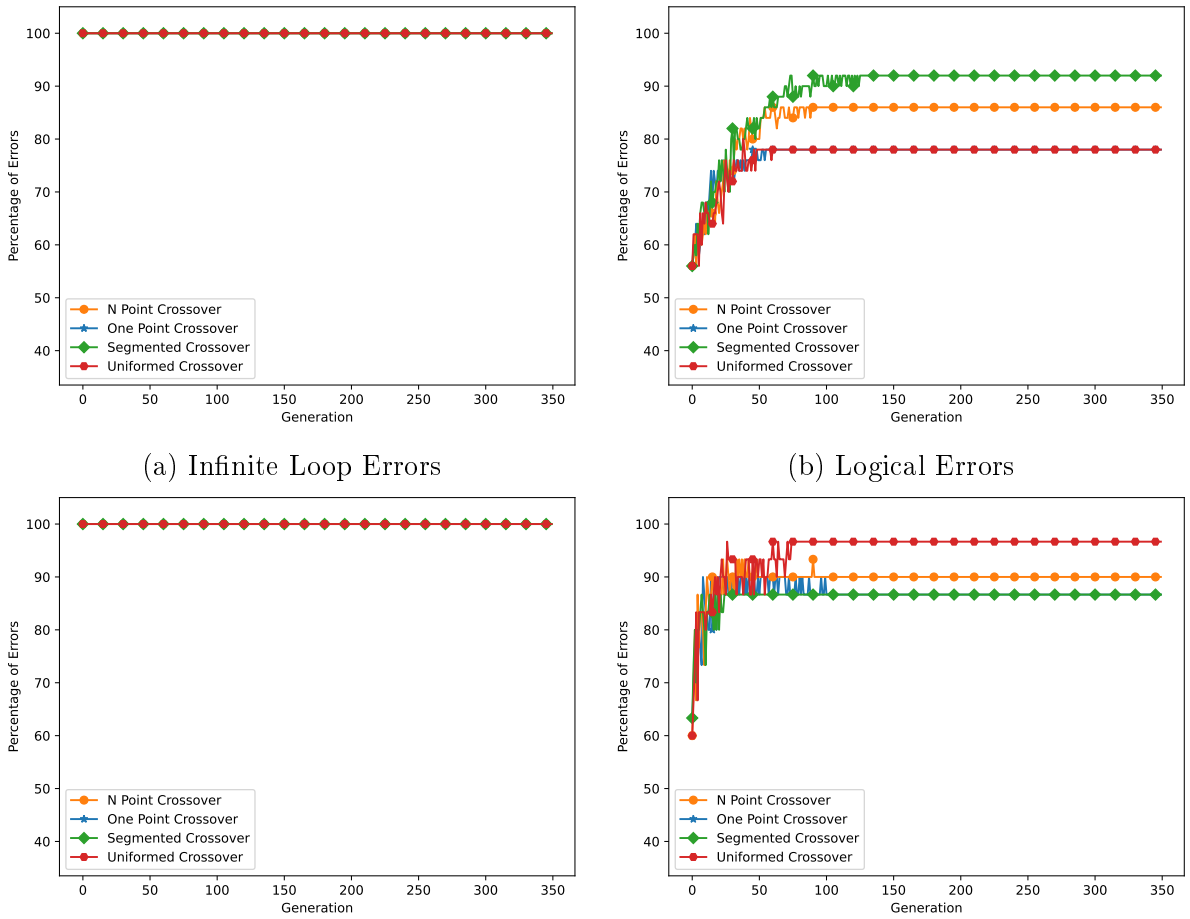
(d) Concurrency Errors

Figure 8.8: Diversity of the population for the different crossover types for different error pools.

much of the important information. Notably, the difference in performance between different mutation probabilities was not statistically significant as the standard deviations overlapped. For the concurrency error pool, a mutation probability of 75% again performed the worst, while a mutation probability of 100% caused GATLAM to perform the best. GATLAM with a mutation probability of 0% performed second best.

When evaluating the decrease in population diversity (Figure 8.12), it is observed that for all of the error pools, GATLAM with a mutation probability of 100% and 75% converged at the highest rate. In all of the error pools apart from the concurrency error pool, a mutation probability of 0% yielded the slowest convergence. This can be

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.9: Percentage of errors found by the best chromosome for the different crossover types for different error pools.

attributed to the lower exploration that a mutation probability of 0% causes. In the concurrency error pool, a mutation probability of 75% caused GATLAM to converge the slowest. At the end of execution, all of the mutation probabilities caused GATLAM to converge onto the same population diversity.

The percentage of errors found (Figure 8.13) by GATLAM using the different mutation probabilities tells a different story compared to the fitness value. Looking solely at the fitness values, it was observed that GATLAM with a mutation probability of 0% had a slightly lower fitness value compared to GATLAM with the other mutation probability configurations for the infinite and memory error pools. Yet GATLAM, with

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

a mutation probability of 0%, was able to find the same number of errors as GATLAM configured with different mutation probabilities. This can be attributed to GATLAM with a mutation probability of 0% exploring the search space at a slower rate, which negatively affected the contribution of the memory component of the fitness function. GATLAM with a mutation probability of 0% was also able to find the most errors in both the logical and concurrency error pools. GATLAM with a mutation probability of 75% found the lowest number of errors in both cases. It seems that exploitation is more beneficial than exploration in this case.

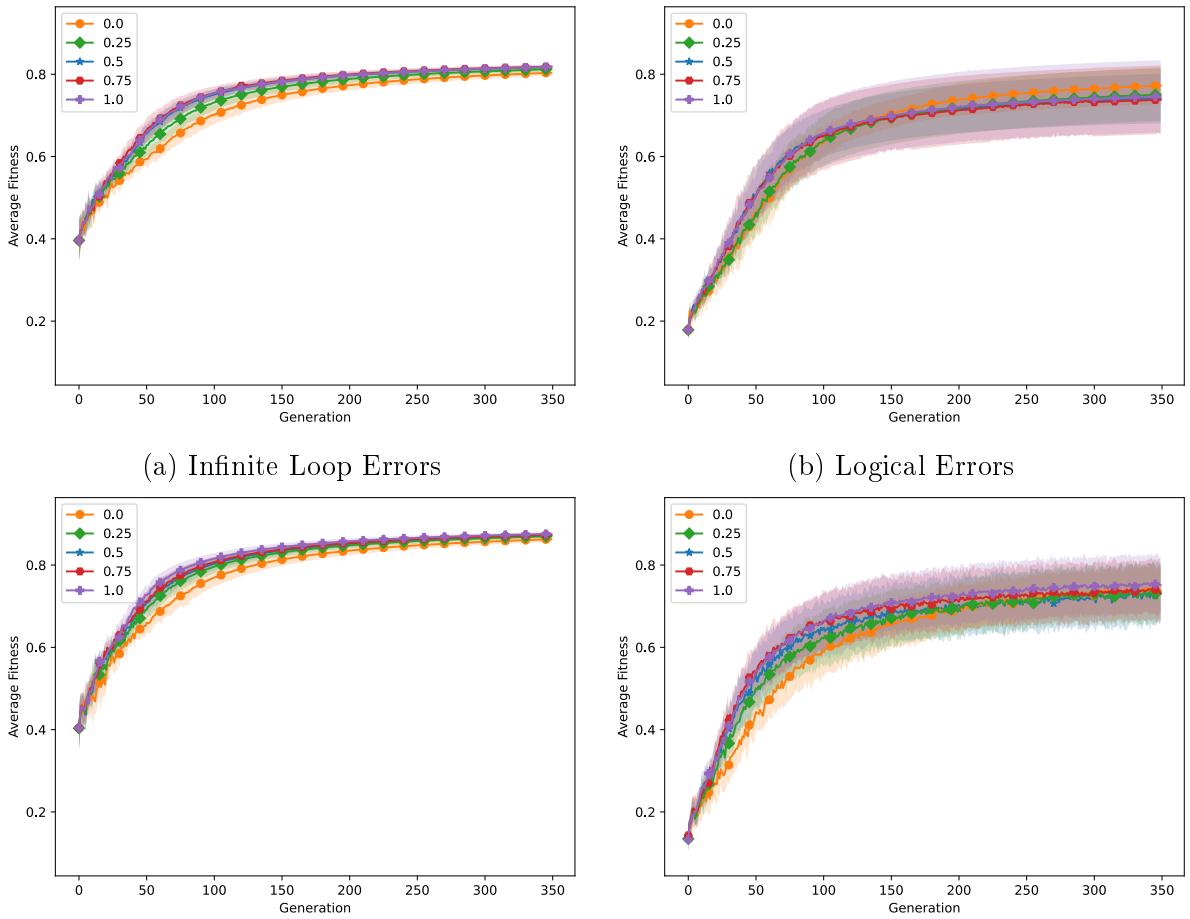
Based on the discussion above, the best mutation probability is chosen to be 0% due to its slower convergence rate, ability to find the most errors, and the difference in fitness performance being slight. A mutation probability of 75% is chosen to be the worst due to its rapid convergence and low error-finding abilities. A mutation probability of 0% brings into question how GATLAM is still able to search through the search space if it is totally reliant on just the initial randomness with which the population was created. The reason for the success of a mutation probability of 0% can be attributed to the genetic operators being applied at a binary level, which is then converted to numbers before they are passed off to the interpreter.

8.3.4 Mutation Type

This section provides a brief summary of the results of GATLAM using different mutation types. Although the best mutation probability was 0%, it is still important to investigate how different mutation types affect GATLAM's performance. The types evaluated were: Bit-wise inversion, random inversion, and single-bit inversion. Single-bit inversion was the default mutation type.

When comparing the average and best fitness (Figures 8.14 and 8.15) of the different mutation types, it is clear that GATLAM with a random inversion did not perform well compared to GATLAM with the other mutation types. Random inversion had the worst-performing fitness value in all of the error pools, except for the logic error pool. This can be attributed to the randomness associated with the number of bits flipped in the random inversion. In contrast, single-bit inversion had either the highest or second-highest average and final fitness values across the four error pools. Bitwise inversion had

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

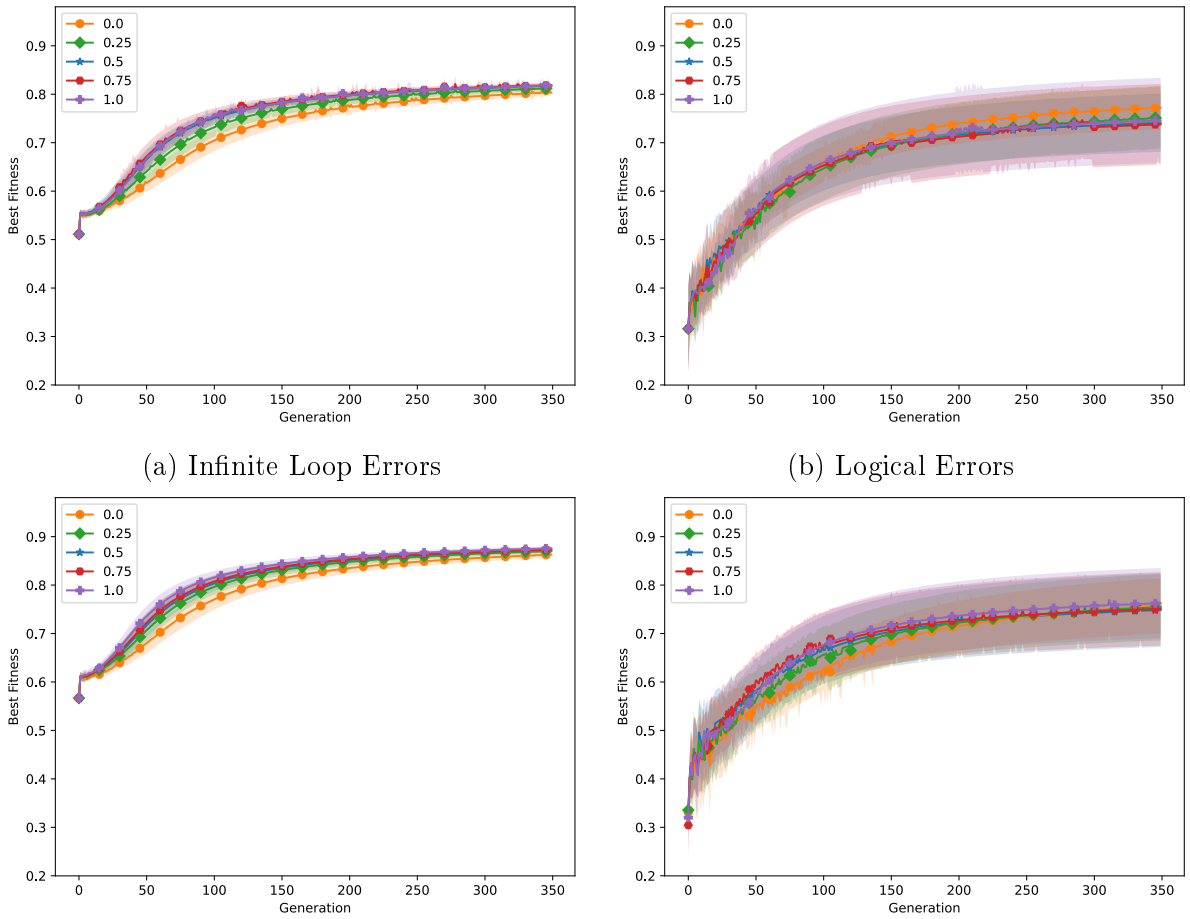
(d) Concurrency Errors

Figure 8.10: Average fitness for the different mutation probabilities for different error pools.

the highest performance in two of the four error pools, but had the worst performance in the logical error pool.

The population diversity graphs (Figure 8.16) for the random inversion and single-bit inversion show a difference in behaviour. Only bit-wise inversion was able to converge onto a population diversity of 0. Neither single-bit inversion nor random inversion was able to converge onto a population diversity below 0.18. It is interesting to note that both random inversion and single-bit inversion created a type of "braided" pattern where the two mutation types would bypass each other constantly. This could again be attributed to the randomness associated with single-bit inversion and random inversion.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

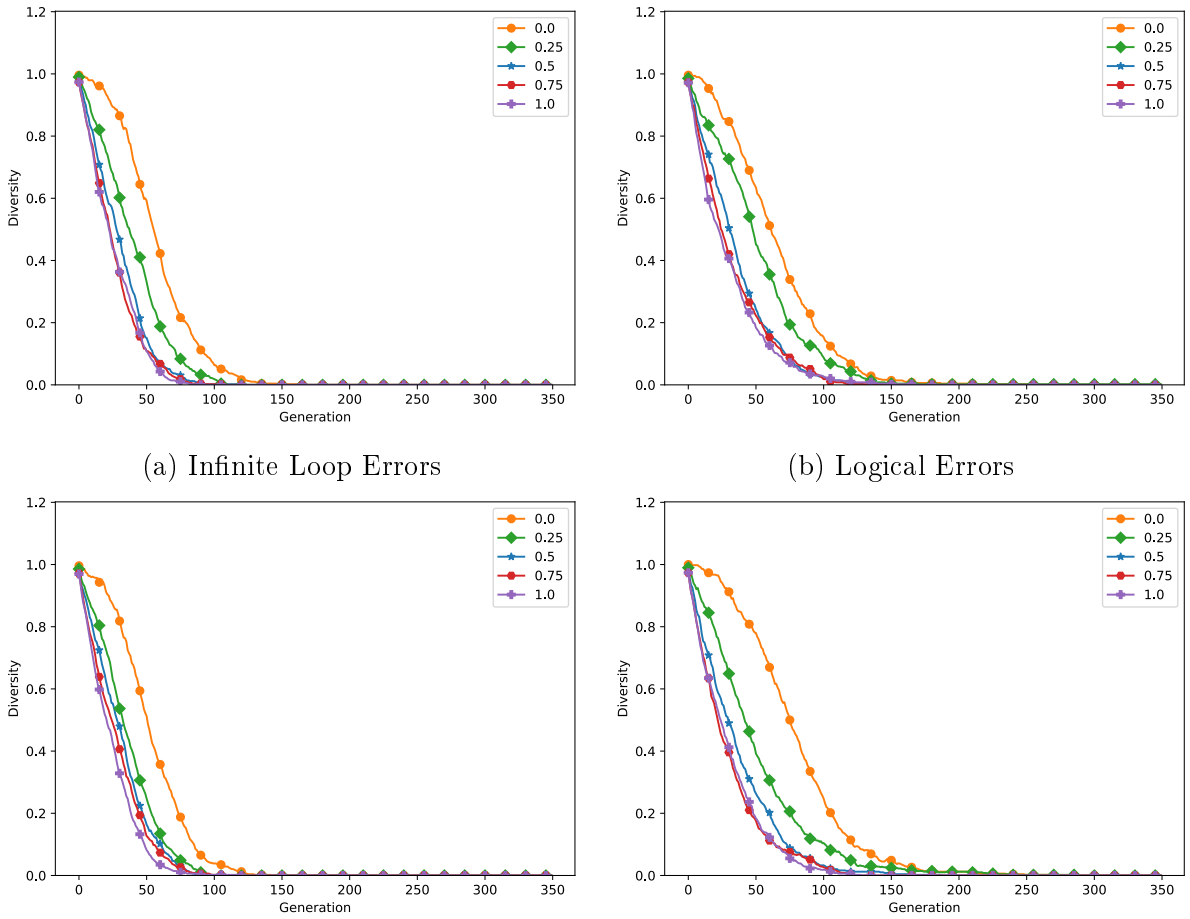
(d) Concurrency Errors

Figure 8.11: Best fitness for the different mutation probabilities for different error pools.

When comparing the percentage of error types found (Figure 8.17), random inversion was able to find the majority of the errors in all of the error pools, while single-bit inversion was able to achieve the same performance as random-bit inversion except for the logical error pool. Bitwise inversion was not able to achieve good performance in the logical or concurrency error pools but was able to find all of the errors consistently in the infinite loop and memory error pool.

For the mutation type hyperparameter, it is difficult to clearly state which mutation type performed better, as in each metric, a different mutation type excelled. For this reason, single-bit inversion is chosen as the average best-performing mutation type due

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

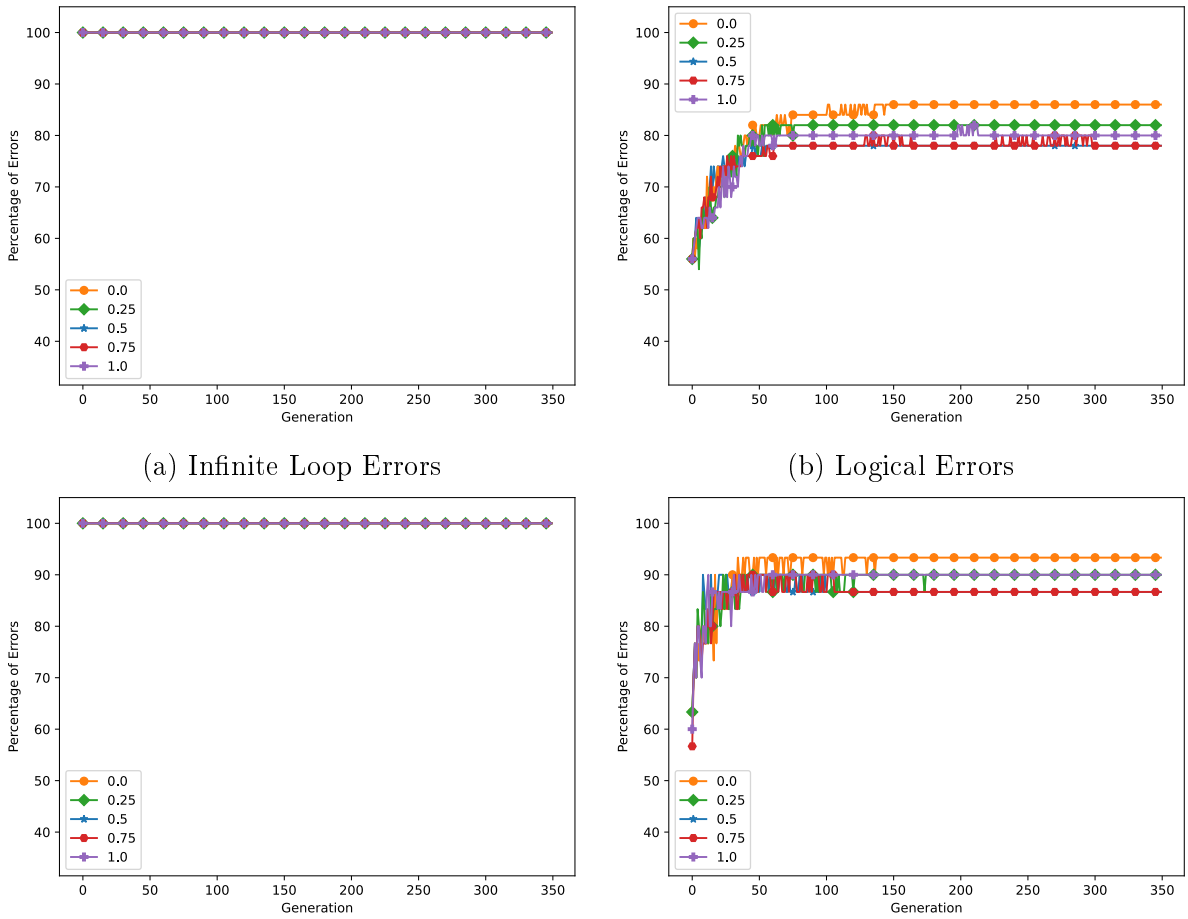
Figure 8.12: Diversity of the population for the different mutation probabilities for different error pools.

to its average good performance, and random inversion is chosen to be the average worst-performing due to the heavy reliance on random numbers.

8.3.5 Population Size

This section discusses the results of the population size hyperparameter. The different values that were evaluated were 5, 10, 25, 50, 75, 100, 125, and 150, with 50 being the default population size.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

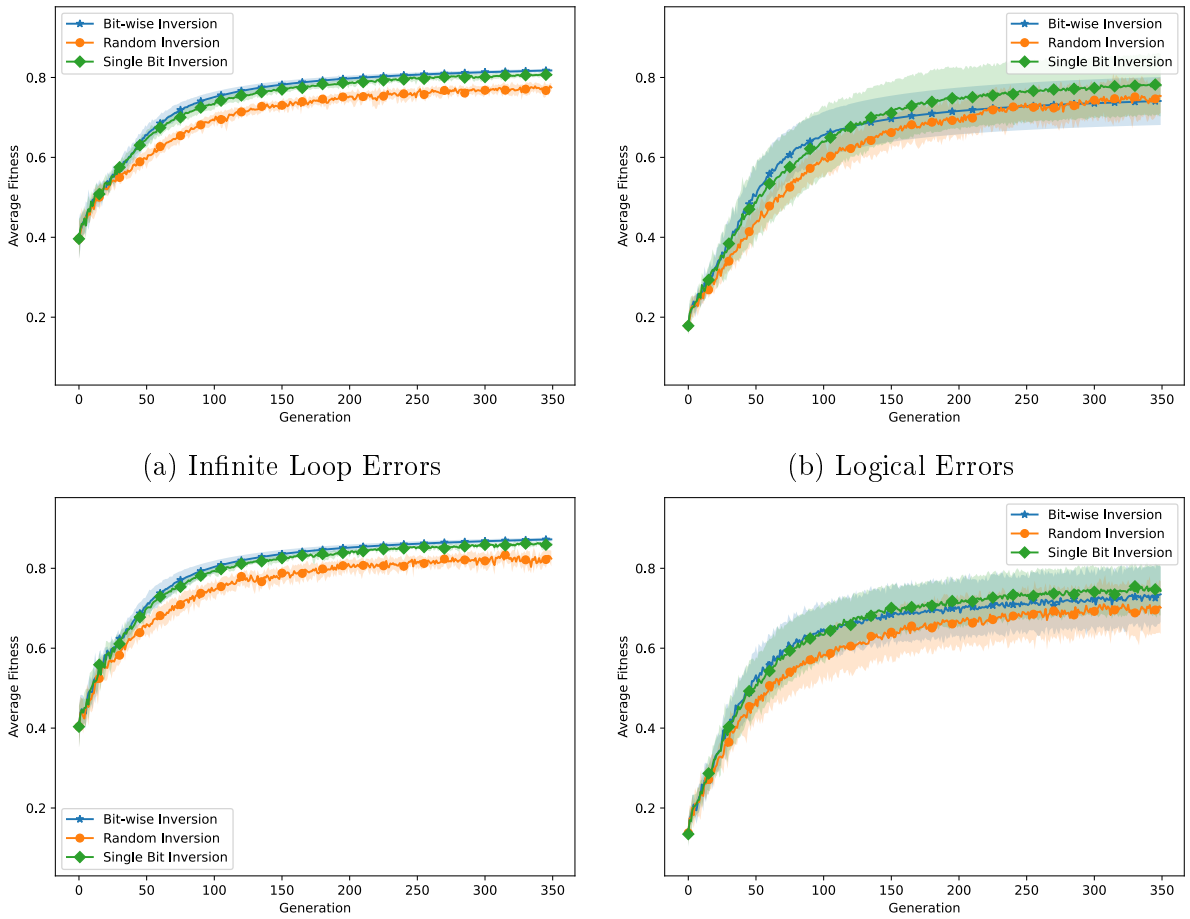
(c) Memory Errors

(d) Concurrency Errors

Figure 8.13: Percentage of errors found by the best chromosome for the different mutation probabilities for different error pools.

Observed in every metric used to evaluate the population size hyperparameter (Figures 8.18, 8.19, 8.20, and 8.21), is that the smaller the population size the quicker the metric changes. In other words, a population size of 5 was able to achieve a higher fitness quicker than a population size of 150. This is attributed to the probability of picking a chromosome for evaluation in tournament selection, which is the default selection method. This caused smaller populations to evolve quicker than larger populations, but could also lead to smaller populations having fewer chromosomes to exploit, i.e. less information to work with.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

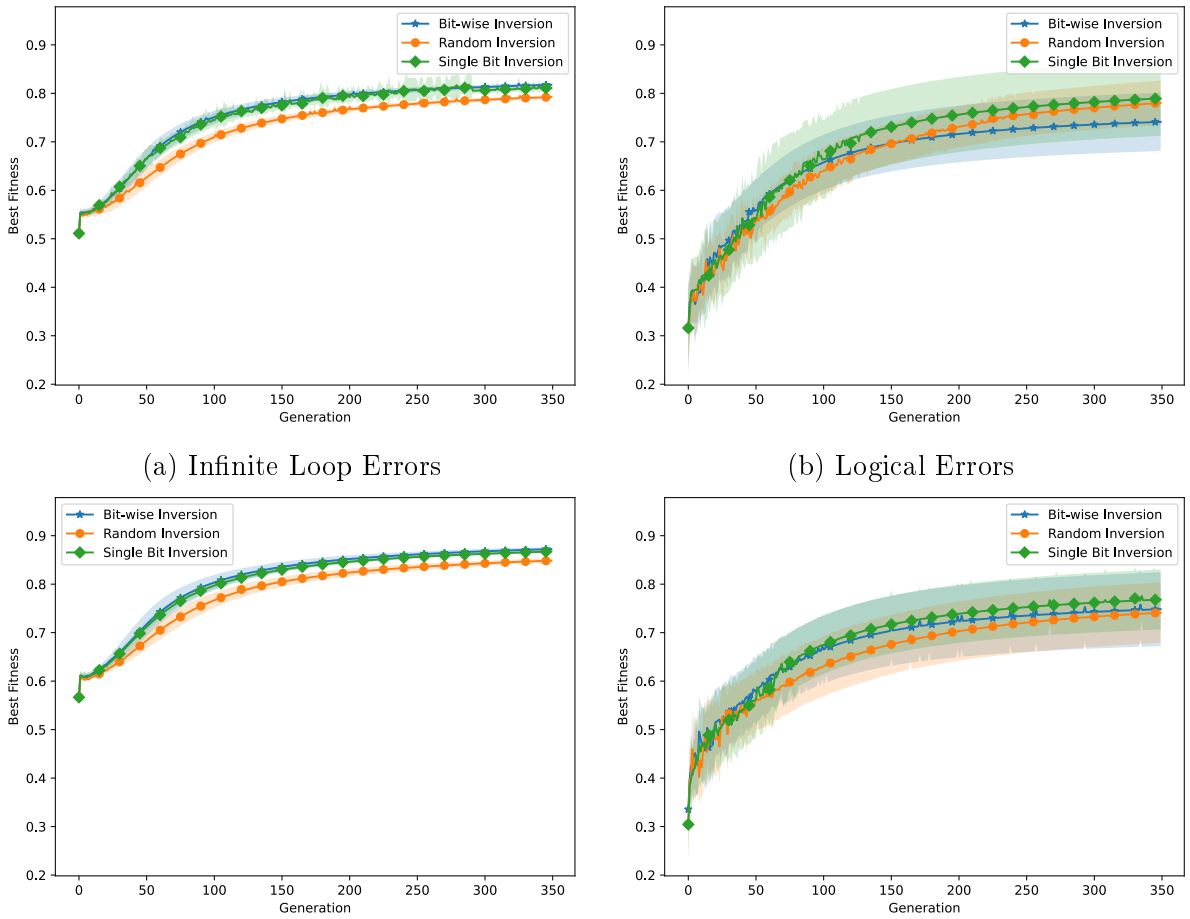
(c) Memory Errors

(d) Concurrency Errors

Figure 8.14: Average fitness for the different mutation types for different error pools.

The observation is seen in the average and best fitness metric (Figures 8.18 and 8.19) where a population size of 5 was able to converge to a high fitness value quickly, while a population size of 150 took longer to converge. One contributing factor can be the memory component, which negatively affects slow "learning" configurations of GATLAM. A population size of 50 was able to achieve good results in a relatively short amount of time compared to GATLAM, with a population size of 100.

The observation is also reflected in population diversity (Figure 8.20). Larger population sizes caused GATLAM to converge at a slower rate compared to smaller population sizes. A population size of 5 was able to converge onto a diversity score of 0 before



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

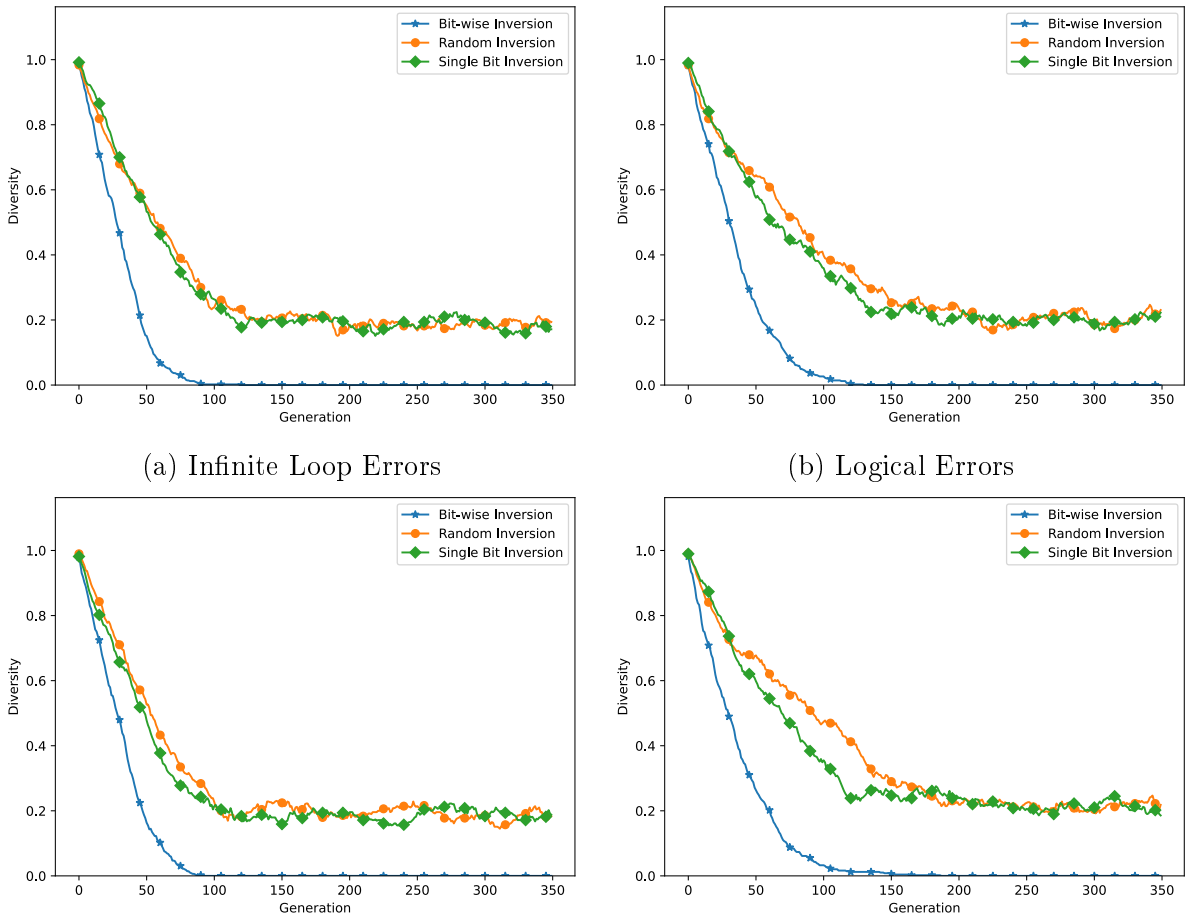
(d) Concurrency Errors

Figure 8.15: Best fitness for the different mutation types for different error pools.

50 generations were completed, while a population size of 150 only converged below a diversity score of 0.18 after around 200 generations.

For the infinite loop and memory error pools, GATLAM was able to constantly find the errors, according to the percentage found metric (Figure 8.21), regardless of the population size. In contrast, for the logic and concurrency error pools, GATLAM with a population size of 5 was able to find the lowest number of errors, compared to a population size of 150 which was able to find the highest number of errors. GATLAM, with a population size of 150 did took longer to find a high number of errors compared to a population size of 50, which was able to find a relatively high number of errors in

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

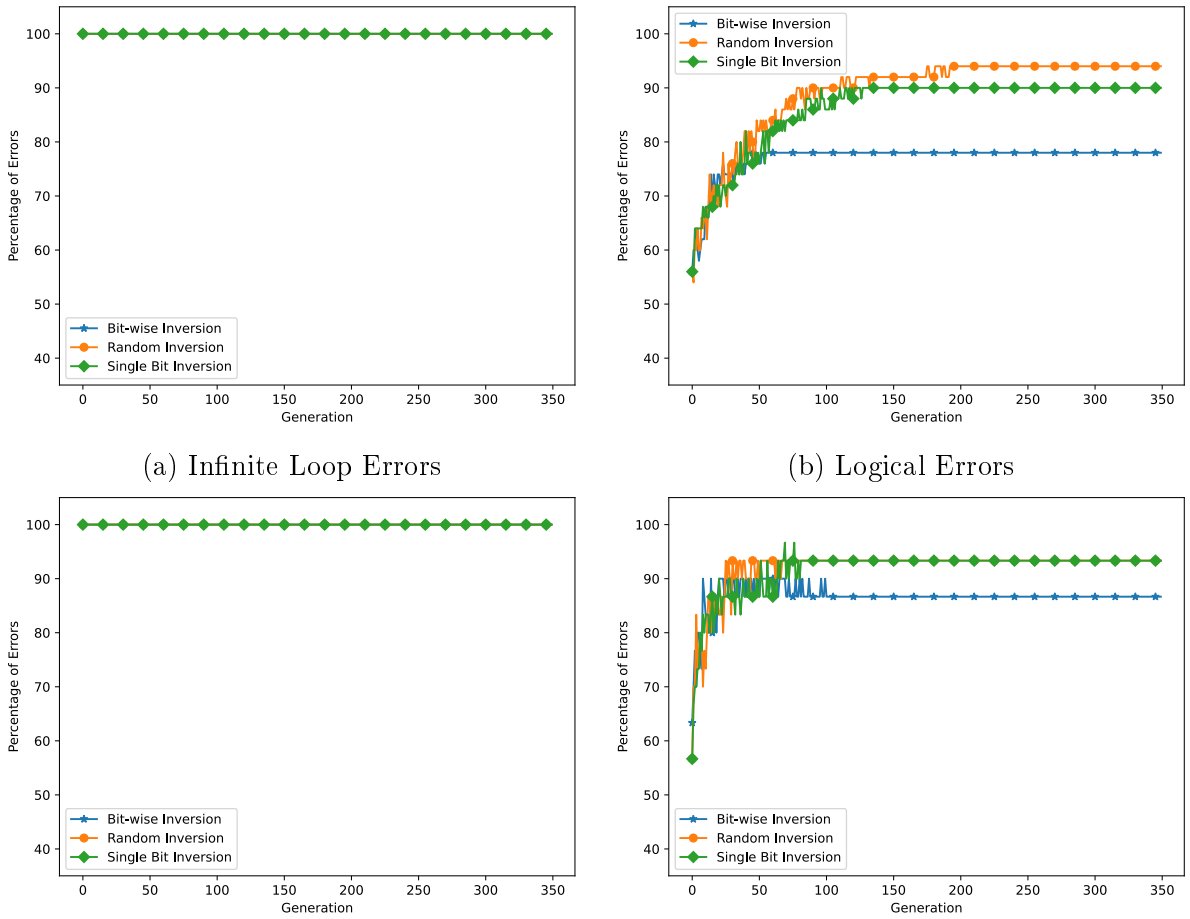
(d) Concurrency Errors

Figure 8.16: Diversity of the population for the different mutation types for different error pools.

a shorter amount of time. A contributing factor to why GATLAM, with a population size of 150, could find more errors using the best chromosome is due to the low selection probability of each chromosome, which could lead to slower population evolution. This could ultimately cause GATLAM with large population sizes to be dependent on the random of the initial population generation. The author recommends that alternative population generation methods are evaluated in future work, possibly where GATLAM's population generation is influenced by previously evaluated submitted assignments.

Based on the discussed results, the average best population size is selected to be 50, due to a population size of 50 showing good performance in a relatively short number of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

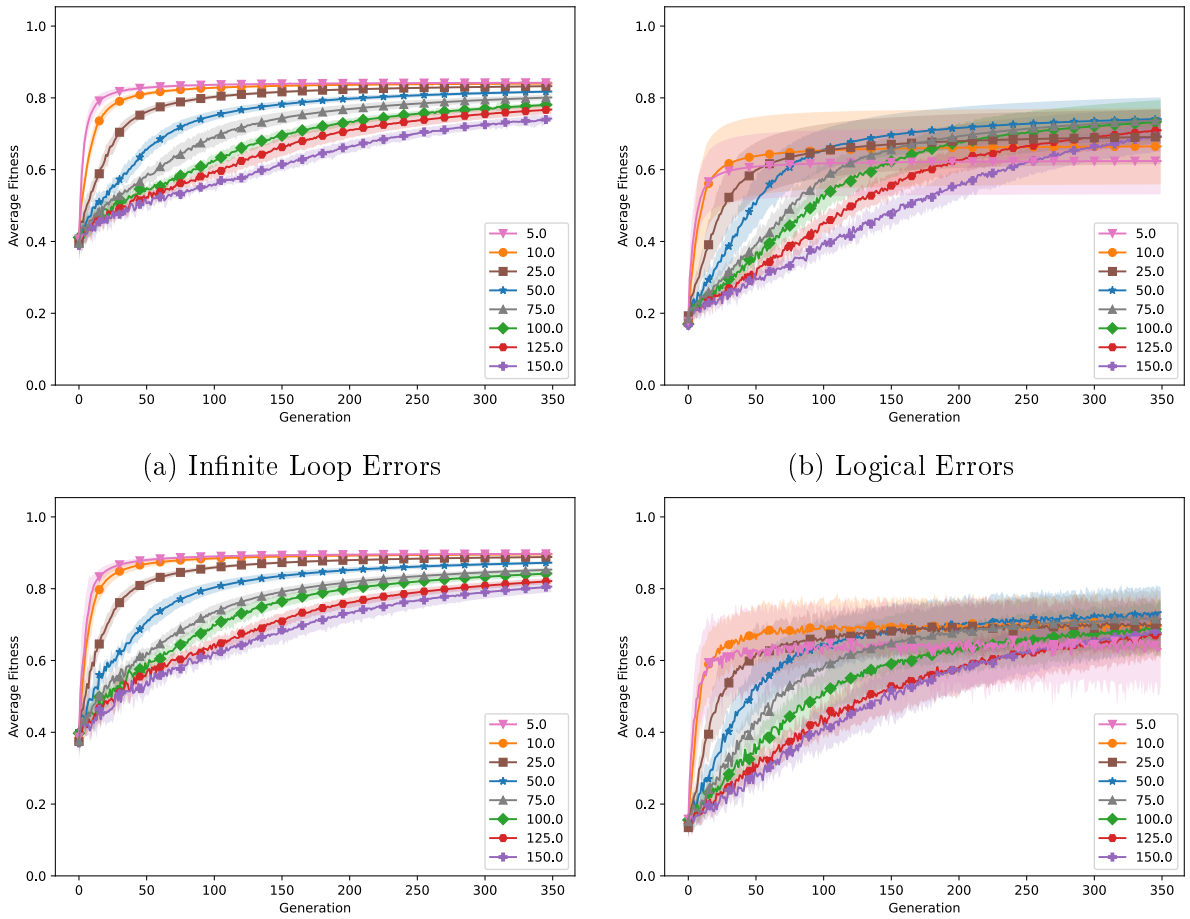
(d) Concurrency Errors

Figure 8.17: Percentage of errors found by the best chromosome for the different mutation type for different error pools.

generations. The worst population size is chosen to be 5, due to its quick convergence rate in both fitness value and population diversity, while finding the lowest number of errors.

8.3.6 Reproduction Probability

This section provides a brief summary of the results for the different reproduction probabilities evaluated. The probabilities evaluated are: 0%, 25%, 50%, 75%, and 100%. A reproduction probability of 50% is the default parameter value.



(a) Infinite Loop Errors

(b) Logical Errors

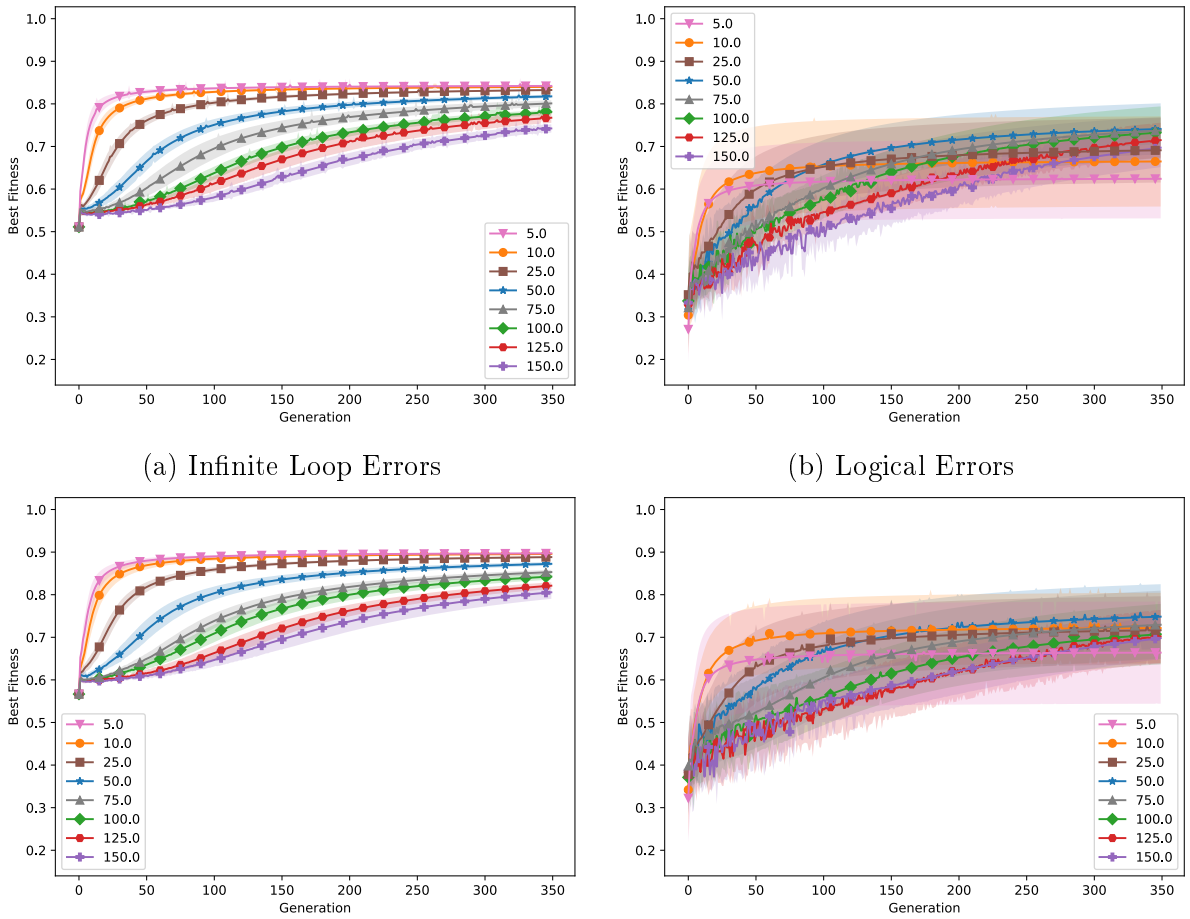
(c) Memory Errors

(d) Concurrency Errors

Figure 8.18: Average fitness for the different population sizes for different error pools.

The effect that the reproduction probability hyperparameter had on GATLAM's behaviour differs from the behaviour seen so far. Different reproduction probabilities were able to produce nearly identical results for both the average and the best fitness metrics (Figures 8.22 and 8.23) in all of the error pools apart from the concurrency error pool. In the concurrency error pool, a reproduction probability of 100% was able to marginally outperform the other reproduction probabilities. A reproduction probability of 25% performed marginally worse than the other settings.

The same observation is reflected in the population diversity (Figure 8.24). For all of the error pools apart from the concurrency error pool, GATLAM performed nearly



(a) Infinite Loop Errors

(b) Logical Errors

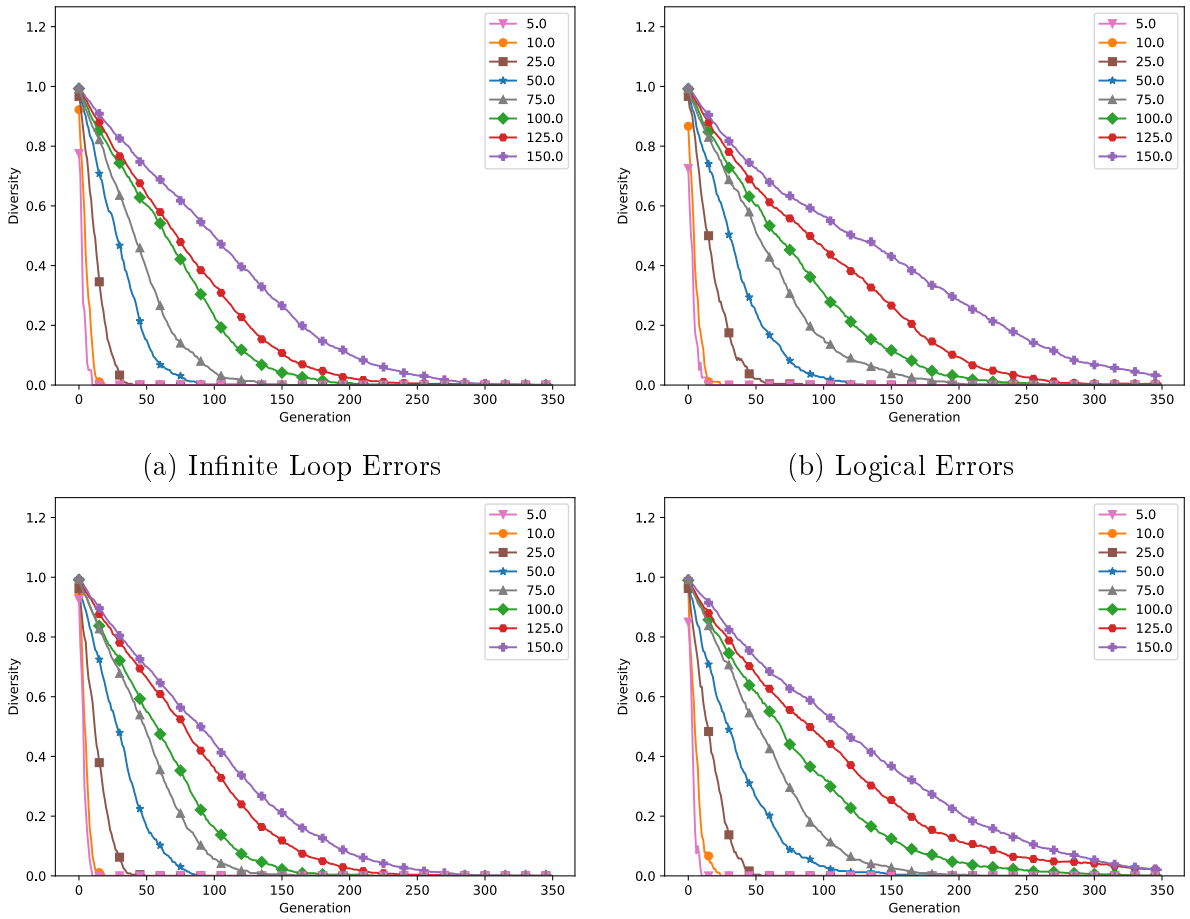
(c) Memory Errors

(d) Concurrency Errors

Figure 8.19: Best fitness for the different population sizes for different error pools.

identically irrespective of the reproduction probability used. In the infinite loop error pool, it was observed that a reproduction probability of 0% and 75% converged slightly slower compared to the other reproduction probabilities. In the concurrency error pool, it was seen that different reproduction probabilities converged at different rates, with a reproduction probability of 50% converging the quickest and reproduction probabilities of 75% and 100% converging the slowest. This can be attributed to the types of errors that are present in the concurrency pool and the chromosome structures needed to find the errors.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

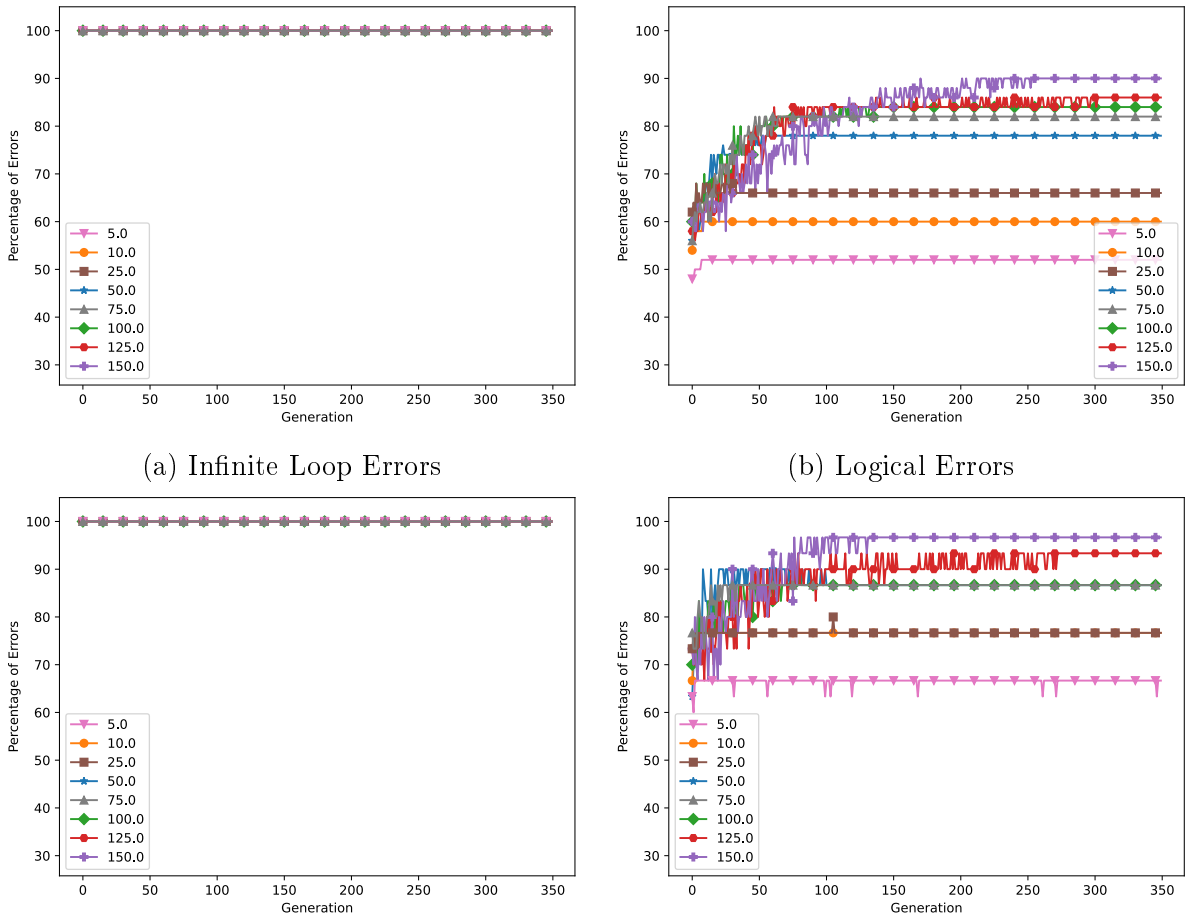
(c) Memory Errors

(d) Concurrency Errors

Figure 8.20: Diversity of the population for the different population sizes for different error pools.

For the percentage of errors found (Figure 8.25), all error pools except for the concurrency pool exhibited identical behaviour. In the concurrency error pool, a reproduction probability of 100% yielded the most errors, while a reproduction probability of 50% found the least errors. Reproduction probabilities of 0% and 25% were able to converge on the same value which was both the second highest and second lowest percentage of errors found in the final generation. In each error pool, GATLAM with all reproduction probabilities was able to find a constant percentage of errors after 150 generations.

It is thus concluded that the reproduction probabilities had a minimal influence on the average performance of GATLAM across the different error pools. For this reason, a



(a) Infinite Loop Errors

(b) Logical Errors

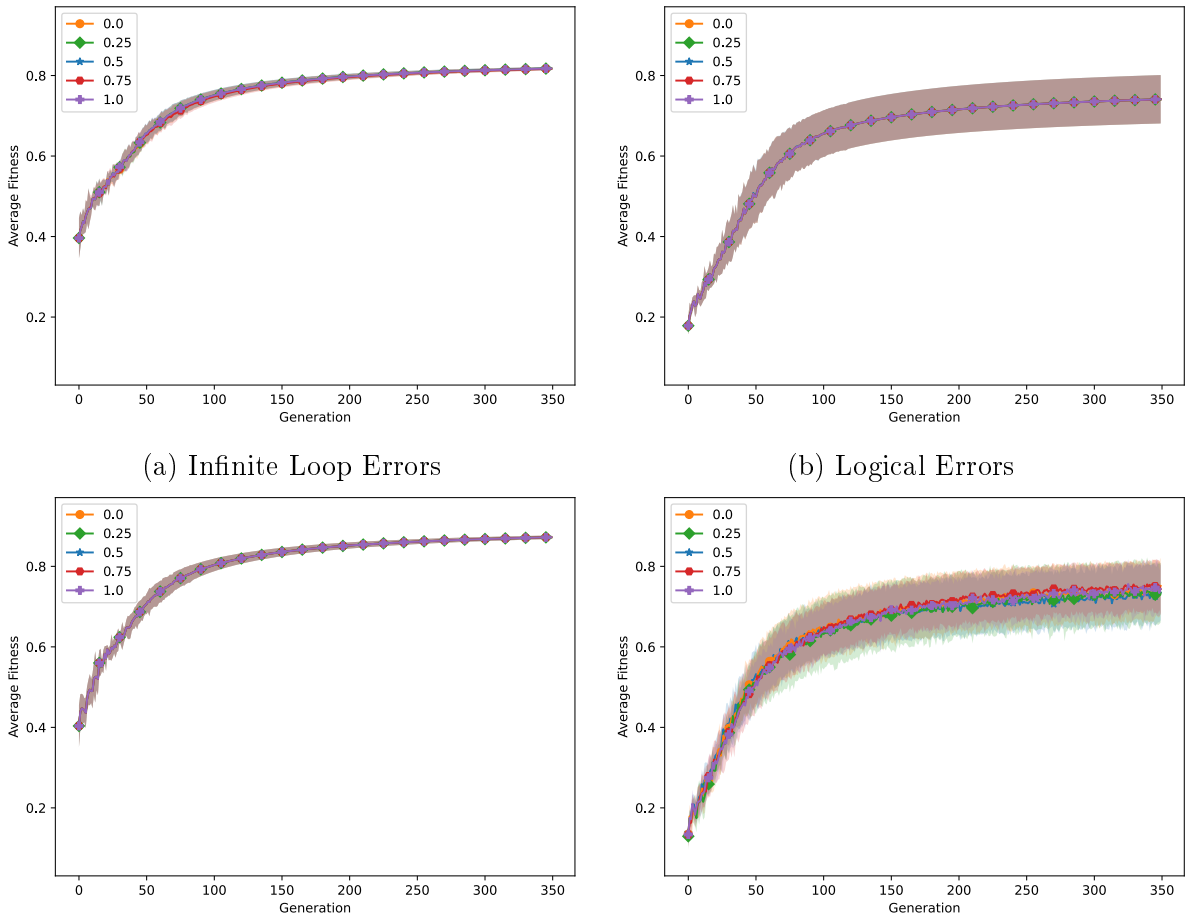
(c) Memory Errors

(d) Concurrency Errors

Figure 8.21: Percentage of errors found by the best chromosome for the different population sizes for different error pools.

reproduction probability of 100% is chosen as the average best-performing reproduction probability and a value of 25% as the average worst-performing reproduction probability. Although the reproduction probability does not significantly impact the performance of GATLAM, a best and worst are still chosen to evaluate the influence they have when combined with other hyperparameters.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

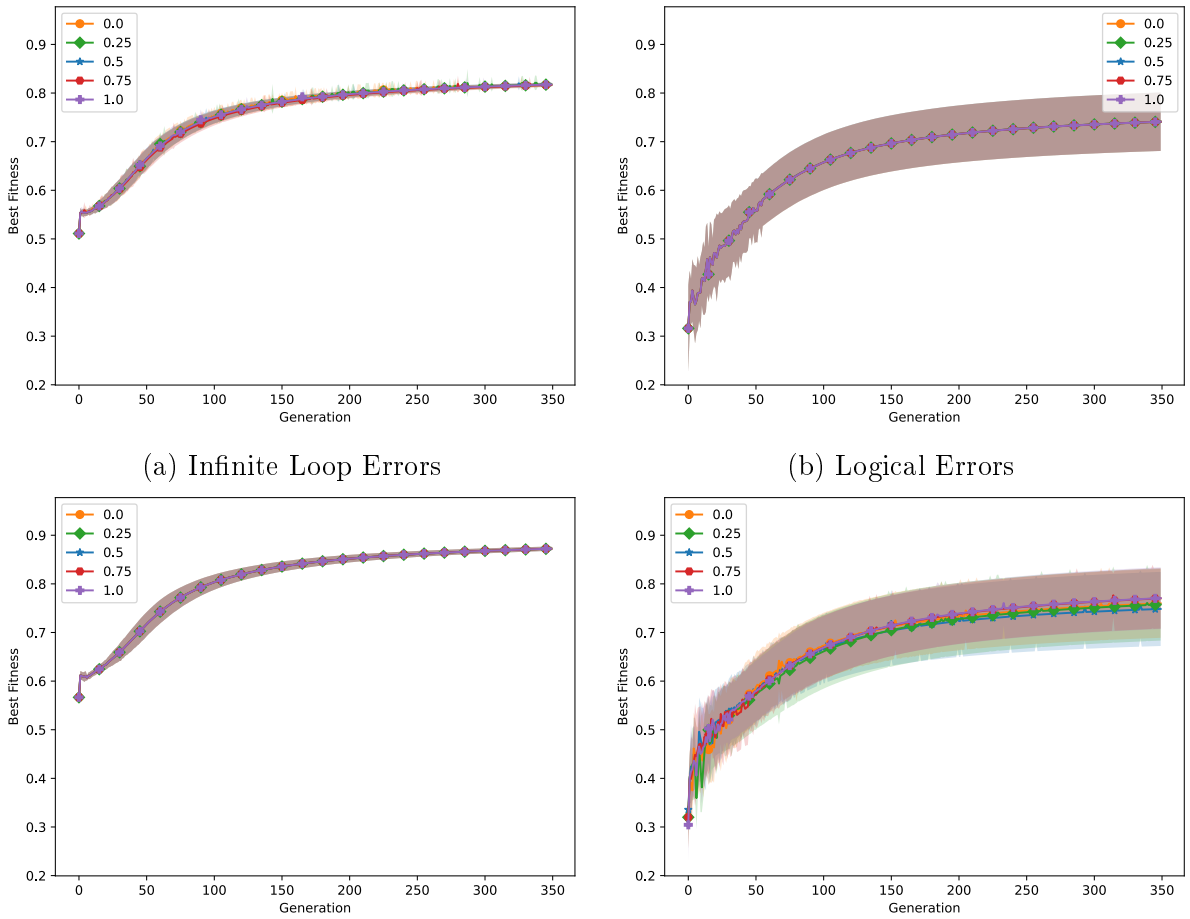
(d) Concurrency Errors

Figure 8.22: Average fitness for the different reproduction probability for different error pools.

8.3.7 Selection Method

This section discusses the results of the different selection methods evaluated, which were: exponential rank selection, roulette wheel selection, stochastic universal sampling (SUS), tournament selection, and truncation selection. The default selection method was tournament selection.

When comparing the results of the average and best fitness (Figures 8.26 and 8.27), a pattern emerges between truncation selection and SUS. For the infinite loop and memory error pools, truncation selection was not able to produce high fitness values, but



(a) Infinite Loop Errors

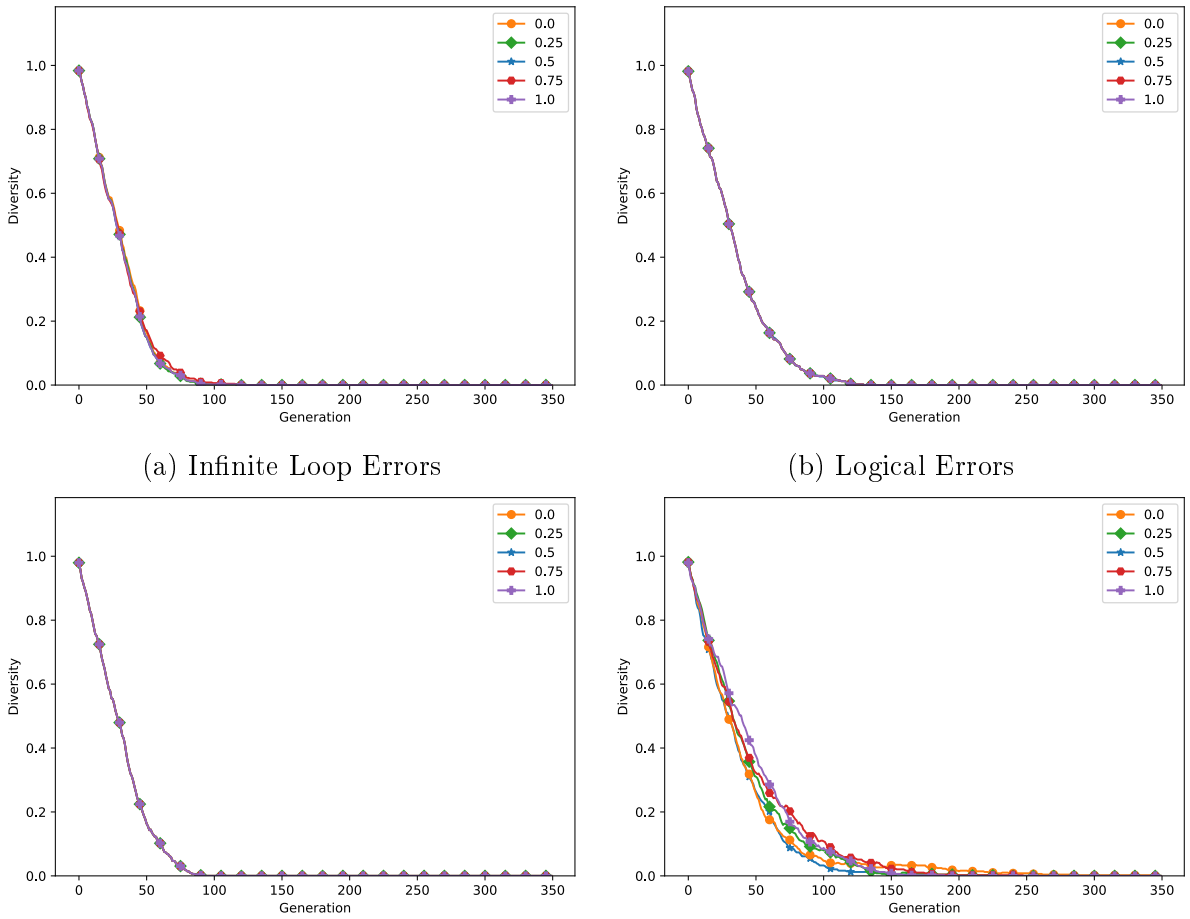
(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.23: Best fitness for the different reproduction probability for different error pools.

was able to maintain a stable fitness value for the majority of its execution. No other selection method was able to produce a constant fitness metric, which can be attributed to truncation selection only selecting chromosomes that are above a certain threshold. For the same two error pools, SUS was able to produce higher fitness values, but at a slower rate compared to truncation selection. When comparing the results of the concurrency and logical error pools, it can be seen that truncation selection was able to produce higher fitness values compared to SUS at a quicker pace. For the infinite loop and memory error pool, SUS was not able to surpass the truncation selection's average and best fitness values. Tournament selection was able to produce the highest average



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

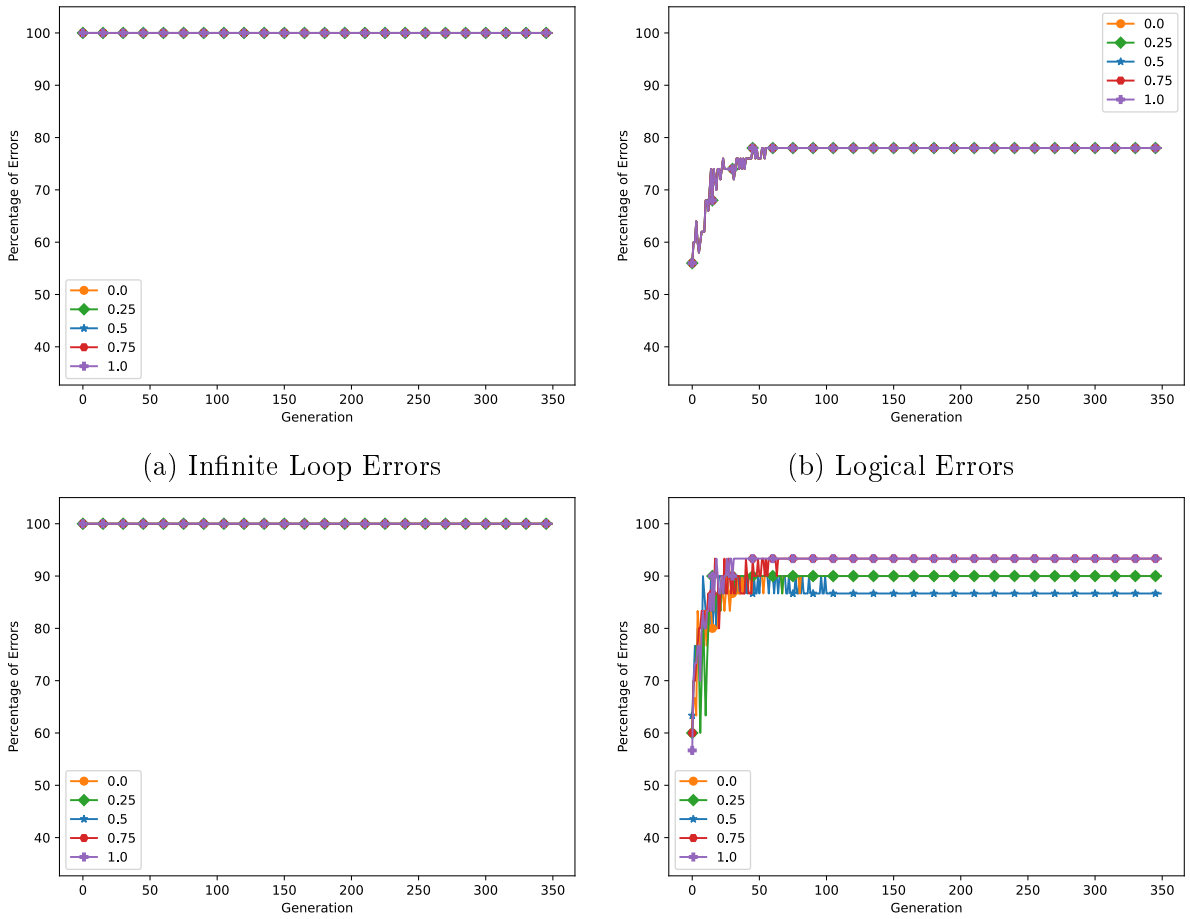
(d) Concurrency Errors

Figure 8.24: Diversity of the population for the different reproduction probabilities for different error pools.

and best fitness metric for the majority of executions in all of the error pools. This can be attributed to tournament selection not evaluating the entire population, but rather a randomly selected subset. Roulette wheel and exponential rank selection were not able to match the performance of tournament selection in any of the error pools, but they were able to exceed the performance of truncation selection and SUS.

The same pattern between SUS and truncation selection is also observed in the diversity of the population (Figure 8.28). For the infinite loop and memory error pools, SUS converged onto a lower population diversity compared to truncation selection. In contrast to the logical and concurrency error pools, truncation selection was able to

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.25: Percentage of errors found by the best chromosome for the different reproduction probability for different error pools.

converge quicker and to a lower population diversity than SUS. Tournament selection was able to converge the quickest to the lowest population diversity score. The same observations about the roulette wheel selection and exponential rank selection were seen in the population diversity metric.

Investigating the percentage of errors found (Figure 8.29), it can be seen that for the infinite loop and memory error pools, all the different selection methods were able to find all of the errors from the start, which differs from the results in the logical and concurrency error pools. In the logical error pool, it can be seen that roulette wheel selection was able to find the majority of the errors consistently after the first 50

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

generations, while SUS was only able to surpass tournament selection after about 200 generations. Between generations 20 and 200, SUS found the lowest number of errors, which can be attributed to SUS requiring a longer execution time to outperform the other selection methods. Looking at the concurrency error pool, it can be seen that the best-performing selection method is truncation selection, and the worst-performing is exponential rank selection and tournament selection. In the concurrency error pool, SUS was never able to outperform truncation selection with respect to the percentage of errors found.

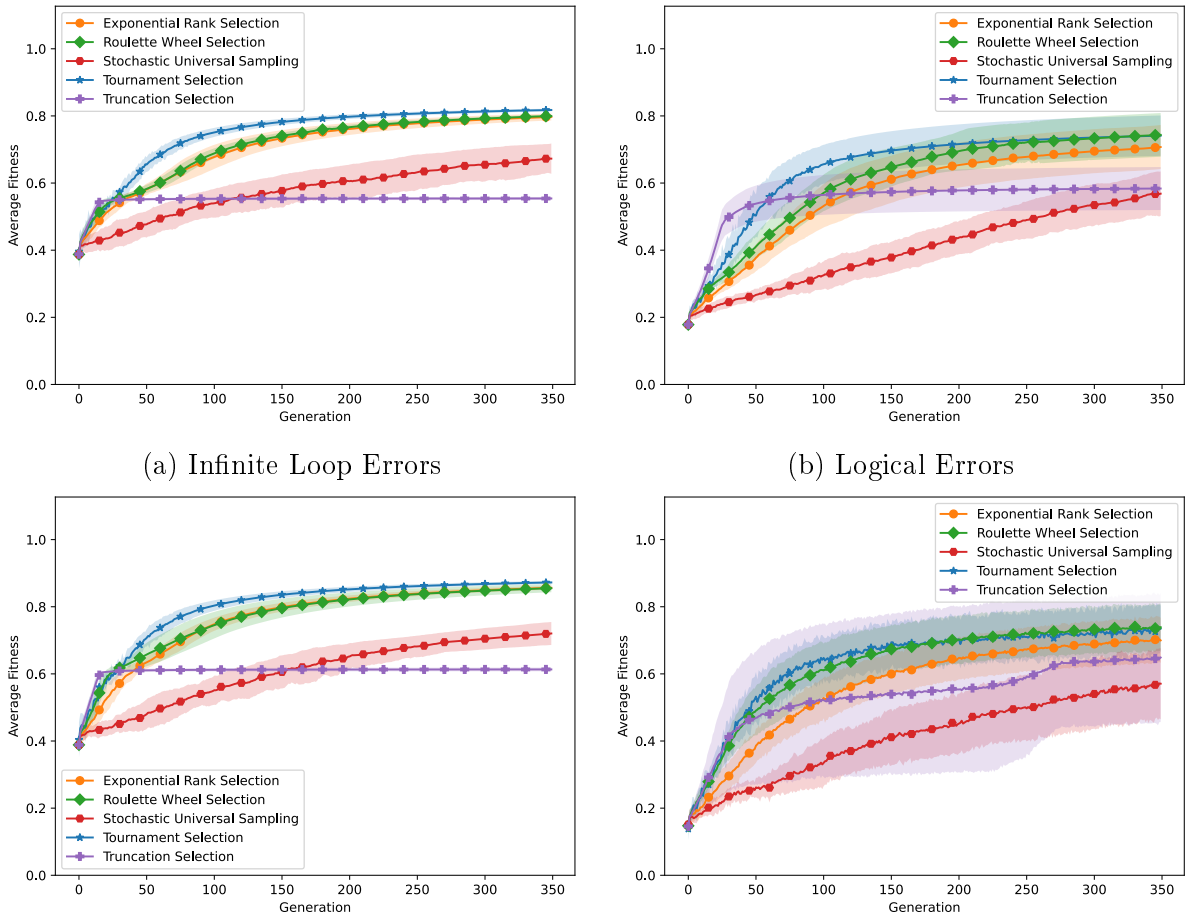
Another metric to consider is, execution duration per generation, introduced that is. This metric measures the time it takes to conclude a single generation. This includes selection, creating offspring, and repopulating the population with the new offspring. The results of the execution duration metric are shown in Figure 8.30, which illustrates the average execution duration per generation for GATLAM equipped with the different selection methods. The shaded area shows the standard deviation across the multiple runs. Tables 8.1 and 8.2 show the average execution duration per generation across multiple runs, and the average execution duration for the final generation, respectively.

The results in Figure 8.30, Table 8.1, and Table 8.2 show a clear trend amongst the different selection methods. Tournament selection was able to execute significantly quicker than any of the other selection methods. This is vital for software testing and automated assessment, as often the luxury of excess time and computing power is limited. The reason for tournament selection's excellent performance is due to tournament selection only evaluating a portion of the population, whereas other selection methods evaluate the entire population. Thus, if a bigger population size is used, the execution time will also drastically increase. For the concurrency error pool, SUS initially performed quicker than truncation selection, but after 200 generations, the average execution time of SUS exceeded that of truncation selection. The same trend can be seen in both the logical and infinite loop error pools, where SUS's execution time was increasing in later generations.

Based on the above discussion, tournament selection is chosen as the average best-performing selection method. Although tournament selection performed average in all of the metrics, tournament selection's excellent performance in the execution time metric

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

compensated for the average performance. For the worst-performing selection method, either truncation selection or SUS are viable as the worst selection method, but ultimately SUS is chosen due to the increased execution time and never being able to find more errors than what truncation selection could.



(a) Infinite Loop Errors

(b) Logical Errors

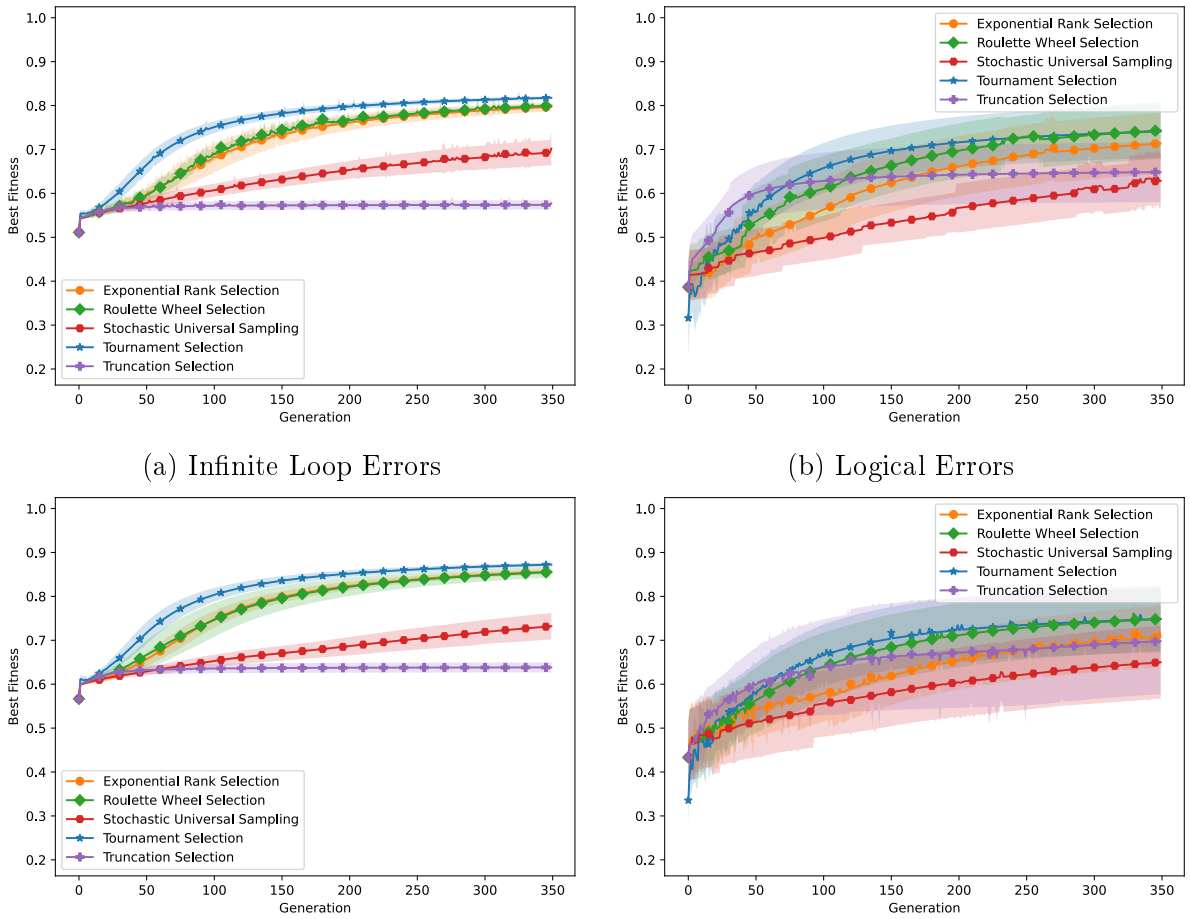
(c) Memory Errors

(d) Concurrency Errors

Figure 8.26: Average fitness for the different selection methods for different error pools.

8.3.8 Tournament Size

This section discusses the results of the tournament size hyperparameter. The following tournament sizes were evaluated: 1, 2, 5, and 10, where 5 is the default tournament size.



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

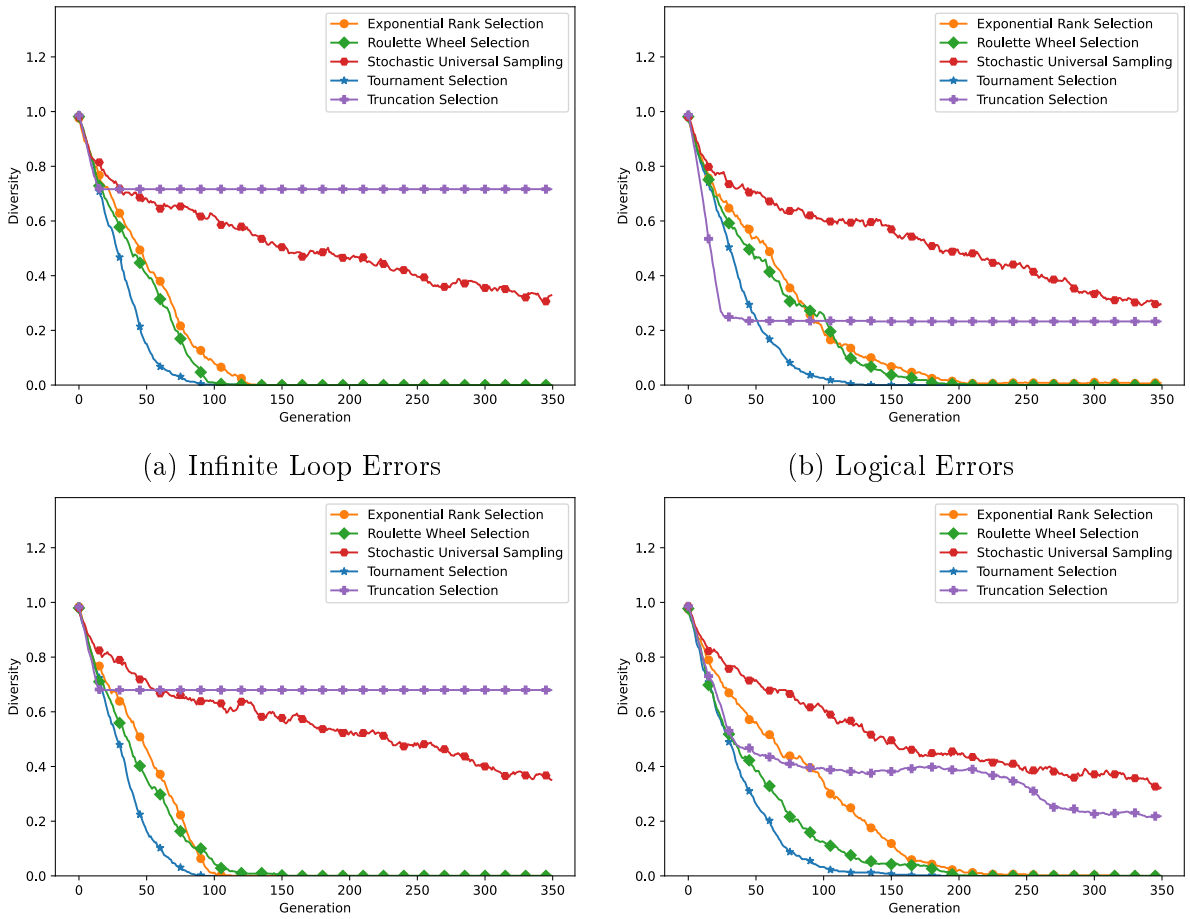
(d) Concurrency Errors

Figure 8.27: Best fitness for the different selection methods for different error pools.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	1.280	0.075	0.054	0.174
Roullette Wheel Selection	1.285	0.076	0.055	0.196
Stochastic Universal Sampling	1.225	0.075	0.054	0.257
Tournament Selection	**0.478**	**0.028**	**0.021**	**0.119**
Truncation Selection	1.294	0.073	0.054	0.255

Table 8.1: Average values for the duration (minutes) for each error pool using different values for the selection method hyperparameter.

The results of the tournament size hyperparameter also have an observable pattern, where the higher the tournament size, the higher the performance. This is due to a higher tournament size evaluating a larger portion of the population, which in turn produces



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

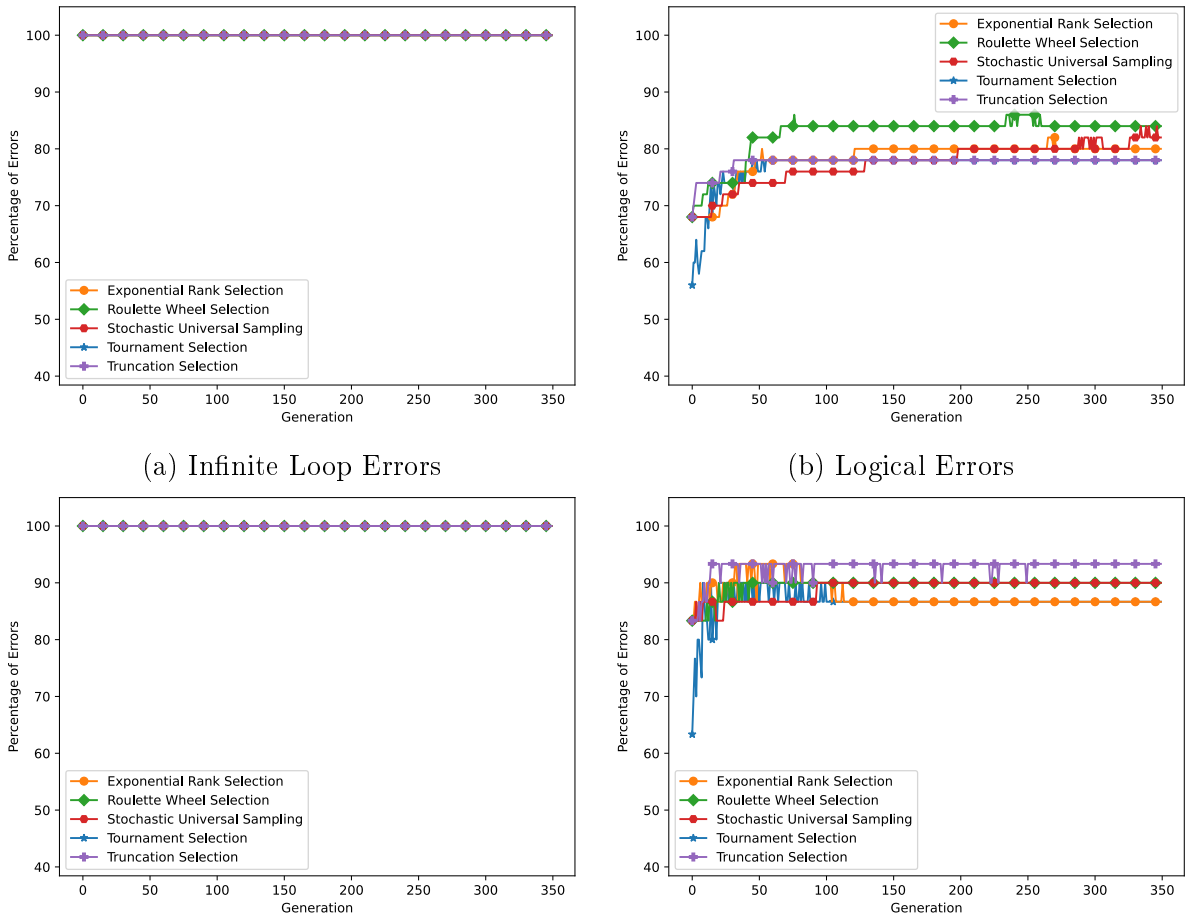
Figure 8.28: Diversity of the population for the different selection methods for different error pools.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	1.293	0.073	0.054	0.154
Roullette Wheel Selection	1.292	0.072	0.055	0.201
Stochastic Universal Sampling	1.272	0.073	0.054	0.322
Tournament Selection	**0.522**	**0.030**	**0.021**	**0.122**
Truncation Selection	1.301	0.073	0.054	0.267

Table 8.2: Final values for the duration (minutes) for each error pool using different values for the selection method hyperparameter.

a bigger mating pool to generate "fitter" chromosomes from. The drawback of a higher tournament size is that the execution time increases.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

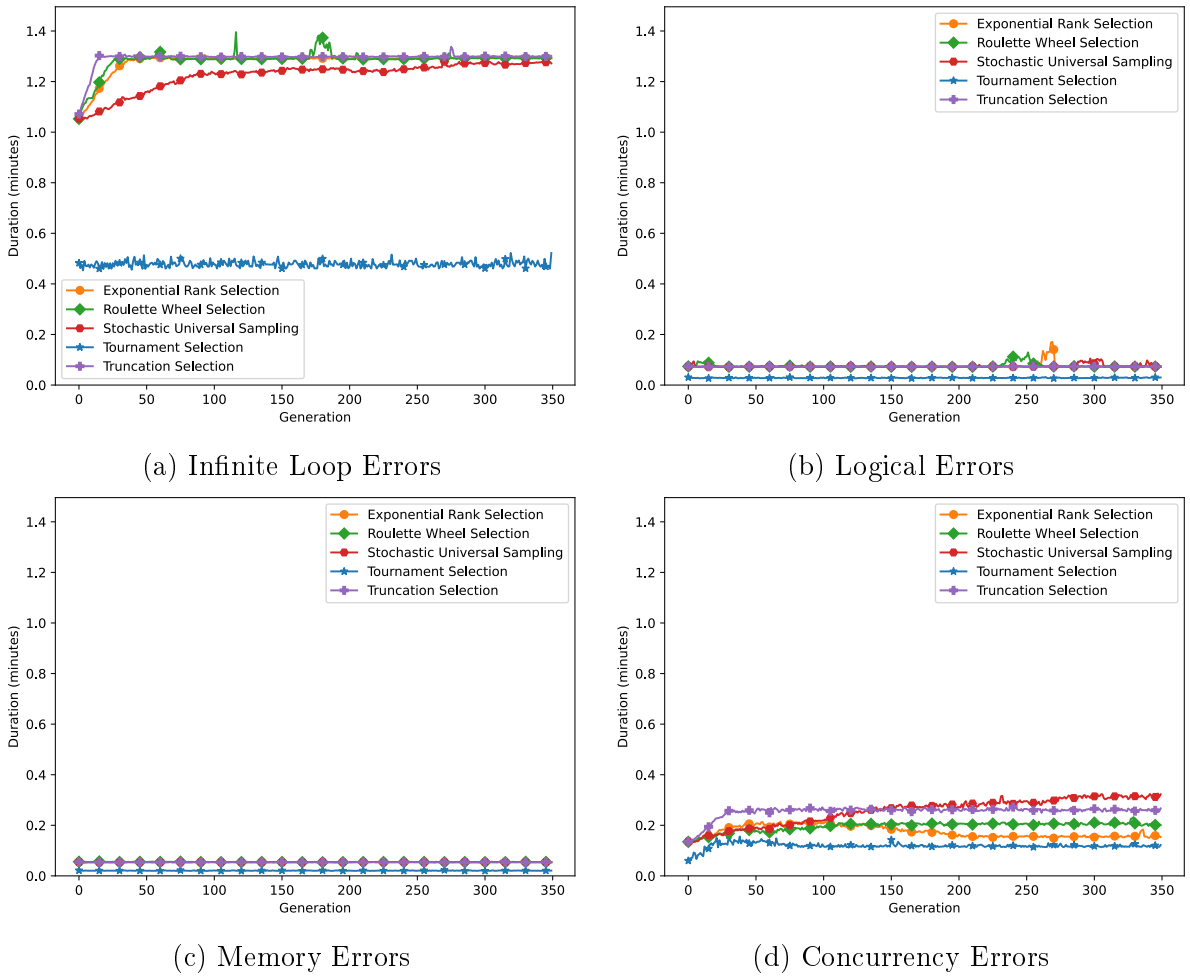
(d) Concurrency Errors

Figure 8.29: Percentage of errors found by the best chromosome for the different selection method for different error pools.

Evaluating the average and best fitness (Figures 8.31 and 8.32), the above-mentioned pattern is already visible in the sense that a tournament size of 1 took 350 generations and still was not able to fully converge on a fitness value, while a tournament size of 10 was able to converge on a fitness value before 150 generations for all of the error pools. A tournament size of 5 was able to achieve a slightly lower final average fitness compared to a tournament size of 10.

Evaluating the population diversity (Figure 8.33), it can be seen that a tournament size of 1 converged at the slowest rate and could only stabilise on the lowest diversity score at around 250 generations. In comparison, a tournament size of 10 was able to

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.30: Average duration (minutes) per generation for the different selection methods for different error pools.

converge quickly on the lowest diversity score before 50 generations. A tournament size of 5 achieved the same performance as a tournament size of 10, but slightly slower, converging on the smallest diversity score before generation 75.

Looking at the percentage of errors found (Figure 8.34), it can be seen that for both the infinite loop and memory error pools, all of the tournament sizes evaluated were able to find all of the errors. In contrast, for the logic error pool, it was seen that a tournament size of 5 and 10 enabled GATLAM to find the highest number of errors. A tournament size of 10 did however find the highest number of errors quicker than a tournament size of 5, albeit by only a few generations. A tournament size of 1 and 2

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

found the lowest number of errors. For the concurrency errors, only a tournament size of 2 was not able to find the maximum number of errors.

The execution time increases as the tournament size increases. To illustrate this, the execution duration metric, introduced in Section 8.3.7, is used. Figure 8.35 shows the average execution duration per generation for GATLAM equipped with different tournament sizes. Tables 8.3 and 8.4 show the average execution duration across multiple runs as well as the average execution duration for the final generation, respectively. Based on Figure 8.35, Table 8.3, and Table 8.4, it can be seen that a tournament size of 10 took the longest to execute per generation for each of the error pools, while a tournament size of 1 took the least amount of time to execute in each generation.

Based on the discussion above, the best tournament size for the error pools is chosen to be 5 due to its average high performance for a relatively short execution time. A tournament size of 1 is chosen to be the worst due to its slow convergence behaviour in both fitness and population diversity.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**0.312**	**0.018**	**0.014**	**0.056**
2	0.478	0.028	0.021	0.119
5	1.015	0.060	0.044	0.209
10	2.080	0.122	0.090	0.414

Table 8.3: Average values for the duration (minutes) for each error pool using different values for the tournament size hyperparameter.

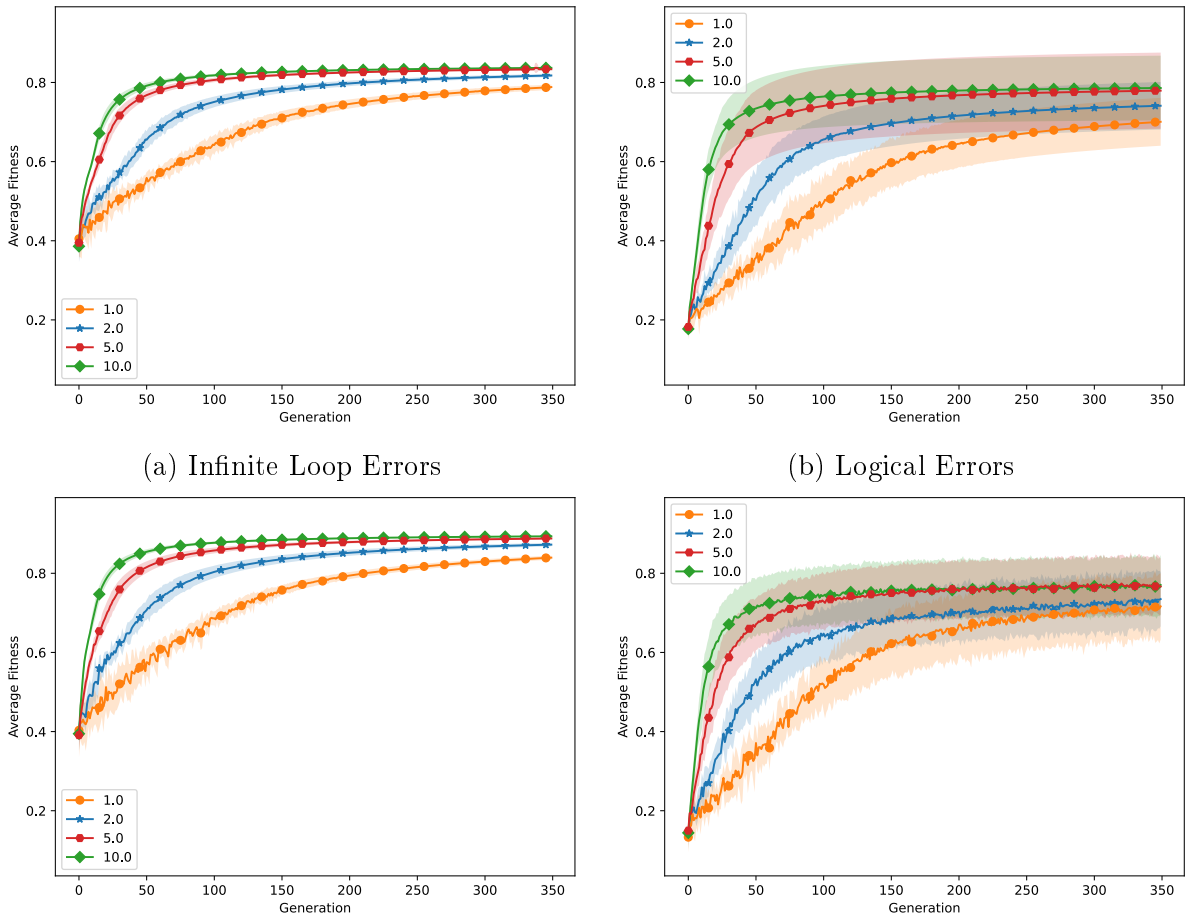
Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**0.306**	**0.018**	**0.014**	**0.052**
2	0.522	0.030	0.021	0.122
5	0.985	0.058	0.044	0.203
10	2.142	0.122	0.090	0.401

Table 8.4: Final values for the duration (minutes) for each error pool using different values for the tournament size hyperparameter.

8.3.9 Summary and Discussion

Table 8.5 contains the average best and worst value for each hyperparameter as discussed in this chapter. The best and worst hyperparameters were chosen based on a mixture of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

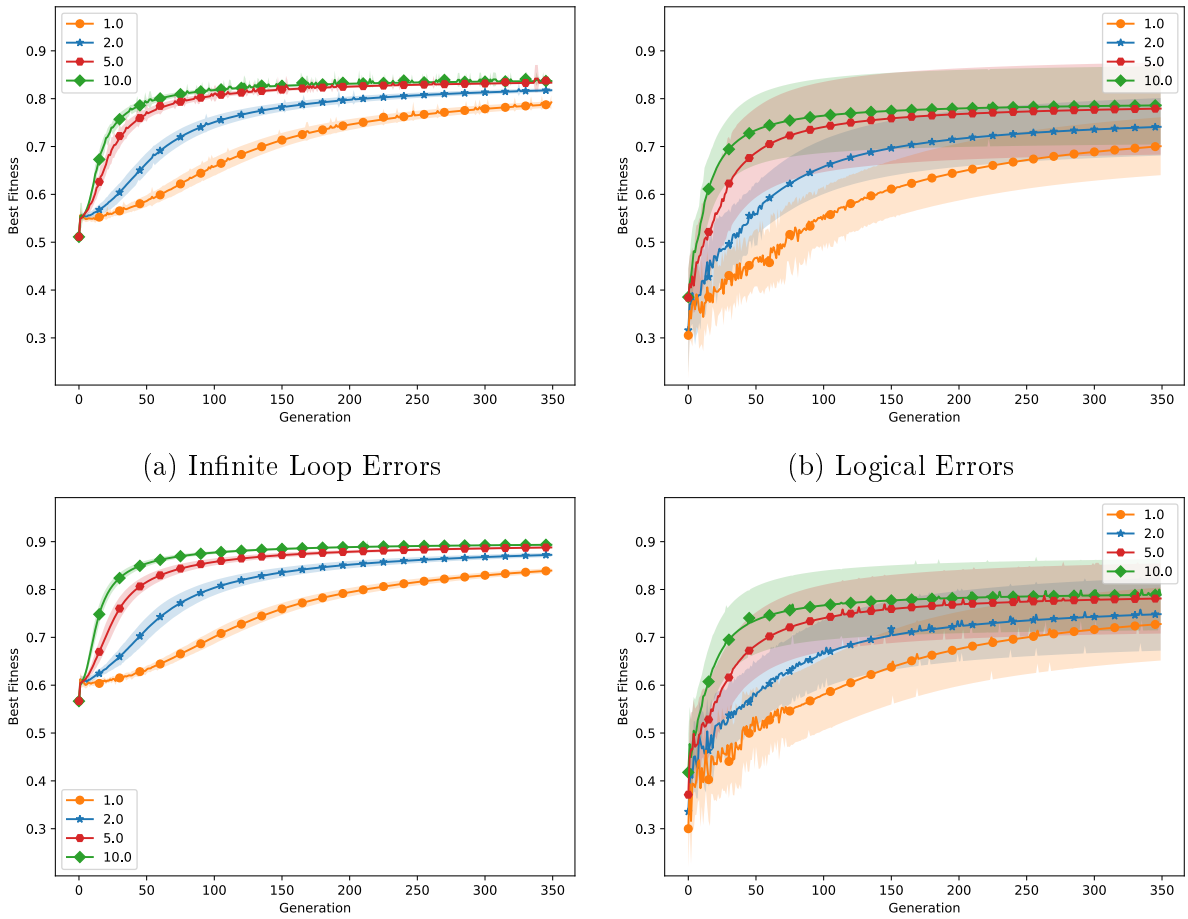
(d) Concurrency Errors

Figure 8.31: Average fitness for the different tournament sizes for different error pools.

the performances of the different hyperparameter values using the metrics discussed in Section 8.1, and, where applicable, the execution duration of a single generation.

The infinite loop and memory errors were the easiest to detect. Students are prone to bad memory management, and this behaviour seems favourable for the task of automated assessment. Further, note that different error pools benefited from different parameter settings. As such, in environments with fewer computational constraints, it may be beneficial to run a few versions of GATLAM in parallel, each tailored for the discovery of a particular kind of error.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

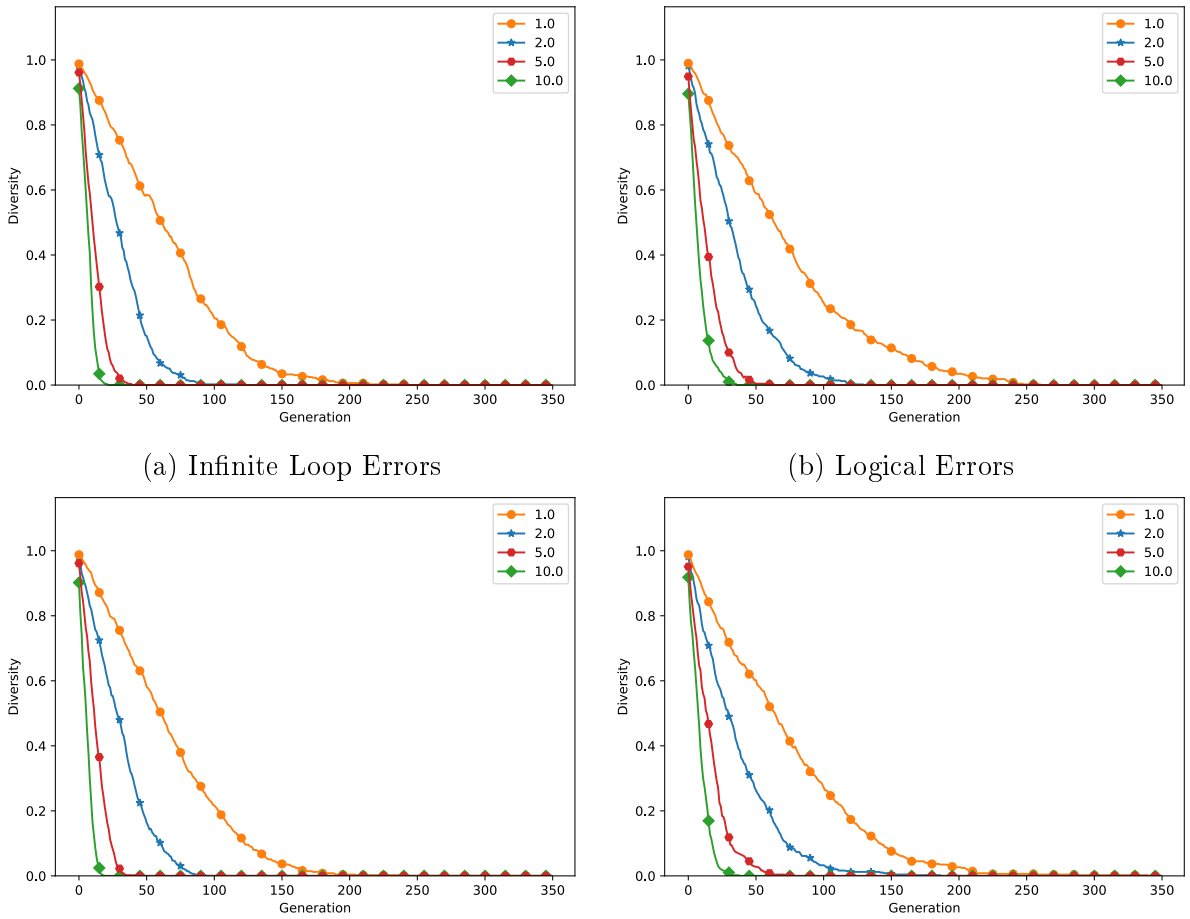
Figure 8.32: Best fitness for the different tournament sizes for different error pools.

It is the recommendation of the author that future works could create more hyperparameter configurations to evaluate. These configurations should be created by choosing the best and worst hyperparameter values solely based on a single metric.

8.4 Summary

This chapter discussed the results of the GA-specific hyperparameter tuning experiments that were performed to establish the influence of the different GA-specific hyperparameters on GATLAM. The chapter first introduced the different metrics that were used

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

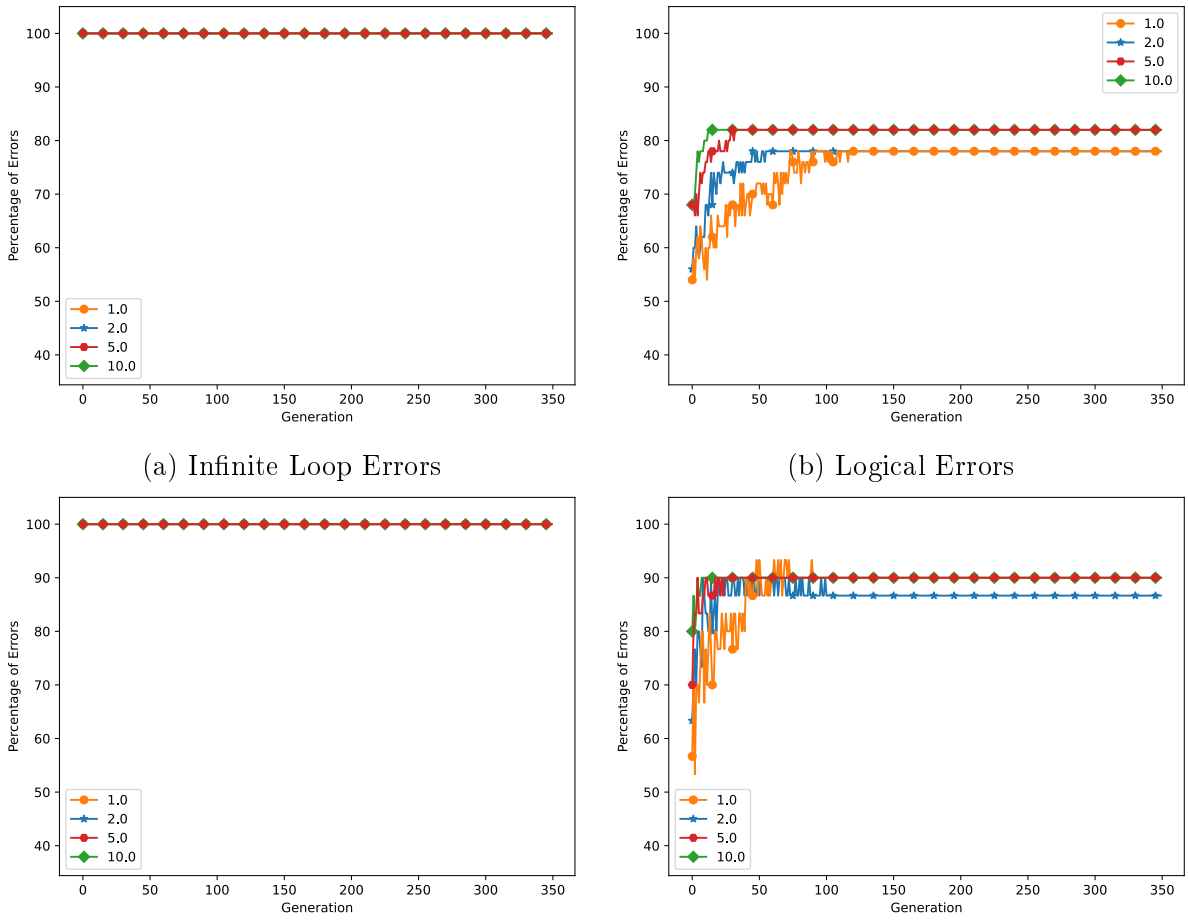
(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.33: Diversity of the population for the different tournament sizes for different error pools.

to evaluate the performance of the different hyperparameter values. After the metrics were defined, the first experiment group, the number of generation experiments, was discussed. It was observed that a maximum generation count of 350 is sufficient for the remaining hyperparameter experiments. The GA-specific experiments were then discussed, and a summary table was provided for the average best and worst GA-specific hyperparameters. Using the average best-performing GA-specific hyperparameters, the results of the fitness function (ω-weight) experiments are discussed in Chapter 9.



(a) Infinite Loop Errors

(b) Logical Errors

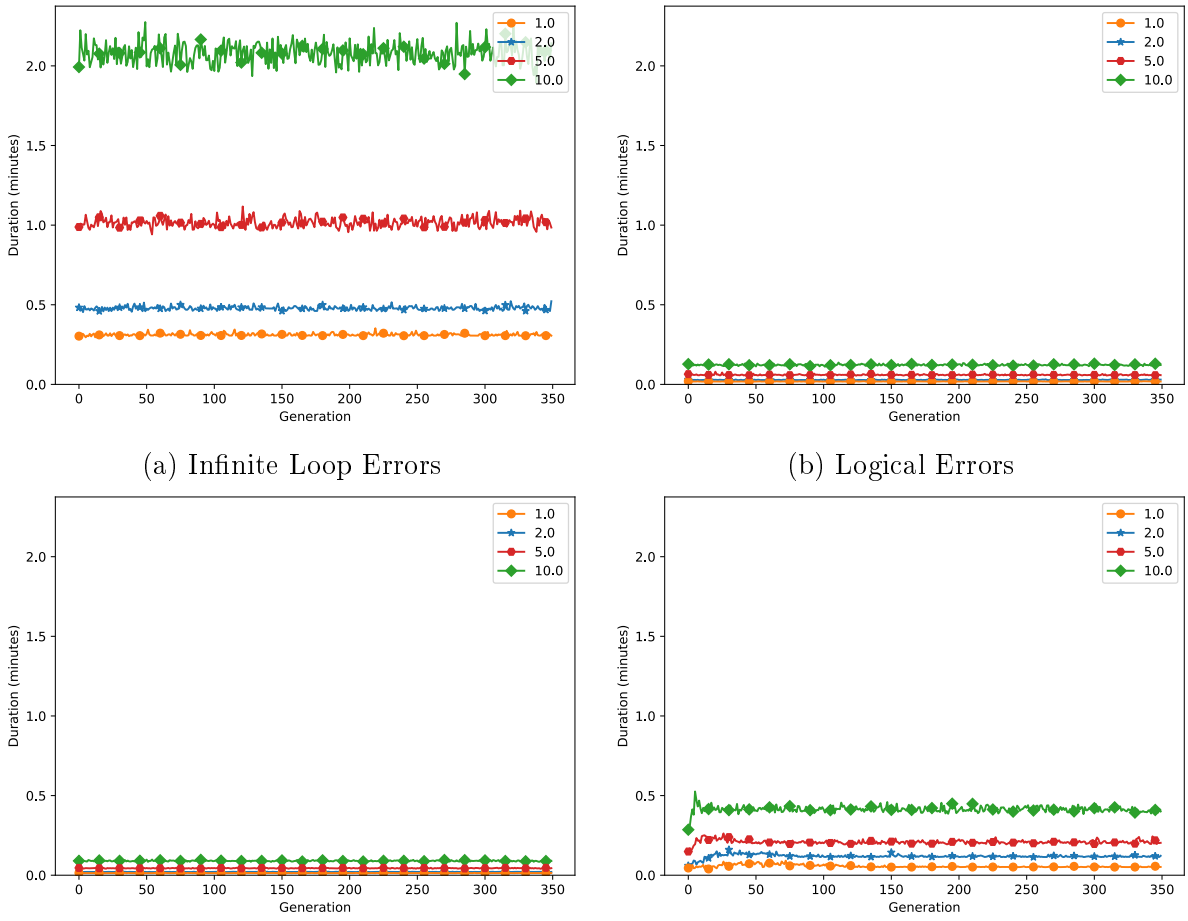
(c) Memory Errors

(d) Concurrency Errors

Figure 8.34: Percentage of errors found by the best chromosome for the different tournament sizes for different error pools.

Hyperparameter	Best value/type	Worst value/type
Reproduction probability	100%	25%
Crossover probability	100%	0%
Mutation type	Single	Random
Population size	50	5
Crossover type	N-Point	One Point
Mutation probability	0%	75%
Selection method	Tournament selection	SUS
Tournament size	5	1

Table 8.5: Summary of the best and worst hyperparameters for the GA specific hyperparameters of GATLAM.



(a) Infinite Loop Errors

(b) Logical Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 8.35: Average duration per generation for the different tournament sizes for different error pools.

Chapter 9

Omega (ω) Weights Experimental Results

This chapter discusses the results of the ω-weight fine-tuning experiments. The purpose of these experiments was to find the approximate best and worst ω-weight configurations for GATLAM using different error pools. This chapter is laid out as follows: Section 9.1 discusses the average fitness results, Section 9.2 discusses the best fitness results, Section 9.3 discusses the population diversity results, and Section 9.4 discusses the percentage of errors found results. Lastly, Section 9.5 contains a short summary of the chapter.

9.1 Average Fitness Value

This section discusses the average fitness results for GATLAM configured with different ω-weights. Figure 9.1 contain the results for the different error pools and ω-weight configurations in the form of Experiment Sets (ES). Tables C.97 and C.99 contain the average and final values of the average fitness metric for the different ω-weights. The tables can be found in Appendix C.2.1.

Figure 9.1 shows that different ω-weights combinations influenced GATLAM's average fitness metric, in the sense that some configurations were able to achieve a near-maximal fitness value while other configurations converged at relatively low fitness values.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

ES 1, 5, and 15 each have only one of the fitness function components contributing to the fitness function. ES 1 only considers the memory component, ES 5 the failure component, and ES 15 the LTL component. ES 5 yielded the highest fitness value close to 1, while ES 1 converged onto a smaller but still relatively high fitness value above 0.89. Lastly, ES 15 converged quickly on a low fitness value, between 0.51 and 0.7. The reason for ES 15's low results can be attributed to not all of the LTL properties being violated in any of the error pools, where maximal fitness is associated with the violation of all LTL properties.

ES 9, 12, 14, and 15 reached a plateau during the final stages of execution, while the remaining ES slowly increased the average fitness during the final stages of execution. This is due to the memory component being omitted from the fitness value calculations in ES 9, 12, 14, and 15. The memory component has the ability to slowly and continually increase the fitness value even when no new information is learned. This also implies that the memory component can easily dominate the fitness value. As such, care should be taken with the weighting given to the memory component.

The default ω-weight configuration yielded a lower average fitness value when compared to ES 1 to 5. This can be attributed to ES 1 to 5 not accounting for the LTL component of the fitness function, and the default configuration having equal ω-weights.

Thus, different ω-weight configurations can influence the fitness function's final result in both good and bad ways. It should be noted that, just because one ω-weight configuration yields a lower fitness value than another configuration, it does not directly imply that the ω-weight configuration associated with the lower fitness value is "worse" than other configurations with a higher fitness value. It only implies that for certain configurations, maximising the fitness value is easier than for others, i.e., some configurations are "stricter" than others. For example, the difference in performance between ES 5 and ES 15 is attributed to the "strictness" of the fitness function employed by ES 15. In ES 5, only one of the LTL properties needs to be violated in each of the active error programs in the error pool to receive a high fitness value, while in ES 15, all of the LTL properties need to be violated across all of the active error programs in the error pool for the chromosome to receive a high fitness value. This "strictness" needs to be taken into account when setting up the ω-weights for the fitness function of GATLAM.

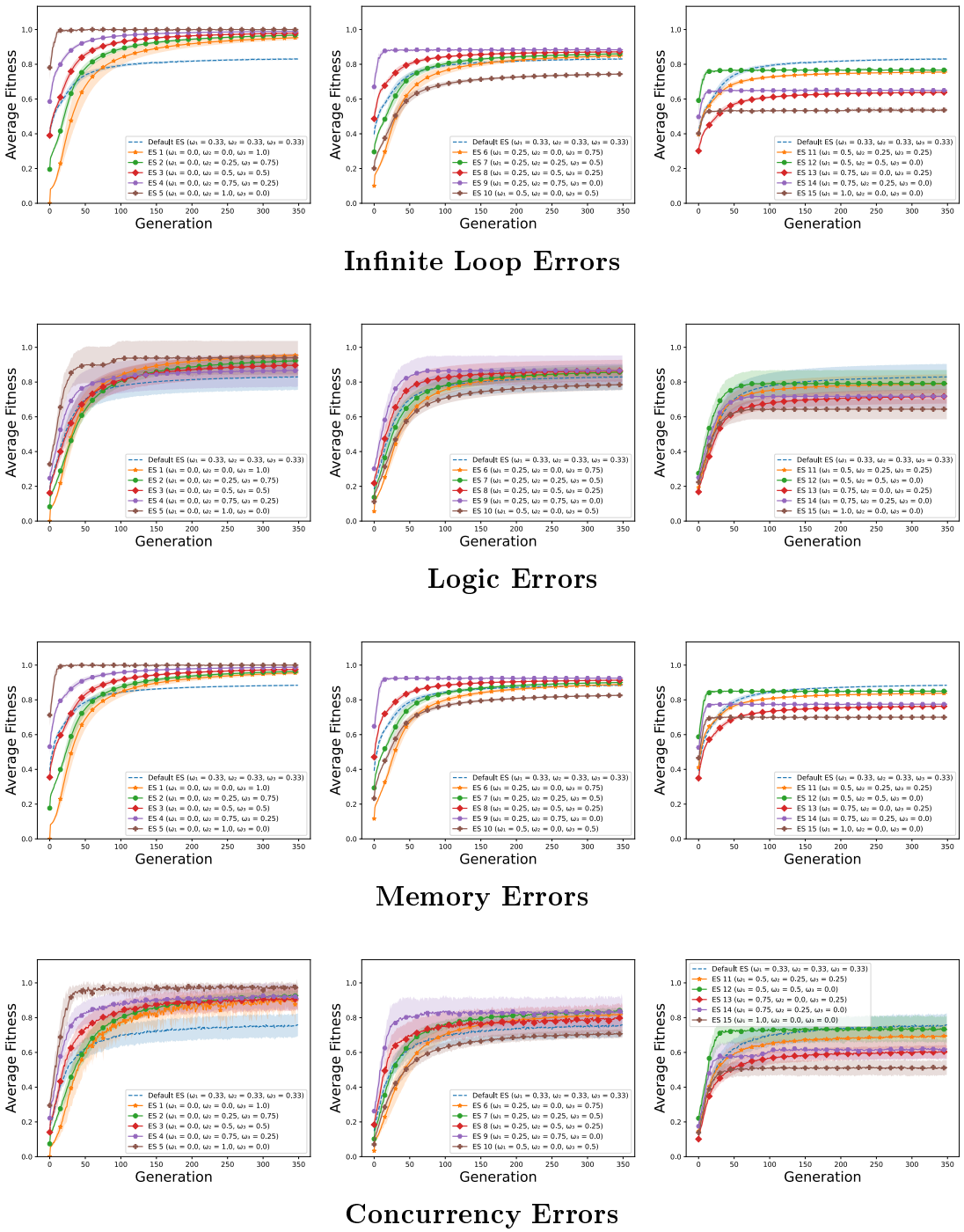


Figure 9.1: Average fitness value with GATLAM configured to use different ω-weights for different error pools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

9.2 Best Fitness

This section discusses the best fitness results for the different ω-weight combinations. Most of the results that appear in this section are similar to the results discussed in Section 9.1, and as such, only key points are highlighted.

From Figure 9.2, it can be seen that in ES 5, 9, 12, 14, and 15, the best fitness values were relatively stable throughout execution. This can be attributed to these ES having an $\omega_3 = 0$ (Memory component), which implies that the memory component is not contributing to the overall performance. This observation further supports the idea that the memory component can over-inflate the chromosome's actual performance and should be used with care.

ES 1 to 5 converged on similar fitness values, which is attributed to all of them having a $\omega_1 = 0$ (LTL component). ES 5's configuration achieved the highest best fitness value, which is attributed to the lack of a memory component. The same trend can be seen in the experiments using the configurations of ES 6 to 9. In these experiments, $\omega_1 = 0.25$, which resulted in each of the weight combinations converging onto roughly the same value as what ES 9 was able to achieve. With ES 9, no memory component was used, and the fitness function solely relied on the other components.

ES 11 and 13 exhibited consistently increasing best fitness values compared to ES 12, 14, and 15. This is attributed to ES 11 and 13 having $\omega_3 = 0.25$ whereas ES 12, 14, and 15 have an $\omega_3 = 0$.

9.3 Population Diversity

This section discusses the population diversity results of the different ω-weight combinations. The supporting tables can be found in Appendix C.2.3.

Upon inspection of the results in Figure 9.3, it is clear that GATLAM, with the majority of the ω-weight combinations, converged onto relatively low population diversities. ES 5, 9, 12, 14, and 15 all struggled to converge, which is attributed to the lack of a memory component in these configurations. For each Experimental Set with $\omega_3 \neq 0$,

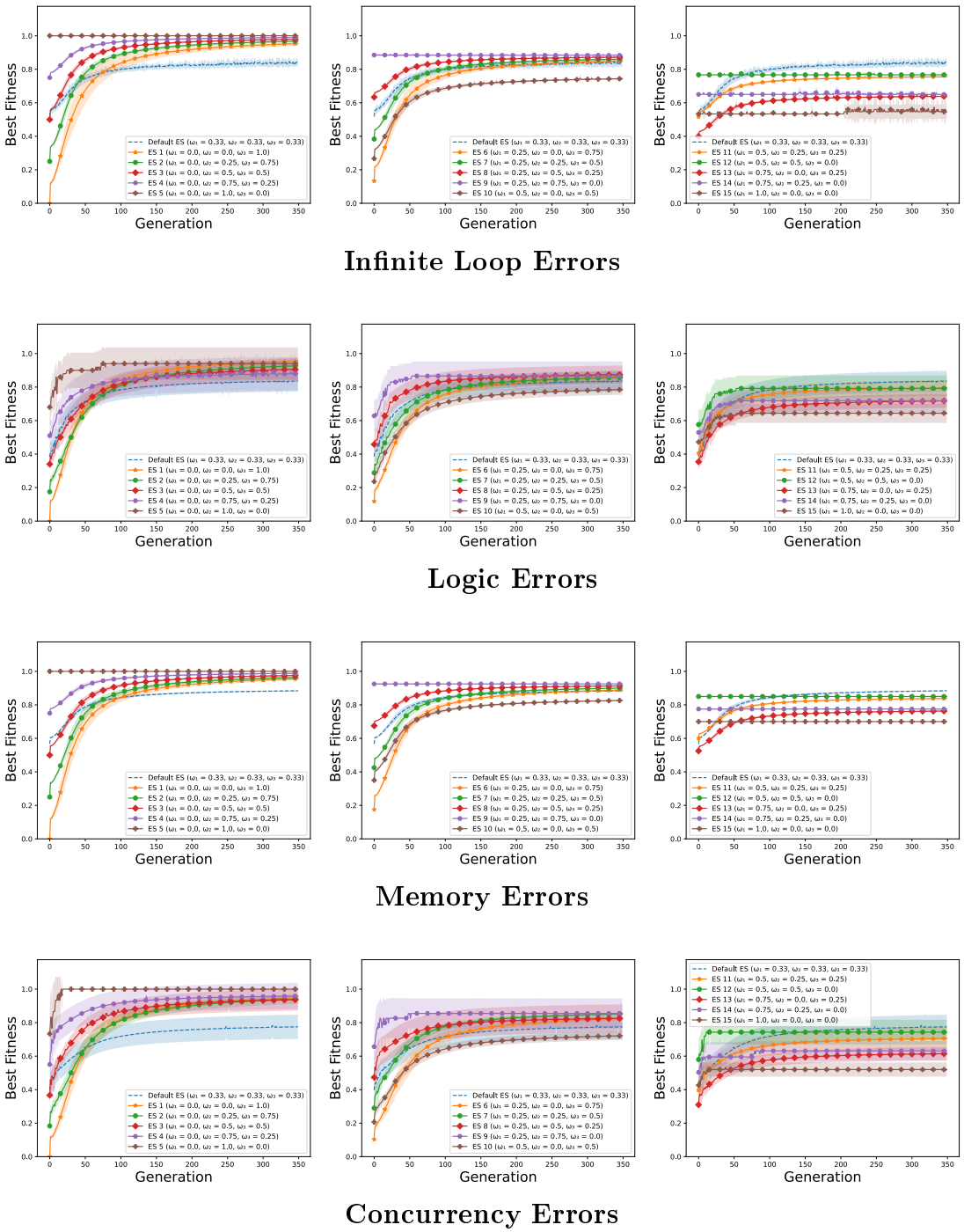


Figure 9.2: Best fitness value per generation with GATLAM configured to use different ω-weights for different error pools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

GATLAM converged quickly, which shows that although the memory component can over-inflate a chromosome's fitness, it also helps GATLAM converge at a quicker pace.

This leads to the conclusion that the memory component helps reduce the search space, which in turn speeds up the search, but can also negatively influence GATLAM's performance if incorrectly configured. For the logic and concurrency error pools, ES 1 converged the quickest. ES 1 consists of just the memory component contributing to the fitness value, which supports the idea that a poorly configured memory component can negatively affect the convergence of GATLAM, as a chromosome's fitness is incorrectly inflated, leading to possible premature convergence.

Based on the population diversity results depicted in Figure 9.3, it is observed that the configurations used in the ES that enabled GATLAM to converge on relatively low diversities did so before 150 generations. This observation is reflected in Sections 9.1 and 9.2, where the rapid increase of fitness function value stopped before 150 generations. After 150 generations, GATLAM, with various ω-weight combinations, experienced a slow increase in the fitness value, likely due to the memory component's contribution to the fitness value increasing with each generation.

9.4 Errors Found

This section discusses the results of the percentage of errors found for GATLAM configured with the different ES. The supporting tables can be found in Appendix C.2.4.

The first observation from Figure 9.4 is that ES 1 found the lowest number of errors in all of the error pools, even in the infinite loop and memory pool, which were configured to have errors that are easy to detect. This is a direct result of the memory component over-inflating the fitness of the chromosomes early on in execution.

For the infinite loop and memory error pools, ES 2 to 15 found all of the errors. A reason for GATLAM's success in these error pools is attributed to the "ease" of discovering the errors in these pools. GATLAM, configured with ES 5, experienced a slight deviation in both the infinite loop and memory error pools for two generations, which resulted in it finding an average of 99.99% of the errors across multiple runs. This deviation can be attributed to a single run not finding all of the errors in a single generation,

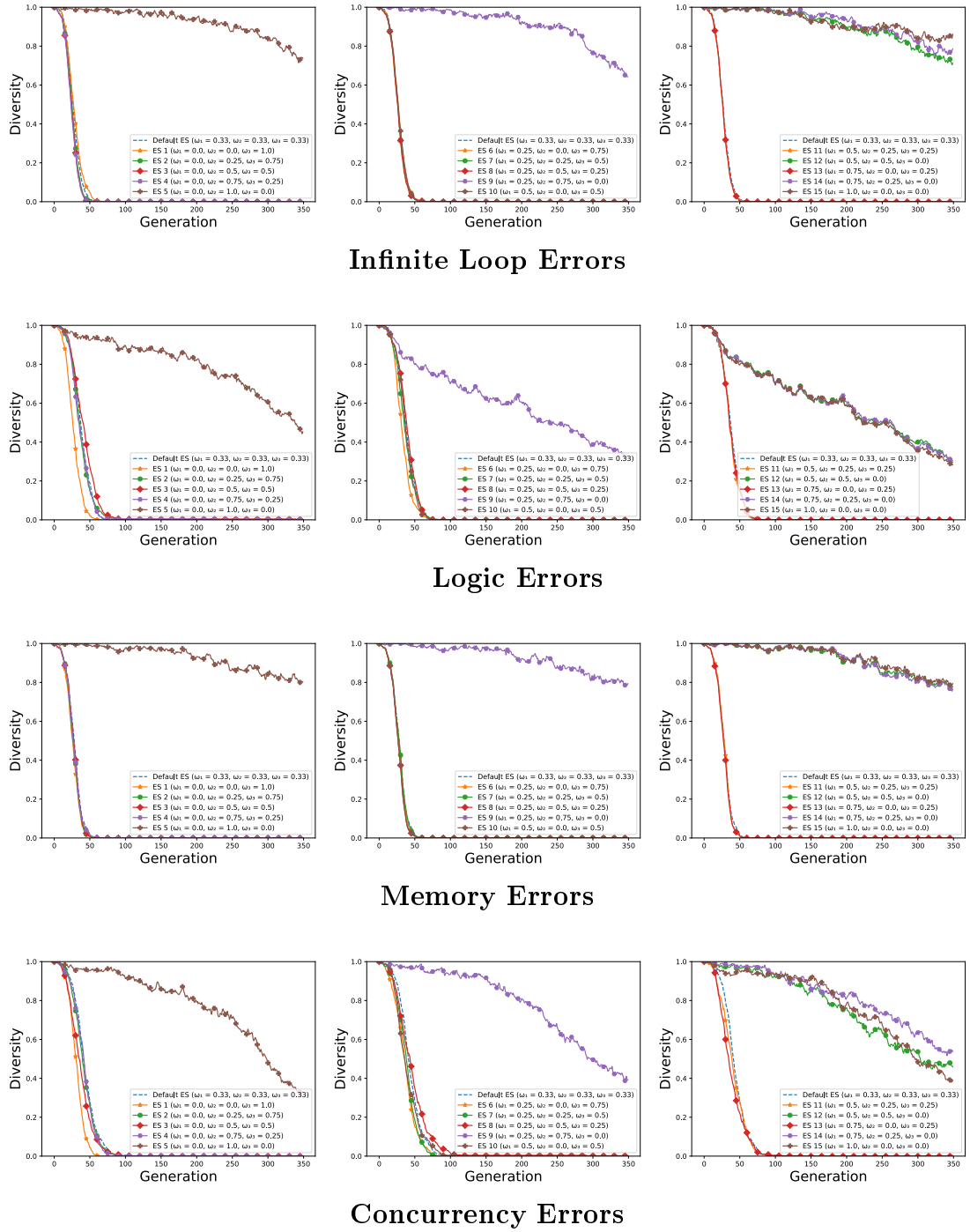


Figure 9.3: Diversity in the population with GATLAM configured to use different ω-weights for different error pools.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

as well as the fact that GATLAM with ES 5 relies only on the failure component to guide the search.

Investigating the logic error pool, we see that, apart from ES 1, the other ES were able to find more than 80% of the errors by generation 100. An interesting pattern is that ES 2 to 5 resulted in a wider spread of the final percentage of errors found, compared to ES 6 to 10 and 11 to 15. This is attributed to ES 2 to 5 not accounting for the specific LTL properties that were violated; in other words, the LTL component was not contributing to the fitness function. It can be seen that the higher the LTL component's contribution is, the smaller the variation in the final percentage of errors found becomes. This implies that although including the LTL component increases the "strictness" of the fitness function, it does yield more robust results for the logic error pool.

For the concurrency error pool, ES 5's configuration enabled GATLAM to find the highest number of errors, with ES 7's configuration coming second. The pattern that was observed in the logic error pool concerning the influence of the LTL component of the fitness function is not shared in the results of the concurrency error pool. Looking at the final average percentage of errors found, it can be seen that only ES 1 did not find more than 85% of the errors.

9.5 Discussion

This chapter presented the analysis of the influence of the ω-weights on the performance of GATLAM. From the results discussed in this chapter, it is difficult to choose a single optimal ω-weight configuration that performs best in all of the metrics across all of the error pools. As such, a single configuration that had a good performance in every metric across all of the error pools is chosen. This is the configuration used in ES 7, which assigned $\omega_1 = 0.25$ (LTL component), $\omega_2 = 0.25$ (Failure component), and $\omega_3 = 0.5$ (Memory component). The configuration used in ES 7 enabled GATLAM to converge on a low population diversity before 75 generations in all four of the error pools, while still being able to achieve high fitness values. ES 7's configuration enabled GATLAM to also find the majority of the errors in all of the error pools, even being able to find nearly all of the errors in the concurrency error pool. For the worst-performing Experiment Set,

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

the choice is easier. ES 1's configuration is chosen due to its poor performance. ES 1 consistently found the fewest errors by a large margin compared to the other ω-weight configurations. ES 1 created a situation where the performance of GATLAM (fitness value) was completely the opposite of the performance of GATLAM in terms of the percentage of errors found. This "delusion" is the main reason why ES 1 is declared the worst performing ω-weight combination.

Using the best and worst combinations of GA-specific hyperparameters and ω-weight combinations, the alternative tool comparisons and real-world experiments are discussed in Chapter 10.

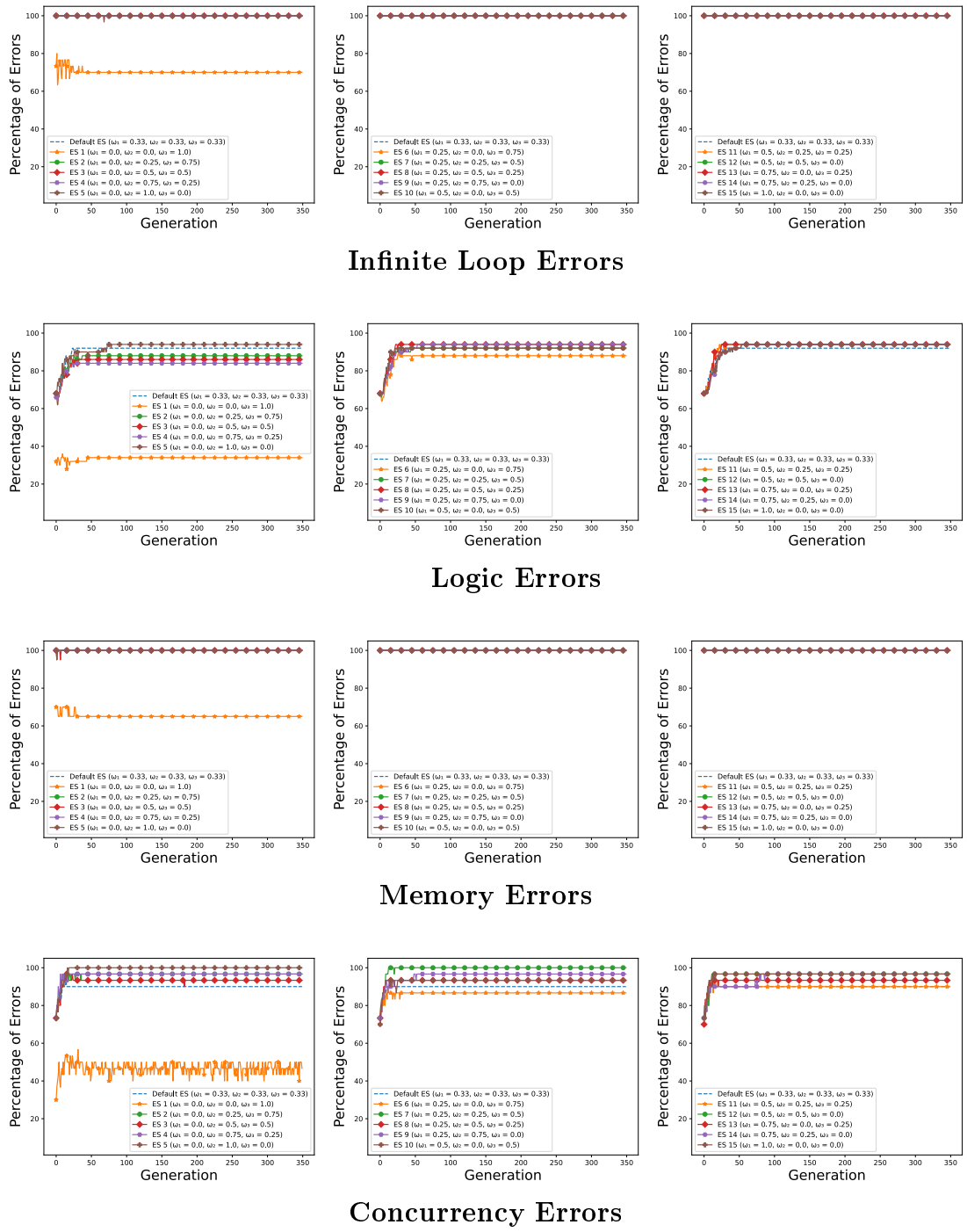


Figure 9.4: Average percentage of errors found by GATLAM when configured to use different ω-weights for different error pools.

Chapter 10

Benchmarking and Real-World Experimental Results

To evaluate the performance of GATLAM, two groups of experiments were conducted. The first group, referred to as the *benchmarking experiments*, compared GATLAM with different configurations to alternative automated software test generation methods. The second group compared GATLAM with different configurations to the alternative automated assessment strategies utilising a real-world CS1[1] level programming assignment[2] completed by students in 2022. A key difference between the experiments in this chapter compared to the experiments discussed in Chapter 8 is that the number of generations the different methods/strategies were allowed to execute for was reduced to 150. From observations made in Chapter 8, it is clear that GATLAM converges around 150 generations. Thus, to avoid wasting computing resources, the number of generations was reduced to 150.

This chapter is laid out as follows: Section 10.1 discusses the results of the benchmarking experiments, Section 10.2 discusses the mass single run experiment for the real-world use case, and Section 10.3 discusses the selected multiple run experiment for the real-world use case. Lastly, Section 10.4 contains a summary of this chapter.

[1]First-year computer science module.

[2]The specification for the programming assignment can be found in Appendix B.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.1 Benchmarking Experimental Results

The purpose of the benchmarking experiments was to evaluate how GATLAM performs compared to fuzzing[3], as well as a GA combined with the code coverage software testing metric. Both of these methods are recognised methods for automatically generating software tests for a SUT as discussed in Chapter 3.

Two metrics are used to compare the different automated software test generation methods, namely the average percentage of errors found and the execution duration in minutes. Both of these metrics were introduced and discussed in Chapter 8. The reason why the fitness value and diversity are not considered is due to GATLAM and CCGA utilising different fitness functions, which are not comparable, as they rely on different inputs. GATLAM's fitness function relies on the hard-coded implementation of the LTL properties to determine the quality of the software test generated by the decoded chromosome whereas CCGA utilises the lines of code that were executed to determine the quality of the software test generated by the decoded chromosome. RNG generates chromosomes randomly and does not use a fitness function to evolve the population.

This section is laid out as follows: Section 10.1.1 discusses the results for the average percentage of errors and Section 10.1.2 discusses the results for the average execution duration. Lastly, Section 10.1.3 summarises the main findings from the benchmarking experiments.

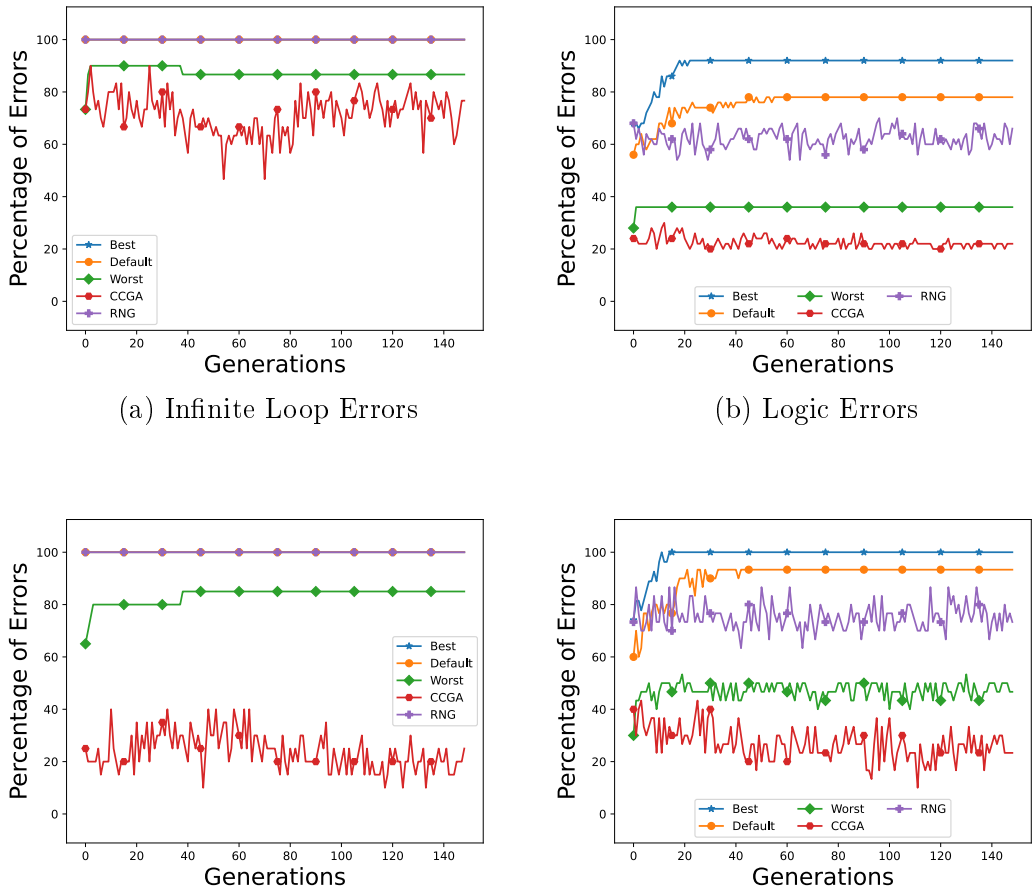
10.1.1 Average Percentage of Errors Found

Figure 10.1 shows the average percentage of errors found across multiple runs for different automated software test generation methods using different error pools.

For all of the error pools, GATLAM's best configuration was able to find all of the errors before 50 generations. GATLAM's default configuration was able to find over 70% of the errors by generation 80. GATLAM's worst configuration, which contains the worst hyperparameter values/types, was also able to perform well in the infinite loop and memory error pools. This good performance can be attributed to the relative

[3]Using random number generators to create software tests

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) Infinite Loop Errors

(b) Logic Errors

(c) Memory Errors

(d) Concurrency Errors

Figure 10.1: Different automated software test generation methods' average percentage of errors found for different error pools. Best, Worst, and Default refer to the different configurations of GATLAM.

ease of finding the errors in these error pools. In the logic error and concurrency error pools, it can be seen that GATLAM's worst configuration was not able to achieve good performance, but was still able to find some errors.

When considering RNG's performance, it can be seen that in the infinite loop and memory error pool RNG was also able to find all of the errors. In the logic error pool, RNG found between 50% and 70% of the errors for the entire execution, continuously oscillating between these two bounds. The reason for the oscillation is attributed to the randomness of RNG chromosome creation. In all of the error pools, GATLAM, with the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

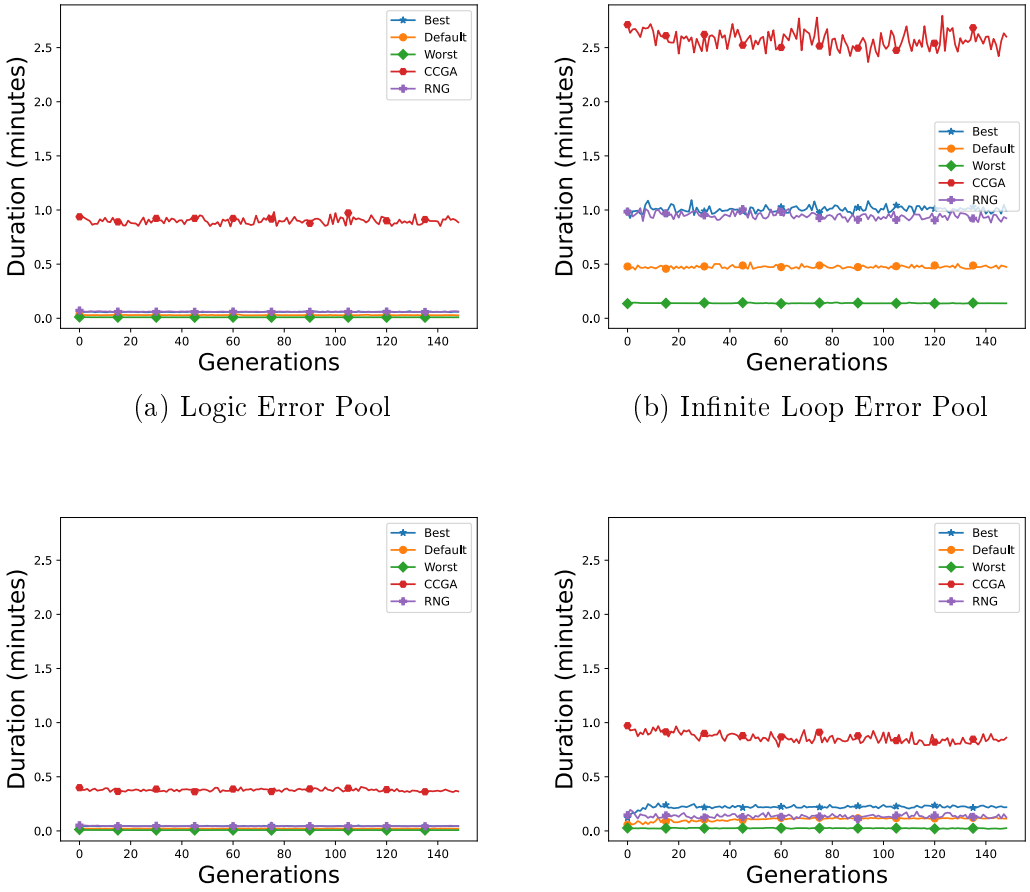
best and default configurations, was able to match or surpass RNG's performance after 40 generations, whereas RNG was able to consistently find more errors than GATLAM's worst configuration and CCGA. Note that RNG used GATLAM's fitness function with the default fitness function configuration for selecting the "best" chromosome. This "best" was then used to determine the number of errors found. In each generation, a purely random set of chromosomes was created, and interpreted to test cases for the error pools. GATLAM's fitness function was then used to determine the best random chromosome. This was done to evaluate the fitness function without the GA, and it can be seen that the fitness function used by GATLAM is able to detect violations of the LTL properties even when the chromosomes are purely random.

CCGA, using the same GA configuration as GATLAM, performed poorly, which is attributed to the fact that if certain errors occur, such as a segmentation fault, no code coverage data can be calculated by gcov. CCGA is dependent on code coverage tools to calculate fitness, and a default dummy value is used if code coverage cannot be calculated. As explained in Chapter 7, a default value of 0 was chosen when no coverage data could be collected. Another reason CCGA performed poorly is due to CCGA trying to maximise the length of the execution path, but if an error lies on a shorter execution path, that path may be skipped in favour of a longer execution path.

10.1.2 Execution Duration

Figure 10.2 shows the average execution duration for each generation for different automated software test generation methods over multiple runs. In all cases, CCGA had the longest average execution duration compared to the other automated software test generation methods. This is attributed to the delay caused by the code coverage tools in completing their analysis and formulating the results.

It can also be seen that GATLAM, with the worst configuration, had the shortest average execution duration per generation in all of the error pools. This is attributed to the small population size of the worst configuration. It is also important to note that RNG was not quicker than GATLAM's default configuration, which is attributed to RNG having to create a new set of chromosomes after each generation.



(a) Logic Error Pool

(b) Infinite Loop Error Pool

(c) Memory Error Pool

(d) Concurrency Error Pool

Figure 10.2: Different automated software test generation methods' average execution duration for different error pools.

10.1.3 Summary

GATLAM, with the best and default configurations, outperformed CCGA in terms of finding errors and execution duration per generation. RNG was not able to outperform GATLAM with the best configuration, and relied on GATLAM's fitness function to determine the best chromosome. All of GATLAM's configurations detected more errors than CCGA, but RNG detected more errors than GATLAM's worst configuration. This leads to the conclusion that GATLAM's hyperparameter configuration is vital to performance, yet the default configuration of GATLAM is still able to find the majority of

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

the errors. It should be noted that due to the repeated process of selecting, evaluating and generating chromosomes performed by GATLAM, RNG, and CCGA, instantaneous results cannot be obtained. The process is time-consuming, which can be a negative factor for any time-pressed software engineer. In terms of automated assessment, this factor also violates the quick feedback benefit that was discussed in Chapters 4 and 5. Arguably, while manual testing might seem like an alternative, it would not necessarily be faster or more accurate. Thus, despite the computational cost, GATLAM still has merit due to its systematic approach to error detection.

10.2 Mass Single Run Real-World Experimental Results

This section discusses the results of the mass single-run real-world experiments. In these experiments, 52 first-year students volunteered their code submissions for a practical assignment. These volunteered submissions were evaluated using automated assessment strategies (previously known as automated software test generation methods). Due to the ethical clearance obtained, the original assessment of the assignments could not be included in this dissertation, as this could lead to the identity of the student being revealed. This is a realistic use-case scenario for GATLAM. For the experiments in this section, GATLAM is configured in ISS mode and not SS mode which was used in previous experiments. The reason for this is due to the availability of an instructor-provided solution, which GATLAM can use to guide its search.

This section is laid out as follows: Section 10.2.1 discusses whether, for each generation of each student's code evaluation, the different automated assessment strategies could find an error or not. Section 10.2.2 discusses the percentage of the instructor's output that the student's output matched before the first error occurred. The overall types of errors found by the different automated assessment strategies are discussed in Section 10.2.3, and the final types of errors found are discussed in Section 10.2.4. Section 10.2.5 discusses an example of the feedback that GATLAM can provide to the students, which is contained in Appendix D. Lastly. Section 10.2.6 contains a summary for this section.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.2.1 Errors Found Per Generation

The errors found per generation metric, look at each student's code evaluation, and for each generation, whether an error was found or not. In order to determine if an error was found or not, a specially designed program is used, which processes the results of GATLAM. This program is similar to the program used for the percentage of errors found metric in Section 8.1.4. Figure 10.3 shows the errors found per generation for each student, by each of the automated assessment strategies. Red represents that an error was found, blue represents that no error was found and grey represents that a compilation failure occurred. This corresponds to a hot-cold colour scheme where an error is deemed as "hot" and no error found is deemed as "cold". This representation is useful, as it indicates how successful each of the automated assessment strategies was at detecting errors per generation for each student.

During the analysis of the results, the cause of the compilation failure was determined to be a print function that was not fully implemented by some students, and due to the way C++ programs are compiled, the incomplete function is only compiled if and only if the function is invoked by the `main`[4] function. The testing `main` function is created by the interpreter which, maps gene values in the chromosome to function calls. Further research can be done into automating the creation of the rules by which the interpreter performs this mapping.

Comparing the errors found across the different automated assessment strategies (Figure 10.3), it can be seen that only GATLAM was able to find the compilation errors. It should be noted that all of the automated assessment strategies, apart from CCGA, used the same interpreter. CCGA used a modified version of the interpreter, with the only difference being that it also stored the code coverage metric. RNG was able to consistently find errors in each generation of each student's code evaluation. This can be attributed to GATLAM's fitness function being used to choose the "best" chromosome to be used for this metric, and the general effectiveness of randomised testing. Although RNG was able to consistently find errors, it failed to find test cases that caused compilation failures, which may be detrimental to assessment. Compilation

[4]The main function is the driver function for C++ programs.

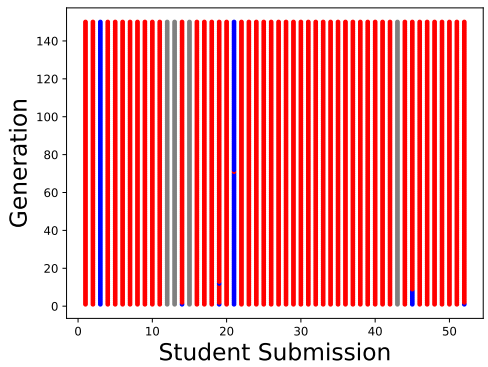
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

failures are common in student submissions, and an automated assessment strategy needs to be able to identify such cases.

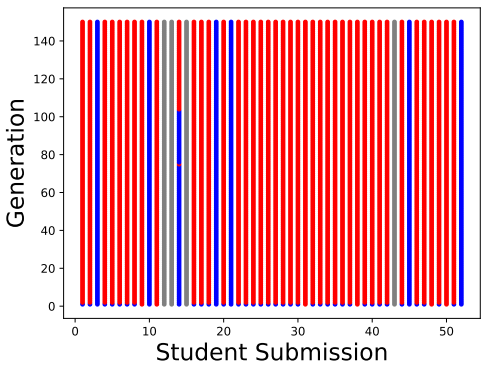
An analysis of Student 21 revealed a case where RNG was able to identify an error that none of the GATLAM configurations detected. The discrepancy was due to a single print statement in the student's output differing from the expected output produced by the instructor's solution. This difference likely stemmed from a misunderstanding of the function responsible for generating the incorrect output. The reason RNG succeeded in detecting this error, while GATLAM did not under any configuration, may be attributed to the randomness inherent in RNG, which allowed it to explore a broader and less structured input space. The same error occurred in the bands in the results for CCGA around the 42^{nd} generation.

CCGA was the only automated assessment strategy that triggered errors in the instructor's solution. These errors occurred in cases not anticipated by the instructor's implementation, as it was not tested for during the original assessment of the assignment. However, these errors did not affect CCGA's performance, as only the coverage data from the student solution was used to calculate chromosome fitness. As previously noted, when the instructor's solution encountered an error, the resulting instructor output was inspected and compared to the student's output to determine if there were any output errors. Overall, CCGA was not effective at identifying errors compared to GATLAM (under any of its configurations) or RNG. CCGA's poor performance is again attributed to the limited effectiveness of the code coverage metric as a fitness function.

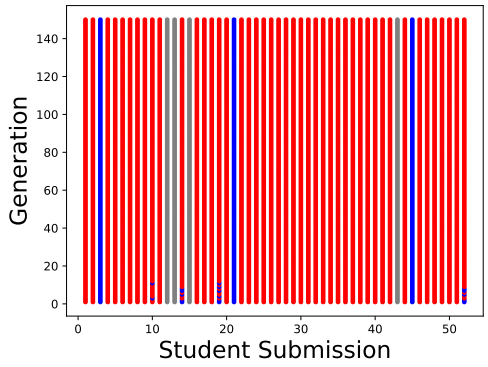
Based on the results in Section 10.1 and this section, it can be seen that the different GATLAM configurations produced similar error-finding capabilities across different SUT. This fact speaks to GATLAM's robustness and shows that GATLAM is a viable option for automated assessment, as it mostly fulfils the consistency requirement for automated assessment. Although GATLAM's worst configuration caused GATLAM to perform poorly compared to GATLAM with best and default configurations as well as RNG, it still significantly outperformed CCGA. This leads to the conclusion that even a poorly configured GATLAM still has the ability to find errors in the majority of student submissions.



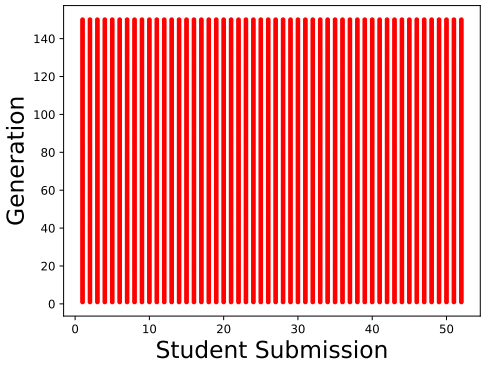
(a) GATLAM: Best



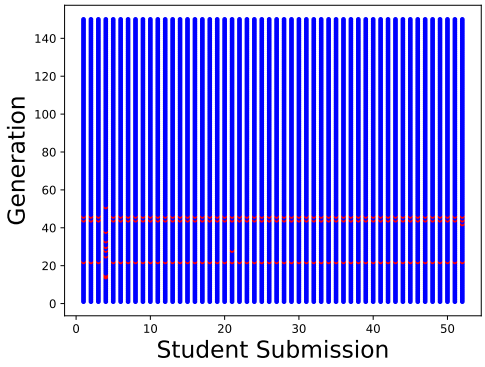
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Figure 10.3: Errors found per student per generation. Red: error found, blue: no error found, grey: compilation failure.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.2.2 Percentage of the Output that Matched Before First Error Occurred

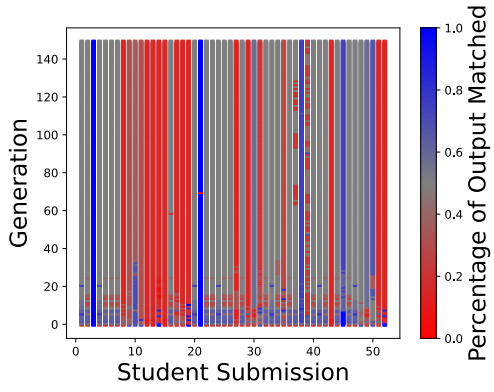
This section discusses the results of comparing the amount of the student's solution's output that matched with the instructor's solution's output before the first error occurred. This metric can give an indication as to where in the execution path the error occurred. If the majority of the student's output matches the instructor's output before the first error was found, the error was caused by a test case near the end of the execution path, while a student whose output hardly matches the instructor's output likely experienced an error at the start of the execution path. The metric is expressed as a percentage of the output matched prior to failure, and is shown in Figure 10.4, where each sub-figure corresponds to a different automated assessment strategy. The colour scale used in Figure 10.4 is a red-grey-blue scheme, where red corresponds to 0% errors found, grey corresponds to 50% errors found, and blue corresponds to 100% errors found. The gradient between the three colours is used to represent other percentages.

Figure 10.4 shows that CCGA was only able to find errors in the early stages of execution. GATLAM's best configuration found errors in both the early and middle stages of execution, with few errors being found in the late stages of execution.

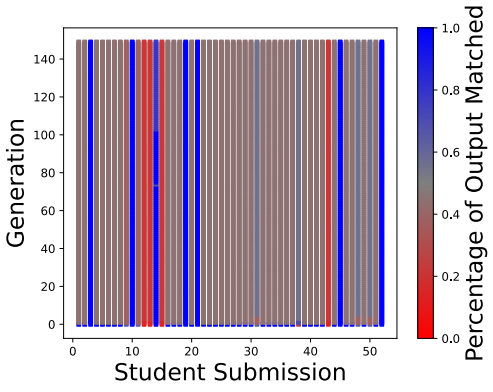
RNG, as expected, detects both early and late errors, where the percentage of output matched is not correlated with the generation number, i.e., no convergence takes place. A deviation can be seen in two of the student's results in terms of a different pattern of execution length being observed. The reason for the general overall pattern is attributed to each RNG evaluation of a student's code using the same-seeded random values. Another contributing factor is the way in which the fitness function chooses the best chromosome. To investigate this deviation, one of the deviating students was selected for further investigation, discussed in Section 10.3.

GATLAM with the worst configuration found the majority of errors in the middle stages of execution, while GATLAM with the default configuration found the majority of errors in the later stages of execution. This leads to the conclusion that the hyperparameter configuration does not affect the error's location, but rather the types of errors found, which is explained in Section 10.2.3 below.

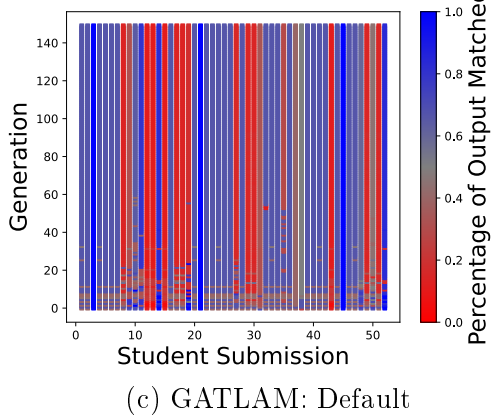
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



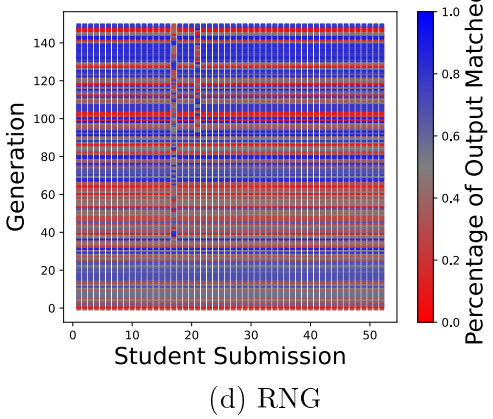
(a) GATLAM: Best



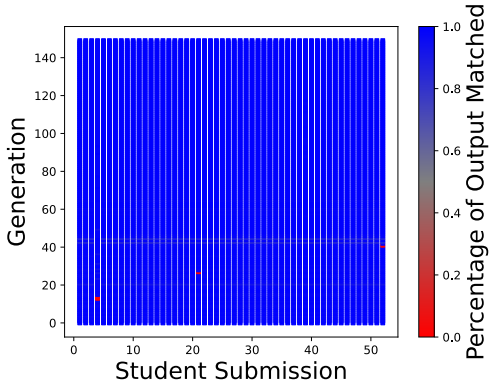
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Figure 10.4: Percentage of the output matched between the student and the instructor solutions before the first error occurred for different automated assessment strategies.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.2.3 Types of Errors Found

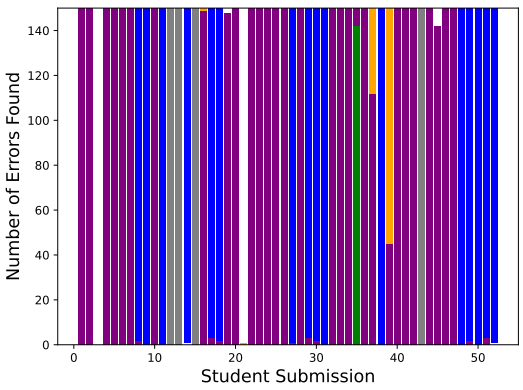
During the evaluation of the students' code submissions, types different of errors were found. This section looks at the types of errors found during the evaluation of each student by the different automated assessment strategies, as well as the overall types of errors found across all of the student code evaluations.

Error Types Found Per Student

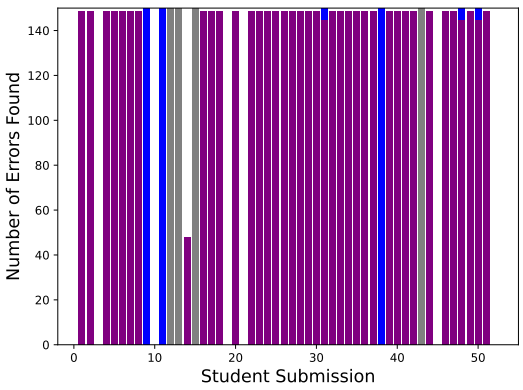
Figure 10.5 shows the number of errors from each error type that different automated assessment strategies were able to find in each of the student code submissions. Note that Figure 10.5 reflects the number of errors found, not which generation the error was found in.

As shown in Figure 10.5, the majority of errors found are of type "incorrect output". Incorrect output could vary from a single incorrect line to the entire output generated by the student's solution being incorrect. Some of the automated assessment strategies discussed in Chapter 5 classify incorrect output as a minor error, and only deduct partial marks for this. Fitchfork is an example of such a strategy. On the other hand, the other errors found are more serious, and affect a student's marks strongly.

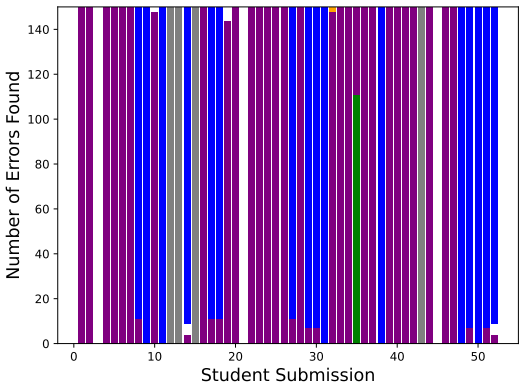
Apart from the two infinite loop errors detected, CCGA was only able to find incorrect output errors across all of the student code evaluations. The same observation is made for RNG; only a single infinite loop was detected, and the remaining errors were of type incorrect output. RNG's poor performance could be attributed to the lack of any "knowledge" being passed on to new chromosomes in the population, as RNG replaces the population at each generation. CCGA's poor performance can be attributed to the lack of code coverage data being collected if the program exits abnormally, and the fact that the error may be present on a shorter execution path, which was skipped in favour of a longer execution path.



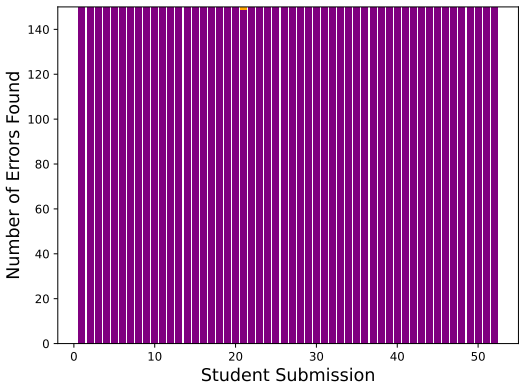
(a) GATLAM: Best



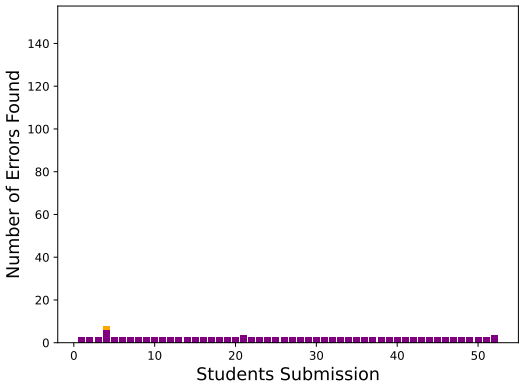
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Error Type	Color
Compilation Failure	
Floating Point Exception	
Incorrect Output	
Infinite Loop	
No Errors	
Segmentation Fault	

Legend

Figure 10.5: Count of each type of error that different automated assessment strategies were able to find in different student code submissions.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

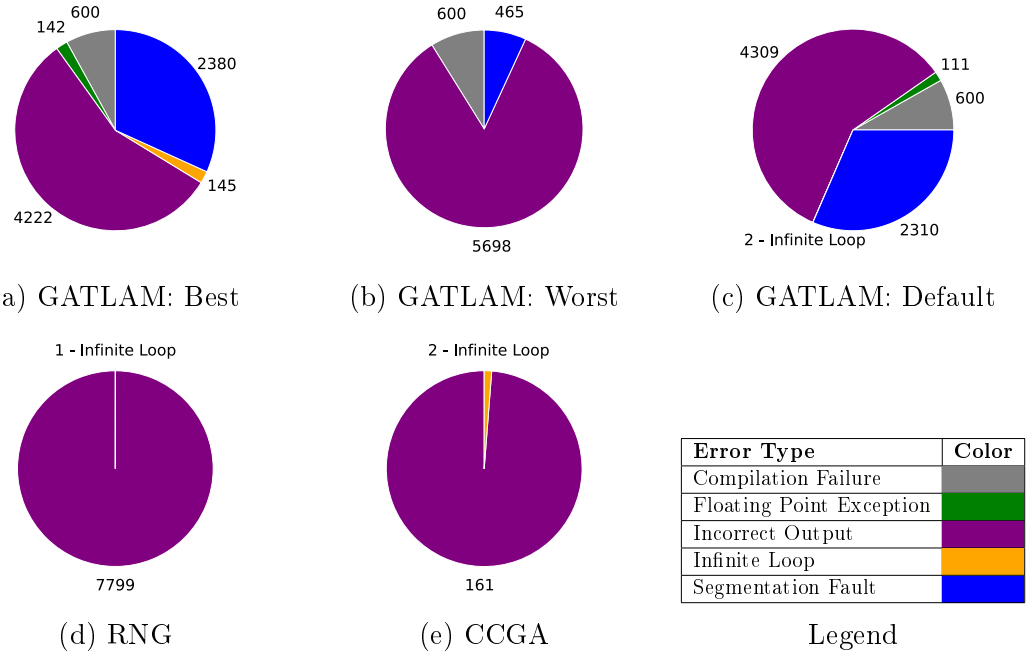
GATLAM, with different configurations, found a wider variety of errors. GATLAM, with the best configuration, found more infinite loop errors than any of the other automated assessment strategies, including GATLAM with other configurations. It can also be seen that GATLAM, with both best and default configurations, was able to find a floating-point exception error. This error occurs when a number is divided by 0. It is interesting to note that GATLAM, with the best configuration, was able to find the floating point exception more often than GATLAM with its default configuration. GATLAM with all three configurations was able to find segmentation faults, with GATLAM with the best configuration finding the most segmentation faults. Student code is prone to segmentation fault failures; as such, an automated assessment strategy capable of detecting such errors is favourable.

Distribution of Error Types Found in Total

The total number of each error type each automated software test generation method was able to find across all student code evaluations is shown in Figure 10.6. For the RNG and CCGA sub-figures, the non-purple pie chart wedges indicate infinite loop errors, which are represented by a yellow colour. GATLAM's default configuration (sub-figure (c) of Figure 10.6), the error type that occurred twice is an infinite loop error.

Figure 10.6 shows that GATLAM's best and default configurations yielded 5 different error types, which is the highest variation of different errors types found. GATLAM's worst configuration found compilation failures, as well as finding two different runtime errors, which are the same compared to RNG and CCGA. GATLAM's best configuration found the highest number of infinite loop errors, floating point exceptions, and segmentation faults, but at the cost of finding a low number of incorrect output errors.

As stated previously, CCGA was not able to find errors as frequently as the other software test generation methods, which is reflected in the low number of incorrect output errors found. Although GATLAM experienced a high variation in several errors found, which can be attributed to how the hyperparameters were configured, it was still able to outperform CCGA and RNG. Arguably, even if two instructors created a set of test cases for the same assignment, the two instructors' created test cases could also have varying results, and as such, this variety in GATLAM's performance is acceptable.



(a) GATLAM: Best (b) GATLAM: Worst (c) GATLAM: Default

(d) RNG (e) CCGA Legend

Figure 10.6: Total number of error types that different automated assessment strategies were able to find.

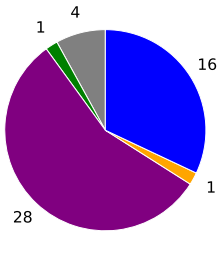
10.2.4 Final Generation Error Types Found

This section only considers what types of errors were detected in the final generation of evaluation of all of the students' submitted code by the different automated assessment strategies. The results are shown in Figure 10.7. Due to CCGA not finding any errors, it was omitted from Figure. The numbers indicate the number of the respective error type that the automated assessment strategy was able to find.

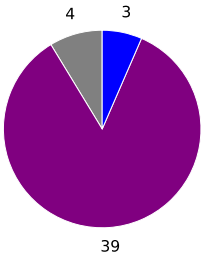
The results in Figure 10.7 reflect the same observation as was made in Section 10.2.3, i.e., that GATLAM with all three configurations found a wider variety of error types in the final generation of execution compared to RNG. GATLAM with the best configuration was the only automated assessment strategy to find 5 different types of errors in the final generation of execution.

In the final generation of evaluation, CCGA was not able to find any errors, while RNG was able to find an incorrect output error in all of the students' submitted code.

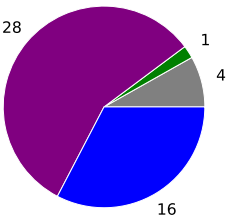
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



(a) GATLAM: Best



(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG

Error Type	Color
Compilation Failure	
Floating Point Exception	
Incorrect Output	
Infinite Loop	
Segmentation Fault	

Legend

Figure 10.7: Number of types of errors that different automated assessment strategies were able to find in the final generation.

10.2.5 Example Feedback

Appendix D provides an example of the feedback generated by GATLAM that could possibly be returned to a student. The feedback would be the example test data, in this case, a full testing program that was able to find an error in the student's code. The feedback also includes the instructor's output as well as the student's output for that program until an error is found. The student's code in this case produced a segmentation fault, which explains the big difference between the instructor's output and the student's output.

Using the feedback, the student will be able to see an example main function which caused an error in their code, which aids debugging and the learning experience of the student while completing the practical. Providing a student with a concrete example of where their submitted assignment failed will remove the frustration of trying to interpret the feedback of the automated assessment strategy, and aid the instructor, as feedback for each line of output does not need to be created. This novel approach to feedback is

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

what makes GATLAM drastically different from other automated assessment strategies that were discussed in Chapter 5.

10.2.6 Summary

GATLAM was able to find a wider range of different types of errors compared to the alternative automated assessment strategies, RNG and CCGA. RNG was able to find more errors than any other automated assessment strategy, which can in part be attributed to GATLAM's fitness function selecting which chromosomes to use. Although RNG did find the majority of the errors, it was only able to find an infinite loop error on one occasion, and the remaining were incorrect output errors. CCGA showed the worst performance, only finding two infinite loop errors. Another important takeaway is that GATLAM's hyperparameter configuration influences the types of errors that were detected, but not the stage during execution time when the errors will be detected. This can be seen as either a positive or a negative aspect, depending on the type of assignment that is being assessed, as well as the marking strategy that is followed.

10.3 Selected Multiple Run Experimental Results

This section focuses on the results of experiments where a selection of a few students' code submissions was evaluated multiple times using the different automated assessment strategies. The students' submissions that were selected are listed below together with a reason why they were chosen.

- **Student 3's submission** due to GATLAM never being able to find errors in this code, in any of GATLAM's configurations.
- **Student 4's submission** due to the deviation in the error detection pattern for CCGA in Figure 10.3.
- **Student 12's submission** due to GATLAM finding compilation failures during this student's submission evaluation.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

- **Student 17's submission** due to the deviation in the error location pattern for RNG in Figure 10.4.
- **Student 30's submission** due to GATLAM in all three of its configurations being able to find an error in this student's code.

Each of these student submissions can be thought of as a representative of a type of submission that experienced the same automated assessment evaluation by an automated assessment strategy. The purpose of these experiments is to determine how reproducible the results are from each of the automated assessment strategies. Three metrics, namely the number of errors found, the types of errors found, and the length run before an error occurred, are used to investigate the reproducibility of the results of the automated assessment strategies. The number of errors found are discussed in Section 10.3.1, the types of errors found are discussed in Section 10.3.2, and the length ran before first error occurred is discussed in Section 10.3.3. Lastly, a summary of this section is given in Section 10.3.4.

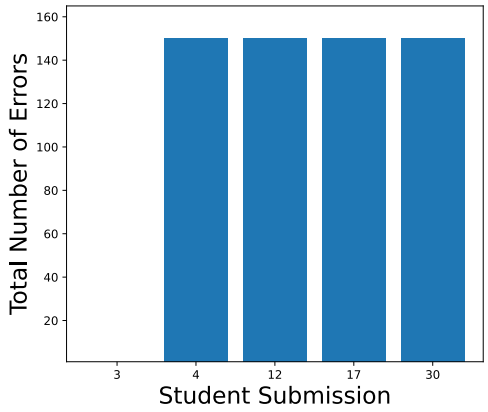
10.3.1 Errors Found

Figure 10.8 shows the results of the number of errors found metric. The results are displayed using a bar graph, where the average total number of errors found are plotted as the bars with the standard deviation overlaid on top of the bars.

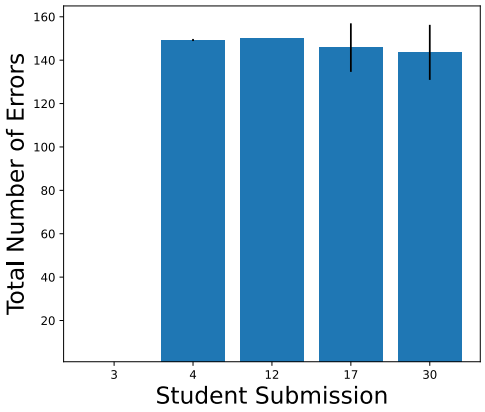
Only CCGA was not able to reproduce the same number of errors found across the multiple independent runs. This shows that CCGA's performance is highly dependent on the random seed utilised by the GA.

GATLAM and RNG, on the other hand, produced consistent results across multiple runs, with GATLAM's worst configuration causing GATLAM to experience a slight deviation in results produced.

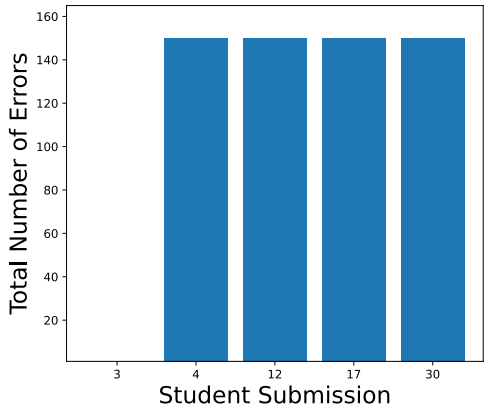
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



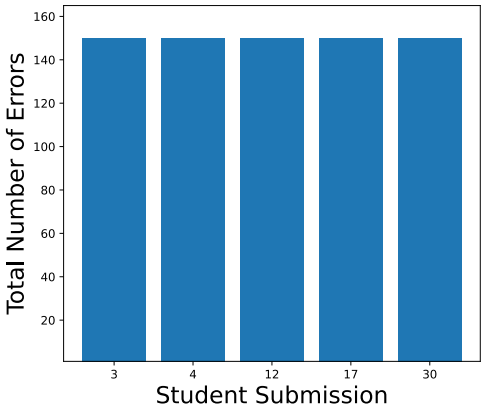
(a) GATLAM: Best



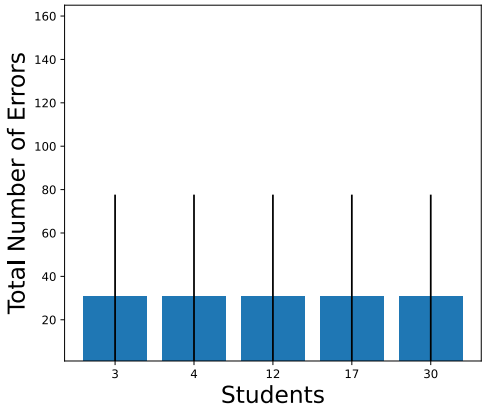
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Figure 10.8: Number of errors different automated assessment strategies were able to find in the final generation with the standard deviation overlaid as a line.

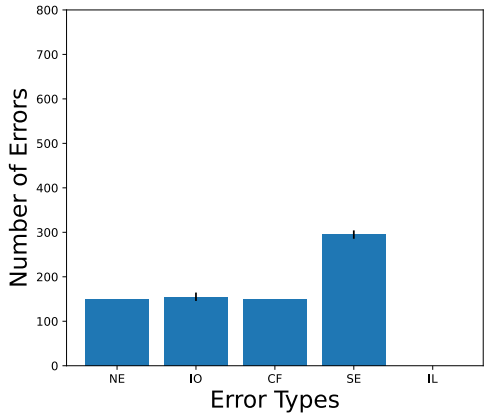
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

GATLAM with any of the three configurations was not able to find the errors in student 3's submission, even across multiple runs, but was able to consistently find the errors in the remaining student submissions. Upon investigation, the error was a single line of output that was incorrectly printed. GATLAM failed to find the error was it was extremely specific, while RNG produced the correct sequence of numbers to detect this error. RNG was again able to find the errors in all of the selected student submissions in each generation of execution. This supports the notion that the proposed fitness function, utilised by both GATLAM and RNG, can determine if the chromosome was able to generate test input causing an error in the student's submission independently of the search algorithm employed.

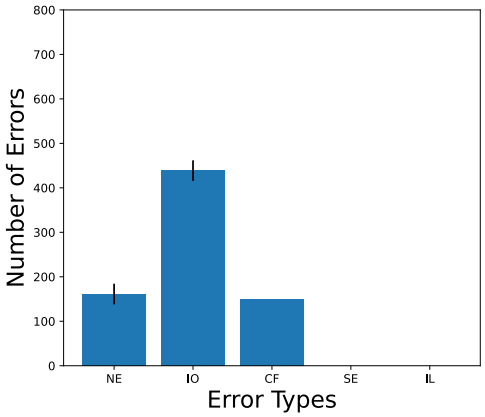
10.3.2 Types of Errors Found

Figure 10.9 shows the results of the type of error that the different automated assessment strategies were able to find. The results are displayed using a bar graph where the average total number of each error type found is displayed as the bar with the standard deviation overlaid on the bar. It should be noted that none of the selected student submissions exhibited floating-point errors.

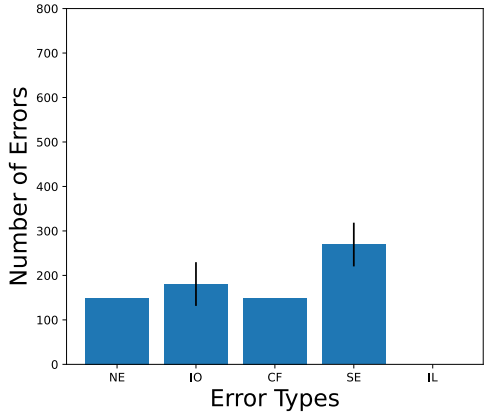
Figure 10.9 shows that GATLAM's best and worst configurations found approximately the same number of errors across the multiple runs across all of the selected student code evaluations. Interestingly, GATLAM's worst configuration yielded the same number of each type of error across multiple runs, with only a few outliers. The results in Figure 10.9 confirm that GATLAM was able to find the highest number of errors for each type of error compared to RNG and CCGA. CCGA and RNG were mostly only able to find incorrect output errors, as discussed in Section 10.2.3. CCGA's ability to find different types of errors is closely related to the random seed, as a high variation in the number of incorrect output errors found across multiple runs is observed.



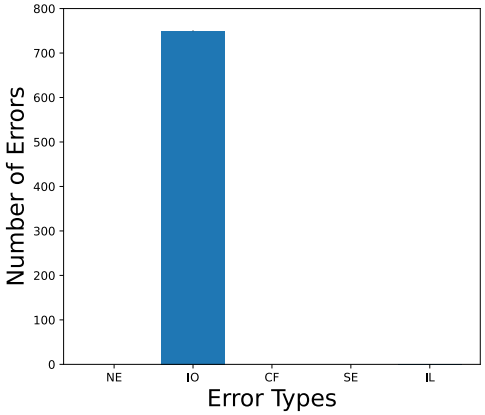
(a) GATLAM: Best



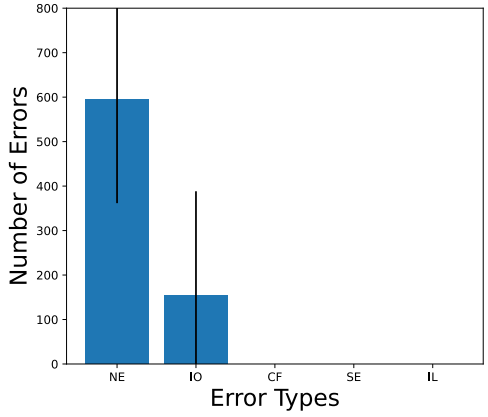
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Acronym	Error Type
NE	No Errors
IO	Incorrect Output
CF	Compilation Failure
SE	Segmentation Fault
IL	Infinite Loop

Legend

Figure 10.9: Types of errors that the different automated assessment strategies were able to find in the final generation.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.3.3 Percentage of the Output that Matched Before First Error Occurred

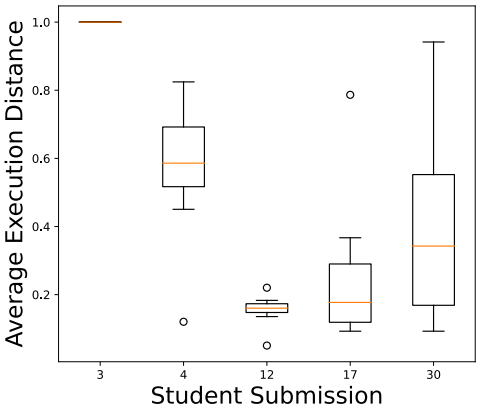
This section discusses the results of the percentage of the instructor's output that the student's output was able to match before the first error was found for the selected student submissions using different automated assessment strategies. The results are displayed in Figure 10.10, where the percentage matched in each independent run is shown as a box plot.

Only CCGA was able to run the same length before a single error occurred in each of the different runs for each of the selected students' submitted programs. This is consistent with the results seen in Section 10.2.2. Although GATLAM was able to consistently find errors in the student's code regardless of the random seed of the independent runs, GATLAM was not able to consistently find the errors at the same location across multiple runs. RNG shares the same trait in the sense that it was able to consistently find errors in the selected student's code, but the location of the execution differed.

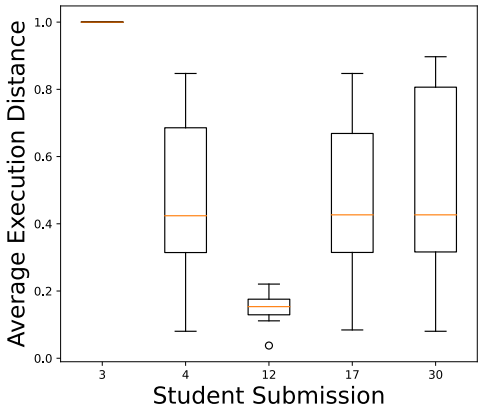
10.3.4 Summary

The results in this section showed that GATLAM can reproduce the similar results across multiple runs, where each run used a different random number seed. The fluctuations observed are due to the different random number seeds that were used. CCGA was seen to not be reliable in reproducing performance across multiple runs , using different random number seeds, and is extremely reliant on the random seed. RNG could reliably find errors in students' code regardless of the random seed, but lacked the ability to find complex errors. Although GATLAM did show fluctuations when comparing different random number seeds, the same seed will be used when assessing submitted assignments, such that all results that were obtained are reproducible.

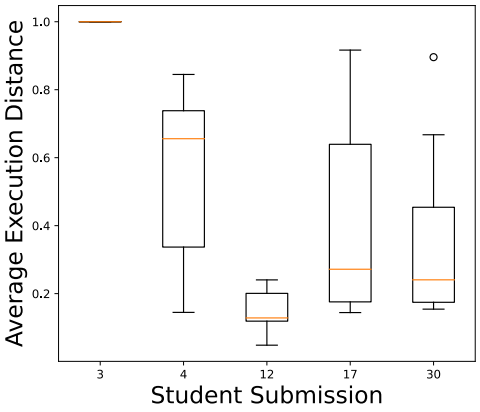
UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA



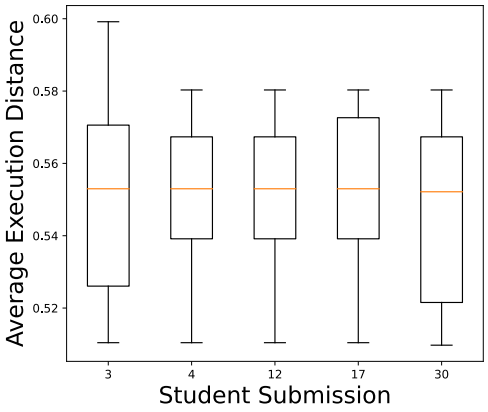
(a) GATLAM: Best



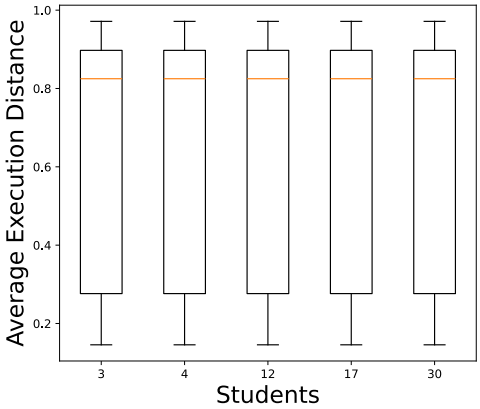
(b) GATLAM: Worst



(c) GATLAM: Default



(d) RNG



(e) CCGA

Figure 10.10: Execution distance before an error was found by different automated assessment strategies.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

10.4 Summary

In this chapter, different configurations of GATLAM were compared to two alternative software generation methods, namely, RNG and CCGA. The comparison was performed under two different scenarios, namely a controlled experiment where a series of error pools were used with artificially injected errors, and an uncontrolled experiment which utilised volunteer code submissions from students for a real world practical completed in 2022. GATLAM was able to find a higher variety of error types compared to the alternative methods in all of the real-world comparisons. CCGA was not able to reliably find a high number of errors due to its preference for longer execution paths, which did not contain errors. RNG was able to find errors more consistently than GATLAM, which can be attributed to GATLAM's fitness function (default fitness function configuration) being used to select the best chromosome for each RNG run. Thus, GATLAM's fitness function can be used independently of the search algorithm. The results presented in this chapter show that GATLAM can be used not only as an automated software test generation method, but also, more importantly, as a reliable automated assessment strategy.

Chapter 11

Conclusion

This chapter presents the concluding remarks of this dissertation. Section 11.1 reflects on the research objective, questions and novelty first laid out in Chapter 1. Section 11.2 proposes the contributions of this dissertation, and Section 11.3 discusses future work that can be derived from this dissertation.

11.1 Attainment of Research Objective

This dissertation proposed a novel automated assessment strategy, **G**enetic **A**lgorithm with **T**emporal **L**ogic for **A**utomated **M**arking (GATLAM). GATLAM comprises three main components, namely:

- Genetic algorithm (GA), which uses the temporal logic formulae in the fitness function (GA was introduced in Chapter 2, and the GA's relevance to GATLAM was discussed in Section 6.2.2);
- an interpreter (Section 6.2.4); and
- solutions (Section 6.2.3).

The GA is used to evolve a set of generic test input data that would cause errors in the solution being evaluated. The interpreter is used to convert the generic test input data to assignment-specific test input data for the solutions. The solutions are then

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

executed with the assignment-specific test input data, and the behaviour of the system is recorded and evaluated. This generation, interpretation, execution, and evaluation process is repeatedly performed during the evaluation of the current solution.

In order to give context for the solutions, GATLAM's running modes need to be defined. GATLAM has two running modes, namely, Single Solution Mode (SS) or Instructor-Student Solutions Mode (ISS). In SS, only a single solution is evaluated for errors, where a solution refers to a programming assignment that is submitted by the student. In ISS, two solutions are used to determine if an error occurred, where the solutions are two programming assignments, one completed by the student, and the second one is a memorandum solution completed by the instructor. The memorandum solution is used to generate the expected output for the assignment for the evolved test input data.

The main objective of this dissertation was to investigate whether a GA, with linear temporal logic (LTL) properties, can be used to enhance the automated assessment of programming assignments. To address this, objective four research questions were identified. A discussion pertaining to each of the research questions is given in the following sections.

11.1.1 Utilising a GA for Automated Assessment

This research question was addressed by firstly giving background on GAs in Chapter 2. After that, Chapter 6 described how the GA can be utilised for automated assessment as part of GATLAM. Chapter 7 discussed the experiments that were conducted to obtain empirical evidence on the GA's ability to be used as part of an automated assessment strategy.

Chapter 8 discussed the influence of the GA's hyperparameters on GATLAM's error-finding and convergence abilities. It was observed that GATLAM is sensitive to different configurations of GA-specific hyperparameters, which did impact GATLAM's error-finding and convergence abilities. It should be noted that even when configured with the worst combination of GA-specific hyperparameters, GATLAM was still able to find errors in both a set of controlled and uncontrolled code samples (further details of these code samples are discussed in Section 11.1.4). Chapter 10 discussed the reliability of the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

GA component to evolve the set of test input data such that GATLAM can repeatedly find errors in the code samples. It was seen that GATLAM does suffer from minor variations across multiple runs, which do not significantly impact GATLAM's error-finding abilities.

The fact that GATLAM is reliably able to find errors, using different GA-specific hyperparameter configurations, shows that GATLAM can be used as an automated assessment strategy. This in turn shows that a GA can be used as part of an automated assessment strategy, which fully answers *Research Question 1: Can a GA be utilised as a part of an automated assessment strategy?*

11.1.2 LTL Properties as Part of GATLAM's Fitness Function

This research question asked if it is possible to utilise LTL properties as part of GATLAM's fitness function. LTL properties are used to describe the desired behaviour of software over time, and a fitness function evaluates to what degree a chromosome was able to solve the problem the GA is trying to solve. A chromosome is a candidate solution to the problem the GA is trying to solve. In the context of GATLAM, a candidate solution is a set of numbers that, when interpreted into assignment-specific test input data, could expose errors in the student's submitted coding assignment.

The research question was addressed by first giving context to pre-existing software test creation methods and background on LTL formulae in Chapter 3. Chapter 6 introduces GATLAM's fitness function, which utilises LTL properties in each of the sub-components of the fitness function:

- The LTL component utilises the LTL properties directly by determining how many of the LTL properties were violated.
- The failure component determines how many of the system under test's modules violated the LTL properties.
- The memory component determines to what degree parts of a given chromosome have a history of creating test input data which will cause the system under test to violate the LTL properties.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

GATLAM's fitness function assigns a weight to each of the fitness function sub-components. Chapter 6 also introduced the LTL properties which are used by GATLAM to determine, if the submitted assignment violated correctness, for a given set of assignment-specific test input data.

To determine the effectiveness of LTL properties as part of GATLAM's fitness function, a series of experiments were conducted. The results were discussed in Chapters 8, 9, and 10. It was observed that the LTL properties could be utilised by the fitness function such that it can identify erroneous behaviour in various code samples. In Chapter 10, it was also observed that the fitness function, when used with other automated assessment strategies, could select the best-performing assignment-specific test input data out of a selection of assignment-specific test input data. The influence on the weights of the LTL component of GATLAM's fitness function is also discussed in Chapter 9. A series of different weight combinations was evaluated, and GATLAM was able to find a high percentage of errors in the code samples with different fitness function sub-component weights.

In conclusion, a GA can be combined with LTL formulae as part of its fitness function to evolve a set of chromosomes which, when translated to test input data, expose errors in student-submitted coding assignments. Thus, *Research Question 2: Can a GA be combined with LTL formulae as part of its fitness function in order to produce test input data which can expose errors in a student's submitted code assignment?* is mostly answered, as the LTL properties utilised by GATLAM are currently hard-coded. In future work, it is recommended that the possibility of providing user-created LTL properties for GATLAM be investigated.

11.1.3 Memory Component as Part of GATLAM's Fitness Function

This research question asked if it is viable to incorporate some type of memory component in GATLAM's fitness function. This was addressed by calculating how frequently a certain value appeared at the same position in a chromosome that caused the submitted assignment to violate the LTL properties. Chapter 6 formally introduced GATLAM

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

and how the memory component is utilised by GATLAM's fitness function. Chapter 7 described the experiments that were conducted to evaluate GATLAM.

It was observed in Chapter 8 that the memory component has the ability to easily dominate the search, but can help to reduce the search space of GATLAM, thus making the search faster. In conclusion, it is possible to incorporate some form of memory component into the fitness function of a GA to speed up the search for test input data that exposes errors in the submitted coding assignment. This discussion answers *Research Question 3: Can some type of memory component be incorporated into the fitness function of the GA?*.

11.1.4 Benchmarking GATLAM

The last research question required GATLAM to be compared to alternative automated assessment tools (also known as automated software test generation methods depending on the use-case). In order to address this research question, GATLAM was compared to other automated assessment strategies both theoretically and empirically.

GATLAM was firstly theoretically compared to different automated assessment strategies, which were discussed in Chapter 5, using the theory discussed in Chapter 4. It was observed that theoretically, due to the iterative algorithm of a GA, GATLAM does not share the same speed benefit that the other automated assessment strategies have. In turn, GATLAM does have the ability to provide a student with a detailed example of test input data which violates the correctness properties of the assignment. Due to the dynamic generation of the test cases, the instructor can release the test cases to the students, which is not always possible or practical in the other discussed automated assessment strategies. These strategies utilise static test cases, where the test cases do not change with each new student submission, and as such could lead to test cases being reverse-engineered by the students.

In order to compare GATLAM to alternative approaches empirically, the experiments outlined in Chapter 7, and the results discussed in Chapter 10, were used. GATLAM was compared against different configurations of itself, as well as to a random number generator (RNG), and a code coverage-based GA (CCGA). RNGs and code coverage-based software testing techniques were discussed in Chapter 3, and are industry-standard

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

methods for automatically creating software tests. The different configurations of GATLAM were derived as a result of the experiments discussed in Chapter 8. Two sets of experiments were conducted. The first compared GATLAM, RNG, and CCGA using hand-crafted error pools. Each error pool contained different examples of related error types, namely: logical errors, concurrency errors, infinite loop errors, and memory-related errors. The second set of experiments utilised a real-world assignment that was given to first-year students to complete in 2022. A selection of students volunteered their assignments to be used in this dissertation to determine how applicable GATLAM is in real-world scenarios.

Based on the results in Chapter 10, it was observed that with the error pools, GATLAM with different hyperparameter configurations was able to reliably find more errors than CCGA. RNG was able to consistently find more errors than GATLAM's worst hyperparameter configuration, but the other GATLAM configurations were able to find more errors than RNG. For the real-world assignment, it was observed that GATLAM was able to find a higher number of errors as well as a wider variety of error types compared to RNG and CCGA. GATLAM was also able to reliably find the errors, which mostly satisfy the automated assessment's benefit of consistent marking.

In conclusion, this section fully answers *Research Question 4: How does the newly proposed automated assessment strategy compare to other automated assessment strategies (or automated software test generation methods) used in the real world?* It should be noted that, as future work, GATLAM can be used for multiple real-world assignments over a longer time period to determine the practical challenges for GATLAM in the wild.

11.2 Contributions

The contributions of this dissertation are twofold. The first novel contribution is the combination of GAs and LTL properties as part of a fitness function. This allows GATLAM to generate dynamic personalised test input data for which the submitted assignment violates correctness. This is followed by incorporating a memory component into the fitness function such that recurring genes are given priority. The introduction of the

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

memory component aids GATLAM in reducing the search, which in turn reduces the overall runtime of GATLAM.

This combination of novel algorithmic contributions led to the main contribution of the dissertation, which is GATLAM, a new, unique automated assessment strategy that introduces two novel features. These features are an evolutionary approach to automatically generate dynamic personalised test input data that violates correctness for a given submission, and a new way of providing constructive submission-specific feedback to students by giving them examples of test input data that caused their code to fail without concern about leaking test cases.

11.3 Future Work

This dissertation opened the door for various future works. A short, but non-exhaustive, list of possible future topics is listed below:

1. Future work with GATLAM itself:
   (a) Can different evolutionary algorithms be utilised, and how do they compare to GATLAM?
   (b) What should happen if the instructor's solution fails, and not the student's solution?
   (c) Longitudinal study about the real-world use of GATLAM as an automated assessment strategy.
   (d) Incorporating some sort of multi-agent search, such that test input data developed for one student is incorporated in the search for other students to improve convergence.
   (e) Expand and refine the LTL properties used to describe the behaviour of the student's solution under test.
   (f) Experiment with how the noise of the different hyperparameter default values affects the performance of GATLAM.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

(g) Experiment with how different chromosome structures can affect GATLAM's performance.

2. Further work with the interpreter used to convert the candidate solution to test input data:

   (a) Formalising the algorithm used for interpretation of the candidate solution to test input data.

   (b) Developing a generic interpreter that can easily be applied and adapted to different assignments.

3. Using GATLAM for other use cases besides automated marking of programming assignments. Examples of some possibilities are:

   (a) Automated software test creation.

   (b) Evolution of hyperparameters for simulations of models.

   (c) Mathematical test generations for high-school mathematical subjects.

As such, it is evident that GATLAM is the starting point for a wide range of possible novel contributions to the scientific fields of evolutionary computing, automated software testing, and automated assessment.

Bibliography

[1] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, pages 153–160, 2010.

[2] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation Analysis. Technical report, Georgia Inst of Tech Atlanta School of Information And Computer Science, 1979.

[3] Tuukka Ahoniemi and Tommi Reinikainen. ALOHA-a Grading Tool for Semi-Automatic Assessment of Mass Programming Courses. In *Proceedings of the 6th Baltic Sea Conference on Computing Education Research: Koli Calling 2006*, pages 139–140, 2006.

[4] Kirsti M Ala-Mutka. A Survey of Automated Assessment Approaches for Programming Assignments. *Computer Science Education*, 15(2):83–102, 2005.

[5] Samia AlBlwi, Imen Marsit, Besma Khaireddine, Amani Ayad, JiMeng Loh, and Ali Mili. Subsumption, Correctness and Relative Correctness: Implications for Software Testing. *Science of Computer Programming*, 239:103177, 2025.

[6] Frances E. Allen. Control Flow Analysis. *SIGPLAN Not.*, 5(7):1–19, jul 1970.

[7] Shaull Almagor, Udi Boker, and Orna Kupferman. Formally Reasoning About Quality. *Journal of the ACM (JACM)*, 63(3):1–56, 2016.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[8] Fatima AlShamsi and Ashraf Elnagar. An Automated Assessment and Reporting Tool for Introductory Java Programs. In *International Conference on Innovations in Information Technology*, pages 324–329. IEEE, 2011.

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2017.

[10] TA Angelo. Defining (And Re-Assessing) Assessment: A Second Try. *AAHE Bulletin*, 48(3):7, 1995.

[11] I Aryandana, A Permanasari, and Taufik Adji. Comparing Method Equivalence Class Partitioning and Boundary Value Analysis With Study Case Add Medicine Module. *IOP Conference Series: Materials Science and Engineering*, 732:012072, 01 2020.

[12] Paul Bachmann. *Number Theory: The Analytical Number Theory*, volume 2. Teubner, 1894.

[13] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT press, 2008.

[14] James E Baker et al. Reducing Bias and Inefficiency in the Selection Algorithm. In *Proceedings of the Second International Conference on Genetic Algorithms*, volume 206, pages 14–21, 1987.

[15] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, and A. Watkins. Breeding Software Test Cases With Genetic Algorithms. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences, 2003.*, page 10, 2003.

[16] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded Model Checking. *Handbook of Satisfiability*, 185(99):457–481, 2009.

[17] Ulrich Bodenhofer. Genetic Algorithms: Theory and Applications. *Lecture Notes, Fuzzy Logic Laboratorium Linz-Hagenberg, Winter*, 2003.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[18] Bryce Boe, Charlotte Hill, Michelle Len, Greg Dreschler, Phillip Conrad, and Diana Franklin. Hairball: Lint-Inspired Static Analysis of Scratch Projects. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, pages 215–220, 2013.

[19] Simone Bova, Robert Ganian, and Stefan Szeider. Model Checking Existential Logic on Partially Ordered Sets. *ACM Trans. Comput. Logic*, 17(2), nov 2015.

[20] Martin D Braine. On the Relation Between the Natural Logic of Reasoning and Standard Logic. *Psychological Review*, 85(1):1, 1978.

[21] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing Finite Kripke Structures in Propositional Temporal Logic. *Theoretical Computer Science*, 59(1-2):115–131, 1988.

[22] Peter Brusilovsky and Sergey Sosnovsky. Individualized Exercises for Self-Assessment of Programming Knowledge: An Evaluation of QuizPACK. *J. Educ. Resour. Comput.*, 5(3):6, September 2005.

[23] Richard J Buchi. On a Decision Method in Restricted Second Order Arithmetic. *Proc. Internat. Congr. On Logic, Methodology and Philosophy of Science*, 1960.

[24] Kenneth Leroy Busbee and Dave Braunschweig. Programming Style. *Programming Fundamentals*, 2018.

[25] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 56(2):82–90, feb 2013.

[26] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Experimental Assessment of Random Testing for Object-Oriented Software. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 84–94, New York, NY, USA, 2007. Association for Computing Machinery.

[27] Edmund M Clarke. Model Checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 54–56. Springer, 1997.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[28] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, Roderick Bloem, et al. *Handbook of Model Checking*, volume 10. Springer, 2018.

[29] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT press, 2022.

[30] Draylson Micael De Souza, Jose Carlos Maldonado, and Ellen Francine Barbosa. ProgTest: An Environment for the Submission and Evaluation of Programming Assignments Based on Testing Activities. In *2011 24th IEEE-CS Conference on Software Engineering Education and Training (CSEE&T)*, pages 1–10. IEEE, 2011.

[31] Kyle Dewey, Phillip Conrad, Michelle Craig, and Elena Morozova. Evaluating Test Suite Effectiveness and Assessing Student Code via Constraint Logic Programming. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 317–322, 2017.

[32] Nafi Diallo, Wided Ghardallou, and Ali Mili. Correctness and Relative Correctness. In *IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 591–594. IEEE, 2015.

[33] Cambridge Dictionary. Assessment, 2024. Accessed: 2024-11-28.

[34] Cambridge Dictionary. Automation, 2024. Accessed: 2024-11-28.

[35] Cambridge Dictionary. Testing, 2024. Accessed: 2024-11-28.

[36] Jens Dolin, Paul Black, Wynne Harlen, and Andrée Tiberghien. Exploring Relations Between Formative and Summative Assessment. *Transforming Assessment: Through an Interplay Between Practice, Research and Policy*, pages 53–80, 2018.

[37] Christopher Douce, David Livingstone, and James Orwell. Automatic Test-Based Assessment of Programming: A Review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4, 2005.

[38] Priyom Dutta and B.S. Mahanand. Chapter 9 - Affordable Energy-Intensive Routing Using Metaheuristics. In Sushruta Mishra, Hrudaya Kumar Tripathy,

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Pradeep Kumar Mallick, Arun Kumar Sangaiah, and Gyoo-Soo Chae, editors, *Cognitive Big Data Intelligence With a Metaheuristic Approach*, Cognitive Data Science in Sustainable Computing, pages 193–210. Academic Press, 2022.

[39] Emma Enström, Gunnar Kreitz, Fredrik Niemelä, Pehr Söderman, and Viggo Kann. Five Years With Kattis—using an Automated Assessment System in Teaching. In *2011 Frontiers in Education Conference (FIE)*, pages T3J–1. IEEE, 2011.

[40] Mehdi Esnaashari and Amir Hossein Damia. Automation of Software Test Data Generation Using Genetic Algorithm and Reinforcement Learning. *Expert Systems With Applications*, 183:115446, 2021.

[41] Carol Evans. Making Sense of Assessment Feedback in Higher Education. *Review of Educational Research*, 83(1):70–120, 2013.

[42] Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

[43] George Fink and Matt Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, 1997.

[44] Thom Fruehwirth. Temporal Logic and Annotated Constraint Logic Programming. In *International Joint Conference on Artificial Intelligence*, pages 58–68. Springer, 1993.

[45] GCC Developers. *Gcov: A Test Coverage Program*. GNU Project, 2024. Accessed: 2024-08-19.

[46] Robert Gold. Control Flow Graphs and Code Coverage. *International Journal of Applied Mathematics and Computer Science*, 20(4):739–749, 2010.

[47] Jarrod Goschen, Anna S Bosman, and Stefan Gruner. Genetic Micro-Programs for Automated Software Testing With Large Path Coverage. In *IEEE Congress on Evolutionary Computation (CEC)*, pages 1–8. IEEE, 2022.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[48] H Grgic, Branko Mihaljević, and Aleksander Radovan. Comparison of Garbage Collectors in Java Programming Language. In *41st International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1539–1544. IEEE, 2018.

[49] Alan Hartman. Model Based Test Generation Tools. *Agedis Consortium*, 2002.

[50] Hadi Hemmati. How Effective Are Code Coverage Criteria? In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 151–156. IEEE, 2015.

[51] Irman Hermadi, Chris Lokan, and Ruhul Sarker. Genetic Algorithm Based Path Testing: Challenges and Key Parameters. In *Second World Congress on Software Engineering*, volume 2, pages 241–244. IEEE, 2010.

[52] Raymond R Hill. A Monte-Carlo Study of Genetic Algorithm Initial Population Generation Methods. In *Proceedings of the 31st Conference on Winter Simulation: Simulation—a Bridge to the Future-Volume 1*, pages 543–547, 1999.

[53] Jack Hollingsworth. Automatic Graders for Programming Classes. *Communications of the ACM*, 3(10):528–529, 1960.

[54] Richard C Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys (CSUR)*, 4(3):179–196, 1972.

[55] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[56] William Hudson. User Stories Don't Help Users: Introducing Persona Stories. *Interactions*, 20(6):50–53, nov 2013.

[57] John Hunt. Gang of Four Design Patterns. In *Scala Design Patterns*, pages 135–136. Springer, 2013.

[58] Patrick Indri, Alberto Bartoli, Eric Medvet, and Laura Nenzi. One-Shot Learning of Ensembles of Temporal Logic Formulas for Anomaly Detection in Cyber-Physical

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Systems. In *European Conference on Genetic Programming (Part of EvoStar)*, pages 34–50. Springer, 2022.

[59] JaCoCo Contributors. *JaCoCo Java Code Coverage Library*. Eclipse Foundation, 2024. Accessed: 2024-08-19.

[60] Joxan Jaffar and J-L Lassez. Constraint Logic Programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 111–119, 1987.

[61] Lalita Jategaonkar Jagadeesan, Carlos Puchol, and James E Von Olnhausen. Safety Property Verification of Esterel Programs and Applications to Telecommunications Software. In *Computer Aided Verification: 7th International Conference, CAV'95 Liège, Belgium, July 3–5, 1995 Proceedings 7*, pages 127–140. Springer, 1995.

[62] Muhammad Abid Jamil, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. Software Testing Techniques: A Literature Review. In *6th International Conference on Information and Communication Technology for the Muslim World (ICT4M)*, pages 177–182, 2016.

[63] Debasish Jana. Discussion about parameter passing in c/c++/java/c, 2011. Accessed: 2024-09-08.

[64] Khalid Jebari and Mohammed Madiafi. Selection Methods for Genetic Algorithms. *International Journal of Emerging Sciences*, 3(4):333–344, 2013.

[65] Irena Jovanović. Software Testing Methods and Techniques. *The IPSI BgD Transactions on Internet Research*, 30, 2006.

[66] Mike Joy, Nathan Griffiths, and Russell Boyatt. The Boss Online Submission and Assessment System. *Journal on Educational Resources in Computing (JERIC)*, 5(3):2, 2005.

[67] Monika Kaczorowska. Analysis of Typical Programming Mistakes Made by First and Second Year IT Students. *Journal of Computer Sciences Institute*, 15:202–205, 2020.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[68] Mohd Ehmer Khan, Farmeena Khan, et al. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *Int. J. Adv. Comput. Sci. Appl*, 3(6), 2012.

[69] Kenneth E. Kinnear and Peter J. Angeline. Advances in Genetic Programming. [Voume 1], 1994.

[70] Savas Konur. Real-Time and Probabilistic Temporal Logics: An Overview. *arXiv Preprint arXiv:1005.3200*, 2010.

[71] John R Koza et al. *Genetic Programming II*, volume 17. MIT press Cambridge, 1994.

[72] Saul A Kripke. Semantical Considerations on Modal Logic. *Acta Philosophica Fennica*, 16, 1963.

[73] Ampomah Ernest Kwame, Ezekiel Mensah Martey, and Abilimi Gilbert Chris. Qualitative Assessment of Compiled, Interpreted and Hybrid Programming Languages. *Communications*, 7(7):8–13, 2017.

[74] Xiao Li, Cristian-Ioan Vasile, and Calin Belta. Reinforcement Learning With Temporal Logic Rewards. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3834–3839. IEEE, 2017.

[75] Tom V Mathew. Genetic Algorithm, 2012. Report submitted at IIT Bombay. Accessed: 2023-07-23. URL: https://datajobstest.com/data-science-repo/Genetic-Algorithm-Guide-[Tom-Mathew].pdf.

[76] Igor Mekterović, Ljiljana Brkić, Boris Milašinović, and Mirta Baranović. Building a comprehensive automated programming assessment system. *IEEE Access*, 8:81154–81172, 2020.

[77] Christoph C. Michael, Gary McGraw, and Michael A Schatz. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[78] Roy Miller and Christopher T Collins. Acceptance Testing. *Proc. XPUniverse*, 238, 2001.

[79] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[80] Seyedali Mirjalili. *Genetic Algorithm*, pages 43–55. Springer International Publishing, Cham, 2019.

[81] Kevin A Naudé, Jean H Greyling, and Dieter Vogts. Marking Student Programs Using Graph Similarity. *Computers & Education*, 54(2):545–561, 2010.

[82] Robert H. B. Netzer and Barton P. Miller. What Are Race Conditions? Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, mar 1992.

[83] Srinivas Nidhra and Jagruthi Dondeti. Black Box and White Box Testing Techniques-a Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, 2012.

[84] José Carlos Paiva, José Paulo Leal, and Álvaro Figueira. Automated Assessment in Computer Science Education: A State-of-the-Art Review. *ACM Transactions on Computing Education (TOCE)*, 22(3):1–40, 2022.

[85] Vreda Pieterse. Automated Assessment of Programming Assignments. *Cserc*, 13:4–5, 2013.

[86] Federico Pigozzi, Eric Medvet, and Laura Nenzi. Mining Road Traffic Rules with Signal Temporal Logic and Grammar-Based Genetic Programming. *Applied Sciences*, 11(22), 2021.

[87] Muthu Ramachandran. Testing Software Components Using Boundary Value Analysis. In *Proceedings 29th Euromicro Conference*, pages 94–98, 2003.

[88] Dharashree Rath, Swarnalipsa Parida, Deepti Bala Mishra, and Sonali Pradhan. Evolutionary Algorithms for Path Coverage Test Data Generation and Optimization: A Review. *Optimization of Automated Software Testing Using Meta-Heuristic Techniques*, pages 91–103, 2022.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[89] G. Venkat Reddy and Ahalya Chandrasekhar. Tools and Techniques for Testing of Flight Critical Software. *Defence Science Journal*, 49(4):317–322, 1999.

[90] S.C. Reid. An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73, 1997.

[91] Nicholas Rescher and Alasdair Urquhart. *Temporal Logic*, volume 3. Springer Science & Business Media, 2012.

[92] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: Programming for All. *Communications of the ACM*, 52(11):60–67, 2009.

[93] Martín Safe, Jessica Carballido, Ignacio Ponzoni, and Nélida Brignole. On stopping criteria for genetic algorithms. In *Brazilian Symposium on Artificial Intelligence*, pages 405–413. Springer, 2004.

[94] Jacques Philippe Sauvé, Osório Lopes Abath Neto, and Walfredo Cirne. Easyaccept: A Tool to Easily Create, Run and Drive Development With Automated Acceptance Tests. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, pages 111–117, 2006.

[95] Sanjay Kumar Singh and Amarjeet Singh. *Software Testing*. Vandana Publications, 2012.

[96] Draylson M Souza, Katia R Felizardo, and Ellen F Barbosa. A Systematic Literature Review of Assessment Tools for Programming Assignments. In *2016 IEEE 29th International Conference on Software Engineering Education and Training (CSEET)*, pages 147–156. IEEE, 2016.

[97] Praveen Srivastava and Tai-Hoon Kim. Application of Genetic Algorithm in Software Testing. *International Journal of Software Engineering and Its Applications*, 3, 11 2009.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[98] Praveen Ranjan Srivastava, Parshad Patel, and Siddharth Chatrola. Cause Effect Graph to Decision Table Generation. *SIGSOFT Softw. Eng. Notes*, 34(2):1–4, feb 2009.

[99] Hussein Suleman. Automatic Marking With Sakai. In *Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology*, pages 229–236, 2008.

[100] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software Assurance by Bounded Exhaustive Testing. *SIGSOFT Softw. Eng. Notes*, 29(4):133–142, jul 2004.

[101] Richard S Sutton. Reinforcement Learning: An Introduction. *A Bradford Book*, 2018.

[102] Luay Ho Tahat, Atef Bader, Boris Vaysburg, and Bogdan Korel. Requirement-Based Automated Black-Box Test Generation. In *Proceedings of the 25th International Computer Software and Applications Conference on Invigorating Software Development*, COMPSAC '01, page 489–495, USA, 2001. IEEE Computer Society.

[103] Kuo-Chung Tai Kuo-Chung Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 69–72. IEEE, 1994.

[104] Wang Tiantian, Su Xiaohong, Ma Peijun, Wang Yuying, and Wang Kuanquan. Autolep: An Automated Learning and Examination System for Programming and Its Application in Programming Course. In *2009 First International Workshop on Education Technology and Computer Science*, volume 1, pages 43–46. IEEE, 2009.

[105] Mustafa M Tikir and Jeffrey K Hollingsworth. Efficient Instrumentation for Code Coverage Testing. *ACM SIGSOFT Software Engineering Notes*, 27(4):86–96, 2002.

[106] Newcastle University. Assessment Criteria, 2023. Accessed on 9 December 2023. URL: https://www.ncl.ac.uk/learning-and-teaching/effective-practice/assessment/assessment-criteria/.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

[107] Gissel Velarde. Artificial Intelligence and Its Impact on the Fourth Industrial Revolution. *International Journal*, 10:41–48.

[108] Tom Verhoeff. Programming Task Packages: Peach Exchange. *Olympiads in Informatics*, page 192, 2008.

[109] Ljuba Veselinova. The Negative Existential Cycle Revisited. *Linguistics*, 52(6):1327–1389, 2014.

[110] Burns Victoria. *53 Interesting Ways to Assess Your Students.*, volume 3rd edition of *Professional and Higher Education*. Frontinus Ltd, 2015.

[111] Neil Walkinshaw, Marc Roper, and Murray Wood. The Java System Dependence Graph. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pages 55–64. IEEE, 2003.

[112] Paul Weeden, Jan Winter, and Patricia Broadfoot. *Assessment*. Psychology Press, 2002.

[113] Darrell Whitley. A Genetic Algorithm Tutorial. *Statistics and Computing*, 4(2):65–85, Jun 1994.

[114] Michael Wooldridge, Michael Fisher, Marc-Philippe Huget, and Simon Parsons. Model Checking Multi-Agent Systems With MABLE. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 2*, pages 952–959, 2002.

[115] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined Static and Dynamic Automated Test Generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, page 353–363, New York, NY, USA, 2011. Association for Computing Machinery.

[116] Soundous Zougari, Mariam Tanana, and Abdelouahid Lyhyaoui. Hybrid Assessment Method for Programming Assignments. In *2016 4th IEEE International Colloquium on Information Science and Technology (CiSt)*, pages 564–569. IEEE, 2016.

Appendix A

LTL Negations

In this appendix, the conversion of the LTL formulas listed in Table 6.1 to their negated form is given. The first section, Section A.1 gives some of the general notation that will be used. Following this section each formula will be negated.

A.1 General Notation:

Let the following labels and atomic propositions be defined as follows:

1. **Exception:**
   When an exception occurs.

2. **Segfault:**
   When a segmentation fault occurs.

3. **InvalidReturnCode:**
   When the return code that is returned is not 0.

$$\text{exit_value} \neq 0$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

4. **ValidReturnCode:**
   When the return code that is returned is 0.

$$\text{exit_value} == 0$$

5. **Termination:**
   When a module/program finishes or is stopped.

A.2 Safety:

This property uses the standard definition of safety: *Something will never go wrong*
Let the label unsafe be defined as:

$$unsafe = \text{Exception} \vee \text{SegFault} \vee \text{InvalidReturnCode}$$

Thus the LTL formula would be:

$$G\neg\left(unsafe\right)$$

This implies that the property is: "always not unsafe".
Taking its negation:

$$\neg\left(G\neg\left(unsafe\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg\left(unsafe\right)\right)\right)$$

$$\Leftrightarrow F\left(unsafe\right)$$

So not eventually unsafe. This property will check the stdout for any exceptions and the stderr. If the exit value is also not 0 then an error will be assumed.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A.3 Proper Termination:

This property look at that the module will make progress and terminate in the correct exit value.

$$G\left(termination \Rightarrow ValidReturnCode\right)$$

So always eventually progress will be made. Taking its negation:

$$\neg\left(G\left(termination \Rightarrow ValidReturnCode\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(termination \Rightarrow ValidReturnCode\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg termination \vee ValidReturnCode\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \neg ValidReturnCode\right)$$

So eventually false until progress-will-be-made. Progress is defined as the system finishing with an exit value of 0.

A.4 Segmentation faults:

This property will check that no seg faults occurred:

$$G\left(\neg\left(segfault\right)\right)$$

So always there will not be a segfault:
Taking its negation:

$$\neg\left(G\left(\neg\left(segfault\right)\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg\left(segfault\right)\right)\right)$$

$$\Leftrightarrow F\left(segfault\right)$$

So eventually a segfault will occur. Thus the std error output will be checked for a segfault.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A.5 Exceptions:

This property will check that no uncaught exception occurred.

$$G\left(\neg\left(exception\right)\right)$$

So always there will not be a segfault: Taking its negation:

$$\neg\left(G\left(\neg\left(exception\right)\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg\left(exception\right)\right)\right)$$

$$\Leftrightarrow F\left(exception\right)$$

So eventually an exception will occur. Thus the std error and std out output will be checked for an exceptions.

A.6 Execution time:

This property will check that r which represents the actual execution time is less than or equal to e which represents the maximum time the system may execute.

$$r \leq e$$

The LTL property will then be:

$$G\left(termination \Rightarrow \left(r \leq e\right)\right)$$

Thus the negation:

$$\neg G\left(termination \Rightarrow \left(r \leq e\right)\right)$$

$$\Leftrightarrow \neg\left(G\left(\neg termination \vee \left(r \leq e\right)\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg termination \vee \left(r \leq e\right)\right)\right)$$

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

$$\Leftrightarrow F\left(termination \wedge \neg\left(r \leq e\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(r > e\right)\right)$$

So eventually the execution time will be greater then the maximum execution time.

A.7 Illegal output:

Let X represent the set of all illegal output consisting of strings and let output represent the output generated by system under test. The LTL property is thus as follows:

$$G\left(termination \Leftrightarrow \left(\wedge_{o \in output} \wedge_{x \in X} \left(x \neq o\right)\right)\right)$$

Thus the negation:

$$\neg\left(G\left(termination \Leftrightarrow \left(\bigwedge_{o \in output} \bigwedge_{x \in X} \left(x \neq o\right)\right)\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(termination \Leftrightarrow \left(\bigwedge_{o \in output} \bigwedge_{x \in X} \left(x \neq o\right)\right)\right)\right)$$

$$\Leftrightarrow F\left(\neg\left(\neg termination \vee \left(\bigwedge_{o \in output} \bigwedge_{x \in X} \left(x \neq o\right)\right)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \neg\left(\bigwedge_{o \in output} \bigwedge_{x \in X} \left(x \neq o\right)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(\bigvee_{o \in output} \bigvee_{x \in X} \left(x = o\right)\right)\right)$$

Thus eventually an element of X will be found in Output.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A.8 Expected output:

Let Y represent the set of expected output. Two options will be considered:

A.8.1 Exact match:

This is where the output received from the system being tested has to be exactly the same as the expected output. So the property can be described as follows:
Let Y be the ordered set of expected output and S the ordered set of output of the system. Let x and s be elements of Y and S respectively. Let the LTL property defined as follows:

$$G\left(termination \Rightarrow \left(\bigwedge_{i=1}^{length(Y)} (y_i = s_i)\right)\right)$$

Thus the negation would be:

$$\neg\left(G\left(termination \Rightarrow \left(\bigwedge_{i=1}^{length(Y)} (y_i = s_i)\right)\right)\right)$$

$$\Leftrightarrow \neg\left(G\left(\neg termination \vee \left(\bigwedge_{i=1}^{length(Y)} (y_i = s_i)\right)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \neg\left(\bigwedge_{i=1}^{length(Y)} (y_i = s_i)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(\bigvee_{i=1}^{length(Y)} \neg(y_i = s_i)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(\bigvee_{i=1}^{length(Y)} (y_i \neq s_i)\right)\right)$$

So the property we will check is that eventually the output will differ between the expected and the system output.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

A.8.2 Contains:

Using the same idea as above except that the indexes of the elements of each set is irrelevant. So the LTL property is defined as follows:

$$G\left(termination \Rightarrow \left(\bigwedge_{i=1}^{length(Y)} \bigwedge_{j=1}^{length(S)} (x_i == s_j)\right)\right)$$

Thus the negation would be:

$$\neg\left(G\left(termination \Rightarrow \left(\bigwedge_{i=1}^{length(Y)} \bigwedge_{j=1}^{length(S)} (x_i == s_j)\right)\right)\right)$$

$$\Leftrightarrow \neg\left(G\left(\neg termination \vee \left(\bigwedge_{i=1}^{length(Y)} \bigwedge_{j=1}^{length(S)} (x_i == s_j)\right)\right)\right)$$

$$\Leftrightarrow F\neg\left(\neg termination \vee \left(\bigwedge_{i=1}^{length(Y)} \bigwedge_{j=1}^{length(S)} (x_i == s_j)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \neg\left(\bigwedge_{i=1}^{length(Y)} \bigwedge_{j=1}^{length(S)} (x_i == s_j)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(\bigvee_{i=1}^{length(Y)} \bigvee_{j=1}^{length(S)} \neg(x_i == s_j)\right)\right)$$

$$\Leftrightarrow F\left(termination \wedge \left(\bigvee_{i=1}^{length(Y)} \bigvee_{j=1}^{length(S)} (x_i \neq s_j)\right)\right)$$

So the property we will check is that eventually there will be an element in Y that is not in S.

Appendix B

Real Assignment

This appendix contains the practical specification given as a formal practical assignment to first-year students partaking in an introduction programming module at the University of Pretoria. The aim of this assignment was to demonstrate the concept of polymorphism. This was done by using a class hierarchy namely the NumberTester class hierarchy. The purpose of the practical was to allow the user to load a set of "number evaluators" or NumberTester objects into a dynamic 1D array of dynamic pointers. The user will then be able to pass a number in to be evaluated. The student's code should then determine if the passed-in number satisfies all of the NumberTester objects. If and only if it does should the evaluate function return true, else the function should return false. Below the exact unaltered specification can be found that was given to the students. Chapters 7 and 10 use this assignment to evaluate the newly proposed automated assessment tool, introduced in Chapter 6.



UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA
Denkleiers • Leading Minds • Dikgopolo tša Dihlalefi

Department of Computer Science
Faculty of Engineering, Built Environment & IT
University of Pretoria

COS110 - Program design: Introduction

Practical 5 Specifications: Polymorphism

Release date: 26-09-2022 at 06:00
Due date: 30-09-2022 at 23:59
Total Marks: 41

Contents

1 General instructions: 2

2 Plagiarism 3

3 Outcomes 3

4 Introduction 3

5 Classes: 5
5.1 NumberTester: . . . 5
5.2 ValueDependantTester . . . 5
5.3 ValueIndependantTester: . . . 6
5.4 IsDivisible . . . 7
5.5 IsGreater . . . 7
5.6 IsSmaller . . . 8
5.7 IsEvenOdd: . . . 9
5.8 IsPrimeNumber: . . . 9
5.9 TesterInterface . . . 10

6 Source files 12

7 Allowed libraries 12

8 Submission 12

1 General instructions:

- This assignment should be completed individually, no group effort is allowed.
- Be ready to upload your assignment well before the deadline as no extension will be granted.
- You may not import any of C++'s built-in data structures. Doing so will result in a mark of zero. You may only make use of 1-dimensional native arrays where applicable. If you require additional data structures, you will have to implement them yourself.
- If your code does not compile you will be awarded a mark of zero. Only the output of your program will be considered for marks, but your code may be inspected for the presence or absence of certain prescribed features.
- If your code experience a runtime error you will be awarded a mark of zero. Runtime errors are considered as unsafe programming.
- All submissions will be checked for plagiarism.
- Read the entire specification before you start coding.
- Ensure your code compiles with C++98

2 Plagiarism

The Department of Computer Science considers plagiarism as a serious offence. Disciplinary action will be taken against students who commit plagiarism. Plagiarism includes copying someone else's work without consent, copying a friend's work (even with consent) and copying material (such as text or program code) from the Internet. Copying will not be tolerated in this course. For a formal definition of plagiarism, the student is referred to http://www.library.up.ac.za/plagiarism/index.htm (from the main page of the University of Pretoria site, follow the Library quick link, and then choose the Plagiarism option under the Services menu). **If you have any form of question regarding this, please ask one of the lecturers, to avoid any misunderstanding.** Also note that the OOP principle of code re-use does not mean that you should copy and adapt code to suit your solution.

3 Outcomes

The aim of this practical is to gain experience with polymorphism in a somewhat complex class hierarchy.

4 Introduction

Implement the UML diagram and functions as described on the following pages.



Figure 1: Meme

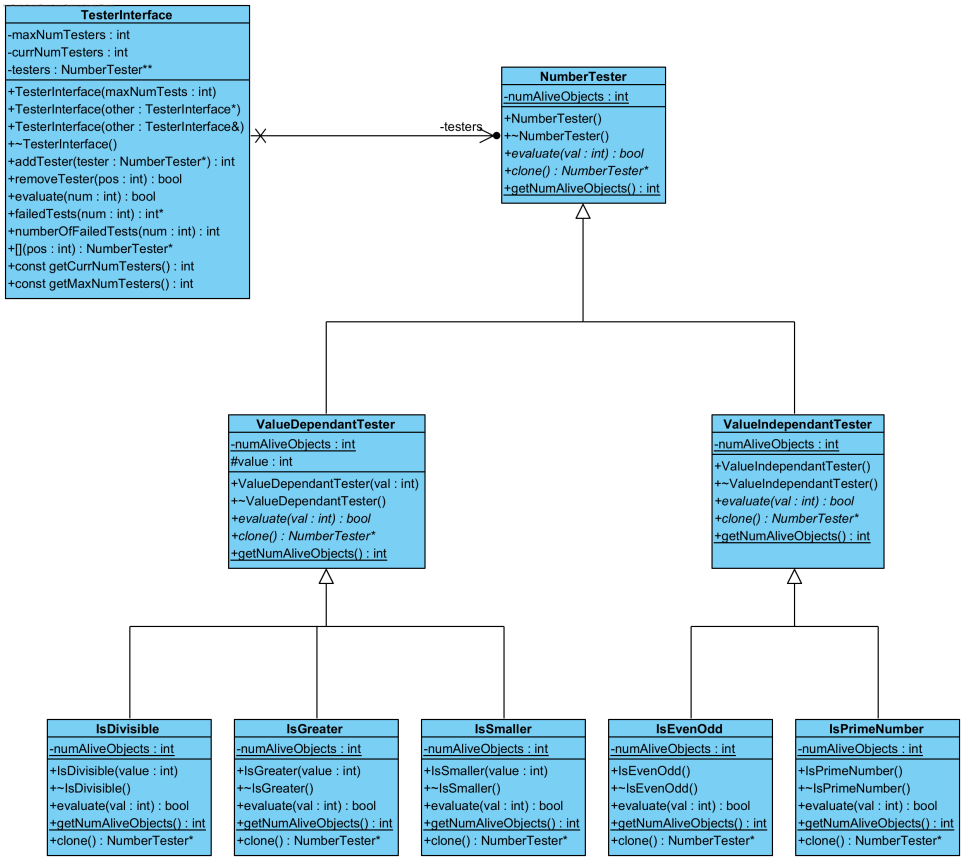


Figure 2: Class diagrams

5 Classes:

5.1 NumberTester:

- members:
  - numAliveObjects: int
    - This is a static variable of the NumberTester class.
    - This variable will keep track of the number of current instantiated NumberTester objects.
    - This variable should be initially initialized to a value of 0.
- functions:
  - NumberTester():
    - This is the constructor for the NumberTester class.
    - This function should increment the numAliveObjects variable.
  - ~NumberTester()
    - This is the destructor for the NumberTester class.
    - This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    - This is a pure virtual function.
  - clone(): NumberTester*
    - This is a pure virtual function.
  - getNumAliveObjects(): int
    - This is a static function.
    - This function should return the static numAliveObjects member of NumberTester class.

5.2 ValueDependantTester

This class has a **public inheritance** relationship with NumberTester.

- member:
  - numAliveObjects: int
    - This is a static variable of the ValueDependantTester class.
    - This variable will keep track of the number of current instantiated ValueDependantTester objects.
    - This variable should be initially initialized to a value of 0.
  - value: int
    - This is the value that will be used by objects of this class.
- functions:
  - ValueDependantTester(val: int)
    - This is the constructor for the ValueDependantTester class.
    - This function should initialize the value member with the passed in parameter.
    - This function should also increment the numAliveObjects variable.

  - ~ValueDependantTester()
    * This is the destructor for the ValueDependantTester class.
    * This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    * This is a pure virtual function.
  - clone(): NumberTester*
    * This is a pure virtual function.
  - getNumAliveObjects(): int
    * This is a static function.
    * This function should return the static numAliveObjects member of ValueDependantTester class.

5.3 ValueIndependantTester:

This class has a **public inheritance** relationship with NumberTester.

- members:
  - numAliveObjects: int
    * This is a static variable of the ValueIndependantTester class.
    * This variable will keep track of the number of current instantiated ValueIndependantTester objects.
    * This variable should be initially initialized to a value of 0.
- functions:
  - ValueIndependantTester():
    * This is the constructor for the ValueIndependantTester class.
    * This function should increment the numAliveObjects variable.
  - ~ValueIndependantTester()
    * This is the destructor for the ValueIndependantTester class.
    * This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    * This is a pure virtual function.
  - clone(): NumberTester*
    * This is a pure virtual function.
  - getNumAliveObjects(): int
    * This is a static function.
    * This function should return the static numAliveObjects member of ValueIndependantTester class.

5.4 IsDivisible

This class has a **public inheritance** relationship with ValueDependantTester.

- members:
  - numAliveObjects: int
    - This is a static variable of the IsDivisible class.
    - This variable will keep track of the number of current instantiated IsDivisible objects.
    - This variable should be initially initialized to a value of 0.
- functions:
  - IsDivisible(int value):
    - This is the constructor for the IsDivisible class.
    - This function should increment the numAliveObjects variable.
    - This function should also initialize the inherited value member with the passed in member
  - ~IsDivisible()
    - This is the destructor for the IsDivisible class.
    - This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    - This function should determine if the passed in value is divisible with the inherited value member.
    - If it is, then the function should return true else false.
  - clone(): NumberTester*
    - This function should return a new IsDivisible pointer initialized with the inherited variable member.
  - getNumAliveObjects(): int
    - This is a static function.
    - This function should return the static numAliveObjects member of IsDivisible class.

5.5 IsGreater

This class has a **public inheritance** relationship with ValueDependantTester.

- members:
  - numAliveObjects: int
    - This is a static variable of the IsGreater class.
    - This variable will keep track of the number of current instantiated IsGreater objects.
    - This variable should be initially initialized to a value of 0.

- functions:
  - IsGreater(int value):
    - This is the constructor for the IsGreater class.
    - This function should increment the numAliveObjects variable.
    - This function should also initialize the inherited value member with the passed in member
  - ~IsGreater()
    - This is the destructor for the IsGreater class.
    - This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    - This function should determine if the passed in value is strictly greater than the inherited value member.
    - If it is then, the function should return true else false.
  - clone(): NumberTester*
    - This function should return a new IsGreater pointer initialized with the inherited variable member.
  - getNumAliveObjects(): int
    - This is a static function.
    - This function should return the static numAliveObjects member of IsGreater class.

5.6 IsSmaller

This class has a **public inheritance** relationship with ValueDependantTester.

- members:
  - numAliveObjects: int
    - This is a static variable of the IsSmaller class.
    - This variable will keep track of the number of current instantiated IsSmaller objects.
    - This variable should be initially initialized to a value of 0.
- functions:
  - IsSmaller(int value):
    - This is the constructor for the IsSmaller class.
    - This function should increment the numAliveObjects variable.
    - This function should also initialize the inherited value member with the passed in member
  - ~IsSmaller()
    - This is the destructor for the IsSmaller class.
    - This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    - This function should determine if the passed in value is strictly smaller than the inherited value member.
    - If it is, then the function should return true else false.

  - clone(): NumberTester*
    * This function should return a new IsSmaller pointer initialized with the inherited variable member.
  - getNumAliveObjects(): int
    * This is a static function.
    * This function should return the static numAliveObjects member of IsSmaller class.

5.7 IsEvenOdd:

This class has a **public inheritance** relationship with ValueIndependantTester.

- members:
  - numAliveObjects: int
    * This is a static variable of the IsEvenOdd class.
    * This variable will keep track of the number of current instantiated IsEvenOdd objects.
    * This variable should be initially initialized to a value of 0.
- functions:
  - IsEvenOdd():
    * This is the constructor for the IsEvenOdd class.
    * This function should increment the numAliveObjects variable.
  - ~IsEvenOdd()
    * This is the destructor for the IsEvenOdd class.
    * This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    * This function should test if the passed in value is even or odd.
    * If it is even the function should return true else false.
  - clone(): NumberTester*
    * This function should return a new IsEvenOdd pointer.
  - getNumAliveObjects(): int
    * This is a static function.
    * This function should return the static numAliveObjects member of IsEvenOdd class.

5.8 IsPrimeNumber:

This class has a **public inheritance** relationship with ValueIndependantTester.

- members:
  - numAliveObjects: int
    * This is a static variable of the IsPrimeNumber class.
    * This variable will keep track of the number of current instantiated IsPrimeNumber objects.
    * This variable should be initially initialized to a value of 0.

- functions:
  - IsPrimeNumber():
    - This is the constructor for the IsPrimeNumber class.
    - This function should increment the numAliveObjects variable.
  - ~IsPrimeNumber()
    - This is the destructor for the IsPrimeNumber class.
    - This function should decrement the numAliveObjects variable.
  - evaluate(val: int): bool
    - This function should test if the passed in value is a prime number or not.
    - If it is a prime number the function should return true else false.
  - clone(): NumberTester*
    - This function should return a new IsPrimeNumber pointer.
  - getNumAliveObjects(): int
    - This is a static function.
    - This function should return the static numAliveObjects member of IsPrimeNumber class.

5.9 TesterInterface

- members:
  - maxNumTesters: int
    - This is the maximum number of testers that can be held by the interface.
    - This is also the size of the testers array.
  - currNumTesters: int
    - This is the current amount of tests held by the interface.
  - testers: NumberTester**
    - This is a dynamic array of dynamic NumberTester objects.
    - The array should initially be populated with nulls.
- functions:
  - TesterInterface(maxNumTests: int)
    - This is the parameterized constructor for the TesterInterface class and should initialize all appropriate member variables.
    - If maxNumTests is less than 1 then initialize the array with a size of 0 and initialize maxNumTests with 0.
  - TesterInterface(other: TesterInterface*)
    - This is a copy constructor for the TesterInterface class.
    - This function should make a deep copy of the passed in parameter
    - If the passed in parameter is NULL initialize all the int variables with a value of 0 and the array with a size of 0.
    - *Hint: remember there is a difference between a null array and an array with a size of 0*
  - TesterInterface(other: TesterInterface&)
    - This is a copy constructor for the TesterInterface class.
    - This function should make a deep copy of the passed in parameter

- ~TesterInterface()
  - This is the destructor for the TesterInterface class.
  - This function should deallocate all the dynamic memory allocated.
- addTester(tester: NumberTester*): int
  - This function should add a deep copy of the passed in NumberTester object to the first index containing null of the testers array and increment the currNumTesters.
  - The function should return the index that the new object was inserted into.
  - If the passed in parameter is null the function should return -1 and not alter the array.
  - If the array is full the function should return -1 and not alter the array.
  - *Hint: use the clone function to make a deep copy of the passed in parameter*
- removeTester(pos: int): bool
  - This function should remove the NumberTester* at the passed in parameter's index in the array and decrement the currNumTesters variable.
  - The NumberTester* object should be deleted and set to null.
  - If the function was able to succesfully remove the NumberTester then the function should return true.
  - If the passed in parameter's index in the array is null then the function should return false.
  - If the passed in parameter's index is outside the bounds of the array the function should return false.
- evaluate(num: int): bool
  - This function should iterate through all the NumberTesters currently in the testers array and pass the passed in parameter to the their evaluate function.
  - If all the NumberTesters in the array return true then the function should return true else it should return false.
  - If the array is empty the function should return false.
- failedTests(num: int): int*
  - This function should return an array containing all the indexes of tests that returned an evaluation result of false.
  - The array should be sized exactly to the number of tests that failed.
  - If no tests failed or if the array does not contain any NumberTesters the function should return an array of size 0.
- numberOfFailedTests(num: int): int
  - This function should return the amount of NumberTester's evaulations that failed.
  - If no tests failed or if the array is empty the function should return 0.
- [ ] (pos: int): NumberTester*
  - This is the overloaded form of the operator[ ].
  - This function should return the pointer at passed in index.
  - If the index is outside of the bounds of the array the function should return null.
- const getCurrNumTesters(): int
  - This is a const function and should return the currNumTesters variable.
  - *In the appropriate h file decalre the function as: int getCurrNumTesters() const.*
- const getMaxNumTesters(): int
  - This is a const function and should return the maxNumTesters variable.
  - *In the appropriate h file decalre the function as: int getMaxNumTesters() const.*

6 Source files

- IsDivisible.h,cpp
- IsEvenOdd.h, cpp
- IsGreater.h, cpp
- IsPrimeNumber.h, cpp
- IsSmaller.h, cpp
- NumberTester.h, cpp
- TesterInterface.h, cpp
- ValueDependantTester.h, cpp
- ValueIndependantTester.h, cpp

7 Allowed libraries

- cstddef
  - Note this is imported such that the NULL constant is defined.
  - Add this to all your h files.

8 Submission

You need to submit your source files, only the cpp files, on the Fitch Fork website (https://ff.cs.up.ac.za/). All methods need to be implemented (or at least stubbed) before submission. Place the above mentioned files in a zip named uXXXXXXXX.zip where XXXXXXXX is your student number. There is no need to include any other files or h files in your submission. Your code should be able to be compiled with the C++98 standard
For this practical you will have 10 upload opportunities. Upload your archive to the Practical 5 slot on the Fitch Fork website.

Appendix C

Supporting Results

C.1 GA Specific Experiments

This section contains the tabulated results of the GA-specific hyperparameter experiments. The discussion of the results can be found in Section 8.3.

C.1.1 Crossover Probability

This section contains the tabulated results of the crossover probability hyperparameter experiments. The discussion of the results can be found in Section 8.3.1.

Average Fitness Value

Table C.1 contains the average fitness value across multiple runs, Table C.2 contains the standard deviation for Table C.1, and Table C.3 contains the final fitness value across multiple runs.

Best Fitness Value

Table C.4 contains the average best fitness value across multiple runs, Table C.5 contains the standard deviation for Table C.4, and Table C.6 contains the final best fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.744	0.586	0.79	0.605
0.25	0.744	0.632	0.799	0.642
0.5	0.749	0.647	0.801	0.636
0.75	0.75	0.648	**0.81**	0.629
1	**0.753**	**0.661**	0.807	**0.646**

Table C.1: Average values for the average fitness for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.097	**0.122**	0.106	0.138
0.25	0.095	0.133	0.102	0.139
0.5	0.094	0.134	0.101	**0.13**
0.75	**0.092**	0.137	**0.098**	0.136
1	0.093	0.14	0.099	0.137

Table C.2: Standard deviation for the average fitness for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.816	0.679	0.869	0.708
0.25	0.815	0.726	0.872	**0.746**
0.5	0.818	0.741	0.872	0.735
0.75	0.817	0.747	**0.876**	0.723
1	**0.819**	**0.759**	0.875	**0.746**

Table C.3: Final values for the average fitness for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.753	0.603	0.802	0.645
0.25	0.752	0.65	0.809	0.68
0.5	0.757	0.667	0.811	0.679
0.75	0.757	0.668	**0.819**	0.665
1	**0.76**	**0.681**	0.816	**0.685**

Table C.4: Average values for the best fitness for each error pool using different values for the crossover probability hyperparameter.

Population Diversity

Table C.7 contains the average diversity score across multiple runs, Table C.8 contains the standard deviation for Table C.7, and Table C.9 contains the final diversity score across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.076	**0.085**	0.078	0.088
0.25	0.077	0.092	0.076	0.088
0.5	**0.075**	0.093	0.075	**0.083**
0.75	**0.075**	0.097	**0.073**	0.089
1	0.076	0.098	0.074	0.089

Table C.5: Standard deviation for the best fitness for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.816	0.679	0.869	0.725
0.25	0.815	0.726	0.872	0.758
0.5	0.818	0.741	0.872	0.748
0.75	0.817	0.747	**0.876**	0.743
1	**0.819**	**0.759**	0.875	**0.761**

Table C.6: Final values for the best fitness for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.064**	**0.069**	**0.064**	**0.075**
0.25	0.074	0.084	0.076	0.092
0.5	0.086	0.101	0.087	0.103
0.75	0.095	0.118	0.087	0.128
1	0.098	0.126	0.095	0.136

Table C.7: Average values for the diversity for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.156**	**0.16**	**0.157**	**0.162**
0.25	0.17	0.18	0.173	0.182
0.5	0.195	0.202	0.195	0.196
0.75	0.211	0.22	0.201	0.233
1	0.22	0.24	0.215	0.246

Table C.8: Standard deviation for the diversity for each error pool using different values for the crossover probability hyperparameter.

Percentage of Errors Found

Table C.10 contains the average percentage of errors found across multiple runs, Table C.11 contains the standard deviation for Table C.10, and Table C.12 contains the final percentage of errors found across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	**0.0**	**0.0**	**0.0**
0.25	**0.0**	**0.0**	**0.0**	**0.0**
0.5	**0.0**	**0.0**	**0.0**	**0.0**
0.75	**0.0**	**0.0**	**0.0**	**0.0**
1	**0.0**	**0.0**	**0.0**	**0.0**

Table C.9: Final values for the diversity for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	67.76	**100.0**	83.029
0.25	**100.0**	75.24	**100.0**	**89.438**
0.5	**100.0**	77.011	**100.0**	86.676
0.75	**100.0**	78.714	**100.0**	85.943
1	**100.0**	**80.663**	**100.0**	89.333

Table C.10: Average values for the percentage of errors for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	**1.177**	**0.0**	**2.453**
0.25	**0.0**	2.854	**0.0**	3.002
0.5	**0.0**	3.299	**0.0**	2.542
0.75	**0.0**	4.063	**0.0**	3.057
1	**0.0**	4.287	**0.0**	3.059

Table C.11: Standard deviation for the percentage of errors for each error pool using different values for the crossover probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	68.0	**100.0**	83.333
0.25	**100.0**	76.0	**100.0**	**90.0**
0.5	**100.0**	78.0	**100.0**	86.667
0.75	**100.0**	80.0	**100.0**	86.667
1	**100.0**	**82.0**	**100.0**	**90.0**

Table C.12: Final values for the percentage of errors for each error pool using different values for the crossover probability hyperparameter.

C.1.2 Crossover Type

This section contains the tabulated results of the crossover type hyperparameter experiments. The discussion of the results can be found in Section 8.3.2.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Average Fitness Value

Table C.13 contains the average fitness value across multiple runs, Table C.14 contains the standard deviation for Table C.13, and Table C.15 contains the final fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.745	0.658	0.798	0.646
One Point Crossover	**0.749**	0.647	**0.801**	0.636
Segmented Crossover	0.745	**0.673**	0.789	0.637
Uniformed Crossover	0.741	0.627	0.798	**0.648**

Table C.13: Average values for the average fitness for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	**0.094**	0.147	**0.099**	0.14
One Point Crossover	**0.094**	**0.134**	0.101	**0.13**
Segmented Crossover	0.095	0.156	0.102	0.135
Uniformed Crossover	0.096	0.138	0.101	0.144

Table C.14: Standard deviation for the average fitness for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.816	0.77	0.871	0.746
One Point Crossover	**0.818**	0.741	**0.872**	0.735
Segmented Crossover	0.816	**0.799**	0.867	0.733
Uniformed Crossover	0.814	0.731	0.871	**0.76**

Table C.15: Final values for the average fitness for each error pool using different values for the crossover type hyperparameter.

Best Fitness Value

Table C.16 contains the average best fitness value across multiple runs, Table C.17 contains the standard deviation for Table C.16, and Table C.18 contains the final best fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.753	0.68	0.808	0.685
One Point Crossover	**0.757**	0.667	**0.811**	0.679
Segmented Crossover	0.753	**0.7**	0.799	0.669
Uniformed Crossover	0.748	0.648	0.808	**0.698**

Table C.16: Average values for the best fitness for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.076	0.105	**0.075**	0.086
One Point Crossover	**0.075**	**0.093**	**0.075**	**0.083**
Segmented Crossover	0.077	0.112	0.078	0.084
Uniformed Crossover	0.078	0.096	0.077	0.094

Table C.17: Standard deviation for the best fitness for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.816	0.77	0.871	0.76
One Point Crossover	**0.818**	0.741	**0.872**	0.748
Segmented Crossover	0.816	**0.799**	0.867	0.744
Uniformed Crossover	0.814	0.731	0.871	**0.783**

Table C.18: Final values for the best fitness for each error pool using different values for the crossover type hyperparameter.

Population Diversity

Table C.19 contains the average population diversity score across multiple runs, Table C.20 contains the standard deviation for Table C.19, and Table C.21 contains the final diversity score across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.094	0.14	0.098	0.129
One Point Crossover	0.086	**0.101**	0.087	**0.103**
Segmented Crossover	0.096	0.157	0.101	0.121
Uniformed Crossover	**0.085**	0.113	**0.076**	0.118

Table C.19: Average values for the diversity for each error pool using different values for the crossover type hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	0.203	0.227	0.201	0.223
One Point Crossover	0.195	0.202	0.195	0.196
Segmented Crossover	0.207	0.223	0.208	0.215
Uniformed Crossover	**0.191**	**0.201**	**0.176**	**0.195**

Table C.20: Standard deviation for the diversity for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	**0.0**	0.002	**0.0**	**0.0**
One Point Crossover	**0.0**	**0.0**	**0.0**	**0.0**
Segmented Crossover	**0.0**	**0.0**	**0.0**	**0.0**
Uniformed Crossover	**0.0**	**0.0**	**0.0**	**0.0**

Table C.21: Final values for the diversity for each error pool using different values for the crossover type hyperparameter.

Percentage of Errors Found

Table C.22 contains the average percentage of errors found across multiple runs, Table C.23 contains the standard deviation for Table C.22, and Table C.24 contains the final percentage of errors found across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	**100.0**	83.817	**100.0**	89.552
One Point Crossover	**100.0**	77.011	**100.0**	86.676
Segmented Crossover	**100.0**	**88.8**	**100.0**	86.267
Uniformed Crossover	**100.0**	76.84	**100.0**	**95.0**

Table C.22: Average values for the percentage of errors for each error pool using different values for the crossover type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	**0.0**	5.871	**0.0**	2.848
One Point Crossover	**0.0**	**3.299**	**0.0**	2.542
Segmented Crossover	**0.0**	7.072	**0.0**	**2.088**
Uniformed Crossover	**0.0**	3.629	**0.0**	4.47

Table C.23: Standard deviation for the percentage of errors for each error pool using different values for the crossover type hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
N Point Crossover	**100.0**	86.0	**100.0**	90.0
One Point Crossover	**100.0**	78.0	**100.0**	86.667
Segmented Crossover	**100.0**	**92.0**	**100.0**	86.667
Uniformed Crossover	**100.0**	78.0	**100.0**	**96.667**

Table C.24: Final values for the percentage of errors for each error pool using different values for the crossover type hyperparameter.

C.1.3 Mutation Probability

This section contains the tabulated results of the mutation probability hyperparameter experiments. The discussion of the results can be found in Section 8.3.3.

Average Fitness Value

Table C.25 contains the average fitness value across multiple runs, Table C.26 contains the standard deviation for Table C.25, and Table C.27 contains the final fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.719	**0.651**	0.778	0.613
0.25	0.737	0.64	0.795	0.627
0.5	0.749	0.647	0.801	0.636
0.75	**0.752**	0.644	0.804	0.651
1	0.75	0.65	**0.809**	**0.658**

Table C.25: Average values for the average fitness for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.098	0.153	0.106	0.151
0.25	0.095	0.145	0.102	0.138
0.5	0.094	0.134	0.101	**0.13**
0.75	**0.093**	**0.133**	**0.099**	0.132
1	**0.093**	0.135	**0.099**	0.138

Table C.26: Standard deviation for the average fitness for each error pool using different values for the mutation probability hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.804	**0.772**	0.863	0.739
0.25	0.812	0.751	0.87	0.732
0.5	0.818	0.741	0.872	0.735
0.75	**0.819**	0.738	0.873	0.73
1	0.818	0.746	**0.876**	**0.751**

Table C.27: Final values for the average fitness for each error pool using different values for the mutation probability hyperparameter.

Best Fitness Value

Table C.28 contains the average best fitness value across multiple runs, Table C.29 contains the standard deviation for Table C.28, and Table C.30 contains the final best fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.729	**0.68**	0.791	0.664
0.25	0.745	0.664	0.806	0.674
0.5	0.757	0.667	0.811	0.679
0.75	**0.76**	0.665	0.813	0.684
1	0.758	0.668	**0.819**	**0.688**

Table C.28: Average values for the best fitness for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.078	0.106	0.078	0.094
0.25	0.078	0.101	0.076	0.089
0.5	**0.075**	**0.093**	0.075	**0.083**
0.75	**0.075**	0.095	0.075	**0.083**
1	0.076	0.098	**0.073**	0.09

Table C.29: Standard deviation for the best fitness for each error pool using different values for the mutation probability hyperparameter.

Population Diversity

Table C.31 contains the average population diversity score across multiple runs, Table C.32 contains the standard deviation for Table C.31, and Table C.33 contains the final population diversity score across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.804	**0.772**	0.863	0.757
0.25	0.812	0.751	0.87	0.754
0.5	0.818	0.741	0.872	0.748
0.75	**0.819**	0.738	0.873	0.751
1	0.818	0.746	**0.876**	**0.763**

Table C.30: Final values for the best fitness for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.165	0.187	0.151	0.224
0.25	0.113	0.147	0.101	0.143
0.5	0.086	0.101	0.087	0.103
0.75	**0.074**	0.092	0.078	**0.084**
1	**0.074**	**0.085**	**0.068**	0.085

Table C.31: Average values for the diversity for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.289	0.291	0.277	0.318
0.25	0.227	0.248	0.215	0.235
0.5	0.195	0.202	0.195	0.196
0.75	**0.176**	0.185	0.178	**0.174**
1	**0.176**	**0.173**	**0.167**	0.177

Table C.32: Standard deviation for the diversity for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	**0.0**	**0.0**	**0.0**
0.25	**0.0**	0.002	**0.0**	**0.0**
0.5	**0.0**	**0.0**	**0.0**	**0.0**
0.75	**0.0**	**0.0**	**0.0**	**0.0**
1	**0.0**	**0.0**	**0.0**	**0.0**

Table C.33: Final values for the diversity for each error pool using different values for the mutation probability hyperparameter.

Percentage of Errors Found

Table C.34 contains the average percentage of errors found across multiple runs, Table C.35 contains the standard deviation for Table C.34, and Table C.36 contains the final percentage of errors found across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	**83.417**	**100.0**	**91.857**
0.25	**100.0**	80.314	**100.0**	88.952
0.5	**100.0**	77.011	**100.0**	86.676
0.75	**100.0**	77.206	**100.0**	86.543
1	**100.0**	78.554	**100.0**	89.095

Table C.34: Average values for the percentage of errors for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	5.571	**0.0**	4.313
0.25	**0.0**	4.767	**0.0**	3.088
0.5	**0.0**	**3.299**	**0.0**	**2.542**
0.75	**0.0**	3.415	**0.0**	2.847
1	**0.0**	4.386	**0.0**	3.198

Table C.35: Standard deviation for the percentage of errors for each error pool using different values for the mutation probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	**86.0**	**100.0**	**93.333**
0.25	**100.0**	82.0	**100.0**	90.0
0.5	**100.0**	78.0	**100.0**	86.667
0.75	**100.0**	78.0	**100.0**	86.667
1	**100.0**	80.0	**100.0**	90.0

Table C.36: Final values for the percentage of errors for each error pool using different values for the mutation probability hyperparameter.

C.1.4 Mutation Type

This section contains the tabulated results of the mutation type hyperparameter experiments. The discussion of the results can be found in Section 8.3.4.

Average Fitness Value

Table C.37 contains the average fitness value across multiple runs, Table C.38 contains the standard deviation for Table C.37, and Table C.39 contains the final fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.749**	0.647	**0.801**	0.636
Random Inversion	0.703	0.621	0.754	0.598
Single Bit Inversion	0.739	**0.663**	0.791	**0.646**

Table C.37: Average values for the average fitness for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	0.094	**0.134**	0.101	0.13
Random Inversion	**0.085**	0.146	**0.093**	**0.128**
Single Bit Inversion	0.091	0.15	0.098	0.138

Table C.38: Standard deviation for the average fitness for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.818**	0.741	**0.872**	0.735
Random Inversion	0.775	0.754	0.824	0.702
Single Bit Inversion	0.807	**0.78**	0.859	**0.743**

Table C.39: Final values for the average fitness for each error pool using different values for the mutation type hyperparameter.

Best Fitness Value

Table C.40 contains the average best fitness value across multiple runs, Table C.41 contains the standard deviation for Table C.40, and Table C.42 contains the final best fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.757**	0.667	**0.811**	0.679
Random Inversion	0.727	0.673	0.783	0.659
Single Bit Inversion	0.753	**0.694**	0.805	**0.689**

Table C.40: Average values for the best fitness for each error pool using different values for the mutation type hyperparameter.

Population Diversity

Table C.43 contains the average population diversity score across multiple runs, Table C.44 contains the standard deviation for Table C.43, and Table C.45 contains the final population diversity score across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	0.075	**0.093**	0.075	**0.083**
Random Inversion	**0.073**	0.11	**0.073**	0.084
Single Bit Inversion	0.075	0.112	0.075	0.093

Table C.41: Standard deviation for the best fitness for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.818**	0.741	**0.872**	0.748
Random Inversion	0.792	0.78	0.849	0.741
Single Bit Inversion	0.811	**0.79**	0.867	**0.768**

Table C.42: Final values for the best fitness for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.086**	**0.101**	**0.087**	**0.103**
Random Inversion	0.303	0.355	0.298	0.378
Single Bit Inversion	0.299	0.334	0.279	0.345

Table C.43: Average values for the diversity for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	0.195	0.202	0.195	0.196
Random Inversion	**0.182**	0.188	0.185	0.192
Single Bit Inversion	0.189	**0.187**	**0.174**	**0.185**

Table C.44: Standard deviation for the diversity for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.0**	**0.0**	**0.0**	**0.0**
Random Inversion	0.194	0.231	0.184	0.218
Single Bit Inversion	0.171	0.222	0.188	0.186

Table C.45: Final values for the diversity for each error pool using different values for the mutation type hyperparameter.

Percentage of Errors Found

Table C.46 contains the average percentage of errors found across multiple runs, Table C.47 contains the standard deviation for Table C.46, and Table C.48 contains the final percentage of errors found across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**100.0**	77.011	**100.0**	86.676
Random Inversion	**100.0**	**89.223**	**100.0**	**92.181**
Single Bit Inversion	**100.0**	86.297	**100.0**	91.705

Table C.46: Average values for the percentage of errors for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**0.0**	**3.299**	**0.0**	**2.542**
Random Inversion	**0.0**	8.041	**0.0**	4.081
Single Bit Inversion	**0.0**	7.03	**0.0**	4.332

Table C.47: Standard deviation for the percentage of errors for each error pool using different values for the mutation type hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Bit-wise Inversion	**100.0**	78.0	**100.0**	86.667
Random Inversion	**100.0**	**94.0**	**100.0**	**93.333**
Single Bit Inversion	**100.0**	90.0	**100.0**	**93.333**

Table C.48: Final values for the percentage of errors for each error pool using different values for the mutation type hyperparameter.

C.1.5 Population Size

This section contains the tabulated results of the population size hyperparameter experiments. The discussion of the results can be found in Section 8.3.5.

Average Fitness Value

Table C.49 contains the average fitness value across multiple runs, Table C.50 contains the standard deviation for Table C.49, and Table C.51 contains the final fitness value across multiple runs.

Best Fitness Value

Table C.52 contains the average best fitness value across multiple runs, Table C.53 contains the standard deviation for Table C.52, and Table C.54 contains the final best fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.831**	0.611	**0.882**	0.629
10	0.82	0.644	0.875	**0.676**
25	0.792	0.64	0.847	0.649
50	0.749	**0.647**	0.801	0.636
75	0.713	0.605	0.757	0.599
100	0.676	0.583	0.735	0.551
125	0.655	0.538	0.7	0.505
150	0.622	0.493	0.677	0.495

Table C.49: Average values for the average fitness for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.038**	**0.041**	**0.049**	**0.046**
10	0.051	0.056	0.056	0.064
25	0.074	0.091	0.08	0.097
50	0.094	0.134	0.101	0.13
75	0.1	0.153	0.111	0.146
100	0.101	0.158	0.115	0.146
125	0.1	0.157	0.113	0.148
150	0.092	0.152	0.111	0.157

Table C.50: Standard deviation for the average fitness for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.842**	0.624	**0.897**	0.633
10	0.84	0.665	0.896	0.696
25	0.833	0.691	0.888	0.699
50	0.818	**0.741**	0.872	**0.735**
75	0.801	0.737	0.853	0.715
100	0.781	0.733	0.843	0.687
125	0.767	0.71	0.821	0.673
150	0.742	0.693	0.806	0.677

Table C.51: Final values for the average fitness for each error pool using different values for the population size hyperparameter.

Population Diversity

Table C.55 contains the average population diversity score across multiple runs, Table C.56 contains the standard deviation for Table C.55, and Table C.57 contains the final population diversity score across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.831**	0.612	**0.884**	0.651
10	0.821	0.646	0.878	**0.702**
25	0.796	0.648	0.852	0.675
50	0.757	**0.667**	0.811	0.679
75	0.723	0.638	0.772	0.642
100	0.69	0.626	0.755	0.61
125	0.672	0.592	0.724	0.585
150	0.643	0.566	0.706	0.587

Table C.52: Average values for the best fitness for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.032**	**0.035**	**0.035**	**0.035**
10	0.043	0.044	0.042	0.045
25	0.061	0.065	0.06	0.066
50	0.075	0.093	0.075	0.083
75	0.081	0.101	0.081	0.086
100	0.08	0.1	0.08	0.085
125	0.077	0.099	0.077	0.087
150	0.07	0.099	0.072	0.085

Table C.53: Standard deviation for the best fitness for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.842**	0.624	**0.897**	0.664
10	0.84	0.665	0.896	0.722
25	0.833	0.691	0.888	0.717
50	0.818	**0.741**	0.872	**0.748**
75	0.801	0.737	0.853	0.729
100	0.781	0.733	0.843	0.707
125	0.767	0.715	0.821	0.702
150	0.742	0.697	0.806	0.698

Table C.54: Final values for the best fitness for each error pool using different values for the population size hyperparameter.

Percentage of Errors Found

Table C.58 contains the average percentage of errors found across multiple runs, Table C.59 contains the standard deviation for Table C.58, and Table C.60 contains the final percentage of errors found across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.008**	**0.009**	**0.011**	**0.01**
10	0.016	0.015	0.015	0.018
25	0.037	0.051	0.039	0.047
50	0.086	0.101	0.087	0.103
75	0.127	0.168	0.142	0.162
100	0.192	0.217	0.172	0.227
125	0.226	0.283	0.234	0.297
150	0.302	0.388	0.276	0.359

Table C.55: Average values for the diversity for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**-0.184**	**-0.185**	**-0.168**	**-0.173**
10	-0.014	-0.02	-0.013	-0.011
25	0.107	0.124	0.106	0.119
50	0.195	0.202	0.195	0.196
75	0.239	0.254	0.253	0.257
100	0.278	0.273	0.268	0.272
125	0.287	0.287	0.294	0.275
150	0.305	0.268	0.298	0.282

Table C.56: Standard deviation for the diversity for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.0**	**0.0**	**0.0**	**0.0**
10	**0.0**	**0.0**	**0.0**	**0.0**
25	**0.0**	**0.0**	**0.0**	**0.0**
50	**0.0**	**0.0**	**0.0**	**0.0**
75	**0.0**	0.001	**0.0**	**0.0**
100	**0.0**	**0.0**	**0.0**	0.001
125	0.001	0.006	**0.0**	0.021
150	0.001	0.031	**0.0**	0.02

Table C.57: Final values for the diversity for each error pool using different values for the population size hyperparameter.

C.1.6 Reproduction Probability

This section contains the tabulated results of the reproduction probability hyperparameter experiments. The discussion of the results can be found in Section 8.3.6.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**100.0**	51.949	**100.0**	66.552
10	**100.0**	59.943	**100.0**	76.619
25	**100.0**	66.046	**100.0**	76.648
50	**100.0**	77.011	**100.0**	86.676
75	**100.0**	80.246	**100.0**	86.229
100	**100.0**	81.629	**100.0**	85.781
125	**100.0**	81.817	**100.0**	89.21
150	**100.0**	**83.04**	**100.0**	**93.152**

Table C.58: Average values for the percentage of errors for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**0.0**	**0.383**	**0.0**	0.658
10	**0.0**	0.424	**0.0**	0.59
25	**0.0**	0.565	**0.0**	**0.356**
50	**0.0**	3.299	**0.0**	2.542
75	**0.0**	4.842	**0.0**	1.938
100	**0.0**	5.091	**0.0**	2.89
125	**0.0**	6.545	**0.0**	5.492
150	**0.0**	8.764	**0.0**	6.947

Table C.59: Standard deviation for the percentage of errors for each error pool using different values for the population size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
5	**100.0**	52.0	**100.0**	66.667
10	**100.0**	60.0	**100.0**	76.667
25	**100.0**	66.0	**100.0**	76.667
50	**100.0**	78.0	**100.0**	86.667
75	**100.0**	82.0	**100.0**	86.667
100	**100.0**	84.0	**100.0**	86.667
125	**100.0**	86.0	**100.0**	93.333
150	**100.0**	**90.0**	**100.0**	**96.667**

Table C.60: Final values for the percentage of errors for each error pool using different values for the population size hyperparameter.

Average Fitness Value

Table C.61 contains the average fitness value across multiple runs, Table C.62 contains the standard deviation for Table C.61, and Table C.63 contains the final fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.748	**0.647**	**0.801**	0.643
0.25	**0.749**	**0.647**	**0.801**	0.634
0.5	**0.749**	**0.647**	**0.801**	0.636
0.75	0.747	**0.647**	**0.801**	**0.645**
1	**0.749**	**0.647**	**0.801**	0.639

Table C.61: Average values for the average fitness for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.094**	**0.134**	**0.101**	0.136
0.25	**0.094**	**0.134**	**0.101**	0.136
0.5	**0.094**	**0.134**	**0.101**	**0.13**
0.75	**0.094**	**0.134**	**0.101**	0.141
1	**0.094**	**0.134**	**0.101**	0.138

Table C.62: Standard deviation for the average fitness for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.817	**0.741**	**0.872**	**0.751**
0.25	**0.818**	**0.741**	**0.872**	0.734
0.5	**0.818**	**0.741**	**0.872**	0.735
0.75	0.817	**0.741**	**0.872**	0.743
1	**0.818**	**0.741**	**0.872**	0.744

Table C.63: Final values for the average fitness for each error pool using different values for the reproduction probability hyperparameter.

Best Fitness Value

Table C.64 contains the average best fitness value across multiple runs, Table C.65 contains the standard deviation for Table C.64, and Table C.66 contains the final best fitness value across multiple runs.

Population Diversity

Table C.67 contains the average population diversity score across multiple runs, Table C.68 contains the standard deviation for Table C.67, and Table C.69 contains the final population diversity score across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.757**	**0.667**	**0.811**	0.685
0.25	**0.757**	**0.667**	**0.811**	0.679
0.5	**0.757**	**0.667**	**0.811**	0.679
0.75	0.755	**0.667**	**0.811**	0.689
1	**0.757**	**0.667**	**0.811**	**0.69**

Table C.64: Average values for the best fitness for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.075**	**0.093**	**0.075**	0.089
0.25	0.076	**0.093**	**0.075**	0.09
0.5	**0.075**	**0.093**	**0.075**	**0.083**
0.75	**0.075**	**0.093**	**0.075**	0.09
1	**0.075**	**0.093**	**0.075**	0.091

Table C.65: Standard deviation for the best fitness for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.817	**0.741**	**0.872**	0.76
0.25	**0.818**	**0.741**	**0.872**	0.758
0.5	**0.818**	**0.741**	**0.872**	0.748
0.75	0.817	**0.741**	**0.872**	0.77
1	**0.818**	**0.741**	**0.872**	**0.771**

Table C.66: Final values for the best fitness for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.087	**0.1**	**0.087**	0.113
0.25	**0.086**	**0.1**	**0.087**	0.119
0.5	**0.086**	0.101	**0.087**	**0.103**
0.75	0.088	**0.1**	**0.087**	0.127
1	**0.086**	**0.1**	**0.087**	0.128

Table C.67: Average values for the diversity for each error pool using different values for the reproduction probability hyperparameter.

Percentage of Errors Found

Table C.70 contains the average percentage of errors found across multiple runs, Table C.71 contains the standard deviation for Table C.70, and Table C.72 contains the final percentage of errors found across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	0.197	**0.202**	**0.195**	**0.194**
0.25	0.196	**0.202**	**0.195**	0.208
0.5	0.195	**0.202**	**0.195**	0.196
0.75	**0.194**	**0.202**	**0.195**	0.206
1	0.195	**0.202**	**0.195**	0.216

Table C.68: Standard deviation for the diversity for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	**0.0**	**0.0**	**0.0**
0.25	**0.0**	**0.0**	**0.0**	**0.0**
0.5	**0.0**	**0.0**	**0.0**	**0.0**
0.75	**0.0**	**0.0**	**0.0**	**0.0**
1	**0.0**	**0.0**	**0.0**	**0.0**

Table C.69: Final values for the diversity for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	**77.011**	**100.0**	89.057
0.25	**100.0**	**77.011**	**100.0**	89.171
0.5	**100.0**	**77.011**	**100.0**	86.676
0.75	**100.0**	**77.011**	**100.0**	92.2
1	**100.0**	**77.011**	**100.0**	**92.476**

Table C.70: Average values for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**0.0**	**3.299**	**0.0**	3.528
0.25	**0.0**	**3.299**	**0.0**	3.808
0.5	**0.0**	**3.299**	**0.0**	**2.542**
0.75	**0.0**	**3.299**	**0.0**	3.642
1	**0.0**	**3.299**	**0.0**	3.721

Table C.71: Standard deviation for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter.

C.1.7 Selection Method

This section contains the tabulated results of the selection method hyperparameter experiments are illustrated. The discussion of the results can be found in Section 8.3.7.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
0	**100.0**	**78.0**	**100.0**	90.0
0.25	**100.0**	**78.0**	**100.0**	90.0
0.5	**100.0**	**78.0**	**100.0**	86.667
0.75	**100.0**	**78.0**	**100.0**	**93.333**
1	**100.0**	**78.0**	**100.0**	**93.333**

Table C.72: Final values for the percentage of errors for each error pool using different values for the reproduction probability hyperparameter.

Average Fitness Value

Table C.73 contains the average fitness value across multiple runs, Table C.74 contains the standard deviation for Table C.73, and Table C.75 contains the final fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.707	0.573	0.764	0.566
Roullette Wheel Selection	0.713	0.61	0.769	0.629
Stochastic Universal Sampling	0.576	0.403	0.606	0.411
Tournament Selection	**0.749**	**0.647**	**0.801**	**0.636**
Truncation Selection	0.55	0.552	0.607	0.536

Table C.73: Average values for the average fitness for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.097	0.143	0.11	0.145
Roulette Wheel Selection	0.094	0.147	0.099	0.136
Stochastic Universal Sampling	0.074	0.108	0.092	0.115
Tournament Selection	0.094	0.134	0.101	0.13
Truncation Selection	**0.018**	**0.071**	**0.026**	**0.097**

Table C.74: Standard deviation for the average fitness for each error pool using different values for the selection method hyperparameter.

Best Fitness Value

Table C.76 contains the average best fitness value across multiple runs, Table C.77 contains the standard deviation for Table C.76, and Table C.78 contains the final best fitness value across multiple runs.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.797	0.707	0.856	0.7
Roullette Wheel Selection	0.8	**0.743**	0.855	**0.739**
Stochastic Universal Sampling	0.673	0.568	0.72	0.571
Tournament Selection	**0.818**	0.741	**0.872**	0.735
Truncation Selection	0.554	0.584	0.613	0.649

Table C.75: Final values for the average fitness for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.716	0.613	0.778	0.623
Roullette Wheel Selection	0.721	0.649	0.778	0.668
Stochastic Universal Sampling	0.635	0.54	0.675	0.582
Tournament Selection	**0.757**	**0.667**	**0.811**	**0.679**
Truncation Selection	0.571	0.623	0.635	0.648

Table C.76: Average values for the best fitness for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.08	0.091	0.081	0.069
Roulette Wheel Selection	0.081	0.092	0.078	0.081
Stochastic Universal Sampling	0.043	0.061	0.037	**0.052**
Tournament Selection	0.075	0.093	0.075	0.083
Truncation Selection	**0.006**	**0.045**	**0.008**	**0.052**

Table C.77: Standard deviation for the best fitness for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.797	0.714	0.856	0.708
Roullette Wheel Selection	0.8	**0.743**	0.855	**0.748**
Stochastic Universal Sampling	0.702	0.629	0.732	0.65
Tournament Selection	**0.818**	0.741	**0.872**	**0.748**
Truncation Selection	0.577	0.648	0.638	0.698

Table C.78: Final values for the best fitness for each error pool using different values for the selection method hyperparameter.

Population Diversity

Table C.79 contains the average population diversity score across multiple runs, Table C.80 contains the standard deviation for Table C.79, and Table C.81 contains the final population diversity score across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.137	0.18	0.131	0.209
Roullette Wheel Selection	0.12	0.162	0.119	0.137
Stochastic Universal Sampling	0.515	0.528	0.564	0.517
Tournament Selection	**0.086**	**0.101**	**0.087**	**0.103**
Truncation Selection	0.722	0.263	0.686	0.384

Table C.79: Average values for the diversity for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	0.23	0.235	0.233	0.248
Roulette Wheel Selection	0.221	0.227	0.212	0.205
Stochastic Universal Sampling	0.127	0.135	0.116	0.136
Tournament Selection	0.195	0.202	0.195	0.196
Truncation Selection	**0.012**	**0.099**	**0.016**	**0.121**

Table C.80: Standard deviation for the diversity for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	**0.0**	0.008	**0.0**	**0.0**
Roullette Wheel Selection	**0.0**	**0.0**	**0.0**	**0.0**
Stochastic Universal Sampling	0.329	0.296	0.351	0.322
Tournament Selection	**0.0**	**0.0**	**0.0**	**0.0**
Truncation Selection	0.716	0.233	0.68	0.218

Table C.81: Final values for the diversity for each error pool using different values for the selection method hyperparameter.

Percentage of Errors Found

Table C.82 contains the average percentage of errors found across multiple runs, Table C.83 contains the standard deviation for Table C.82, and Table C.84 contains the final percentage of errors found across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	**100.0**	78.423	**100.0**	87.886
Roullette Wheel Selection	**100.0**	**82.663**	**100.0**	89.6
Stochastic Universal Sampling	**100.0**	77.549	**100.0**	89.0
Tournament Selection	**100.0**	77.011	**100.0**	86.676
Truncation Selection	**100.0**	77.669	**100.0**	**92.857**

Table C.82: Average values for the percentage of errors for each error pool using different values for the selection method hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	**0.0**	3.241	**0.0**	2.335
Roulette Wheel Selection	**0.0**	3.658	**0.0**	**1.416**
Stochastic Universal Sampling	**0.0**	3.51	**0.0**	1.762
Tournament Selection	**0.0**	3.299	**0.0**	2.542
Truncation Selection	**0.0**	**1.184**	**0.0**	1.682

Table C.83: Standard deviation for the percentage of errors for each error pool using different values for the selection method hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Exponential Rank Selection	**100.0**	80.0	**100.0**	86.667
Roullette Wheel Selection	**100.0**	**84.0**	**100.0**	90.0
Stochastic Universal Sampling	**100.0**	82.0	**100.0**	90.0
Tournament Selection	**100.0**	78.0	**100.0**	86.667
Truncation Selection	**100.0**	78.0	**100.0**	**93.333**

Table C.84: Final values for the percentage of errors for each error pool using different values for the selection method hyperparameter.

C.1.8 Tournament Size

This section contains the tabulated results of the tournament size hyperparameter experiments are illustrated. The discussion of the results can be found in Section 8.3.8.

Average Fitness Value

Table C.85 contains the average fitness value across multiple runs, Table C.86 contains the standard deviation for Table C.85, and Table C.87 contains the final fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.686	0.558	0.727	0.569
2	0.749	0.647	0.801	0.636
5	0.796	0.724	0.847	0.714
10	**0.808**	**0.75**	**0.867**	**0.731**

Table C.85: Average values for the average fitness for each error pool using different values for the tournament size hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.104	0.147	0.117	0.159
2	0.094	0.134	0.101	0.13
5	0.072	0.107	0.078	0.107
10	**0.061**	**0.086**	**0.062**	**0.085**

Table C.86: Standard deviation for the average fitness for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.788	0.701	0.84	0.716
2	0.818	0.741	0.872	0.735
5	0.833	0.779	0.888	0.767
10	**0.837**	**0.786**	**0.893**	**0.77**

Table C.87: Final values for the average fitness for each error pool using different values for the tournament size hyperparameter.

Best Fitness Value

Table C.88 contains the average best fitness value across multiple runs, Table C.89 contains the standard deviation for Table C.88, and Table C.90 contains the final best fitness value across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.702	0.594	0.75	0.626
2	0.757	0.667	0.811	0.679
5	0.799	0.733	0.85	0.735
10	**0.81**	**0.755**	**0.869**	**0.758**

Table C.88: Average values for the best fitness for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.078	0.1	0.079	0.095
2	0.075	0.093	0.075	0.083
5	0.061	0.077	0.063	0.073
10	**0.055**	**0.062**	**0.052**	**0.06**

Table C.89: Standard deviation for the best fitness for each error pool using different values for the tournament size hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.792	0.701	0.84	0.728
2	0.818	0.741	0.872	0.748
5	0.833	0.779	0.888	0.781
10	**0.837**	**0.786**	**0.893**	**0.789**

Table C.90: Final values for the best fitness for each error pool using different values for the tournament size hyperparameter.

Population Diversity

Table C.91 contains the average population diversity score across multiple runs, Table C.92 contains the standard deviation for Table C.91, and Table C.93 contains the final population diversity score across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.189	0.211	0.187	0.204
2	0.086	0.101	0.087	0.103
5	0.033	0.041	0.036	0.046
10	**0.02**	**0.023**	**0.017**	**0.025**

Table C.91: Average values for the diversity for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	0.268	0.262	0.268	0.259
2	0.195	0.202	0.195	0.196
5	0.118	0.125	0.126	0.134
10	**0.09**	**0.089**	**0.08**	**0.097**

Table C.92: Standard deviation for the diversity for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**0.0**	**0.0**	**0.0**	**0.0**
2	**0.0**	**0.0**	**0.0**	**0.0**
5	**0.0**	**0.0**	**0.0**	**0.0**
10	**0.0**	**0.0**	**0.0**	**0.0**

Table C.93: Final values for the diversity for each error pool using different values for the tournament size hyperparameter.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Percentage of Errors Found

Table C.94 contains the average percentage of errors found across multiple runs, Table C.95 contains the standard deviation for Table C.94, and Table C.96 contains the final percentage of errors found across multiple runs.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**100.0**	75.469	**100.0**	88.429
2	**100.0**	77.011	**100.0**	86.676
5	**100.0**	81.446	**100.0**	89.724
10	**100.0**	**81.783**	**100.0**	**89.895**

Table C.94: Average values for the percentage of errors for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinite Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**0.0**	5.186	**0.0**	5.006
2	**0.0**	3.299	**0.0**	2.542
5	**0.0**	2.205	**0.0**	1.52
10	**0.0**	**1.412**	**0.0**	**0.886**

Table C.95: Standard deviation for the percentage of errors for each error pool using different values for the tournament size hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
1	**100.0**	78.0	**100.0**	**90.0**
2	**100.0**	78.0	**100.0**	86.667
5	**100.0**	**82.0**	**100.0**	**90.0**
10	**100.0**	**82.0**	**100.0**	**90.0**

Table C.96: Final values for the percentage of errors for each error pool using different values for the tournament size hyperparameter.

C.2 Omega (ω)-weight Experiments

This section contains the tabulated results of the fitness function weight combinations specific hyperparameter experiments. The discussion of the results can be found in Section 9.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

C.2.1 Average Fitness

Tables C.97, C.98, and C.99 shows the average fitness value, standard deviation, and final fitness value of different ω-weight combinations for different error pools. The discussion of the results can be found in Section 9.1.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.783	0.76	0.835	0.687
Experiment Set 1	0.82	0.822	0.826	0.76
Experiment Set 2	0.872	0.795	0.86	0.793
Experiment Set 3	0.915	0.801	0.904	0.822
Experiment Set 4	0.956	0.81	0.95	0.863
Experiment Set 5	**0.995**	**0.904**	**0.994**	**0.941**
Experiment Set 6	0.749	0.744	0.784	0.697
Experiment Set 7	0.792	0.767	0.828	0.746
Experiment Set 8	0.836	0.804	0.876	0.737
Experiment Set 9	0.88	0.834	0.92	0.804
Experiment Set 10	0.676	0.695	0.756	0.623
Experiment Set 11	0.72	0.733	0.8	0.639
Experiment Set 12	0.763	0.763	0.845	0.712
Experiment Set 13	0.604	0.663	0.726	0.555
Experiment Set 14	0.647	0.691	0.77	0.593
Experiment Set 15	0.532	0.621	0.696	0.495

Table C.97: Average values for the average fitness for each error pool using different weight configurations for the fitness function.

C.2.2 Best Fitness

Tables C.100, C.101,and C.102 shows the average best fitness value, standard deviation, and final best fitness value of different ω-weight combinations for different error pools. The discussion of the results can be found in Section 9.2.

C.2.3 Population Diversity

Tables C.103, C.104, and C.105 shows the average population diversity score, standard deviation, and final population diversity score of different ω-weight combinations for different error pools. The discussion of the results can be found in Section 9.3.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.078	0.123	0.082	0.109
Experiment Set 1	0.205	0.206	0.205	0.207
Experiment Set 2	0.156	0.185	0.163	0.19
Experiment Set 3	0.109	0.148	0.115	0.144
Experiment Set 4	0.062	0.109	0.068	0.117
Experiment Set 5	0.022	0.101	0.03	0.105
Experiment Set 6	0.157	0.175	0.162	0.173
Experiment Set 7	0.11	0.145	0.114	0.144
Experiment Set 8	0.062	0.12	0.067	0.1
Experiment Set 9	0.019	0.096	0.028	0.085
Experiment Set 10	0.108	0.139	0.113	0.127
Experiment Set 11	0.06	0.111	0.066	0.095
Experiment Set 12	0.018	0.088	0.027	0.077
Experiment Set 13	0.058	0.103	0.065	0.085
Experiment Set 14	0.015	0.08	0.025	0.064
Experiment Set 15	**0.013**	**0.071**	**0.024**	**0.054**

Table C.98: Standard deviation for the average fitness for each error pool using different values for the Fitness Function hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.83	0.83	0.884	0.76
Experiment Set 1	0.953	**0.954**	0.956	0.899
Experiment Set 2	0.968	0.923	0.964	0.927
Experiment Set 3	0.979	0.897	0.976	0.912
Experiment Set 4	0.99	0.868	0.988	0.905
Experiment Set 5	**1.0**	0.939	**1.0**	**0.971**
Experiment Set 6	0.849	0.859	0.889	0.817
Experiment Set 7	0.86	0.857	0.9	0.836
Experiment Set 8	0.872	0.866	0.913	0.794
Experiment Set 9	0.883	0.866	0.925	0.824
Experiment Set 10	0.744	0.785	0.826	0.708
Experiment Set 11	0.755	0.79	0.838	0.695
Experiment Set 12	0.767	0.792	0.85	0.732
Experiment Set 13	0.639	0.718	0.763	0.6
Experiment Set 14	0.65	0.718	0.775	0.62
Experiment Set 15	0.537	0.644	0.7	0.512

Table C.99: Final values for the average fitness for each error pool using different weight configurations for the fitness function.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.794	0.778	0.839	0.719
Experiment Set 1	0.824	0.825	0.829	0.808
Experiment Set 2	0.875	0.806	0.864	0.815
Experiment Set 3	0.919	0.819	0.908	0.861
Experiment Set 4	0.959	0.836	0.954	0.917
Experiment Set 5	**1.0**	**0.926**	**1.0**	**0.995**
Experiment Set 6	0.753	0.749	0.788	0.709
Experiment Set 7	0.795	0.777	0.832	0.773
Experiment Set 8	0.839	0.827	0.88	0.784
Experiment Set 9	0.884	0.854	0.925	0.848
Experiment Set 10	0.679	0.704	0.759	0.643
Experiment Set 11	0.723	0.75	0.804	0.665
Experiment Set 12	0.767	0.781	0.85	0.741
Experiment Set 13	0.607	0.674	0.73	0.576
Experiment Set 14	0.651	0.708	0.775	0.623
Experiment Set 15	0.541	0.635	0.7	0.519

Table C.100: Average values for the best fitness for each error pool using different weight configurations for the fitness function.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.067	0.088	0.067	0.076
Experiment Set 1	0.195	0.196	0.195	0.204
Experiment Set 2	0.145	0.167	0.151	0.168
Experiment Set 3	0.096	0.12	0.101	0.109
Experiment Set 4	0.048	0.07	0.05	0.062
Experiment Set 5	**0.0**	0.037	**0.0**	0.029
Experiment Set 6	0.148	0.16	0.151	0.159
Experiment Set 7	0.099	0.117	0.101	0.113
Experiment Set 8	0.05	0.077	0.05	0.059
Experiment Set 9	0.001	0.04	**0.0**	0.022
Experiment Set 10	0.099	0.116	0.101	0.108
Experiment Set 11	0.05	0.072	0.051	0.057
Experiment Set 12	0.001	0.037	**0.0**	0.018
Experiment Set 13	0.05	0.069	0.05	0.056
Experiment Set 14	0.003	0.034	**0.0**	0.019
Experiment Set 15	0.01	**0.029**	**0.0**	**0.01**

Table C.101: Standard deviation for the best fitness for each error pool using different values for the Fitness Function hyperparameter.

C.2.4 Percentage of Errors Found

Tables C.106, C.107, and C.108 shows the average percentage of errors found, standard deviation, and final percentage of errors found using different ω-weight combinations for different error pools. The discussion of the results can be found in Section 9.3.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.833	0.835	0.884	0.775
Experiment Set 1	0.953	**0.954**	0.956	0.948
Experiment Set 2	0.968	0.927	0.964	0.94
Experiment Set 3	0.979	0.906	0.976	0.937
Experiment Set 4	0.99	0.879	0.988	0.959
Experiment Set 5	**1.0**	0.94	**1.0**	**1.0**
Experiment Set 6	0.849	0.859	0.889	0.824
Experiment Set 7	0.86	0.857	0.9	0.852
Experiment Set 8	0.872	0.874	0.913	0.826
Experiment Set 9	0.883	0.866	0.925	0.855
Experiment Set 10	0.744	0.785	0.826	0.722
Experiment Set 11	0.755	0.795	0.838	0.705
Experiment Set 12	0.767	0.792	0.85	0.743
Experiment Set 13	0.639	0.718	0.763	0.615
Experiment Set 14	0.65	0.718	0.775	0.632
Experiment Set 15	0.547	0.644	0.7	0.52

Table C.102: Final values for the best fitness for each error pool using different weight configurations for the fitness function.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.078	0.108	0.081	0.123
Experiment Set 1	0.082	**0.078**	**0.076**	**0.089**
Experiment Set 2	0.073	0.108	0.077	0.118
Experiment Set 3	**0.07**	0.121	0.079	0.105
Experiment Set 4	**0.07**	0.107	0.078	0.12
Experiment Set 5	0.925	0.796	0.929	0.772
Experiment Set 6	0.077	0.093	0.079	0.104
Experiment Set 7	0.076	0.104	0.08	0.11
Experiment Set 8	0.076	0.111	0.077	0.132
Experiment Set 9	0.91	0.615	0.928	0.783
Experiment Set 10	0.078	0.107	0.077	0.112
Experiment Set 11	0.076	0.106	0.08	0.115
Experiment Set 12	0.905	0.614	0.921	0.765
Experiment Set 13	0.076	0.107	0.078	0.108
Experiment Set 14	0.923	0.613	0.919	0.823
Experiment Set 15	0.926	0.605	0.928	0.775

Table C.103: Average values for the diversity for each error pool using different weight configurations for the fitness function.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	0.217	0.255	0.221	0.268
Experiment Set 1	0.22	0.214	0.214	0.233
Experiment Set 2	0.209	0.25	0.217	0.261
Experiment Set 3	0.206	0.259	0.219	0.241
Experiment Set 4	0.205	0.248	0.217	0.264
Experiment Set 5	0.05	**0.123**	**0.041**	0.177
Experiment Set 6	0.215	0.237	0.218	0.242
Experiment Set 7	0.214	0.247	0.22	0.253
Experiment Set 8	0.214	0.258	0.217	0.26
Experiment Set 9	0.075	0.155	0.045	0.173
Experiment Set 10	0.217	0.255	0.216	0.247
Experiment Set 11	0.214	0.254	0.221	0.252
Experiment Set 12	0.066	0.158	0.051	0.163
Experiment Set 13	0.215	0.254	0.218	0.242
Experiment Set 14	0.052	0.159	0.056	**0.123**
Experiment Set 15	**0.034**	0.166	0.045	0.167

Table C.104: Standard deviation for the diversity for each error pool using different values for the Fitness Function hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 1	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 2	**0.0**	0.002	**0.0**	**0.0**
Experiment Set 3	**0.0**	0.002	**0.0**	**0.0**
Experiment Set 4	**0.0**	0.002	**0.0**	**0.0**
Experiment Set 5	0.739	0.453	0.804	0.306
Experiment Set 6	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 7	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 8	**0.0**	**0.0**	**0.0**	0.002
Experiment Set 9	0.643	0.316	0.794	0.392
Experiment Set 10	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 11	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 12	0.712	0.296	0.778	0.459
Experiment Set 13	**0.0**	**0.0**	**0.0**	**0.0**
Experiment Set 14	0.786	0.304	0.765	0.539
Experiment Set 15	0.861	0.284	0.794	0.39

Table C.105: Final values for the diversity for each error pool using different weight configurations for the fitness function.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	**100.0**	91.131	**100.0**	89.762
Experiment Set 1	70.219	33.783	65.214	46.59
Experiment Set 2	**100.0**	87.2	99.986	96.238
Experiment Set 3	**100.0**	85.263	99.986	93.048
Experiment Set 4	**100.0**	83.434	**100.0**	96.352
Experiment Set 5	99.99	92.531	99.986	99.429
Experiment Set 6	**100.0**	87.166	**100.0**	86.457
Experiment Set 7	**100.0**	91.086	**100.0**	**99.571**
Experiment Set 8	**100.0**	93.034	**100.0**	93.105
Experiment Set 9	**100.0**	92.674	**100.0**	95.876
Experiment Set 10	**100.0**	91.24	**100.0**	93.029
Experiment Set 11	**100.0**	**93.086**	**100.0**	89.829
Experiment Set 12	**100.0**	92.629	**100.0**	96.257
Experiment Set 13	**100.0**	93.04	**100.0**	93.038
Experiment Set 14	**100.0**	92.56	**100.0**	95.0
Experiment Set 15	**100.0**	92.611	**100.0**	96.267

Table C.106: Average values for the percentage of errors for each error pool using different weight configurations for the fitness function.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	**0.0**	3.785	**0.0**	1.666
Experiment Set 1	1.353	**0.741**	1.014	3.09
Experiment Set 2	**0.0**	3.293	0.267	2.335
Experiment Set 3	**0.0**	3.128	0.267	2.014
Experiment Set 4	**0.0**	2.638	**0.0**	2.065
Experiment Set 5	0.178	3.863	0.267	2.995
Experiment Set 6	**0.0**	3.635	**0.0**	**1.456**
Experiment Set 7	**0.0**	3.716	**0.0**	2.618
Experiment Set 8	**0.0**	4.2	**0.0**	1.751
Experiment Set 9	**0.0**	4.504	**0.0**	2.654
Experiment Set 10	**0.0**	3.619	**0.0**	1.914
Experiment Set 11	**0.0**	4.083	**0.0**	1.587
Experiment Set 12	**0.0**	4.589	**0.0**	2.608
Experiment Set 13	**0.0**	4.151	**0.0**	1.94
Experiment Set 14	**0.0**	4.774	**0.0**	3.421
Experiment Set 15	**0.0**	4.622	**0.0**	2.427

Table C.107: Standard deviation for the percentage of errors for each error pool using different values for the Fitness Function hyperparameter.

Hyperparameter	Infinity Loop Errors	Logical Errors	Memory Errors	Concurrency Errors
Default experiment	**100.0**	92.0	**100.0**	90.0
Experiment Set 1	70.0	34.0	65.0	46.667
Experiment Set 2	**100.0**	88.0	**100.0**	96.667
Experiment Set 3	**100.0**	86.0	**100.0**	93.333
Experiment Set 4	**100.0**	84.0	**100.0**	96.667
Experiment Set 5	**100.0**	**94.0**	**100.0**	**100.0**
Experiment Set 6	**100.0**	88.0	**100.0**	86.667
Experiment Set 7	**100.0**	92.0	**100.0**	**100.0**
Experiment Set 8	**100.0**	**94.0**	**100.0**	93.333
Experiment Set 9	**100.0**	**94.0**	**100.0**	96.667
Experiment Set 10	**100.0**	92.0	**100.0**	93.333
Experiment Set 11	**100.0**	**94.0**	**100.0**	90.0
Experiment Set 12	**100.0**	**94.0**	**100.0**	96.667
Experiment Set 13	**100.0**	**94.0**	**100.0**	93.333
Experiment Set 14	**100.0**	**94.0**	**100.0**	96.667
Experiment Set 15	**100.0**	**94.0**	**100.0**	96.667

Table C.108: Final values for the percentage of errors for each error pool using different weight configurations for the fitness function.

Appendix D

Example Feedback

This section contains an example of the feedback that can be provided to the students after evaluating their assignment. This example was obtained from the experiments conducted in Chapter 10, and discussed in Section 10.2.5.

Listing D.1: An example of the feedback that GATLAM can provide back to a student.

```
#include <iostream>
#include "TesterInterface.h"
#include "IsDivisible.h"
#include "IsSmaller.h"
#include "IsGreater.h"
#include "IsEvenOdd.h"
#include "IsPrimeNumber.h"
void printAliveObjects(){
std::cout << "AliveObjects:␣";
std::cout << IsEvenOdd::getNumAliveObjects() << ",";
std::cout << IsDivisible::getNumAliveObjects() << ",";
std::cout << IsGreater::getNumAliveObjects() << ",";
std::cout << IsPrimeNumber::getNumAliveObjects() << ",";
std::cout << IsSmaller::getNumAliveObjects() << ",";
std::cout << ValueDependantTester::getNumAliveObjects() << ",";
std::cout << ValueIndependantTester::getNumAliveObjects() << ",";
std::cout << NumberTester::getNumAliveObjects() << std::endl;
}
int main(){
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
TesterInterface* t1 = new TesterInterface(7);
std::cout << "Finished setup" << std::endl;
printAliveObjects();NumberTester* n1 = NULL;
std::cout << "IsSmaller objects: " << IsSmaller::getNumAliveObjects()
    << std::endl;
n1 = new IsSmaller(7);std::cout << "IsSmaller objects: " <<
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add tester: " << t1->addTester(n1) << std::endl;
if( n1 != NULL)
{
delete n1;
n1 = NULL;
};if((*t1)[7] != NULL){
std::cout << "Result of test 7: " << (*t1)[7]->evaluate(7) <<
    std::endl;
} else {
std::cout << "Result of test 7: " << "NULL" << std::endl;
}printAliveObjects();printAliveObjects();NumberTester* n2 = NULL;
std::cout << "IsSmaller objects: " << IsSmaller::getNumAliveObjects()
    << std::endl;
n2 = new IsSmaller(7);std::cout << "IsSmaller objects: " <<
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add tester: " << t1->addTester(n2) << std::endl;
if( n2 != NULL)
{
delete n2;
n2 = NULL;
};if((*t1)[7] != NULL){
std::cout << "Result of test 7: " << (*t1)[7]->evaluate(7) <<
    std::endl;
} else {
std::cout << "Result of test 7: " << "NULL" << std::endl;
}printAliveObjects();printAliveObjects();NumberTester* n3 = NULL;
std::cout << "IsSmaller objects: " << IsSmaller::getNumAliveObjects()
    << std::endl;
n3 = new IsSmaller(7);std::cout << "IsSmaller objects: " <<
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add tester: " << t1->addTester(n3) << std::endl;
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
if( n3 != NULL)
{
delete n3;
n3 = NULL;
};if((*t1)[7] != NULL){
std::cout << "Result␣of␣test␣7:␣" << (*t1)[7]->evaluate(7) <<
    std::endl;
} else {
std::cout << "Result␣of␣test␣7:␣" << "NULL" << std::endl;
}printAliveObjects();printAliveObjects();NumberTester* n4 = NULL;
std::cout << "IsSmaller␣objects:␣" << IsSmaller::getNumAliveObjects()
    << std::endl;
n4 = new IsSmaller(7);std::cout << "IsSmaller␣objects:␣" <<
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add␣tester:␣" << t1->addTester(n4) << std::endl;
if( n4 != NULL)
{
delete n4;
n4 = NULL;
};if((*t1)[7] != NULL){
std::cout << "Result␣of␣test␣7:␣" << (*t1)[7]->evaluate(7) <<
    std::endl;
} else {
std::cout << "Result␣of␣test␣7:␣" << "NULL" << std::endl;
}printAliveObjects();printAliveObjects();NumberTester* n5 = NULL;
std::cout << "IsSmaller␣objects:␣" << IsSmaller::getNumAliveObjects()
    << std::endl;
n5 = new IsSmaller(7);std::cout << "IsSmaller␣objects:␣" <<
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add␣tester:␣" << t1->addTester(n5) << std::endl;
if( n5 != NULL)
{
delete n5;
n5 = NULL;
};if((*t1)[7] != NULL){
std::cout << "Result␣of␣test␣7:␣" << (*t1)[7]->evaluate(7) <<
    std::endl;
} else {
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
std::cout << "Result␣of␣test␣7:␣" << "NULL" << std::endl;                    81
}printAliveObjects();printAliveObjects();NumberTester* n6 = NULL;           82
std::cout << "IsSmaller␣objects:␣" << IsSmaller::getNumAliveObjects()       83
    << std::endl;
n6 = new IsSmaller(7);std::cout << "IsSmaller␣objects:␣" <<                  84
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add␣tester:␣" << t1->addTester(n6) << std::endl;               85
if( n6 != NULL)                                                             86
{                                                                           87
delete n6;                                                                  88
n6 = NULL;                                                                  89
};if((*t1)[7] != NULL){                                                     90
std::cout << "Result␣of␣test␣7:␣" << (*t1)[7]->evaluate(7) <<               91
    std::endl;
} else {                                                                    92
std::cout << "Result␣of␣test␣7:␣" << "NULL" << std::endl;                    93
}printAliveObjects();printAliveObjects();NumberTester* n7 = NULL;           94
std::cout << "IsSmaller␣objects:␣" << IsSmaller::getNumAliveObjects()       95
    << std::endl;
n7 = new IsSmaller(7);std::cout << "IsSmaller␣objects:␣" <<                  96
    IsSmaller::getNumAliveObjects() << std::endl;
std::cout << "Add␣tester:␣" << t1->addTester(n7) << std::endl;               97
if( n7 != NULL)                                                             98
{                                                                           99
delete n7;                                                                  100
n7 = NULL;                                                                  101
};if((*t1)[7] != NULL){                                                     102
std::cout << "Result␣of␣test␣7:␣" << (*t1)[7]->evaluate(7) <<               103
    std::endl;
} else {                                                                    104
std::cout << "Result␣of␣test␣7:␣" << "NULL" << std::endl;                    105
}printAliveObjects();                                                       106
std::cout << "Evaluate␣tester:␣" << (t1->evaluate(11)  ? "True" :           107
    "False") << std::endl;
printAliveObjects();if((*t1)[1] != NULL){                                   108
std::cout << "Result␣of␣test␣1:␣" << (*t1)[1]->evaluate(1) <<               109
    std::endl;
} else {                                                                    110
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
std::cout << "Result␣of␣test␣1:␣" << "NULL" << std::endl;
}std::cout << "Remove␣tester:␣" << (t1->removeTester(1)  ? "True" :
    "False") << std::endl;
if((*t1)[1] != NULL){
std::cout << "Result␣of␣test␣1:␣" << (*t1)[1]->evaluate(1) <<
    std::endl;
} else {
std::cout << "Result␣of␣test␣1:␣" << "NULL" << std::endl;
}printAliveObjects();
printAliveObjects();NumberTester* n8 = NULL;
std::cout << "IsDivisible␣objects:␣" <<
    IsDivisible::getNumAliveObjects() << std::endl;
if(0 != 0){
n8 = new IsDivisible(0);
}
std::cout << "IsDivisible␣objects:␣" <<
    IsDivisible::getNumAliveObjects() << std::endl;
std::cout << "Add␣tester:␣" << t1->addTester(n8) << std::endl;
if( n8 != NULL)
{
delete n8;
n8 = NULL;
};if((*t1)[0] != NULL){
std::cout << "Result␣of␣test␣0:␣" << (*t1)[0]->evaluate(0) <<
    std::endl;
} else {
std::cout << "Result␣of␣test␣0:␣" << "NULL" << std::endl;
}printAliveObjects();
if( t1 != NULL)
{
delete t1;
t1 = NULL;
};
return 0;
}
/*Instructor output:
rm *.o main -f
g++ -g -std=c++98 *.cpp -o main -w
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
./main
Finished setup
AliveObjects: 0,0,0,0,0,0,0,0
IsSmaller objects: 0
IsSmaller objects: 1
Add tester: 0
Result of test 7: NULL
AliveObjects: 0,0,0,0,1,1,0,1
AliveObjects: 0,0,0,0,1,1,0,1
IsSmaller objects: 1
IsSmaller objects: 2
Add tester: 1
Result of test 7: NULL
AliveObjects: 0,0,0,0,2,2,0,2
AliveObjects: 0,0,0,0,2,2,0,2
IsSmaller objects: 2
IsSmaller objects: 3
Add tester: 2
Result of test 7: NULL
AliveObjects: 0,0,0,0,3,3,0,3
AliveObjects: 0,0,0,0,3,3,0,3
IsSmaller objects: 3
IsSmaller objects: 4
Add tester: 3
Result of test 7: NULL
AliveObjects: 0,0,0,0,4,4,0,4
AliveObjects: 0,0,0,0,4,4,0,4
IsSmaller objects: 4
IsSmaller objects: 5
Add tester: 4
Result of test 7: NULL
AliveObjects: 0,0,0,0,5,5,0,5
AliveObjects: 0,0,0,0,5,5,0,5
IsSmaller objects: 5
IsSmaller objects: 6
Add tester: 5
Result of test 7: NULL
AliveObjects: 0,0,0,0,6,6,0,6
```

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA


```
AliveObjects: 0,0,0,0,6,6,0,6
IsSmaller objects: 6
IsSmaller objects: 7
Add tester: 6
Result of test 7: NULL
AliveObjects: 0,0,0,0,7,7,0,7
Evaluate tester: False
AliveObjects: 0,0,0,0,7,7,0,7
Result of test 1: 1
Remove tester: True
Result of test 1: NULL
AliveObjects: 0,0,0,0,6,6,0,6
AliveObjects: 0,0,0,0,6,6,0,6
IsDivisible objects: 0
IsDivisible objects: 0
Add tester: -1
Result of test 0: 1
AliveObjects: 0,0,0,0,6,6,0,6

Student output:
rm *.o main -f
g++ -g -std=c++98 *.cpp -o main -w
./main
Finished setup
AliveObjects: 0,0,0,0,0,0,0,0
IsSmaller objects: 0
IsSmaller objects: 1
Add tester: 0
*/
```

Listing D.1: An example of the feedback that GATLAM can provide back to a student.