TEST DATA GENERATION: TWO EVOLUTIONARY APPROACHES TO MUTATION TESTING

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of doctor of philosophy.

By

Peter Stephen May

May 2007

© Copyright 2007

by

Peter Stephen May

Contents

List of Tables **xvi**

List of Figures **xxvii**

Abstract **xxviii**

Acknowledgements **xxix**

1 Introduction **1**

1.1 Motivation 1

1.1.1 Software Testing Concerns 1

1.1.2 Inspiration from Nature 5

1.2 Contributions 7

1.3 Thesis Structure 8

1.4 Publications 9

2 Mutation Testing **10**

2.1 Introduction 10

2.2 Methodology 12

2.2.1 Mutation Operators 14

2.3 Problems with Mutation Testing 16

2.4 Reducing Computational Expense 17

2.4.1 Do Fewer 17

2.4.2 Do Faster 23

2.4.3 Do Smarter . . . 26
2.5 Increasing Automation . . . 29
2.5.1 Automatically Generating Test Data . . . 30
2.5.2 Automatically Detecting Equivalent Mutants . . . 40
2.5.3 Oracle Problem . . . 43
2.6 Summary . . . 44
2.6.1 Reducing Computational Expense . . . 44
2.6.2 Increasing Automation . . . 46

3 Evolutionary Computation 50
3.1 Introduction . . . 50
3.2 Species Evolution . . . 51
3.2.1 Biological Species . . . 53
3.2.2 Evolution of Species . . . 54
3.2.3 Genetic Algorithms . . . 55
3.3 Immune Systems . . . 58
3.3.1 Biological Immune Systems . . . 59
3.3.2 Evolution in the Immune System . . . 64
3.3.3 Artificial Immune Systems . . . 65
3.4 Summary . . . 70

4 Evolving Test Data 71
4.1 Introduction . . . 71
4.2 Approach Outline . . . 73
4.3 The Mutation Testing System . . . 75
4.3.1 Initialisation . . . 76
4.3.2 Operation . . . 78
4.4 Engineering Framework . . . 83
4.5 Genetic Algorithm . . . 85
4.5.1 Variables . . . 89
4.6 Immune Inspired Algorithm . . . 89

4.6.1 Variables . . . 93
4.7 Differences in the Algorithms . . . 93
4.8 Programs Under Test . . . 94
4.8.1 CalDay . . . 95
4.8.2 DateRange . . . 95
4.8.3 Select . . . 97
4.8.4 TriangleSort . . . 97
4.9 Summary . . . 98

5 Algorithm Comparison **99**
5.1 Introduction . . . 99
5.2 Hypotheses . . . 99
5.3 Qualifiers . . . 102
5.3.1 Statistics . . . 104
5.4 H1 - Primary Hypothesis . . . 104
5.4.1 Conclusion . . . 107
5.5 H2 - Usefulness of Immune Inspired Algorithms . . . 108
5.5.1 Conclusion . . . 111
5.6 H3 - Number of Program Executions . . . 111
5.6.1 Conclusion . . . 118
5.7 H4 - Number of Hard-To-Kill Mutants Found . . . 119
5.7.1 Conclusion . . . 123
5.8 H5 - Test Set Size . . . 123
5.8.1 Conclusion . . . 126
5.9 H6 - Mutation Score per Iteration . . . 126
5.9.1 Conclusion . . . 130
5.10 H7 - Number of Cloned and Random Tests . . . 131
5.10.1 Conclusion . . . 132
5.11 Summary . . . 132

6 Parameter Analysis **135**

6.1 Introduction . . . 135

6.2 Methodology . . . 135

6.2.1 Statistics . . . 136

6.3 Genetic Algorithm for Mutation Testing . . . 137

6.3.1 indSize . . . 138

6.3.2 crossRate . . . 153

6.3.3 mutRate . . . 157

6.4 Immune Inspired Algorithm for Mutation Testing . . . 167

6.4.1 nFittest . . . 167

6.4.2 nWorst . . . 183

6.4.3 cloneRate . . . 192

6.5 Summary . . . 199

6.5.1 Genetic Algorithm for Mutation Testing . . . 200

6.5.2 Immune Inspired Algorithm for Mutation Testing . . . 202

7 Conclusion **207**

7.1 Introduction . . . 207

7.2 Revisiting the Problem Domain . . . 207

7.3 Evolutionary Approaches . . . 210

7.3.1 Parameter Values . . . 213

7.4 Further Work . . . 215

Appendices **220**

A Complementary Functions **220**

B Test Programs **223**

B.1 CalDay . . . 223

B.1.1 Example Mutation Adequate Test Set . . . 223

B.1.2 Initial Test Values . . . 224

B.1.3 Code . . . 224

B.2 DateRange . . . 226
B.2.1 Example Mutation Adequate Test Set . . . 226
B.2.2 Initial Test Values . . . 226
B.2.3 Code . . . 226
B.3 Select . . . 229
B.3.1 Example Mutation Adequate Test Set . . . 229
B.3.2 Initial Test Values . . . 232
B.3.3 Code . . . 232
B.4 TriangleSort . . . 238
B.4.1 Example Mutation Adequate Test Set . . . 238
B.4.2 Initial Test Values . . . 238
B.4.3 Code . . . 239

C Parameter Analysis Results **241**
C.1 Genetic Algorithm for Mutation Testing . . . 241
C.1.1 indSize: Effect on Number of Mutant Executions . . . 241
C.1.2 indSize: Effect on Mutation Score per Iteration . . . 243
C.1.3 indSize: Effect on Number of HTK identified . . . 245
C.1.4 indSize: Effect on Number of Tests . . . 246
C.1.5 crossRate: Effect on Number of Mutant Executions . . . 249
C.1.6 crossRate: Effect on Mutation Score per Iteration . . . 252
C.1.7 crossRate: Effect on Number of HTK identified . . . 255
C.1.8 crossRate: Effect on Number of Tests . . . 256
C.1.9 mutRate: Effect on Number of Mutant Executions . . . 259
C.1.10 mutRate: Effect on Mutation Score per Iteration . . . 262
C.1.11 mutRate: Effect on Number of HTK identified . . . 265
C.1.12 mutRate: Effect on the Number of Tests . . . 266
C.2 Immune Inspired Algorithm for Mutation Testing . . . 268
C.2.1 nFittest: Effect on Number of Mutant Executions . . . 268
C.2.2 nFittest: Effect on Mutation Score per Iteration . . . 272

C.2.3 nFittest: Effect on Number of HTK identified 274
C.2.4 nFittest: Effect on Number of Tests 275
C.2.5 nWorst: Effect on Number of Mutant Executions 277
C.2.6 nWorst: Effect on Mutation Score per Iteration 279
C.2.7 nWorst: Effect on Number of HTK identified 281
C.2.8 nWorst: Effect on Number of Tests 282
C.2.9 cloneRate: Effect on Number of Mutant Executions 284
C.2.10 cloneRate: Effect on Mutation Score per Iteration 288
C.2.11 cloneRate: Effect on Number of HTK identified 290
C.2.12 cloneRate: Effect on Number of Tests 291

Bibliography **293**

List of Tables

2.1 The standard 22 mutagens used in Mothra 15

3.2 Example Roulette Wheel Selection for a 5 individual population. n is a random number generated for selecting individuals. Antibodies are trying to match the binary string 01111110. 56

3.3 Antibody clones generated for the top 3 individuals in a population during clonal expansion. Antibodies are trying to match the binary string 01111110. 68

3.4 Mutated clones generated for the top 3 individuals in a population during clonal expansion. 69

4.5 Mean and σ values for the inputs to the four tested programs, along with an approximate range of input values that will most likely be generated. 83

5.6 Parameter values used for each algorithm. 103

5.7 Average mutation scores and standard deviations achieved for each program after 1, 50 and 500 iterations using an Immune Inspired Algorithm. *All results are to 2 decimal places.* 105

5.8 Mean number of mutants executed for each algorithm achieving its highest mutation score. The number executed (and any standard deviation) has been rounded up to the nearest whole number. . . 106

5.9 The mean average number of hard-to-kill (HTK) mutants killed by each algorithm, and percentage of the total number of HTK mutants, after 500 iterations. *All results are to 2 decimal places.* . 107

5.10 Average mutation scores and standard deviations achieved for each program after 1, 50 and 500 iterations using an Immune Inspired Algorithm. *All results are to 2 decimal places.* 110
5.11 T-test (0.05 level) results for the significance between the mean number of mutants executed for each algorithm, at the highest mutation score obtained by both algorithms in at least 25 runs. The mean number of mutants executed (and the standard deviation) is rounded up to the nearest integer to reflect that a mutant is either executed completely or not at all. *T-test results are to 2 decimal places.* 117
5.12 Mean number of mutants executed for each algorithm achieving its highest mutation score (i.e. results after 500 iterations). The number executed (and any standard deviation) has been rounded up to the nearest whole number. 118
5.13 The mean average number of hard-to-kill (HTK) mutants killed by each algorithm, and percentage of the total number of HTK mutants, after 500 iterations. *All results are to 2 decimal places.* . 121
5.14 T-test results (0.05 level) for the difference in the mean average percentage of HTK mutants killed by each algorithm after 500 iterations. *All results are to 2 decimal places.* 122
5.15 T-test results (0.05 level) for the significance between the average test set sizes generated by each algorithm, at the highest mutation score achieved by both algorithms in at least 25 runs. *All results are to 2 decimal places.* 124
5.16 Final test set sizes generated for the highest mutation score achieved by each program for each algorithm. *All results are to 2 decimal places.* 126
5.17 T-test results for the significance between the average mutation scores for each algorithm after 500 iterations. *Plus/minus figures are 1 standard deviation. All results are to 2 decimal places.* . . . 130

5.18 The mean average number of tests produced by cloning (and mutating) existing tests or by random generation after 500 iterations 132
6.19 Possible parameter values for the Genetic Algorithm. Default values are shown in **bold** font. 138
6.20 The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean numbers of executions (in **bold**). The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. Values replaced with '-' are not important as the ANOVA results are not significant. *Mutation scores and ANOVA/Scheffé results are to 2 decimal places.* 142
6.21 The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the three individual sizes: 10, 20, 30 (individual size of 5 ignored). ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean numbers of executions (in **bold**). The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. Values replaced with '-' are not important as either the ANOVA results are not significant or Scheffé results are not calculated because the individual size=5 experiment only achieved a lower mutation score. *Mutation scores and ANOVA/Scheffé results are to 2 decimal places.* 144

6.22 The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean mutation scores (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 149

6.23 Possible parameter values for the Immune Inspired Algorithm. Default values are shown in bold font. 167

6.24 The mean number of mutants executed (and standard deviation as a percentage of the mean) at the highest mutation score obtained by at least 25 runs for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nFittest` values result in significantly different (at the 0.05 level) mutant execution numbers (in **bold**). *The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. All other results are to 2 decimal places.* 171

6.25 The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nFittest` values (in **bold**) result in significantly different mean mutation scores. *All results are to 2 decimal places.* 180

6.26 The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the six `nWorst` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nWorst` values (in **bold**) result in significantly different mean mutation scores. *All results are to 2 decimal places.* 189

1.27 Functions used by both the Immune Inspired Algorithm and the Genetic Algorithm. 220

1.28 Complementary functions used by the Immune Inspired Algorithm only. 221
1.29 Complementary functions used by the Genetic Algorithm only. . . 222
2.30 The number of mutant programs created by each mutation operator for each program. 223
2.31 A Mutation adequate test set for the *CalDay* program based on the manually determined equivalent mutants. 224
2.32 A Mutation adequate test set for the *DateRange* program based on the manually determined equivalent mutants. 226
2.33 A Mutation adequate test set for the *Select* program based on the manually determined equivalent mutants. 231
2.34 A Mutation adequate test set for the *DateRange* program based on the manually determined equivalent mutants. 238
3.35 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 245
3.36 The mean number of distinct tests needed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the four individual sizes: 5, 10, 20, and 30. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean number of tests produced by each individual size. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *All results are to 2 decimal places.* 248

3.37 The mean number of mutants executed (and standard deviation) at the highest mutation score, obtained by at least 25 runs for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, 1. The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean values of each rate. *Mutation scores and ANOVA values are to 2 decimal places.* 251

3.38 Mean mutation scores (and standard deviation) achieved by each crossover rate after 500 iterations. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean mutation scores achieved by each crossover rate. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 254

3.39 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, 1. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of crossover rates result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 255

3.40 Mean test set sizes (and standard deviations) at the highest mutation score, obtained by at least 25 runs for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, and 1. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean values of each rate; a f_{obt} value on "-" indicates that the value could not be calculated as all six crossover rates exhibited the same test set size (with no variation). *All results are to 2 decimal places.* 258

3.41 The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the five mutation rates: 0.01, 0.02, 0.04, 0.06, 0.08. ANOVA and Scheffé values (at the 0.05 level) indicating whether differences between the means obtained using different pairs of mutation rates are significant. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. All other results are to 2 decimal places.* 261

3.42 The mean mutation score (and standard deviation) achieved after 500 iterations for each of the five mutation rates (0.01, 0.02, 0.04, 0.06, 0.08). ANOVA and Scheffé values (at the 0.05 level) indicating whether differences between the means obtained using different pairs of mutation rates are significant. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *All results are to 2 decimal places.* 264

3.43 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the five mutation rates: 0.01, 0.02, 0.04, 0.06, 0.08. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of mutation rates result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 265

3.44 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `nFittest` values result in significantly different mean percentages (in **bold**). *All results are to 2 decimal places.* 274

3.45 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six `nWorst` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `nWorst` values result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 281

3.46 The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six `cloneRate` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `cloneRate` values result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.* 290

List of Figures

2.1 The Mutation Testing process. *Diagram reproduced from [91]* . . . 13

2.2 Three possible scenarios for the relationship between the sets of tests that kill a weak and strong mutant. 19

2.3 Mutant Distribution by Mutation Operator. *Diagram reproduced from [92]* 22

2.4 Example of original Java PUT method. 25

2.5 Mutated version of the Java PUT in figure 2.4. 25

3.6 A basic Genetic Algorithm. *Diagram adapted from [25]* 55

3.7 Single-point crossover applied to two binary-string chromosomes. *Diagram adapted from [25]* 57

3.8 Single-point mutation applied to a binary-string chromosome. *Diagram adapted from [25]* 58

3.9 Abstract visualisation of shape-space. Within shape-space S, antibodies are dots with a spherical recognition region (V_ϵ), antigens are crosses. V is the volume containing all antibody and antigen-complement shapes *Diagram reproduced from [25]* 62

3.10 The clonal selection principle in response to foreign antigens. Antibodies matching antigens (*(i) Selection*) undergo cloning (*(ii) proliferation*) and differentiate into memory and antibody secreting plasma cells (*(iii) Differentiation*). *Diagram adapted from [25]* . . 65

3.11 A basic Clonal Selection Algorithm. 67

4.12 Test data evolution using mutation testing. 74

4.13 Outline of the MTAIS process engineered for this research. The process operates in two stages indicated by the different arrows. . 75
4.14 Example of how a mutant array is used to access a symbol lookup table. 78
4.15 Abstract class diagram for the mutation testing system implemented for this research. "Classes" shown may refer to groups of classes. For example, the Harness "class" actually refers to multiple classes necessary to perform all the harness activities. 79
4.16 The sigmoidal graph and the effect of varying the spread (σ). . . . 81
4.17 Engineering Framework for Evolutionary Algorithms. *Diagram adapted from [25]* 84
4.18 Single iteration of the Genetic Algorithm process. 86
4.19 Genetic Algorithm for Mutation Testing 87
4.20 Single iteration of the 'Immune Inspired Algorithm for Mutation Testing' process. 90
4.21 Immune Inspired Algorithm for Mutation Testing 91
5.22 Mean mutation scores achieved at each iteration for all four programs using the Immune Inspired Algorithm for Mutation Testing. *Error bars are ± 1 s.d.* 109
5.23 Zoom in of figure 5.26. Mean number of mutants executed (up to 50,000) to achieve at least a specific mutation score for the *CalDay* program. *Error bars are ± 1 s.d.* 113
5.24 Mean number of mutants executed to achieve at least a specific mutation score for the *DateRange* program. *Error bars are ± 1 s.d.* 115
5.25 Mean number of mutants executed to achieve at least a specific mutation score for the *TriangleSort* program. *Error bars are ± 1 s.d.*115
5.26 Mean number of mutants executed to achieve at least a specific mutation score for the *CalDay* program. *Error bars are ± 1 s.d.* . 116
5.27 Mean number of mutants executed to achieve at least a specific mutation score for the *Select* program. *Error bars are ± 1 s.d.* . . 116

5.28 Mean mutation score per iteration for the *DateRange* program. *Error bars are ± 1 s.d.* 128
5.29 Mean mutation score per iteration for the *TriangleSort* program. *Error bars are ± 1 s.d.* 128
5.30 Mean mutation score per iteration for the *CalDay* program. *Error bars are ± 1 s.d.* 129
5.31 Mean mutation score per iteration for *Select* program. *Error bars are ± 1 s.d.* 129
6.32 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *DateRange* program. 140
6.33 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. 140
6.34 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 141
6.35 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *DateRange* program. 147
6.36 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. 147
6.37 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 148
6.38 Effect of varying the individual size on the mean number of HTK mutants identified for all three programs. 150
6.39 Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program. 152
6.40 Effect of varying the crossover rate on the mean number of HTK mutants identified for all three programs. 156

6.41 Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *DateRange* program. 158
6.42 Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. 159
6.43 Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 159
6.44 Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *DateRange* program. 161
6.45 Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *TriangleSort* program. 162
6.46 Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *CalDay* program. 162
6.47 Effect of varying the mutation rate on the mean number of HTK mutants identified for all three programs. 164
6.48 Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *CalDay* program. 166
6.49 Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *DateRange* program. 169
6.50 Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *TriangleSort* program. 170
6.51 Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *CalDay* program. 170
6.52 Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program. 173
6.53 Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program. 173

6.54 Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program. 174
6.55 Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *Select* program. 174
6.56 Effect of varying `nFittest` on the mean mutation score achieved per iteration for the *TriangleSort* program. 178
6.57 Effect of varying `nFittest` on the mean number of HTK mutants identified for all three programs after 500 iterations. 181
6.58 Effect of varying `nFittest` on the mean number of tests needed to achieve a specific mutation score for the *TriangleSort* program. . . 183
6.59 Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 184
6.60 Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 186
6.61 Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 186
6.62 Effect of varying `nWorst` on the mean mutation score per iteration for the *DateRange* program. 190
6.63 Effect of varying `nWorst` on the mean mutation score per iteration for the *TriangleSort* program. 191
6.64 Effect of varying `nWorst` on the mean number of HTK mutants identified for all three programs after 500 iterations. 192
6.65 Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program. 194
6.66 Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program. 194

6.67 Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program. 195
6.68 Effect of varying the `cloneRate` on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . 196
6.69 Effect of varying the `cloneRate` on the mean mutation scores per iteration for the *DateRange* program. 198
6.70 Effect of varying `cloneRate` on the mean number of HTK mutants identified for all three programs. 199
3.71 Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 241
3.72 Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 242
3.73 Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 242
3.74 Effect of varying indSize on the mean mutation score per iteration for the *DateRange* program. 243
3.75 Effect of varying indSize on the mean mutation score per iteration for the *TriangleSort* program. 244
3.76 Effect of varying indSize on the mean mutation score per iteration for the *CalDay* program. 244
3.77 Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program. 246
3.78 Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program. 247
3.79 Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program. 247

3.80 Effect of varying the crossover rate on the mean number of executions to achieve specific mutation scores for the *DateRange* program. 249
3.81 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. 250
3.82 Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 250
3.83 Effect of varying the crossover rate on the mean mutation score per iteration for the *DateRange* program. 252
3.84 Effect of varying the individual size on the mean mutation score per iteration for the *TriangleSort* program. 253
3.85 Effect of varying the individual size on the mean mutation score per iteration for the *CalDay* program. 253
3.86 Effect of varying the crossover rate on the mean number of Tests generated to achieve specific mutation scores for the *DateRange* program. 256
3.87 Effect of varying the individual size on the mean number of Tests generated to achieve specific mutation scores for the *TriangleSort* program. 257
3.88 Effect of varying the individual size on the mean number of Tests generated to achieve specific mutation scores for the *CalDay* program. 257
3.89 Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 259
3.90 Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 260
3.91 Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 260
3.92 Effect of varying mutRate on the mean mutation score per iteration for the *DateRange* program. 262

3.93 Effect of varying mutRate on the mean mutation score per iteration for the *TriangleSort* program. 263
3.94 Effect of varying mutRate on the mean mutation score per iteration for the *CalDay* program. 263
3.95 Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *DateRange* program. 266
3.96 Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *TriangleSort* program. 267
3.97 Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *CalDay* program. 267
3.98 Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 268
3.99 Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 269
3.100Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 269
3.101Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program. 270
3.102Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program. 270
3.103Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program. 271
3.104Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *Select* program. 271

3.105Effect of varying `nFittest` on the mean mutation score per iteration for the *DateRange* program. 272
3.106Effect of varying `nFittest` on the mean mutation score per iteration for the *TriangleSort* program. 273
3.107Effect of varying `nFittest` on the mean mutation score per iteration for the *CalDay* program. 273
3.108Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program. 275
3.109Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program. 276
3.110Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program. 276
3.111Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 277
3.112Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 278
3.113Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 278
3.114Effect of varying `nWorst` on the mean mutation score per iteration for the *DateRange* program. 279
3.115Effect of varying `nWorst` on the mean mutation score per iteration for the *TriangleSort* program. 280
3.116Effect of varying `nWorst` on the mean mutation score per iteration for the *CalDay* program. 280
3.117Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program. 282
3.118Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program. 283

3.119Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program. . 283
3.120Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *DateRange* program. . . . 284
3.121Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program. . . 285
3.122Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *CalDay* program. 285
3.123Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program. 286
3.124Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program. 286
3.125Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program. 287
3.126Effect of varying cloneRate on the mean mutation score per iteration for the *DateRange* program. 288
3.127Effect of varying cloneRate on the mean mutation score per iteration for the *TriangleSort* program. 289
3.128Effect of varying cloneRate on the mean mutation score per iteration for the *CalDay* program. 289
3.129Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program. 291
3.130Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program. 292

3.131Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program. 292

Abstract

Despite nearly 30 years of research and being widely held as a powerful unit testing technique, mutation testing still suffers from a number of problems. For the most part, its hindrance lies with the large number of mutated program executions that must occur, although there are also a number of other challenges such as automation, test case generation, and identifying semantically equivalent programs. Over the years many techniques have tried to help overcome these problems; this thesis provides another, tailored specifically to the area of automatic test case generation using a biological metaphor.

Nature has been regarded as a plentiful supply of ideas and metaphors for our own engineering needs. In computing terms, this practice has spawned many new algorithms primarily designed at optimisation and adaptation, with one of the most infamous being Genetic Algorithms (GA). Recently however, a new paradigm has emerged that offers promising results compared to GAs - Artificial Immune Systems (AIS). As their name suggests, these algorithms look towards the immune system to provide inspiration for solving complex and adaptive problems, often with favourable results.

Genetic Algorithms have previously been applied to mutation testing in the area of test data generation, with reasonable success. This thesis compares such an approach to an immune system inspired algorithm, indicating that the latter is capable of generating higher mutation scores in lower execution times. In addition, an analysis of each algorithm's parameter space is performed, highlighting to practitioners useful settings for each parameter in respect to the program being tested, as well as possible algorithm refinements.

Acknowledgements

There are a number of people whose support, kindness and insight throughout this work has not gone unnoticed. To these people especially, I offer my sincerest gratitude.

To my friends and supervisors, Jon Timmis and Keith Mander. Thank you so much for your endless support, your encouragement and your wise words of wisdom, both inside and outside of work.

To Hannah. We have been through many tough months together, sailing through the highs and suffering the lows. I thank you for your patience, your understanding and your love.

To my good friend Claire. You have been there for me throughout everything - thank you. You truly are a dear friend.

To Bob Barnes (Philips Electronics UK Ltd.). Thank you for your support and understanding.

And finally, to my parents and sister. Thank you. I could not have achieved this without your continual support through these difficult and stressful times.

"There is nothing magical about testing and test design
that immunizes testers against bugs."
- Boris Beizer [8]

Chapter 1

Introduction

1.1 Motivation

1.1.1 Software Testing Concerns

Software engineering is different from most other engineering disciplines in that it generates a product that cannot be physically touched. Furthermore, unlike most other engineering forms, each piece of software is unique; programs are specifically designed to fulfil some (hopefully) well-defined purpose[1]. Software engineering is not entirely independent of the other engineering streams however: all need testing to ensure correct functionality and safety of a product. Without testing, products could be released that are unsatisfactory, or unsafe, for their operational environment.

Software testing can be considered to have two aims [8]. The primary aim is to prevent *bugs* from being introduced into code - prevention being the best medicine. The second is to discover those un-prevented bugs, i.e. to indicate their *symptoms* and allow the *infection* to be *cured*. This naturally prompts four questions[2]: *What is a bug? What are its symptoms? What is an infection? And, how is it cured?*

[1] Albeit possibly mass produced.

[2] To answer these questions, the author made reference to the terms 'error', 'fault' and 'failure' as defined in [97].

What is a bug? When developing software, people often make *errors*. They maybe misunderstand a specification, underestimate the complexity of the software, or inadvertently press the wrong key. Whatever the reason, these errors manifest themselves as *faults* - physical mistakes in the design or implementation of the code. These are commonly referred to as 'bugs', especially in the context of code development.

What are the symptoms of a bug? Any bugs present in the software may cause it to fail when executed. Software *failure* is an observable event where the software's execution differs from its specification. Therefore, the failure observed is a symptom of the bug. These symptoms can be just as wide ranging as the symptoms of the flu in humans. For example, a programmer may erroneously interpret a specification and insert a fault which incorrectly rounds a value. However, the failure resulting from this could range from a trivial annoyance such as incorrect change at a till, to something as drastic as the loss of human life, such as a death caused by an incorrect drug prescription.

What is an infection? In biology, an infection is due to the presence of a bug in the body that may or may not cause symptoms to be expressed. Similarly, an infection in code refers to software that has at least one fault that may or may not express symptoms when executed. Simply, the code is infected with a bug.

How is an infection cured? Curing an infection can be viewed as a two stage process forming the basis of testing: Every bug has to first be identified, and then corrected. Identification is primarily achieved by executing tests on a program in an attempt to reveal symptoms of a bug. If symptoms are expressed, then the test has caused the program to execute differently from its specification, and so has provided information useful in identifying a fault. Once identified, this fault can be corrected, which in many instances is a trivial matter requiring a simple change to the source code. For example, the wrong variable name, or an incorrect relational operator was used, the correct versions of which should be fairly obvious to the programmer. Other instances however, may need more fundamental changes that require the rewriting of numerous lines of code. In either case though, the faulty

code has been identified and fixed.

This process of identifying and correcting faults continues until all bugs in the code have been found, at which point a set of tests will have been generated that have reduced the failure rate of the program - a *cure*. So far however, the assumption has been made that a programmer knows *a priori* whether faults are present in the software. This is a contradiction. If the faults were known beforehand, then testing (in a fault-detection capacity) would be unnecessary. Not knowing beforehand however, poses an interesting dilemma: *how does a tester distinguish between a "poor" test that is incapable of displaying a fault's symptoms, and a "good" test when there are simply no faults to find?* Neither situation provides a useful metric. A heuristic to help aid this problem uses the notion of *test set adequacy* as a means of measuring how "good" a test set is at testing a program [108]. The key to this is that "goodness" is measured in relation to a predefined *adequacy criteria*, which is usually some indication of program coverage. For example, the statement coverage criteria requires that a test set executes every statement in a program at least once. If a test set is found inadequate relative to the criteria (e.g. not all statements are executed at least once), then further tests are required. The aim therefore, is to generate a set of tests that fully exercise the adequacy criteria.

Typical adequacy criteria such as statement coverage and decision testing (exercising all true and false paths through a program) rely on exercising a program with an increasing number of tests in order to improve the reliability of that program. They do not, however, focus on the cause of a program's failure, namely the faults. One criteria does. Known as *mutation testing*, this criteria generates versions of the program containing simple faults and then finds tests to indicate their symptoms. If an adequate test set can be found that reveals the symptoms in all the faulty versions, then one's confidence that the program is correct increases. This criterion forms an adequacy measure for the cure.

The term "cure", however, tends to suggest some medicine that can be repeatedly applied to "fix" some ailment in a body every time it occurs. Perhaps a more

pertinent view would be to consider the generated test set as a *vaccine*. This could be seen as a simple change of terminology, but it does emphasise the preventative measure of the test set and so enhance the primary aim of testing. A vaccine, for example, does not stop a bug infecting a body, but instead, endeavours to prevent the bug causing symptoms by identifying it every time it is seen, thus enabling its swift removal. In software terms, the test set does not stop faults occurring in the code, but instead, endeavours to prevent them from causing software failures by identifying the faults for subsequent removal. The overall effect is to reduce the failure rate, or improve the reliability, of a program.

Unfortunately, a vaccine rarely remains sufficient. Software is under a constant pressure to change, for two reasons. First, incorrect code needs correcting. It is rare that code is correct from the start and this prompts a cycle of testing and correcting faults. As Beizer [8] noted in his first law of software testing, *The Pesticide Paradox*, "*every method you use to prevent or find bugs leaves a residue of subtler bugs against which those methods are ineffectual*". Subsequently, the tests that were once employed and worked well for a previous software version may no longer be enough. *Test sets wear out!* To compensate for this, new tests are required, but the fact remains that these new tests are likely to expire too. The second reason for software's constant pressure to change is due to humanity's thirst for technological advancement. Again, Beizer sums this up well with his second law, *The Complexity Barrier*, "*software complexity (and therefore that of 'bugs') grows to the limits of our ability to manage that complexity*" [8]. If a piece of software works well, then it becomes less complex as a user's understanding of it grows, allowing new features to be demanded and added until software again reaches the complexity barrier. And so it is with bugs. As a programmer's understanding of them increases, their complexity drops, but as new features increase the software complexity again, the bugs faced become subtler. So once again, the tests that worked well at one level of complexity, may no longer be effective.

This problem is exacerbated further by human nature; we want to learn and

not repeat the same mistakes time and time again. It is likely, therefore, that a programmer who takes note of the faults they so often have to correct, will try not to make them. They will try to adapt their habits, evolve their programming style, and in the process possibly make new faults. So we see again, the tests that once worked need to adapt in order to continue to be effective. *The vaccine needs to evolve.*

1.1.2 Inspiration from Nature

In biology, vaccines are designed to help improve a body's natural defence mechanism against infection from a wide variety of invading organisms and viruses, known as *pathogens*. They do this by providing an artificial exposure to a virus, prompting the body's immune system to pre-develop receptor cells capable of recognising the virus. Subsequent invasions by that pathogen are identified and removed quicker because of presence of these pre-developed receptors, thereby limiting the virus' overall impact on the body. Had the body not received a vaccination (and had not been infected by the pathogen before), then the necessary receptors would also have been evolved as part of the immune response to the infection - indeed it is this immune response that allows vaccines to work. Evolving receptors takes a relatively long time however, giving the virus a greater opportunity to overwhelm the body. Overall then, vaccines are advantageous in quickly identifying viruses and limiting their impact, just as a software vaccines should help quickly identify faults and limit the failures they induce.

Unfortunately, the general process of evolution that helps the immune system recognise new viruses, also aids pathogens in evading the immune system. New virus strains can evolve which an immune system has no receptors for. As a consequence, new vaccines are needed to prevent infections by these new strains. A good example is the influenza (flu) virus which often undergoes genetic changes, forcing vaccines (and therefore the immune system) to be reworked annually [93]. The development of updated vaccines however, provokes the need for the pathogen to mutate again in order to survive. And so the cycle continues. Pathogenic

evolution forces vaccine evolution, which in turn encourages pathogenic evolution. So it is with software testing, as tests (vaccines) evolve to detect new failures, the associated faults are changed, and the software evolves, possibly introducing new faults. Continuously evolving tests helps to limit the failures produced by the evolving code.

Evolution occurs constantly in nature, and on many scales, and is often driven by the need for survival - for example the body wishes to survive against viruses and so the immune system must adapt to defend it. More generally this can be thought of as a change in an organism's environment prompts a need for it to adapt in order to survive. This is seen elsewhere in nature, not just the immune system. Changes in the environment (whether man-made or natural) cause humans (and other animals) to evolve over many generations. Those individuals who develop stronger solutions to the environmental changes are more likely to survive, reproduce and pass on their solution (genetic material) to their children. Individuals who fail to adapt quickly enough are usually overwhelmed by the changes to their environment and fail to reproduce - their weaker solution dies out. This process is often referred to as *Darwinian evolution* and encompasses a *survival of the fittest* metaphor.

Evolution can be seen to provide the key to survival in an ever changing world. It is no surprise then, that engineers should turn to evolution as inspiration to complex problems. If the problem space of a computational task is forever changing, it is likely that the optimal solution will change too. This has led computer engineers to use evolutionary strategies as inspiration to a wide diversity of complex problems, creating techniques known as *Evolutionary Algorithms* (EAs), the most infamous being *Genetic Algorithms* (GAs) [40, 76]. Commonly, these algorithms evolve a population of solutions (to some "environmental" problem) over a number of iterations, favouring the stronger solutions at any one stage. As this process continues, an optimal solution emerges. Over the recent years however, EAs have seen an additional technique develop based on the workings of the

vertebrate immune system. Known as *Artificial Immune Systems* [25], these algorithms evolve a population of artificial immune cells (solutions) through cloning and mutation, in order to survive against dynamic viruses (the problem space).

Having this metaphor in place between natural and artificial immune systems, raises the question: *if the immune system can develop a set of receptors to detect unseen viruses, can Artificial Immune Systems be used to evolve an effective set of tests to detect software faults?*

1.2 Contributions

The principal goal of this thesis is to compare the use of an Artificial Immune System metaphor for generating mutation adequate test data with a Genetic Algorithm approach. As will be seen later, research has previously been conducted in the area of automatic test data generation for mutation testing, but not using an immune system metaphor. Typically, research has focused on a Genetic Algorithm methodology [5, 6, 7, 13, 46], and so this will form the basis for a comparison system. The evaluation criteria itself will predominantly centre on the time the algorithm takes to perform, as this is a fundamental drawback of mutation testing. Other measures will also be used (such as test set size).

The contributions of this thesis include:

1. A thorough review of literature surrounding mutation testing and its current state-of-the-art, with particular emphasis placed on the various problems with the technique and their current work-arounds (Chapter 2);

2. A review of the biological metaphors inspiring both of the major evolutionary algorithms used in this thesis (Genetic Algorithms and Artificial Immune Systems), along with details of typical implementations of each algorithm's component processes (Chapter 3);

3. Mutation testing is revisited and the measures used to fairly compare algorithm effectiveness are described. A new AIS algorithm is then presented

(based on CLONALG [24]), along with a GA, both aimed at improving a set of test data for a given program (Chapter 4);

4. A statistical comparison between the AIS and GA algorithms is performed using non-optimised parameter settings, indicating that the AIS algorithm is at least, and often more, effective than the GA approach (Chapter 5);

5. A detailed statistical analysis of each algorithm's parameters is undertaken providing useful insights into the appropriate parameter values needed when testing a program. Explanations are presented for each parameter's effect, although further evidence is required to ascertain the validity of the explanations (Chapter 6).

1.3 Thesis Structure

This thesis is organised into the following chapters.

Chapter 2 details the principal components of mutation testing, describing the basic methodology and the fundamental assumptions necessary to undertake this approach. Discussion then turns to mutation testing's two broad categories of problems: computational expense and automation. Both areas are explored, with an emphasis on examining past research that attempts to overcome (or at least improve upon) these problems.

Chapter 3 discusses the use of evolution in nature as inspiration for solving computational problems. In particular, focus is placed on inspiration drawn from the Darwinian evolution of species, and evolution within the immune system. Details are given about the current understanding of the biological workings of such systems. Finally each approach is concluded with an explanation of the artificial system derived from its natural counterpart. Every effort is made to keep these discussions relevant to the overall thesis.

Chapter 4 describes the approach taken within this research to applying a Genetic Algorithm and an Artificial Immune System to the field of mutation

testing. Both algorithms are outlined in detail, with the chapter concluded by a discussion on the fundamental differences between the two.

Chapter 5 starts by identifying the main hypothesis of this research, which is broken down into constituent hypotheses. Variable settings are selected for each algorithm, and comparisons made on all sub-hypotheses. Results indicate definite gains from using an Artificial Immune System approach to evolving test data.

Chapter 6 details the effect each algorithm's parameters have on overall performance, in order to provide guidance for practitioners wishing to use this technique. Results are presented for each algorithm, divided into sections for each parameter. Some of the results graphs are presented in this chapter, although all graphs can be found in Appendix C.

Finally, chapter 7 summarises the work undertaken and the overall major conclusions - an Artificial Immune System approach is consistently more effective at generating test data than a Genetic Algorithm. This chapter also provides possible directions for future research.

1.4 Publications

The following two papers were written when researching this thesis:

- P. May, K. Mander, and J. Timmis, "Mutation Testing: An Artificial Immune System Approach", in *UK-Softest*, UK Software Testing Workshop, University of York, Uk. September 2003, pp.

- P. May, K. Mander, and J. Timmis, "Software Vaccination: An Artificial Immune System Approach", in *Proceedings of the 2nd International Conference on Artificial Immune Systems (ICARIS2003)*, ser. Lecture Notes in Computer Science, J. Timmis, P. Bentley, and E. Hart, Eds. no. 2787, Springer-Verlag, September 2003, pp. 81-92.

Chapter 2

Mutation Testing

2.1 Introduction

The effectiveness of tests at indicating program faults is crucial to software testing; some tests are better than others, but it is often difficult to identify which these are. A test may be 'good' in the sense that it is capable of detecting a particular fault that no other test can, but difficult to identify because the exposing fault is not present in the code. Conversely, a test may be 'poor' in the sense that it is incapable of detecting any faults, but again difficult to identify because we do not know if any faults are present in the code. In both cases, there is no way to measure the test's effectiveness. The key is to know whether faults exist prior to testing, but knowing this makes testing redundant. To aid with this paradox, criteria are used to provide a requirement for test data adequacy, and so give a measure for improving a test set. For example, tests are iteratively improved until they execute all statements in the code (statement testing), or execute all true and false branch decisions (branch testing). Compliance with a criterion deems the test data as *adequate* (with respect to that criterion) and therefore more likely to indicate faults if they exist. However, typical criteria do not focus on the *cause* of a program's failures - the faults; the *mutation adequacy* criterion does. This provides a measure for test data effectiveness by showing

that the tests can expose all possible simple[1] faults of a program in much the same way as statement testing displays effectiveness by ensuring that every line of code has been executed. Most likely, however, a test set will not be able to identify all faults, in which case the mutation adequacy criterion gives a measure to determine improvement in selecting a new test set; for example, a test set that detected 80% of faults would allow test generation to focus on the remaining 20%. This measure allows a controlled, iterative improvement of test data, forming the basis of *mutation testing*.

DeMillo et. al. [27] introduced mutation testing to provide a means of iteratively improving test data adequacy with respect to some Program Under Test (PUT). Based on the mutation adequacy criterion, fault induced variants of the PUT are executed with a test set to ascertain how many variants fail[2]. The more that fail, the greater the adequacy of that test set. The tester's aim is to generate new test data that improves upon the adequacy of the existing tests. A rather useful consequence of this approach is that improving the adequacy of the test data improves the tester's confidence in the correctness of the PUT. Arguably it can be reasoned that a tester's confidence in a program is going to be greater where more discriminating tests are used. However, it is the method for improving the test adequacy that offers more substantial support. If a fault induced variant of the PUT fails when executed with some test, and the PUT succeeds, then the PUT itself cannot contain that fault (i.e. the PUT is the correct version of the program with respect to that fault). Testing every possible variant helps test that none of those faults are present in the PUT. Evolving test data in this manner conjointly enables the tester to evolve their confidence in the PUT's correctness, but this is not without restrictions.

Mutation testing makes three fundamental assumptions. The first is the *competent programmer hypothesis*, which states that a competent programmer creates

[1] A simple fault would be produced by a single lexeme difference or variable name replacement, for example.

[2] Produce an incorrect output for that test input.

programs that, if not correct, are close to being correct [27, 37]. Such programmers have an idea of what the desired program should look like and strive to achieve that. The second is the *coupling effect* that states that test data capable of identifying simple errors (single lexeme differences, for example) are implicitly able to identify more complex errors [27]. This effect is supported by empirical evidence [81] and allows mutation testing to operate on a restricted set of simple PUT variations, whilst generating tests effective at identifying complex faults. The third assumption is the presence of an *oracle* for classifying the output of a test execution as correct or not. How this oracle is implemented is in itself a complex problem and beyond the scope of this research. These assumptions underpin the foundations of mutation testing and provide restrictions to its usage.

2.2 Methodology

Mutation testing is an iterative procedure to improve test data with respect to a program, as indicated in Figure 2.1. The initial parameters to the process are the *PUT* (Program Under Test) and a test set population, *T*. Initially, by using an oracle, the PUT must be shown to produce the desired outputs when executed with the tests in T. If any of the outputs are erroneous, then T has demonstrated that the PUT contains a fault. This should be corrected before resuming the process.

Having identified all tests in T produce correct outputs, the next stage is to generate a set, *M*, of fault induced variants of the PUT that correct for simple faults that could have occurred. Each variant, or *mutant*, differs from the PUT by a small amount, such as a single lexeme, and is generated by a *mutation operator*. *Mutagens*, as they are otherwise known, alter the semantics of the PUT depending on the faults they classify. For example, the *relational operator* mutagen will generate a number of mutants where each one has an instance of a relational operator replaced by another. These mutants are then executed with T and their outputs compared against the outputs from the PUT. If a mutant can be shown

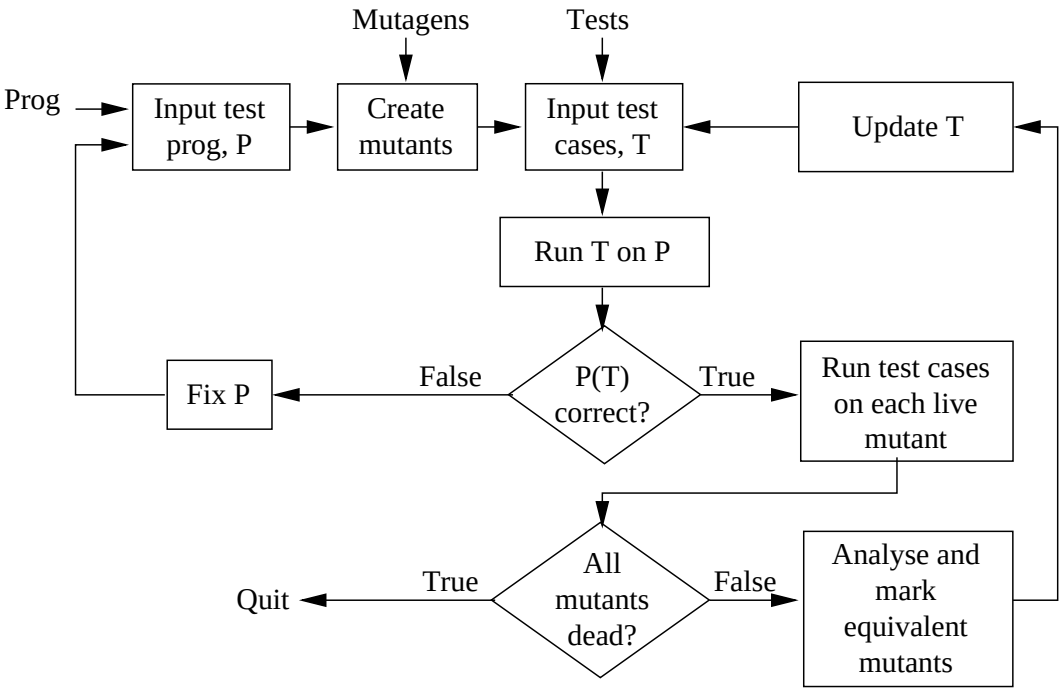

Figure 2.1: The Mutation Testing process. *Diagram reproduced from [91]*

to produce an undesirable result (i.e. its result differ from the PUT's), then the fault that that mutant corrects is proven not to occur in the PUT. Subsequently, the tester's confidence that the PUT is correct, increases. Any such undesirable mutants become superfluous, as a test exists to distinguish them, and so they are *killed* (removed) from the mutant set M.

Once all the tests in T have been executed on all mutants in M, those mutants that remain alive (that still exist in M) are so far indistinguishable from the original. In other words, there does not exist a test in T that will cause these *living* mutants to compute a different output from P. These mutants become the target for the next iteration, where new test data will be generated in the attempt to detect them.

This process continues until all mutants in M are killed. Killing mutants, however, is not a trivial task as some mutants may be semantically the same as the PUT. These mutants are known as *equivalent mutants*, and will always produce the same output as the PUT regardless of the test applied. As a consequence,

M can never be completely emptied when equivalent mutants exist. This has an adverse effect on mutation testing as the tester does not know whether the tests that remain in M are equivalent or not. If they are equivalent then no test will kill them; if not, then a test that will distinguish them has so far not been found. Worse still, determining whether a mutant is equivalent is undecidable [17], and so typically the decision is left for the tester to establish manually.

Reducing the set M to the empty set offers a useful metric for assessing the quality of T with respect to the PUT. If T manages to kill all non-equivalent mutants then the tests are capable of identifying that none of the faults the mutants try to repair are present in the PUT. Before T reaches this adequate state however, it will only discriminate a proportion of the mutants, indicated by the number of non-equivalent mutants killed from M. This proportion is the *mutation score* (MS) and is more formally defined as:

$$MS = \frac{\mid mutants_{killed} \mid}{\mid mutants \mid - \mid equivalents \mid} \tag{2.1}$$

That is, the proportion of mutants killed (identified) out of all non-equivalent mutants. As this proportion increases (i.e. more non-equivalent mutants are killed), so does the adequacy of the test data and the tester's confidence in the correctness of the PUT. Subsequent iterations therefore involve generating new tests to improve the adequacy of T.

2.2.1 Mutation Operators

The majority of research into mutation testing is based around Fortran programs primarily because of the easy availability of the Mothra mutation system [79]. Mothra uses 22 mutagens to generate mutants for Fortran-77 programs [48], as shown in table 2.1. These operators were developed and refined over 20 years ago, with many still used in some form today - for example, [7] uses a logical operator replacement operator and a statement suppression operator similar to Mothra's LCR and SDL mutagens; muJava still uses the ABS, AOR, LCR, ROR and UOI

Mutation Operator	Description
AAR	Array reference for array reference replacement
ABS	Absolute value insertion
ACR	Array reference for constant replacement
AOR	Arithmetic operator replacement
ASR	Array reference for scalar variable replacement
CAR	Constant for array reference replacement
CNR	Comparable array name replacement
CRP	Constant replacement
CSR	Constant for scalar variable replacement
DER	DO statement end replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	Logical connector replacement
ROR	Relational operator replacement
RSR	RETURN statement replacement
SAN	Statement analysis
SAR	Scalar variable for array reference replacement
SCR	Scalar for constant replacement
SDL	Statement deletion
SRC	Source constant replacement
SVR	Scalar variable replacement
UOI	Unary operator insertion

Table 2.1: The standard 22 mutagens used in Mothra

operators [59].

Mutation operators are designed to reveal faults in the PUT when executed. A good example is the relational operator replacement mutagen, *ROR*. A mutant generated from this operator will be identical to the PUT except that a single relational operator will be replaced with another nonidentical relational operator, for example, a statement such as $x < y$ will be replaced by $x \leq y$. The idea behind this is to detect whether, in this case $<$, is the correct relational operator to use. If the PUT is indeed correct, then it is possible to find tests that generate incorrect outputs for all the non-equivalent relational operator variants, thereby eliminating those mutants from being the correct version of the program. However, it is not known, *a priori* to testing, whether the PUT is correct. Instead, the PUT has to be assumed correct, unless a test can prove otherwise. Under these circumstances then, the proportion of non-equivalent mutants the test set kills, is a measure of

how adequate the test set is and a measure of confidence in the correctness of the PUT.

Recently, research into mutation testing has focussed on developing new mutagens specifically for Object-Oriented environments, with particular emphasis on Java [10, 47, 58, 60, 88]. Object-Oriented (OO) languages differ from traditional programs in many ways, in particular in structure and in paradigms such as inheritance and polymorphism. These differences cause possibilities for new faults to be introduced that need to be represented in OO mutation testing systems for them to be effective. This research, however, focusses on the testing of simple programs, where it is reasonably safe to assume that most traditional mutation operators still suffice [58].

2.3 Problems with Mutation Testing

Although widely believed to be a powerful unit testing technique [48, 82, 91, 103], mutation testing suffers from a number of problems that restrain its adoption into mainstream industry. These can be categorised broadly into two groups: *computational expense* - how much time and effort is required to perform mutation testing; and *automation* - how much effort is required on the tester's part.

Mutation testing is expensive because of the large number of mutant programs that need to be created and executed [91]. Researchers conducting empirical studies with mutation testing are therefore usually restricted to using small programs in order to restrain the number of mutants produced [81, 85, 103]. Whilst this constraint is acceptable from an academic stance, it is not for industry which often wishes to test larger, more complex programs. As the complexity increases, so does the execution time for a program and its mutant variants, therefore increasing the overall runtime for mutation testing. Matters are aggravated further due to the difficulties in automating the entire mutation testing process. Despite a large proportion of the process capable of being easily automated, certain tasks such as detecting equivalent mutants and checking output correctness are typically

done by hand. Although these labours do allow the code itself to be scrutinised, it is still a tedious and error prone task that increases the duration of testing at a point in the software life cycle when time is often critical.

2.4 Reducing Computational Expense

Traditional mutation testing systems generate large numbers of mutant programs. For example, 385 mutants are generated for an 18 line Newton square root Fortran procedure [103]. Analysis shows that the number of mutants generated is approximately defined as the product of the number of data references and data objects [18, 20, 85]. Therefore in general, as a program's complexity increases so does the number of mutants. In turn, this increases the execution costs, as every mutant has to execute against at least one test case. Such expense is accepted for academic research where time constraints are flexible, but industry does not generally have that luxury. High runtime is exacerbated further because traditional systems interpret mutant programs. Whilst this is convenient, it does make such systems slow to execute, hard to construct, and difficult to emulate the operational environment [85]. To overcome the costs associated with mutants, most research has concentrated on one of three areas: do fewer, do faster, or do smarter [91].

2.4.1 Do Fewer

An obvious way to reduce the expense of standard mutation testing is to reduce the number of mutant programs that are executed. In particular, three mutation testing techniques exist that attempt to do just this without lessening their fault finding capabilities: *Mutant Sampling*; *Constrained Mutation*; and, *N-Selective Mutation*.

Mutant sampling reduces the computational costs by selecting a small percentage of mutants from each mutagen and ignoring the remainder. Investigations by

Acree [1] and Budd [18] suggested that a 10% sampling rate resulted in the identification of over 99% of all non-equivalent mutants whilst providing considerable cost savings. Wong [108] continued research into the cost benefits of mutant sampling by varying the sampling rate between 10% and 40%. Even at the lowest rate, Wong's results suggest mutant sampling to be an effective cost-cutting strategy producing test sets capable of identifying at least 96.14% of all mutants but only examining 9.8%[‡] of them. Whilst, as Wong suggests, these results are similar to the scores reported by [1, 18], only 5 runs per sample rate were performed implying that the average scores quoted may not be accurate for the 4 programs tested, let alone other programs. Assuming these results are indicative of every program however, the remaining percentage of unidentified mutants could still be significantly large. For example consider Wong's results: the 80 line STRMAT2 C program [108] generated 510 mutants - the smallest for the 4 programs - of which 3.86% (percentage of remaining mutants from the 10% sampling rate test set) represents approximately 20 mutants. As the tested program's complexity increases, so will the overall number of mutants, meaning that the proportion of remaining mutants will be even larger. Therefore despite achieving a high (96%+) mutation score, the number of remaining mutants is likely to be just as significant.

A further point highlighted in [108] is that for sample-adequate test sets, a higher sampling rate does not necessarily generate a higher mutation score than a lower sampling rate. Initially this seems implausible. One would expect tests generated from a larger sample to be more adequate at identifying the remaining mutants than tests from a smaller sample. Wong's results however, suggest otherwise; for the TEXTFMT function, a 25% sampling rate averages a 98.92% mutation score compared with 99.01% for a 10% sampling rate. These results suggest that the number of mutants examined is not a clear indicator of the mutation score (bar examining all mutants), but instead an indication that some

[‡]Although not specified by Wong, the observable differences in the percentage of mutants is presumably caused by approximation errors in calculating the number of mutants for each mutagen. For example, 10% of an ROR operator generating 101 mutants is 10.1 mutants, which is either rounder up to 11, or down to 10 whole mutants.

tests are capable of identifying more mutants than others. In which case, *does the choice of the mutants selected for testing affect the tests generated and therefore the mutation score?*

Mutants could be categorised based on the set of tests that kill them. A *strong* mutant is hard to distinguish, requiring a specific test to kill it. At the other extreme, a *weak* mutant is easily distinguished by any test. In general however, mutants vary in strength between these extremes, with differing sets of tests that kill them. These sets are generally unknown before (and often after!) testing, however it is from a mutant's test set that a test must be generated in order to kill that mutant. Consider the situation where a reasonably weak mutant, W, and a reasonably strong mutant, S, are chosen. W will have a large set of tests that kill it - T_W; S will not - T_S. There are three possible situations that could arise, shown graphically in figure 2.2:

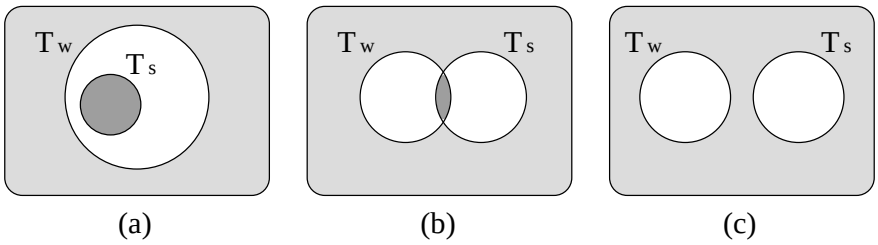

Figure 2.2: Three possible scenarios for the relationship between the sets of tests that kill a weak and strong mutant.

1. $T_S \subseteq T_W$. The test set that kills the stronger mutant is a subset of the test set that kills the weaker mutants - *figure 2.2(a)*.

2. $(T_S \not\subseteq T_W) \wedge (T_S \cap T_W \neq \oslash)$. T_S is not a subset of T_W but the two sets intersect - *figure 2.2(b)*.

3. $T_S \cap T_W = \oslash$. The two test sets are mutually exclusive - *figure 2.2(c)*.

If situation 1 occurs, then selecting the stronger mutant during sampling will ensure a test is generated to kill the weaker mutant - i.e. the generated test is in

both T_S and T_W. If the weaker mutant is selected however, there is a reduced chance that the generated test will kill the stronger mutant - i.e. the test will be in T_W but may or may not be in T_S. If situation 3 arises, then selecting either mutant during sampling will only generate test data capable of killing that particular mutant. Situation 2 is a combination of the other two. Selecting the stronger mutant will generate a test that may or may not kill the weaker mutant, and vice versa. In general, as the size of T_W is likely to be greater than the size of T_S, the probability of a test from T_W intersecting with T_S is likely to be smaller than a test from T_S intersecting with T_W. For example, consider only a single test, x, that intersects the two sets; $\mid T_W \mid = 20$ and $\mid T_S \mid = 5$. If a test is generated to kill the weaker mutant (i.e. the test is in T_W), then the probability of it being x is 1/20. If a test is generated to kill the stronger mutant (i.e. the test is in T_S), then the probability of it being x is 1/5[§]. These three situations indicate that the mutant chosen during sampling will indeed affect the tests generated. Furthermore, they suggest that selecting stronger mutants improves the chances of killing weaker mutants, and so it is not the size of the mutant sample that is important, but the choice of mutant. Sampling a lower percentage of stronger mutants could generate a higher mutation score than sampling a higher percentage of weaker mutants.

Constrained mutation was proposed by Mathur [64] as a variant of mutant sampling. It works by generating all mutants from a subset of mutation operators rather than generating a sample from all possible mutants. Empirical tests have generated compelling results in favour of constrained mutation [65, 66, 108]. Wong's results in particular found that average mutation scores from constrained-adequate (using only the ABS and ROR operators) test sets exceeded 95% for the four programs tested, and reduced the number of mutants to between 14.39% and 19.94% of the total number [108]. Whilst the cost savings are not as great as a 10%

[§]These statistics ignore the probabilities of generating test data belonging to a set. Obviously, if test generation is random, there is a slightly larger chance of generating a test in T_W than T_S. The probability of generating a test in T_W is $20/n$ (where n is the total number of possible tests); for T_S the probability is $5/n$.

sampling rate (which executed between 9.8% and 10.98% of the total mutants), the average mutation scores are approximately equal - 97.56% for 10% sampling rate and 97.18% for ABS/ROR constrained mutation. Mathur et. al.'s findings offer a similar indication, reporting that in 7 of 10 experiments, ABS/ROR constrained mutation is at least as effective as a 10% sampling rate at fault detection. Furthermore, the remaining 3 experiments all involved the same program, which was not used in the other 7 experiments [65]. They conclude that *"when combined with the cost of various criteria and the number of mutants to be examined, the abs/ror variant of mutation appears to be the most promising criterion to use in practice"*.

Offutt et. al. [92, 85] subtly altered Mathur's work by using all mutagens bar the two that generate the most mutants (for their *2-selective mutation* approach) - different from Mathur's approach of constraining the operators to a select few. They then hypothesised an *N-selective* approach where the N highest-yielding mutagens are removed, opting to compare 2-,4- and 6-selective approaches on 10 test programs. Initially 28 programs were examined to determine the percentage of mutants produced by each mutagen, as shown in figure 2.3. These suggest omitting SVR and ASR mutagens for 2-selective mutation, plus SCR and CSR for 4-selective, combined with ACR and SRC for 6-selective. For each selective approach, the Godzilla automatic test data generator [28] was used to create selective-mutation adequate test data. These test sets were then compared against all mutants to ascertain their effectiveness against all mutants - i.e. their true mutation score. The results were promising. The 2-selective approach resulted in a true mutation score of 99.99% and a saving of 23.98% in the number of mutants examined. For 4-selective, these figures were 99.84% and 41.36%, and for 6-selective they were 99.71% and 60.56% respectively. Such high mutation scores from selective-adequate test sets suggest that tests designed to kill mutants from low-yielding mutagens are also capable of killing mutants from high-yielding mutagens, offering credence towards the theory suggested by mutant sampling that the choice of mutants used influences the quality of the tests generated.

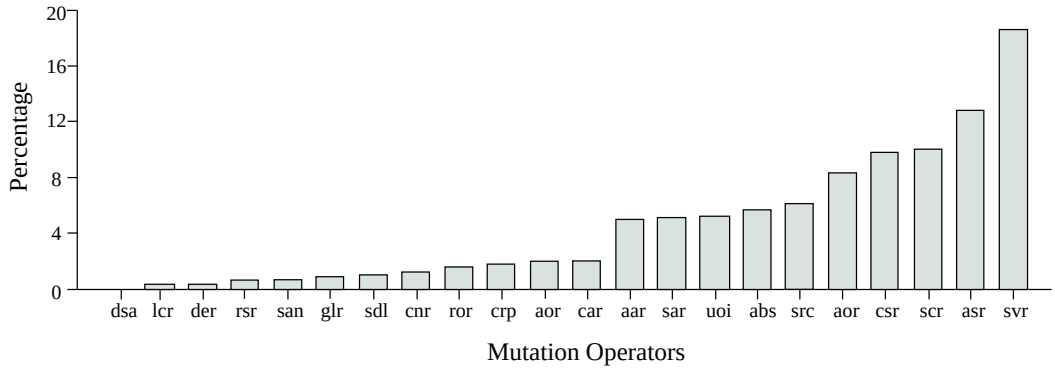

Figure 2.3: Mutant Distribution by Mutation Operator. *Diagram reproduced from [92]*

Offutt et. al. continued researching whether removing further mutants produces increased savings without severely harming the effectiveness of the generated test sets [85]. The 22 Fortran mutagens were divided into three groups: *Replacement of operand* mutagens - replacing each program operand with another legal operand; *Expression modification* mutagens - replacing program operators and inserting new operators; and, *Statement modification* mutagens - modifying entire statements. They performed three experiments to generate adequate test data, excluding a single group of mutagens in each study. When compared against all mutants, the results suggested that the 5 expression operators (ABS, AOR, LCR, ROR, UOI) were the most useful. These 5 operators alter the program's control structure, and so they are also useful in providing branch coverage tests. Unsurprisingly then, they also correspond with Beizer's statistics that 12.8% of faults are due to control flow and sequencing errors - the largest cause of errors [8]. Additional experiments demonstrated that adequate test sets generated from these five produced an average mutation score of 99.51% with a saving of 77.56% on the number of mutants examined. These results are similar to the average mutation scores obtained using mutant sampling with a 25% sampling rate (75% saving): between 98.27% and 99.01%. However, unlike mutant sampling, this approach has emphasised the class of mutants examined, providing evidence that certain operators help generate stronger tests (i.e. they are able to kill more mutants)

than others. This led to the notion of a *sufficient mutagen set* to reduce computational expense whilst maintaining test set effectiveness: given a sufficient set of mutagens, M, and a test set, T, adequate with respect to M, then T will generate a reasonably high mutation score when compared against all mutants [85]. Offutt et. al. extended this work by attempting to reduce the 5 sufficient operators [85]. They performed five more experiments, excluding one of the five operators from each study, but could not produce conclusive evidence to warrant excluding a further mutagen primarily because of the small number and low complexity of their test programs. When testing the LCR operator, for example, eight out of the 10 programs had no LCR mutants making it difficult to draw conclusions from the 2 remaining programs. Offutt et. al. concluded *"in the absence of further evidence, ... all five E-selective mutant operators should be used"* [85].

2.4.2 Do Faster

Do faster techniques aim to generate and run mutant programs quicker than standard systems. Most traditional mutation systems interpret their mutants, and whilst this is convenient, it is also slower than executing compiled code. The Proteum system [26] generates and compiles each mutant before execution, in a *separate-compilation* technique. This can result in a runtime speed increase of 15-20 times compared to a traditional system [91]. However, if the compile time is significantly greater than the run time, a compilation bottleneck occurs, resulting in a build up of programs to be compiled whilst little execution is happening [20].

Krauser avoids bottlenecks using a *compiler-integrated* mutation mechanism [52]. This uses a novel compiler to simultaneously compile the PUT and develop code patches representing mutations. Prior to execution, the necessary code patches are applied to the compiled PUT to deliver a mutant program that executes at compiled speeds. Consequently, the PUT only needs to be compiled once. Krauser's results demonstrate that the compiler-integrated approach gives an appreciable speedup (calculated as a ratio between techniques of the average execution time, per mutant, to generate and execute all mutants against one

test case) against separate-compilation when the execution time of the PUT is low. For example, for the TRANSPOSE program the compiler-integrated approach was 7.58 times faster than by using the separate-compilation technique, and 27.33 times faster for the TRITYP program. As Krauser states, *"this speedup arises from the compile time overhead experienced by the separate compilation approach and results when individual mutant program execution times are small"* [52]. Put simply, program execution time affects the speedup gain for the compiler-integrated approach. With low execution times (i.e. where execution time $<$ compile time), a separate-compilation approach executes mutants faster than it can compile them. The delay in compilation causes the overall time to execute all mutants to increase and so increases the average mutant execution time. With higher execution times however (i.e. where execution time $>$ compile time), the compilation time becomes a less significant factor. Mutants can easily be compiled before execution meaning that the average time to execute each mutant for the separate-compilation approach is only dependent on the execution time - the same as the compiler-integrated technique, meaning the speedup ratio approaches 1/1.

A variation on these approaches which also helps alleviate the bottleneck problem is *mutant schema generation* (MSG) [103]. All possible mutations of a program are encoded in a single source level, *metamutant* program. Mutation points in the PUT (for example, an arithmetic operation) are replaced by syntactically valid function calls, called *metaprocedures*. Each metaprocedure encodes a mutagen and varies its output depending on its arguments. For example, depending on its arguments, an arithmetic metaprocedure will perform addition, subtraction, multiplication, division, or modulus. Mutants are then represented as a set of metaprocedure arguments to be applied at runtime to the metamutant. By varying only a single metaprocedure argument from the original (PUT's) set of metaprocedure arguments, the appropriate mutant can be executed. This means mutants are not separately compiled or interpreted, and are executed in the target environment at compiled-program speeds.

```
// Original PUT method
public boolean sumIsPositive(int n, int m){
  z ← 0;
  sum ← n + m;
  return (sum > z);
}
```

Figure 2.4: Example of original Java PUT method.

A simple metamutant of the code fragment in figure 2.4 is demonstrated in figure 2.5. This metamutant is created using only the relational operator mutagen, which replaces every instance of a relational operator by an appropriate relation operator metaprocedure, in this case *ROR.op*(), as shown in figure 2.5 - line 2. *ROR.op*() has the signature: *ROR.op*(*int lhs*, *int rhs*, *int mutantNumber*)

The metaprocedure itself, is defined elsewhere as indicated in figure 2.5 - line 1. During execution of this metamutant, the appropriate arguments are supplied to the *ROR* metaprocedure allowing any relational operator to be applied to the two values.

```
   // PUT method mutated by ROR mutagen
1  import MTAIS.Mutagens.ROR;
   ...
   public boolean sumIsPositive(int n, int m){
     z ← 0;
     sum ← n + m;
2    return (ROR.op(sum, z, 0));
   }
```

Figure 2.5: Mutated version of the Java PUT in figure 2.4.

Each mutant is represented by an array of values indicating the operation for each metaprocedure. The *mutantNumber* refers to the metaprocedure index within a mutant array. For example, the *mutantNumber* in code 2.5 is 0, indicating that the value in the zeroth index in the mutant array sets the operator used in *ROR.op*(). All relational operators are numbered, for example, $<$ is 0, $\leq$ is

1, == is 2, > is 3 and so on. The value at the *mutantNumber* index determines which relational operator to apply, in this example, the original $sum > z$ statement would be executed by a mutant array with the zeroth index containing 3 (the number for the > relational operator). To execute $sum == z$, the value in the zeroth index would be modified to 2 (the number for the == relational operator). It should be clear that by using this approach all mutants are effectively present within the metamutant, each one being individually selected at runtime by applying the appropriate values from a mutant array to the metaprocedures.

In essence, this technique is similar to the compiler-integrated approach except that patches aren't applied at run-time, they are instead encoded into the PUT before compilation. Furthermore, the metamutant only needs compiling once, but unlike the compiler-integrated approach, this can be done using the standard developmental compiler making the system less complex and more portable [91].

Untch et. al. report that executing the NEWTON square root procedure using mutant schema generation is 4.1 times faster than using an interpretative approach [103]. They argue that this *"is a strong indication that MSG can significantly increase the performance of mutation testing"*, although this appears to be supported only by the single experiment. To this author's knowledge little else has been published detailing the effects of MSG. Despite this lack of research, Untch's evidence for MSG coupled with that given by [52] for compiler-integrated mutation, strongly suggests mutation testing using a single pre-compiled program to be faster than interpretation.

2.4.3 Do Smarter

Do smarter approaches aspire to use "smart" methods to improve execution performance. For example, parallel computation architectures such as SIMD (Single-Instruction, Multiple-Data) [53] and MIMD (Multiple-Instruction, Multiple-Data) [20, 90]) allow mutants to be executed simultaneously. However these techniques require specialist equipment, which for industry, particularly those companies on limited budgets, is not cost effective. A more practical technique, not requiring

parallel computers, is *weak mutation*, proposed by Howden [41].

In order to increase the mutation score, mutation testing kills mutants that express a differing output from the PUT given the same input. Strong mutation testing therefore, continues mutant execution until completion and compares the subsequent outputs, or program states. But this is wasteful. Consider a mutant compared to the PUT. Each program statement is identical except the single statement containing the mutation, implying that to cause varying outputs, this "faulty" code must cause some differing state immediately after its execution. If no varying state is found then the mutant will continue along the same program path and generate the same output as the PUT. However, if a varying state is found at this point, then this *may* indicate a possible difference in program outputs, and save executing the remainder of the mutant. This is the premise behind weak mutation testing. It does have a subtle drawback however: a state difference immediately after execution of a mutated statement does not necessarily imply the final outputs will be different. For example, a program may incorporate error checking functionality that corrects for the state difference caused by the mutation, resulting in the correct output. Incorrectly killing a mutant under these circumstances would result in a higher mutation score giving stronger credibility to the test set adequacy than is true. This is an unavoidable drawback of weak mutation, however the enticing advantage is that execution times are reduced as only part of each mutant requires execution. It is with these pros and cons in mind that additional research has been done into where programs states should be compared.

Weak mutation differs from strong mutation in when the states of the mutant and PUT are compared. Both approaches can be classed as comparing at opposite extremes: weak mutation compares immediately after the mutated statement; strong mutation compares upon program termination. To cover the various approaches between these extremes, Woodward and Halewood [109] introduced the idea of *firm mutation*. This compares the states of the programs at some point after the mutated code is executed, but before the end of the program. By deciding

whether to kill a mutant closer to the natural termination of the program, it is hoped to improve the test set adequacy whilst retaining some of the performance advantages from weak mutation.

These variations of weak mutation have been studied empirically [34, 62, 87], in particular using a modified version of Mothra, called Leonardo (Looking at Expected Output Not After Return but During Operation) [86, 87], which incorporates a working weak mutation system. Offutt and Lee [87] used Leonardo to perform studies on firm mutation using four different comparison points: (i) after the first execution of the innermost expression surrounding the mutation; (ii) after the first execution of the mutated statement - traditional weak mutation; (iii) after the first execution of the basic block (sequence of instructions with single entry and exit points) surrounding the mutation; and (iv) after N executions of the mutation containing basic block where N is greater than 1 and less than equal to the number of times the basic block was executed in the PUT - a mutant is killed upon the first instance of an incorrect state. They performed two comparative studies against these four versions and strong mutation. The first generated 100% firm-mutation scoring test sets for each firm approach and executed them using strong mutation to generate overall mutation scores. The second study generated test sets with less than 90% mutation scores and executed these on the four firm approaches to ascertain their firm-mutation scores. The results were interesting. Intuitively, approach (iv) would test program states closer to the natural program end, and so closer to a strong mutation system, suggesting that this method should provide stronger, more adequate test data. This was not the case. The first study indicated that the most effective point to perform state comparisons was after executing the mutated statement (approach ii) or the first execution of the basic block (approach iii). To explain this Offutt and Lee suggested that performing comparisons after the first execution of the basic block is a more rigorous requirement than causing an incorrect state on any execution of the block, and this therefore requires a stronger test set [86]. A slight variation on this reasoning however, could be that killing mutants whose execution exceeds

the PUT's number of iterations of the basic block is an easily achievable target using low quality tests, and yet does not necessarily imply a differing final output. If this is true, the test sets generated from approach (iv) would be *weaker* than those from approach (iii) and produce a lower mutation score.

The second study was designed to indicate differences amongst the various firm approaches. It would seem natural to expect that as the comparison point approached the natural end of the program, that the firm-mutation scores would approach the strong-mutation score in some fashion; Offutt and Lee suggest that the mutation scores should decrease [86]. This is based on the idea that a firm mutation system will kill more mutants than perhaps it should, the further the comparison point is from the program's natural end (i.e. the firm-mutation score will be at least as high as the strong-mutation score). However, Offutt and Lee's results indicate otherwise. They found that comparing states after N basic block executions for programs containing loops (i.e. closest to the program's end) often resulted in higher mutation scores than the other approaches, however they offered no explanation for this. However, one possible explanation is again that tests cause mutant executions to exceed the PUT's number of basic block executions, resulting in a larger number being killed than otherwise should. This would not occur with approach (iii) as mutants are only killed if a state difference is identified after the first execution of the basic block only.

The results from Leonardo indicate that weak mutation is able to considerably reduce computational expense too, with savings often in excess of 50% of execution times [86, 91]. Offutt and Lee concluded that based on their studies, *"weak mutation [should] be used as a cost-effective alternative to strong mutation for unit testing of non-critical applications."* [91].

2.5 Increasing Automation

Despairingly, mutation testing suffers from further problems besides the computational expense, that also restrict its adoption into mainstream industry. These

additional burdens can be classed as automation difficulties that, with the current state of the art, require considerable effort from the tester. First and foremost, test data must be generated to kill mutants. Whilst, for the majority of mutants, any random test will suffice (providing a use for random test generators), a small percentage of difficult-to-kill mutants require scrutinising the source code and hand-generating tests in order to identify the faults they contain. This is both time consuming and challenging. Furthermore, the mutation score calculated as a measure of test adequacy has to discount *equivalent mutants*. Given that recognising these is undecidable [17], detecting equivalent mutants has traditionally been done by hand. Again, this is an expensive task that prevents further uptake by industry. Equivalent mutants aside, all other mutants have to be shown to generate incorrect outputs compared to the original's correct output. The original program must therefore produce correct results to start with, as determined by an *oracle* (a manual or automated procedure that determines output correctness for any given test). However, deciding whether the executed output is correct for an input is often very difficult, and is known as the *oracle problem* [33].

2.5.1 Automatically Generating Test Data

Both traditional and evolutionary techniques have been applied to help automatically generate test data. Traditional algorithm design often relies on a *divide and conquer* approach to solving problems. An algorithm divides the task into smaller units, and repeats itself on each until a solution is derived. Whilst such algorithms are efficient at providing solutions to certain problems, they usually struggle to solve problems with dynamic solution spaces, such as optimisation problems. These problems generally require more dynamic approaches, typically relying on heuristics rather than hard coded rules. In this respect, evolutionary methods attempt to capture emergent qualities that evolve in other (usually non-computer related) areas, such as learning in the brain or the problem-solving abilities of ants, by using metaphors to help create new approaches to algorithms. Often these make use of *population dynamics*, whereby the interactions between

many small entities, each incapable of solving the problem on its own, combine and present an emergent solution.

Traditional Techniques Offutt developed *constraint-based testing* (CBT) to produce mutation-adequate test sets [80]. This technique is based on the notion that to kill a mutant a test must satisfy three conditions:

1. *Reachability:* Mutant programs differ from the PUT by a small amount usually contained in a single statement. As all other statements are the same, this mutated statement must be responsible for a difference in program outputs. Therefore, for a test to distinguish the mutant from the PUT, this mutated statement must be *reached*.

2. *Necessity:* Having reached the mutated statement, it is *necessary* that it causes an incorrect internal program state immediately after its execution. If it does not, then the mutant cannot fail as, with all other statements being identical, no successive statement can cause the internal state to differ.

3. *Sufficiency:* Finally, the test is *sufficient* if the incorrect state propagates through the mutant resulting in a failure upon termination. If any statement corrects the internal state, then either the mutant is equivalent or the test is not adequate enough in causing a differing program output.

These conditions are used by the *Godzilla* test data generator to automatically create test sets for Mothra [28]. Godzilla represents conditions as input domain based algebraic expressions, known as *constraints*, that assert whether each condition is met. Reachability constraints are expressed as path expressions describing all paths to the mutated statement. Likewise, necessity constraints assert whether execution of the mutated statement results in an incorrect internal state. Sufficiency constraints cannot be generated however, as they require prior knowledge of the complete path the program will take. This is undecidable [80]. Instead, Godzilla combines the reachability and necessity constraints and solves this to provide a test that executes the mutated statement and generates an inconsistent

internal state. Effectively, this is similar to generating test data via weak mutation, and as such suffers from the same pitfalls. Notably, a test may generate an inconsistent internal state in a mutant that is subsequently corrected to produce the desired output; a mutant will be killed when it should remain living. CBT approaches also suffer from problems handling arrays, loops and nested expressions, mainly due to their symbolic execution nature, resulting in an inability to find test cases [91]. Despite these problems, empirical results from [28] suggest that CBT approaches generate high mutation score test data, killing over 90% of mutants. Indeed, [28] claim they repeated their experiments *"for dozens of unit and module-level FORTRAN subprograms of up to a few hundred lines. The test data that Godzilla generated for these programs had an average mutation score of .97 [97%], with very little deviation."*

Offutt et. al [84] developed an alternative test generator, called *dynamic domain reduction* (DDR) that works when CBT does, and in many cases when it does not [91]. DDR uses parts of CBT approach in addition to the dynamic test data generation technique developed in [49]. Initially, each input variable is given a large range of values, known as its domain. The DDR procedure then executes a path through the control flow graph to a specific node - the goal node; for example, in mutation testing this would be the mutated statement. When it reaches a branch point in the path, the variables used within that branch predicate have their domains reduced in accordance with the execution of the desired branch path. If there is a choice of how to reduce the domain, a search is made of the subsequent path so as not to make an inappropriate selection and restrict subsequent branch outputs. Upon reaching the goal node, the remaining values in each variable's domain represent the values that will cause execution of the desired path, i.e. for mutation testing where the mutated statement is the goal node, the remaining domains represent test values that satisfy the reachability constraint. In some cases though, a domain will be empty indicating that the procedure failed either because the path is infeasible or it was too difficult to find

values to execute it [84]. DDR's success however, lies with its combination of multiple test generation techniques which give it an improved performance over those techniques used independently. Indeed Offutt et. al. conclude that DDR *"is less likely to fail to find a test case when a test case exists, and that implementations can be more efficient"*. Compared to a CBT approach, this procedure would seem more favourable.

Evolutionary Techniques A wealth of work exists that use various search optimisation techniques to generate test data: Genetic Algorithms [46, 74, 75, 94]; Simulated Annealing [101]; Tabu Search [29]. Whilst similar to the focus of this research, these techniques use differing methods to assess the quality of the test data generated: Decision (or Branch) coverage [46]; Condition/Decision coverage [74, 75]; Path Coverage [57]; Control-Dependence Path coverage [94]. In general however, it is the limited work on evolutionary techniques using mutation testing as their test adequacy measure that are most pertinent to this research, and these will therefore be the focus. Despite this, there is a common problem surrounding evolutionary testing in general, which practitioners should be aware of, namely how best to direct the generation of test data.

Typically, evolutionary testing (ET) techniques direct test generation through some form of fitness function. As suggested recently by McMinn and Holcombe [72] however, some of these functions can cause problems for ET due to their lack of guidance. In particular, flag variables present problems for the cost metrics used by fitness functions to direct evolution for structural testing. In these situations, a cost metric is usually defined to measure how well a test satisfies a specific predicate. For example, given the predicate `if(a==b){`, the cost metric would evaluate to the absolute difference between `a` and `b` as a means to determine how close the test came to executing the true path (where the true path gives access to code of interest). A number of other rules have also been defined [49, 100]. If instead, the predicate only contains a flag variable (which has previously been assigned the result of the original predicate), e.g. `flag=(a==b);...; if(flag){`,

then the cost metric only provides a binary fitness landscape (true or false). This offers no guidance for finding satisfying tests (i.e. tests that result in the predicate evaluating to true), and so means ET is no better than a random search in this situation [38].

A solution to this flag problem is to associate the cost metric information with the flag variable upon its assignment, and use this information, as necessary, to direct test evolution (i.e. to help direct test evolution for the if statement flag variable) [14]. An alternative approach, indicated by Harman *et. al.* [38], is to transform a flag-containing program into a flag-free program and perform evolutionary testing on this. Their results are favourable, indicating improved execution times and code coverage. They do not, however, indicate whether their method is capable of dealing with the more complex situation described by Bottaci [14]. Bottaci highlights the problem of when a boolean condition that determines the flag value is not directly assigned to the flag, but instead controls the assignment of a "summary" value to that flag, e.g. `flag=false; if(a==b){flag=true;};...; if(flag){`. In this case, the cost of the first `if` statement can be calculated as before ($| a - b |$), but it is only associated with this statement, and not with the setting of `flag` to true. Bottaci suggests that Data Dependency Analysis (DDA) [2] could be used to associate the second assignment of `flag` (and therefore its relation with the first `if` statement) with `flag`'s use in the second `if` statement.

Ferguson and Korel [30] use DDA to search for statements that may affect the execution of a desired statement, linking these statements into the execution 'chain' needed before the desired statement can be executed. McMinn and Holcombe [72] combine this chaining approach with evolutionary testing, with their results (achieved from seven test programs) arguing that this hybridisation can improve ET. They also utilise the chaining approach on another problematic domain for ET, namely states in test objects [71, 73]. States typically imply a test goal may be dependent on previous tests, and not just the current PUT input. To achieve a test goal, the test object needs to be placed into a goal-feasible state; this is achieved by the execution of the correct sequence of statements (possibly

through execution of multiple test inputs) that manipulate the underlying state machine of the test object. DDA can identify these statements, and so the authors suggest its use to help direct ET. Their initial work using ant colony algorithms ([12], cited in [71]) to search and evaluate the sequences gives promising results with regards test generation [71], as does their more recent work on sequence generation hybridised with the chaining approach, which achieved higher coverage levels and success rates than the non-hybridised approach [73].

Focussing on mutation testing based approaches now, one popular technique is the use Genetic Algorithms [76]. These algorithms evolve a population of individuals, searching to find the best one in relation to the environmental problem space (i.e. the ideal solution). Individuals are iteratively reproduced based on how ideal they are (their fitness), combined randomly with other individuals, and then mutated slightly. The key point is that the single best individual is sought. For mutation testing it is unlikely that one test will kill all mutants, requiring that, for a GA, each individual must incorporate a sufficient number of tests to allow that individual to achieve a full mutation score - to kill all mutants. This in itself is a hindrance as the number of tests required to kill all mutants is generally unknown *a priori*.

Jones et. al. [46] initially perform a variety of experiments using a Genetic Algorithm (GA)[5] to develop test data that fulfils branch coverage (e.g. all conditional branches within the PUT must evaluate to true and false). They compare their results from the GA to those from randomly generated test data over several test programs (Quadratic Solver, Triangle Classification, Integer Remainder Calculation, Linear Search, and Binary Search), recording the percentage of test runs achieving full branch coverage and the average number of generations required to do so. Their results argue in favour of using GA's. The triangle classification program, for example, achieves full branch coverage on all runs, taking on average 185 generations to do so, compared with only 69.5% of randomly generated runs attaining full coverage in an average of 1187 generations. The authors then briefly

[5]See Section 3.2.3 for a discussion on Genetic Algorithms.

investigate the use of GA generated branch coverage tests on the mutation score of the program, although few details are given; 70 mutations of the quadratic solver were made, incorporating only predicate mutations (for example, $>$ is replaced by $\geq$). Test data is then generated for each mutant using a GA that achieves full branch coverage, as per their initial experiments, except the additional requirement is made that the tests are close to the input boundary that causes the to-be-mutated predicate to oscillate between true and false. These are then executed on the original program with outputs compared to the mutant outputs. The mutation score obtained was 0.97, similar to the result achieved by the CBT method in [28].

Bottaci [13] described a possible GA fitness function for mutation testing based on the three CBT conditions: reachability; necessity; and sufficiency. The overall fitness of a test is based on a combination of costs for each of the three conditions, equating to zero if the test kills a mutant, or a positive score otherwise.

- *The reachability cost* (cost of a test *reaching* the mutated statement) is based on the control flow graph and is calculated, for the case when only one goal path exists (i.e. only one path to the mutated statement), as the number of nodes in the goal path minus the number of nodes in the longest common prefix with the path a test executes. Therefore, the later a test causes divergence from the goal path, the smaller its cost is. In the case where two inputs result in the same non-zero costs, a further cost metric is introduced, indicating the amount that the failing branch predicate (that caused path divergence from the goal path) differs from the goal-required predicate value. For example, if the predicate tests for equality, say $a = b$, then the cost is 0 when a equals b, otherwise its the absolute difference between a and b, bounded by an upper limit[6]. Obviously as both arguments approach equality, the cost minimises. Usually however, it is the case that more than one goal path exists to the mutated statement. Under such conditions, the

[6]Other rules exist, see [13] for details.

reachability constraint is the disjunct of all the goal paths, with the costs defined as above.

- *The necessity cost* is the difference by which the necessity condition fails. The necessity condition must be true in order to kill the mutant and is therefore dependent on the mutation applied. For example, if $a = b$ is mutated to $a \leq b$ then the necessity condition required to kill the mutant is a less than, but not equal to, b (or: $a < b$). In this way, the necessity cost can be calculated in a similar manner to the reachability cost.

- *The sufficiency cost* defines a cost based on the number of equal data states encountered after the mutated statement is executed. A mutant will only be killed if a differing data state (from the original) is propagated through to the output, resulting in a different output from the original. If a mutant's output does not vary, then at least one data state since the mutated statement's execution is equal to the original's. Counting the number of equal states gives a cost value to minimise, however, how states are counted is an interesting problem, that can widely affect the cost (for details, see [13]).

Bottaci highlights one further concern of using a GA: two parent tests combine to create offspring, but these could be *worse* than the parents. Consider the case where each parent test follows a different path through the program. At least one branch predicate will produce contradictory outputs between the two inputs (i.e. cause path divergence). If two high-fitness parents with very different paths crossover, there is a possibility that the offspring will be a test with a worse path than either parent. To help alleviate this problem, Bottaci suggests the idea of *species*. Only tests from the same (or possibly similar) species are allowed to combine, with species being determined by execution paths; tests that fail to reach the mutated statement are classified into a single species (those failing the reachability condition); tests that reach the mutated statement are classified based on their path from the mutated statement to the exit (partitioning according to sufficiency).

Whilst the ideas presented by Bottaci are intriguing, no empirical evidence is given to support them. Indeed, to this author's knowledge, no evidence has been given since this work. The approach does however, combine a CBT technique (similar to that used by the Godzilla test data generator [28]) with the search optimisation of a GA. It is therefore feasible that this combination has the ability to generate test data at least as adequate as tests generated by other CBT approaches. If Godzilla is reported to average a 97% mutation score on the *"dozens of unit and module-level FORTRAN subprograms"* [28], then this could be a positive evolutionary approach.

Baudry et. al. [5, 6, 7] initially demonstrate the use of a GA for automatically generating mutation adequate test data. They then compare this work with the use of another biological analogy, termed Bacteriological Algorithms (BAs), and show that this evolutionary strategy outperforms the GA for their mutation testing system. Their work uses only two mutagens (the equivalent of Mothra's LCR and SDL operators) and focusses on a PUT (program under test) that translates input data from one form to another, for example, a parser or compiler. In particular, they focus on optimising test data for a C# parser in the .NET framework, meaning that their test data is in fact a program. Despite this (unusual) difference from traditional mutation testing approaches, their methodology could be applied to any PUT.

Baudry et. al. [6] used a population of 12 individuals consisting of 4 tests each. Using a 2% mutation rate over 200 iterations, the mutation score reaches a peak of 80%, with an average of 65-70%. Increasing the mutation rate to 10% achieves a reasonably constant score of between 85-90%. These later values are just below those attained by Godzilla.

Bacteriological algorithms are similar in nature to elitist GAs (see section 3.2.3 for details about elitist GAs), with two main exceptions: (i) a BA individual is an atomic unit - it cannot be divided; (ii) BAs maintain a memory set consisting of the best individual(s) from each generation. As an atomic unit, individuals cannot be mated (due to the divisions occurring in the crossover mating process),

and so variation stems purely from the reproduction and mutation process. A memory set is also maintained, where new individuals are added if their fitness exceeds some threshold. Individual fitness is based on a narrowing search space, with calculations based on what is left to optimise. For example, applied to mutation testing, fitness is calculated based on how many mutants a test kills that the memory set does not. If this fitness exceeds a threshold, that individual is reproduced and placed into memory. Applying a threshold however, does seem a negative thing to do. Although it helps to minimise the memory size, it does restrict the tests that can enter it. A test that kills a single mutant for example, may not kill a large enough percentage of the remaining mutants, and not be incorporated. This is still a valuable test though, and should be kept, especially if the mutant it kills is hard-to-kill. Main population individuals (for mutation testing, each individual is a test) are randomly chosen for mutation (i.e. the selected individuals are mutated to form a new individuals), and those whose fitness is zero (e.g. for mutation testing, they do not kill any new mutants), or who are redundant (e.g. kill the same mutants as another test) are removed.

Results from [7] report that using a BA generates a memory set with a mutation score of 96% in just 30 generations. The initial population consisted of 30 tests and a memory threshold of 20% (i.e. a test had to kill over 20% of remaining mutants to be added to the memory set). Comparisons were also made with a GA approach in [6], however it is difficult to ascertain the fairness of the experiments. For example, the GA consisted of 12 members of 4 tests each, equating to 48 tests, where as (in the comparative trial) the BA used between 3 and 10 tests for its initial main population. There is also a potential discrepancy with the size of each test. The BA approach claims *"the size of a bacterium [ed - individual] is an important parameter. The bigger a bacterium (test case) the longer it takes to run"*. It is then stated *"we finally chose 15 as a good size for a bacteria"*, referring to the number of nodes in a test case's syntax tree (a test is a program to be parsed by the C# parser). No mention is made to this variable within the GA, but presumably its effects are the same, implying that the BA and GA should be

compared on similar sized tests. Assuming a comparative study is made however, the results are favourable for the BA, which attains an average mutation score of 96% by executing only 46,375 mutants, compared with the GA attaining an average mutation score of 85% in 480,000 mutant executions.

2.5.2 Automatically Detecting Equivalent Mutants

An additional burden in mutation testing is determining whether a living mutant is equivalent to the original program or not. Equivalent mutants display the same output as the original for all possible inputs and must be removed before calculating the mutation score. If they are not removed, the percentage of killed mutants can never reach 100% (in cases where equivalent mutants exist), thereby unnecessarily reducing the tester's confidence in the correctness of the PUT and the test data. A test set killing less than 100% of generated mutants is either insufficient to kill the remaining mutants, or the remaining mutants are equivalent, or both; ultimately, a tester will be unsure about which situation has arisen unless they can detect equivalents. Equivalent mutants must be removed however, and so traditional mutation systems burden testers with the error prone task of manually detecting them. Worse still, there is potential for a large number of equivalents to exist, each one having to be detected individually. Offutt and Pan report that 9.1% of mutants they studied were equivalent [89]. Furthermore, they reported that 47% of them came from the ABS operator alone[7]. If accurate, these figures would suggest, for example, that on the modest 18 line Newton Square Root Fortran procedure [103], approximately 35 of the 385 mutants would be equivalent. This figure would increase as the number of mutants increased. Detecting even 35 equivalents, however, is prone to error. Acree [1] found that in a test on 50 mutants, half of which were equivalent, testers only correctly identified equivalence 80% of the time; 12% of the time equivalents were marked non-equivalent

[7]This figure is understandable given the nature of the ABS operator. For example, if a statement x is mutated to $ABS(x)$, then this is equivalent if x is restricted to positive numbers only; likewise for $-ABS(x)$ when x is restricted to negative numbers.

(type II errors), and 8% of the time non-equivalents were marked equivalent (type I errors). This analysis prompts the conservative approach of reducing type I errors, as incorrectly marking equivalents as non-equivalent can be corrected in later iterations (as opposed to incorrectly marking non-equivalents as equivalent which is more difficult to correct later) [89]. Therefore, to help alleviate human error, efforts have been made to automate equivalence detection using a variety of approaches, such as compiler optimisation techniques [83], constraint based testing approaches [89] and program slicing [39]. These solutions however, are only based on heuristics.

Offutt and Craft [83] implemented algorithms based on data flow and compiler optimisation techniques to automatically detect equivalent programs. Compiler optimisation techniques use the fact that code optimisation transformations generate equivalent programs that are either an optimisation or de-optimisation of the original program, therefore if a mutant satisfies a code optimisation rule, it can be shown to be equivalent [89]. This compiler optimisation idea was originally suggested by Baldwin and Sayward in [4]. Offutt and Craft use six techniques to detect equivalent mutants, which are described in more detail in [83] (examples in the text below are repeated from this publication):

- Dead Code Detection - most obvious form: inexecutable code that contains the mutated statement will be equivalent to the PUT. Detected via a control flow graph.

- Constant Propagation - a variable whose value can be defined as constant at runtime may produce equivalent mutants in some circumstances. For example, consider the ABS operator. A mutated statement executing the absolute value of a constant (≥ 0) will always produce an equivalent mutant.

- Invariant Propagation - an invariant is a true relation between two variables or a variable and a constant which is known at a particular point in the program. For example, to kill a mutant that replaces one variable with

another, the two variables must have different values. If it is known that they are equal at this point, then the mutant is likely to be equivalent.

- Common Subexpression Detection - Used in conjunction with the invariant propagation technique, it determines equivalence of variables based on common subexpressions. For example, if `A = B+C-D` and `X = B+C-D` then `A==X` after `X`'s assignment. This information is used by the invariant propagation technique to detect equivalents.

- Loop Invariant Detection - code optimisation often moves invariant code outside of a loop. If a mutation alters this positioning (i.e. moves code inside or outside of a loop), then it is equivalent.

- Hoisting and Sinking - code optimisation attempts to move code that is executed multiple times to a position where it is executed once. Mutants that affect this position are equivalent.

Offutt and Craft tested their implementation (which was developed as a plug-in to Mothra) on 15 test programs. Their results detected between 0 and 100% of equivalent mutants, although the median value was around 10% (but with a high standard deviation of 25%). They observed that the detection ability of their implementations depended heavily upon the program being tested. For example, the FIND program contained arrays and backwards GOTO statements, both of which are problematic for Offutt and Craft's implementation [83]. Considering equivalents usually have to be detected by hand however, this approach does go some way to reducing the burden on testers.

Constraint-Based Testing (CBT) was also used by Offutt and Pan [89] to implement a system to detect equivalent mutants. CBT was initially developed to help automatically generate test data, where algebraic constraints are used to specify which tests will kill a mutant [28]. If however, a constraint cannot be satisfied, then no test will kill that mutant and so it is equivalent. However, recognising infeasible constraints is undecidable, meaning that potential solutions

using CBT are again based on heuristics [89, 91]. The tool implemented by Offutt and Craft had an average detection ability of 47.63%, with detection in 7 out of the 11 programs being greater than 60% [89]. Again, although not all equivalent mutants are detected, this approach does present a rewarding alternative to detection by hand.

2.5.3 Oracle Problem

A key aspect for mutation testing, and testing in general, is determining whether the output from executing the PUT with a specific test is correct or not. Often this task is attempted manually, requiring a great deal of effort on the tester's part. This situation only worsens when automatic test generators are involved, which can create obscure tests whose output is difficult to determine. Ideally the tester requires an automated procedure that can decide the correctness of a test based on the program's specifications. These are called *oracles*. Unfortunately, it is not always possible to define such procedures; this is the *oracle problem* [33, 61]. For example, it may be the case that outputs are not defined for the entire input domain. Alternatively the process of deciding correctness may be undecidable, for example determining equivalence on programs when testing a compiler [33]. Furthermore however, where it is possible to create an automated oracle, it can often prove just as costly as implementing the original program, as well as being prone to errors itself [42].

This research assumes that an appropriate oracle exists for the program being tested, thereby avoiding the oracle problem. Despite this also being true for considerable amounts of testing literature, especially within the mutation testing domain, some research has specifically focussed on the oracle problem. For example, approaches based on using pseudo-oracles (independently written programs that calculate the same function as the PUT) [23], specifications [61] and heuristics/statistics [11, 69] have all been suggested. Interested readers should consult these references for further details.

2.6 Summary

Mutation testing was introduced to provide a means to iteratively improve test data adequacy with respect to some program. A useful side effect of this technique however, is that the program itself is tested against containing a set of faults. Unfortunately though, mutation testing suffers from a number of problems which prevent its adoption into industry. These problems can broadly be categorised into two areas: computational expense and automation difficulties. It is expensive because of the large number of variant programs that need to be executed by at least one test; and, it is difficult to automate because strong tests need to be generated, equivalent mutants need to be removed, and the PUT's output needs to be tested for correctness. Because of these problems, and the strength of mutation testing, a lot of research has been undertaken into these areas.

2.6.1 Reducing Computational Expense

Research has typically focussed on three approaches to reducing computational expense: do fewer (mutant executions); do faster (executions); and, do smarter (executions).

Do fewer approaches reduce expense by reducing the number of mutant programs executed. Three techniques in particular are strongly supported by empirical evidence: Mutant Sampling; Constrained Mutation; and, N-Selective Mutation. Wong's results [108] indicate mutant sampling is capable of producing test sets with mutation scores on average in excess of 96% whilst reducing the number of mutants examined by approximately 60-90%. A higher sampling rate however, does not necessarily imply a higher mutation score (against all mutants) suggesting that the choice of mutants sampled has an impact on the quality of the tests generated. Constrained mutation addresses this need for selecting appropriate mutagens, and is again supported by Wong's experiments, which found restricting the mutagens to ABS and ROR resulted in an average mutation score of 97.18% with a reduction in the number of mutants examined by at least 80%

[108]. Offutt et. al. provided a subtle alternative to constrained mutation called N-selective mutation [92, 85]. This reduces the number of mutagens used by removing the N operators that generate the most mutants. The authors report a 60.56% saving in mutants examined for a mutation score of 99.71% using 6-selective mutation. Further research suggested this saving could be increased to 77.56% with an average mutation score of 99.51% using just 5 operators: ABS; AOR; LCR; ROR; and UOI (see table 2.1). These results again support the notion that the choice of mutants used impact on the quality of the tests produced; a result further compounded by Budd's experiments which found mutants obtained from the ABS operator are more likely to detect faults than those from other operators [18]. However, one concern that is often ignored is that despite some novel approach achieving close to a 100% mutation score, there still remains a small percentage of living mutants which are often hard to kill. Although this percentage may be small, when compared against the large numbers of mutants often produced, the number remaining alive could be dramatic. As Offutt et. al. mention, *"the software testing literature offers no clear evidence that 100% coverage provides better testing than coverage at a lower level"*. After all, the mutants produced from the 22 Fortran mutagens are just a subset of a much larger group of possible mutants.

Research is also in favour of do-faster approaches. Traditional mutation systems interpret the programs they test, often applying patches to mutate a statement. In most instances, compiled code executes quicker than interpreting it, so research in this category has focussed on appropriate methods to execute compiled code. Separate compilation techniques improve runtime speeds as much as 20 times [91], however they can lead to bottlenecks when compilation time exceeds runtime [20]. Krauser avoids this bottleneck and shows an appreciable speed increase over separate compilation techniques using a compiler-integration approach [52]. Mutant Schema Generation also avoids the bottleneck problem with the single compilation of a metamutant program containing all mutations in one program using metaprocedures. A mutant is generated by setting appropriate arguments

to the metaprocedures. Untch et. al. report that this technique executed on the NEWTON square root procedure is 4.1 times faster than using an interpretative approach [103].

Do smarter approaches use "smart" methods that cannot be classified under do fewer or do faster. Traditional strong mutation systems execute mutants to completion and compare the final output state with that of the original, however this is wasteful. A mutant differs from the PUT by a single statement. For this mutant's output to differ, this mutated statement must cause a difference in the internal state of the program. Comparing states immediately after executing this statement should indicate whether a mutant will be killed - this is weak mutation. Unfortunately, state differences do not guarantee a mutant will be killed. For example, a program may have error checking code in place to correct for erroneous states. Firm mutation is similar to weak mutation, except that the state comparison point is made after the mutated statement, but before the end of the program. Offutt and Lee [87] compared weak mutation and several firm mutation points against strong mutation. They found that comparing after executing the mutated statement (e.g. weak mutation) performed better than comparing closer to the program's end, and demonstrated considerable savings, with execution times reduced by over 50%.

2.6.2 Increasing Automation

Increasing automation within mutation testing suffers three main hurdles: generating test data; detecting equivalent mutants and determining the correctness of a program's output. These need to be overcome, to a certain degree, for industry to adopt mutation testing as a viable tool.

Both traditional approaches and evolutionary techniques have been applied to automatically generating test data. Offutt, in particular, has been a keen advocator of traditional techniques, developing a constraint based testing (CBT) approach [80] and an alternative dynamic domain reduction (DDR) technique [84]. CBT uses the fact that mutants differ from the PUT by a single statement,

and that for a test to kill a mutant, it must satisfy three conditions: Reachability; Necessity; and, Sufficiency. Expressing the first two conditions as algebraic constraints (the third is undecidable [80]) and solving them, generates test data equivalent to weak mutation testing. Empirical results are promising, with a CBT killing on average, 97% of mutants [28]. CBT approaches do suffer from problems handling arrays and loops however. To help overcome this, Offutt developed the DDR technique which improves on the CBT approach, working when CBT does, and in many cases when it does not [91]. DDR generates test data by following paths in the control flow graph to the mutated statement and reducing each variables' domain at branch points. Upon reaching the mutated statement, if the domain still contains elements then these are suitable tests that satisfy the reachability constraint. For mutation testing, these can be reduced further by constraining them to those that satisfy the necessity constraint.

Evolutionary techniques have also been applied to automatically *evolve* test data. In particular Genetic Algorithms (GA) have been applied in a variety of approaches to mutation testing. Jones et. al. [46] use branch testing as a fitness measure for each test, developing tests that have full branch coverage. These are then used with mutation testing to generate a mutation score. Their experiments generated a mutation score of 97% using this approach, equivalent to the CBT average score. Bottaci [13] describe a possible GA fitness function based on the CBT conditions, however no empirical evidence is given. The fitness function calculates a positive cost for each constraint, which the GA has to reduce to zero. Bottaci also highlights the problem that two fit parents could combine and produce a test with a worse control flow path than either of them. To help stop this from happening, Bottaci suggests dividing tests in *species* based on their execution paths. Baudry et. al. [5, 6, 7] provide another variation on using a GA for mutation testing, using the mutation score itself as a fitness value. With a population of 12 individuals, each consisting of 4 tests, and a 10% mutation score, their experiments evolved tests to a mutation score of between 85 and 90%. However, a problem with a standard GA is that it searches for the best individual, meaning

that each individual must contain enough tests to allow all non-equivalent mutants to be killed. To help overcome this, Baudry et. al. developed a Bacteriological Algorithm (BA) based on a GA. This has two main differences: each individual is an atomic unit; and, a memory set is maintained of the best individual in any iteration, on condition that the individual provides new information to the memory set. Results from this approach were promising, with it achieving a 96% mutation score in only 30 iterations. Furthermore, in comparisons with a GA, the BA achieved its 96% score in 46,375 mutant executions as opposed to the GA which achieved an average 85% in 480,000 mutant executions. However, in detriment to the BAs achievement, it is unclear from [5, 6, 7] whether a fair comparison was made in terms of the initial number of tests used in each technique and the difficulty of the PUT.

Equivalent mutants need to be removed otherwise they unnecessarily reduce the tester's confidence in the correctness of the PUT. Unfortunately, detecting equivalents is undecidable [17], although a variety of heuristics exist to help detect a proportion of them. Compiler Optimisation based techniques have been applied to detecting equivalent mutants with limited success [83]. These approaches revolve around the fact that compiler optimisations are either an optimisation or de-optimisation of the PUT, and so if a mutant satisfies an optimisation rule, it can be shown to be equivalent. Results in [83], tested on 15 programs, suggest these approaches detect a median average of 10% of equivalent mutants. CBT approaches offer a more rewarding approach, with empirical evidence suggesting they can detect an average of 47.63% of equivalent mutants [89]. Under this technique, a mutant is considered equivalent if the constraints cannot be solved - i.e. no test can be found to kill the mutant. However, detecting infeasible constraints is undecidable.

Finally, software testing requires the correctness of a test executed on the PUT to be determined. Often this is completed as a labour-intensive manual task by the tester. Automating this task would help improve the reliability and efficiency of testing, however this is difficult to accomplish, and is known as the oracle

problem. This research circumvents this problem by assuming that appropriate oracles exist for each PUT.

Of the problems described, this thesis focuses on improving the automation of test data generation. To do this, it looks at the immune system as inspiration for the evolution of a set of tests and compares this with the more traditional evolutionary approach - genetic algorithms. The following chapter provides an overview of the inspiration and mechanics of both algorithms.

Chapter 3

Evolutionary Computation

3.1 Introduction

Computational problems present many difficult challenges. Some problems, for example, require searching through large domains of possible solutions, such as for computational protein engineering (searching for an amino acid sequences with particular properties) [21, 102], or the travelling salesman problem (minimising the distance travelled between N nodes, visiting each node exactly once) [35, 56]. The challenge in these situations is how to search the solution set to quickly find a "good" answer[1]. Other problems are caused by an ever changing environment; as the environment changes, so does the solution required. Often, however, it is not possible to predict the change in the environment, and so hard-coding a solution to work in every situation is not always possible. This is common in robotics, where a robot has to adapt to a varying environment in order to perform a task. The challenge is how to create a solution capable of working in any environment. Finally, another challenge is that some problems are simply too complex to be described by a set of hard-coded rules. This is typified in the artificial intelligence (AI) world, where a top-down architecture defining rules to govern intelligence is too difficult. An alternative is a bottom-up approach where

[1] A "good" answer can be defined as a solution meeting some user-defined acceptance level.

simple rules controlling simple behaviours are defined. The desired complexity of the system as a whole is an emergent property of the many interactions between these simple rules [76].

So, how are these challenges solved? In most cases it is too complex, or even impossible, to hard-code a single strategy that works in every instance. To do so would require knowledge of every possible environmental or problem combination, which could be infinite. Instead the solution must develop through time; it must *evolve*. This notion of evolution draws obvious similarities with biological evolution, providing good analogies for creating new computational solutions. Evolution in the biological setting is driven by survival. It encourages useful characteristics that result in organisms best adapted for their environment [76]. In this manner, the solution to an environmental problem is developed through evolution. In the real world however, evolution occurs at many levels and in many timescales, with each providing a subtly different metaphor for evolutionary computing solutions. For example, the connectionist adaptation of synapses between brain neurons occurs throughout our life times, providing the basis for *artificial neural networks* (ANNs) [36], used in particular to learn and classify information; or the evolution of advantageous traits between generations of organisms happens on a larger timescale and provides inspiration for *genetic algorithms* (GAs) used for search and optimisation problems [76]. Most recently however, the ability of the vertebrate immune system to learn to ward off previously unseen viruses has sparked interest in the *artificial immune system* (AIS) paradigm [25]. Of these three approaches, GAs and AIS are the most similar and the most pertinent to this research due to their abilities to evolve near-optimal solutions from a population; this research will focus on these two areas.

3.2 Species Evolution

In the early 19th Century, Jean-Baptiste Lamarck outlined his initial explanation for the theory of evolution [54] (English translation [45]). This was based on the

idea that individuals adapt during their lifetimes and pass these learnt traits onto their offspring. The offspring continue this same process, adapting from where their parents finished, and enable the species to evolve [106].

Although his theory was subsequently disproved by the discovery of the particulate nature of genetic inheritance (traits determined by discrete units of inheritance passed from one generation to the next) by Gregor Mendel [107], Lamarck is considered by many as a fundamental influence in early ideas about evolution. In particular, such acclaim comes from Charles Darwin whose ideas about evolution shape scientific views today. Darwin identified two primary factors in evolution: *natural selection* and *genetic variation*. He used these ideas, in his works on the origin of species [22], to formulate four hypothesis. These are summarised concisely in [25] and repeated below:

1. The number of offspring tends to be larger than the number of parents;
2. The number of individuals in a species remains approximately constant;
3. From (1) and (2) one can conclude that there will be competition to survive;
4. There are genetic variations within the same species of individuals.

Survival within a population favours those individuals best adapted to their environment - *survival of the fittest*. These individuals will in turn, because of their increased rate of survival, be more likely to reproduce and continue their strong traits. Therefore, because of this natural competition and the favouritism of strong individuals, the overall population strength should increase as the population evolves, until an optimal strength is reached. To complement this process, genetic variations, such as *recombination* and *mutation*, occur during reproduction to alter (to some degree) the genetic structure of the offspring. They introduce variability, or randomness, into the solution set. Ultimately, these changes may impose a significant impact on an offspring's survivability, either increasing or decreasing it. Any resultant change will affect the likelihood of that offspring reproducing and the chance of that variation impacting on the rest of the population.

Overall, evolution of useful solutions in species is driven through a constant pressure to survive and variations introduced during reproduction. Genetic Algorithms (GAs) try to model these pressures on a population through various processes, in order to optimise a set of solutions to a given problem domain. GAs will be discussed in section 3.2.3. First, a brief biological background will be given.

3.2.1 Biological Species

Humans, amongst all other living organisms, are made up of cells. For any individual, each cell contains the same set, known as a *genome*, of one or more *chromosomes* - a chromosome being a strand of DNA. The genome of any individual forms a 'blueprint' that defines its physical and mental makeup.

Each chromosome consists of a number of *genes*. These are fragments of DNA, each occurring at a particular *locus* (position) in the chromosome, that are usually responsible for a particular protein. Very simply, a gene can be considered as encoding a specific trait, such as eye or hair colour, although it is not limited to these viewable characteristics; for example, other genes are responsible for enzymes. The alternative forms that a particular gene can take (e.g. red, brown, or blonde hair) are called *alleles*. The selection of these alleles for every gene in a genome, or *genotype*, defines an organism's expressed physical and mental characteristics - its *phenotype*.

Chromosomes can either be paired with another chromosome or unpaired. Most sexually reproducing species (between two parent individuals) have paired chromosomes and are known as *diploids*. Asexual reproducing species have unpaired chromosomes, and are called *haploids*. During sexual reproduction, two events occur, *recombination* and *mutation*, both of which cause offspring to express a varied phenotype from either parent (and indeed other siblings). Recombination varies slightly depending on whether the organism is diploid or haploid. For diploids, each parent produces a single chromosome (called a *gamete*), created by taking genes from each pair of chromosomes. Each gamete from both parents then combines to form new chromosome pairs. For haploids however, who have

unpaired chromosomes, genes are taken from chromosomes of each parent directly - gametes cannot be formed. The second process to occur happens regardless of whether the organism is diploid or haploid. Offspring chromosomes undergo mutation, where bits of DNA, called *nucleotides*, are changed between parent and child. Both processes result in a child that is a combination of both parents with possible mutations.

3.2.2 Evolution of Species

The evolution of species is believed to be heavily influenced by two primary factors, as identified by Darwin: natural selection and genetic variation. As described previously, the former works to guide a species' evolution in a favourable direction, whereas the later is responsible for variations in phenotype.

Individuals who are well suited to their environment are more likely to survive and reproduce, passing on their genes to their offspring. The probability that an individual will live to reproduce is usually a measure of *fitness*. Competition to survive (indicated by Darwin's second hypothesis) means that weaker individuals in any generation are likely to die out, leaving individuals with stronger traits that are useful for survival. As long as these traits continue to be useful, they will become more commonplace in the population, therefore improving the populations' fitness as a whole. None of this can happen though, without a method to incorporate genetic material from parents to their offspring. This is achieved with recombination. Genes from both parents are used to produce a new chromosome (or chromosome pair for diploids), allowing for the possibility that strong genes from each parent could be combined to produce an even stronger child individual. Or course, the reverse is also possible. In addition to recombination, mutation of genes also occurs, introducing randomness into chromosomes, and opening other avenues to evolution that were previously hidden. Both factors work in conjunction to evolve solutions to the environment.

3.2.3 Genetic Algorithms

Genetic Algorithms (GA) try to model the processes of natural selection and genetic variation to stochastically search for solutions. Typically they constitute a fixed-size population of haploid individuals (in particular, single-chromosome individuals with fixed lengths, as used in this work), which represent potential solutions to some computational problem. Searching for potentially better solutions occurs by evolving new (offspring) populations through the processes of selection, recombination and mutation, before assessing each offspring individual's ability. For example, GAs have, amongst many other applications, been successfully used for optimising solutions to the *Travelling Salesman Problem* [55], evolving programs [50], and of course, generating tests for mutation testing [7].

A basic GA, as employed by this research, can easily be described by the flow diagram in figure 3.6. An initial population of individuals are evaluated to measure their competencies at solving the problem. A number of these are selected, based on their ability, and used to generate the next population.

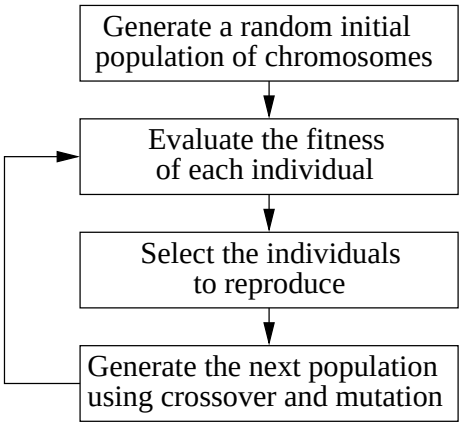

Figure 3.6: A basic Genetic Algorithm. *Diagram adapted from [25]*

Selection There are a number of different methods that can be used to select individuals for reproduction, such as *Roulette Wheel Selection*, *Rank Selection*, *Boltzmann Selection* or *Elitism*. Interested readers should consult [76] for details

on these and others - only methods relevant to this research will be discussed.

A typical means of selecting individuals is known as *Roulette Wheel* selection, and is employed in this research as a simple and effective way of selecting individuals based on their fitness. Essentially, an individual's probability of selection is directly proportional to its fitness. Table 3.2 demonstrates how Roulette Wheel is applied to a population of 5 individuals.

Ind.	**Chromosome**	**Fitness**	**Proportion**	**Range**
1	01011111	6	$(6/20) = 0.3$	$0.0 < n \leq 0.3$
2	11100111	4	$(4/20) = 0.2$	$0.3 < n \leq 0.5$
3	00110101	4	$(4/20) = 0.2$	$0.5 < n \leq 0.7$
4	10110010	4	$(4/20) = 0.2$	$0.7 < n \leq 0.9$
5	00001001	2	$(2/20) = 0.1$	$0.9 < n \leq 1.0$
Total		**20**		

Table 3.2: Example Roulette Wheel Selection for a 5 individual population. n is a random number generated for selecting individuals. Antibodies are trying to match the binary string 01111110.

Each individual has its fitness evaluated. These are summed and the total used to calculate each individual's "proportion of the wheel". For example, individual 1 forms 30% of the total fitness values, and so occupies 30% of the wheel. Normalised ranges are then calculated for each individual, representing a range of values which will select that individual. To select an individual, a random number, n, with uniform distribution in the range [0,1] is generated and matched to a *range*. The individual associated with that range is selected. For example, if the random number 0.31 was generated, individual 2 would be selected.

A problem with selecting individuals in this manner is that there is a high probability of losing the highest fitness individual - it may not be selected at all, or it will be selected but crossover and mutation will reduce its fitness. To stop this from happening, an additional selection operator, referred to as *elitism*, can be applied. Elitism simply copies the highest fitness individual from the parent population to the child population. The remainder of the child population is then generated by a selection method, such as roulette wheel. Elitism is also used in

this research in conjunction with roulette wheel selection.

Recombination Similarly to selection, recombination has many varieties of methods, each attempting to emulate recombination in biology. Again, consult [76] for details of other techniques. The approach used in this research however, is known as *single-point crossover*, as shown in figure 3.7.

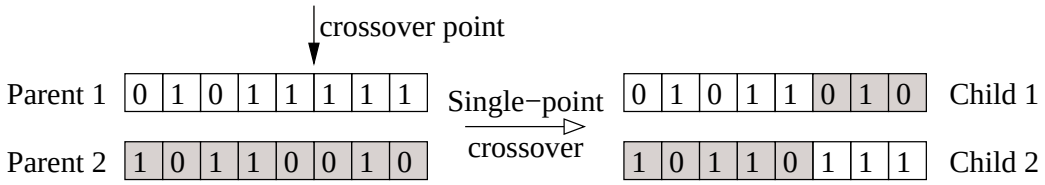

Figure 3.7: Single-point crossover applied to two binary-string chromosomes. *Diagram adapted from [25]*

Having selected two parent individuals (single-chromosome haploid), a random number in the range [0,1] is generated. If this falls below a user-defined threshold probability (the crossover probability), a random locus is chosen on a chromosome. All genes subsequent of this point are swapped between parents. Typically, a high crossover probability (≈ 0.8) is used [76].

Mutation Mutation of an individual is strongly dependent on the choice of representation for an individual. As such, the algorithm used and the rate of mutation vary per implementation. Often, binary-strings are used to represent a chromosome; a different encoding is used for this research however, and will be discussed in chapter 4.

Figure 3.8 demonstrates mutating a single bit of a binary-string individual. Every locus has equal chance of being mutated, based on the mutation rate; if, for a given locus, a random number (in the range [0,1]) falls below the mutation probability, that locus will be mutated (i.e. bit changed from 0 to 1, or vice versa). Usually, a very low mutation rate is used (< 0.05) is used [76].

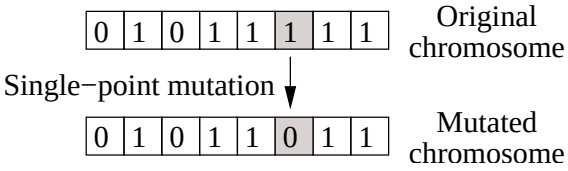

Figure 3.8: Single-point mutation applied to a binary-string chromosome. *Diagram adapted from [25]*

3.3 Immune Systems

Edward Jenner, a pioneer in vaccination, noticed that milkmaids who contracted cowpox did not suffer from its more deadly relative, smallpox. In his famous experiment [77, 98], Jenner inoculated an 8-year old boy, James Phipps, with fluid from a milkmaid's cowpox pustule, before exposing him to smallpox. Thankfully, for Jenner, James survived. Jenner soon realised the enormity of this result, and its implications for vaccination. In less than 4 years from this experiment, around 100,000 people had been vaccinated worldwide [16]. Since this experiment, numerous vaccines have been made against many diseases, such as polio, tetanus, measles and others [77].

The process by which the immune system evolves is key to the success of immunisation. When a person becomes infected with a virus, immune system cells multiply and modify themselves in an effort to detect the particular pathogenic cells (disease causing microorganisms) that have entered the body. Many immune cells are generated to fight off this virus, whilst at the same time a memory of the pathogen is developed, enabling for quicker recognition of the same, or a very similar, virus in the future. Should the worst happen and the person becomes re-infected, this memory helps speed up the identification and response to the virus. In this way, the immune system learns from its past pathogenic encounters, enabling a faster response to kill a repeating virus before it has a chance to significantly infect its host.

Artificial Immune Systems (AIS) use the vertebrate immune system as a

metaphor for developing, in general, optimized receptor patterns capable of solving a particular problem. AIS will be discussed more in section 3.3.3. First, to help understand the artificial metaphor, an overview of biological immune systems will be presented.

3.3.1 Biological Immune Systems

The human immune system is a multi-layered defence mechanism designed to protect our bodies from infectious organisms, or *pathogens*, such as viruses and bacteria. It is composed of physical barriers, such as skin; biochemical barriers, such as saliva and other destructive enzyme containing fluids; and two systems offering complementary forms of immunity: innate and adaptive.

The *innate immune system* is often regarded as a static mechanism, providing an initial response to a wide range of invading pathogens. It is also additionally responsible for initiating and controlling the adaptive immune response. However, because of its fairly static pattern recognition abilities, the innate immune system is often not considered as inspiration for machine learning, although some work does draw inspiration from it [9]. For this reason, and in particular because this research focusses on evolution and learning, this background will centre on the adaptive immune system.

The *adaptive* (or *acquired*) *immune system*, as the name suggests, has the ability to evolve, allowing it to direct its attack to previously unseen pathogens that the innate system is incapable of removing. This system consists of two main cell components that deliver a response to an invading pathogen. These components are lymphocyte cells (white blood cells), named *T-cells* and *B-cells*. Cells of these type originate in the bone marrow, but vary in where they mature - B-cells maturate in the bone marrow, whereas T-cells migrate to the thymus first. Maturation is a process whereby self-harming or ineffective lymphocytes are killed off before they enter the body. In this way the lymphocytes comprising the working immune system are the most effective at recognising pathogens and do not react to their

host's cells. Recognition occurs when a receptor on a lymphocyte binds complimentarily to an *antigen* exhibited on the surface of a pathogen. Reciprocally, an antigen is any molecule recognised by the immune system. B and T-cells that have matured go on to provide defences against invading pathogens in distinct, but interrelated ways.

T-cells display single specificity receptors (i.e. all receptors on one T-cell exhibit the same structure) on their surface which recognise and bind to *antigens* that have been processed and bound to a *major histocompatibility complex* (MHC) molecule. Two classes of MHC molecules exist: MHC class I molecules exist in all cells, whereas MHC class II molecules exist in special *antigen presenting cells* (APCs) such as macrophages, and B-cells. A cell that has been infected, or an APC that has ingested an antigen, digests and fragments the antigen into *peptides* [43, 78, 96]. These fragments then bind to the appropriate MHC molecule, forming a complex which is displayed on the surface of the cell. T-cells that recognise this complex can then directly attack and kill the infected cell. In addition to this however, T-cells also help regulate other cells, for example, they provide a co-stimulatory signal to B-cells; without this signal, B-cells cannot be provoked into an immune response.

Similarly to T-cells, B-cells also display single specificity receptors, known as *antibodies*, on their surface. These receptors however, bind to antigenic patterns on a pathogen's surface rather than peptides presented by MHC molecules. The magnitude of the subsequent immune response is affected by the degree of matching between the antibody and the antigen. How the adaptive immune system functions however, has always been the subject of much speculation. One theory developed to explain it is the *Clonal Selection* theory [19]. Antibodies that match (recognise) an antigen to some degree, undergo cloning dependent on the strength of that match, referred to as the *affinity* of that B-cell. Stronger affinity B-cells have greater ability in identifying the antigen and so produce more offspring than lower affinity B-cells. Furthermore, to cause diversity among B-cells, offspring are also mutated at a rate inversely proportional to their affinity - the stronger the

affinity, the stronger the antigenic match, and therefore the less mutation that is required. Memorisation of good antibodies also occurs, in preparation for a re-infection. Should the same, or similar, antigen invade the body again, this memorisation allows for a quicker response - instead of relatively weak affinity antibodies having to reproduce and mutate in order to increase the match strength, strong affinity antibodies have already been memorised and are able to produce a plethora of strong antibodies. This feature of the adaptive system provides the notion of learning in the immune system, and will be discussed more in section 3.3.2.

Shape and Shape-Space

Central to the antibody/antigen matching process is the idea of *shape* and *shape-space*. Each antibody and antigen has a specific shape - a structure, based on properties such as electrostatic charges or hydrogen bindings - that determines what it will match with [96]. To recognise an antigen, an antibody's shape must bind complementarily with a significant proportion of the antigen's shape. The degree of complementariness defines the strength of the binding and therefore the strength of the immune response.

Given the finite range of the structures involved (i.e. a finite range of possible charges, bindings, etc.), an antibody's shape exists within a range of possibilities, referred to as the immune system's *shape-space* [95]. This shape-space can be represented abstractly as the volume V in figure 3.9. An antibody can be considered as a point within this volume whose location is defined by its shape. Similarly, as antigens are also based on the same finite structures, they too can be considered as points existing within this volume.

As previously mentioned, the strength of the immune response to an antigen depends on the degree of complementariness between it and the binding antibody. Binding must occur over a sizable region of the antibody's and antigen's surfaces, in order for the antigen to be recognised [25]. If the degree is below some threshold (i.e. a small binding region), ϵ, then an immune response is unlikely

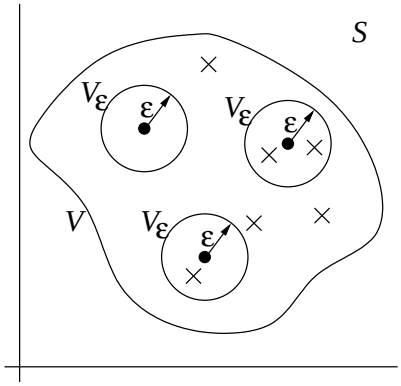

Figure 3.9: Abstract visualisation of shape-space. Within shape-space S, antibodies are dots with a spherical recognition region (V_ϵ), antigens are crosses. V is the volume containing all antibody and antigen-complement shapes *Diagram reproduced from [25]*

to occur. Therefore, this threshold can be thought of as providing a *recognition region* located around each antibody in the shape-space, depicted as V_ϵ in figure 3.9. An antibody will bind to and cause an immune response to any antigen whose **complement** lies within its recognition region (the response is proportional to the binding strength - i.e. the distance within the recognition region that the antigen is from the antibody). In this way, a body's finite number of antibodies can detect all possible antigens allowable in the shape-space. Considered another way, the immune system does not require a specific antibody to target each individual antigen; instead any antibody is capable of detecting a variety of antigens, meaning fewer antibodies are needed to detect a large number of antigens. The immune system generalises the shapes it is capable of detecting to provide maximum coverage through minimal numbers of antibodies.

The Immune Response

The outcome of an infection is strongly influenced, as in typical predator-prey relationships, by the sizes of the two populations involved - the larger population usually has the better chance of survival [96]. To increase the immune system's chance of defeating an infection, an immune response must increase the number

of antibodies that are effective against the infecting pathogen.

An antibody recognising an antigen initiates an immune response directed towards that antigen. The strength of this response is dictated by the degree of complementariness between the antibody and the antigen; a measure known as the antibody's *affinity* to that antigen. As previously mentioned, the most prominent explanation for the immune response is the *Clonal Selection Theory* [19]. Consider the shape-space model in figure 3.9. If two antigens (technically their complements) lie within an antibody's recognition region (e.g. the antibody on the right-hand side in the figure), then the antigen whose complement is closest to the antibody (in the shape-space) causes the strongest immune response. The clonal selection theory suggests that because these antibodies are helpful in fighting the invading pathogen then there should be more of them. The antibodies' respective B-cells produce clones of themselves to strengthen the antibody army against the invading pathogen. These clones mature into non-dividing antibody secreting cells, called *plasma cells*, which account for a large proportion of antibodies produced in an immune response. In the case where more than one antibody recognises a particular antigen, those B-cells with a higher affinity produce more offspring than the lower affinity B-cells, capitalising on the fact that high affinity antibodies are more adept at killing antigens than low affinity ones. Furthermore, in addition to this proliferation, diversity is also added to the army via mutation. The clones' receptors can mutate, inversely proportional to their affinity, in an effort to improve their recognition of the antigen. Those receptors that bind strongly to the antigen require only a slight modification to help them improve, and so mutate a little; receptors whose recognition ability is weaker however, need to mutate more. This process is known as *somatic hypermutation* due to the high mutation rates used within the immune system.

According to clonal selection then, an invading pathogen causes the expansion and diversification of the antibody army. Many new and slightly different antibodies are produced in order to achieve an efficient and effective immune response. With this search for stronger antibodies however, comes the possibility

that some antibodies may not be useful to the immune system; some might have low affinities or worse still, may be harmful to the very body they are trying to protect. Thankfully the immune system contains strategies useful for eliminating these useless or harmful antibodies, such as *negative selection* [63, 105]. Additionally, with the high proliferation rates involved, the large number of new cells produced could overwhelm the body. This does not happen though. Instead, the number of immune cells remains roughly constant throughout a body's lifespan, indicating that there must also be a high degree of cell death; low affinity and self-reactive B-cells suffer a process of programmed cell death, known as *apoptosis*, thereby helping to keep population numbers reasonably constant [105].

3.3.2 Evolution in the Immune System

There are two distinct theories about how the adaptive immune system learns and remembers antigens: *memory cells* and *immune network theory*. This research focusses on the more traditional, and more accepted, idea that long-lived memory cells are created[2].

Memory Cells - Clonal Selection

This is the more accepted idea of how the immune system learns. Because of this, memory cells are often the implied learning mechanism when referring to clonal selection.

When an antibody recognises an antigen and initiates an immune response, it causes the proliferation of its associated B-cell. During this mass-production of immune cells, many mature into plasma cells. Others however, differentiate into long-lived memory cells. These circulate the body, and upon recognition of the same or similar antigen, produce plasma cells. Plasma cells can produce antibodies quicker than standard B-cells, and so immune responses to subsequent exposures of an antigen, or similar antigen, are quicker. This is important for

[2]Immune network theory is based on the interactions amongst immune cells and amongst immune cells and antigens. Interested readers should consult [44].

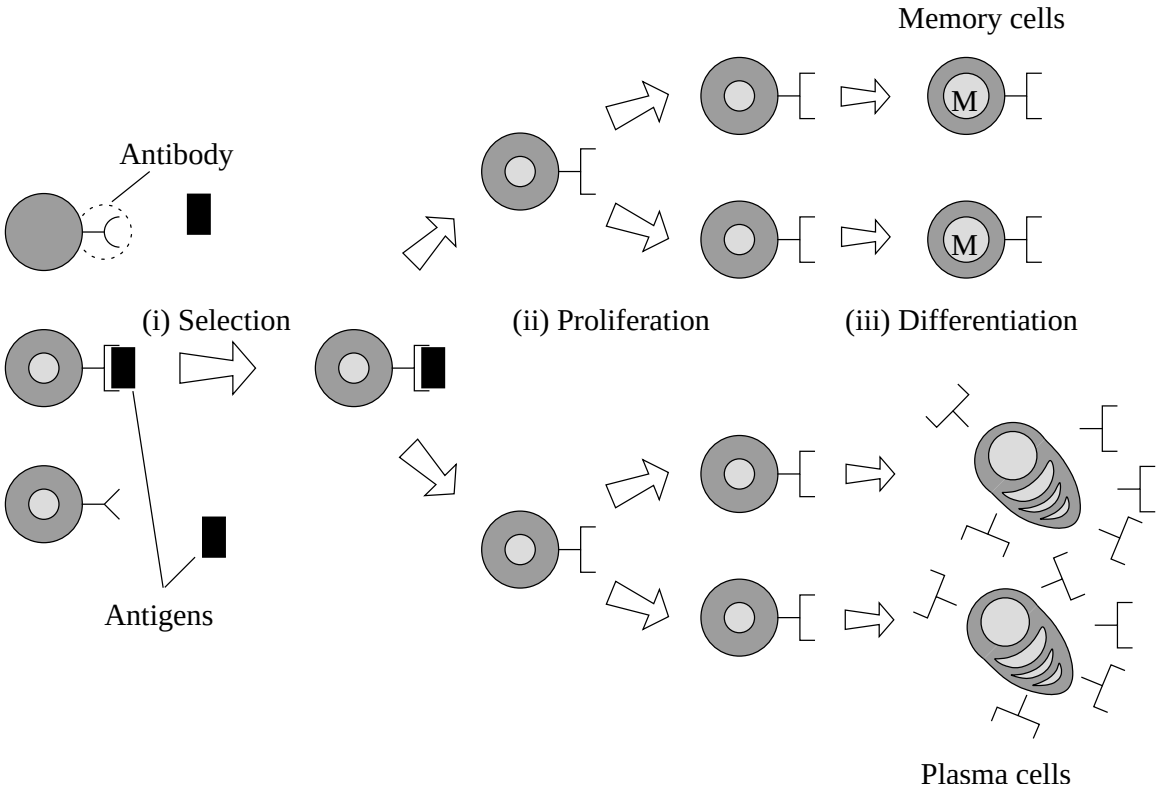

Figure 3.10: The clonal selection principle in response to foreign antigens. Antibodies matching antigens (*(i) Selection*) undergo cloning (*(ii) proliferation*) and differentiate into memory and antibody secreting plasma cells (*(iii) Differentiation*). *Diagram adapted from [25]*

quickly defeating any antigen before it has a chance to multiply significantly. Being long-lived also means that the immune system can maintain this memory of a particular antigen, without the need for constant exposure to it. In many cases, just one exposure is enough to produce life-long resistance.

3.3.3 Artificial Immune Systems

Similarly to GAs, Artificial Immune Systems use their natural biological counterparts as inspiration for creating adaptive, problem solving solutions. To be classified as an AIS therefore, a solution must embody three central principles: a basic model of an immune component; be designed by encompassing ideas from theoretical and/or experimental immunology; and, be aimed at problem solving

[25]. These three constraints are typically satisfied through an arbitrary modelling of antibody and antigenic populations, coupled with some biologically inspired process to modify the population members. Such modelling lends itself to being abstracted into three key areas: *representation*; *evaluation*; and, *adaptation*. Antibodies and antigens are *represented* in some format, often a binary string can be used. A method to *evaluate* how well an antibody performs against an antigen is needed, and this is used in some process to *adapt* the population and improve its solution(s). The choice for these three layers is usually made in consideration for the problem at hand. For example, for computer security, a representation based on network traffic rules (e.g. IP addresses, port numbers, etc.) or system calls, coupled with a *negative selection* algorithm to generate a set of disallowed connections may be more beneficial [31, 32]; alternatively, for pattern recognition, an encoding applicable to the data being searched (e.g. real values, or binary-strings) may be more relevant, combined with clonal selection to evolve the solutions[3]. This later case is most relevant to this research, where each solution encodes a test (see section 4.6) and a memory set is evolved using clonal selection.

The problem domain affects both the representation of antibodies and antigen, the evaluation mechanism, and the algorithm used. In general, the encoding of antibodies (and antigens) should reflect the information pertinent to the solution. Subsequently, the 'evaluation' mechanism used will be determined by the chosen representation, with the choice of algorithm being dependent on what is to be achieved. There are four classes of algorithm, based on effects and processes seen in the immune system, each presenting different attributes and/or results. They are: *bone marrow models* - used to generate repertoires of cells; *thymus models* - used to generate repertoires of cells capable of performing self/nonself discrimination; *clonal selection algorithms* - used to control how immune cells interact with antigens; and, *immune network models* - used to simulate the immune

[3]Note: neither of these examples specifies an 'evaluation' mechanism, although one is needed to allow the 'adaptation' process to function.

network theory [25]. Of these four, clonal selection algorithms are the most useful to this research as they evolve a set of memory cells, and will be described in the rest of this section.

Searching for better solutions to some (antigenic) problem using a clonal selection based AIS tries to follow the same principles used in nature. Solutions (*antibodies*) from the current solution set (*antibody population*) that solve a problem (*recognise an antigen*) are encouraged by **clonal expansion**, with **affinity maturation** serving to improve their effectiveness. Solutions that are good and should not be lost are stored in a **memory** (*memory cells*). These memory solutions, as well as the proliferated solutions, are then conjoined with the current solution set to evolve the next-generation solution set. Additionally, and to reflect the constant birth and death of new cells in the body, solutions in the population die over time and are replaced with random new ones - a process termed **metadynamics**. This algorithm is outlined by the flow diagram in figure 3.11.

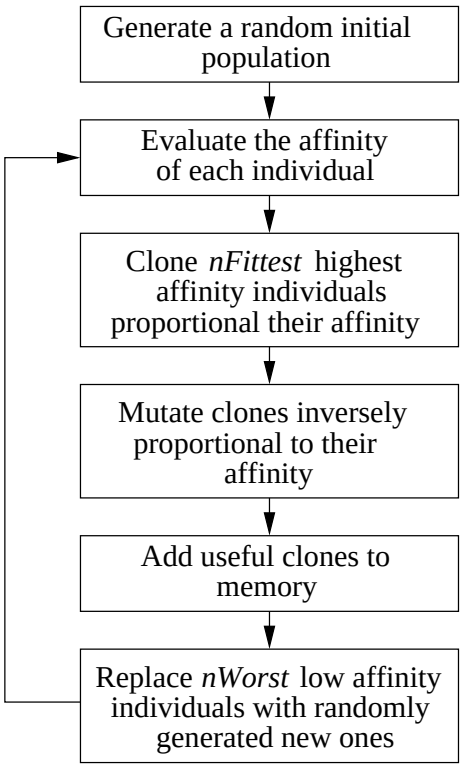

Figure 3.11: A basic Clonal Selection Algorithm.

Clonal Expansion In the biological immune system, all antibodies that bind with (match) an antigen undergo proliferation proportional to their affinity (i.e. proportional to how well they match with the antigen). Therefore cells which would match, but for whatever reason have not, will not multiply. In terms of artificial immune systems, this can be achieved by selecting a number of high affinity solutions (antibodies) to proliferate - the higher the affinity, the more clones produced for that solution. This is the approach taken with the CLONALG algorithm, adapted for this research [24].

As an example, consider the individuals (each individual is an antibody) in table 3.3. If the top 3 individuals are selected for cloning, with the highest affinity individual generating 3 clones, the second highest generating 2, and the third generating 1, then then following clones would be produced (picking individuals in order if they have the same affinity):

Parent				**Clones**		
Ind.	**Antibody**	**Affinity**		**Ind.**	**Antibody**	**Affinity**
1	01011111	6	→	6	01011111	6
				7	01011111	6
				8	01011111	6
2	11100111	4		9	11100111	4
			→	10	11100111	4
3	00110101	4	→	11	00110101	4
4	10110010	4	→	*No clones*		
5	00001001	2				

Table 3.3: Antibody clones generated for the top 3 individuals in a population during clonal expansion. Antibodies are trying to match the binary string 01111110.

Affinity Maturation The clones produced by clonal expansion also mutate at a rate inversely proportional to their affinity. The selective pressure caused by this "inverse" rate attempts to direct the mutation and improve the affinity of the individuals - a local search of the shape-space. Like the GA, how mutation occurs is dependent on the encoding used for the antibodies.

For example, continuing with the binary-string example in table 3.3, all 6 clones suffer mutation. The maximum affinity value achievable is 8 (i.e. the antibody encodes the binary-string 01111110 exactly), therefore the number of bits to be mutated for a clone can be calculated by $8 - \textit{affinity}$. In this case, clones 6-8 incur 2 mutations ($8 - 6 = 2$), and clones 9, 10 and 11 each incur 4 mutations ($8 - 4 = 4$):

Clones				**Mutated Clones**		
Ind.	**Antibody**	**Affinity**		**Ind.**	**Antibody**	**Affinity**
6	01011111	6	→	12	01111110	8
7	01011111	6	→	13	00011101	4
8	01011111	6	→	14	00111111	6
9	11100111	4	→	15	11001001	2
10	11100111	4	→	16	00100010	4
11	00110101	4	→	17	01111110	8

Table 3.4: Mutated clones generated for the top 3 individuals in a population during clonal expansion.

Memory Clones go on to differentiate into memory cells and antibody secreting plasma cells. Whilst plasma cells are not typically modeled in AIS, with algorithms satisfied by producing clones, memory cells (or some form of memory) can be helpful in storing good solutions that should not be lost. There is however, no predefined way of modeling memory as this depends largely on the desired outcome of the algorithm. CLONALG, for example, stores the best solution for each antigenic pattern it encounters. It does this by only replacing the memory solution if a higher-affinity individual exists in the current population. This research however, only stores a solution if it adds value (i.e. kills a mutant) not already contained in the current memory set - i.e. it increases the affinity of the whole memory set.

Metadynamics Finally, to reflect the birth and death of cells in the immune system, a number of low affinity individuals are removed from the population and

replaced by randomly generated individuals. This allows the AIS to perform a global search of the shape-space, thereby avoiding getting stuck in a local maxima.

3.4 Summary

Solutions to difficult computational problems within the computing domain can often be developed using novel algorithms inspired by nature. This chapter presented two such algorithms pertinent to this research: *Genetic Algorithms* and *Artificial Immune Systems*.

Genetic Algorithms are inspired by the evolution of species as theorised by Charles Darwin [22]. They use the theories of *natural selection* and *genetic variation* to evolve a population of solutions (individuals) to some computational problem, in much the same way that a species evolves over time. Natural selection means those individuals who are best adapted to their environment (i.e the problem) are more likely to survive and reproduce - strong solutions survive, whilst weaker ones die out. During reproduction, recombination and mutation in the genetic makeup of individuals causes offspring to vary from their parents. This variation serves to search out stronger individuals who, because of natural selection, are more likely to propagate the species.

Artificial Immune Systems present a different evolutionary approach, inspired by the vertebrate immune system. In particular, they focus on the adaptive immune system, and how it evolves its finite number of antibodies to defend against an infinite number of antigens. To do this, antibodies (representing solutions) that identify antigens (problems) are cloned (proportional to their affinity with the antigen) and mutated (inversely proportional to affinity) to try to improve their matching capabilities. Additionally, new antibodies are constantly being created to replenish dead cells and provide coverage to new areas of the shape-space.

Chapter 4 details how these two approaches can be utilised to automatically evolve test data.

Chapter 4

Evolving Test Data

4.1 Introduction

An initial test set for a program undergoing mutation testing can easily result in 50-70% of non-equivalent mutants being killed [6]. Improving this figure is the prime motivation for a tester to undertake mutation testing's cumbersome manual process. While Genetic Algorithms, amongst other techniques mentioned in Chapter 2, offer a beneficial reduction to the amount of work a tester has to perform, *can Artificial Immune Systems be used to improve upon these results?*

The primary hypothesis of this thesis is that an Immune Inspired Algorithm for Mutation Testing (IIA/MT) is consistently at least as effective at evolving test data as a Genetic Algorithm for Mutation Testing (GA/MT). This is, in itself, a high level hypothesis that requires thorough explanations of "consistency" and "effectiveness" in this context.

Consistency simply refers to the notion that whilst an algorithm could be at least as effective for a given program on a given run, this may not be the case over multiple runs. An algorithm must be at least as effective *on average*, in order to be *consistent*.

Effectiveness is a loose term, given to indicate some measure of performance an algorithm has in improving test data. *But how should this performance be*

measured? Traditionally, mutation testing is a manual process. Recent applications of various flavours of evolutionary algorithms to the domain (see Section 2.5.1) however, have reduced the need to manually develop new tests. Whilst this decreases the tester's work, these approaches do incur their own costs. Predominantly, any newly conceived test has to be executed against the remaining living mutants in order to judge its strength (in identifying mutants as incorrect), just as in traditional mutation testing. Unlike the classical approach however, evolutionary techniques are usually based on optimising a population of tests. In this case, each test within the population requires testing against all remaining living mutants through a series of iterations. As every mutant executed needs a finite amount of time to execute[1], the more mutants an algorithm executes, the longer it will take to run. The number of mutant executions needed depends largely on the algorithm employed and its representation of the problem space. This has direct implications for the *effectiveness* of an algorithm for mutation testing.

As an example, consider the situation where a genetic algorithm requires 5 million mutant executions to achieve a 95% mutation score for a given program, P. For an immune inspired algorithm to be *at least as effective* for the same program P, requires that a 95% mutation score is achieved in 5 million mutant executions or fewer. Alternately, *at least as effective* can be viewed as an immune inspired algorithm achieving a mutation score greater than or equal to 95% after 5 million executions. Either way, emphasis is placed on obtaining higher mutation scores in fewer mutant executions, or, considering the number of executions relates directly to algorithm run times, as achieving a higher mutation score in less time.

As a minimum, consistently speeding up the evolution of test data whilst maintaining high mutation scores increases mutation testing's chances of adoption by industry - and practical usage outside of research is an important goal for any new mutation testing technique. *But are there other criteria on which to judge effectiveness?* Considering the difficulties in finding test data to identify

[1]Mutants that enter infinite loops can have their execution times limited by a multiplication of the original program's execution time.

hard-to-kill mutants, an algorithm which consistently evolves such tests could be considered more effective than an algorithm that does not. In this case, *does an IIA/MT consistently find tests that a GA/MT does not?* Also, although perhaps of less importance, is the size of the generated test set. In comparison with another algorithm, an algorithm which generates a smaller test set attaining at least the same mutation score could be considered more *effective. Is this true of an IIA/MT compared with a GA/MT?*

This thesis makes a comparison between two evolutionary algorithms used to automatically evolve software test data: an Immune Inspired Algorithm and a Genetic Algorithm. This chapter outlines the process by which test data is evolved using mutation testing, and describes the two evolutionary algorithms employed to achieve this.

4.2 Approach Outline

Like all evolutionary algorithms, both the GA and the AIS algorithms iteratively optimise a population of individuals in an attempt to find a good solution - in this case, a set of tests which kill all mutant programs. The overall process by which an individual is evolved is common to both algorithms, and presented in figure 4.12. This is based on the mutation testing process outlined in figure 2.1.

The process starts by creating an initial population of tests, T. It is this population that will evolve. Next, numerous mutated versions of the program under test (PUT) are created based on the set of mutagens in use. Each test in T is then first executed on the PUT and then each mutant in turn. For each test, the result obtained from executing the PUT can be compared to the result obtained from executing each mutant; different outputs indicate erroneous mutants, which are killed. At this point, in traditional mutation testing the tester would then generate new tests to kill the remaining living mutants. For an evolutionary approach however, the population of tests T are evolved (by some mechanism), based on the mutants each test kills, with the aim of evolving a population that

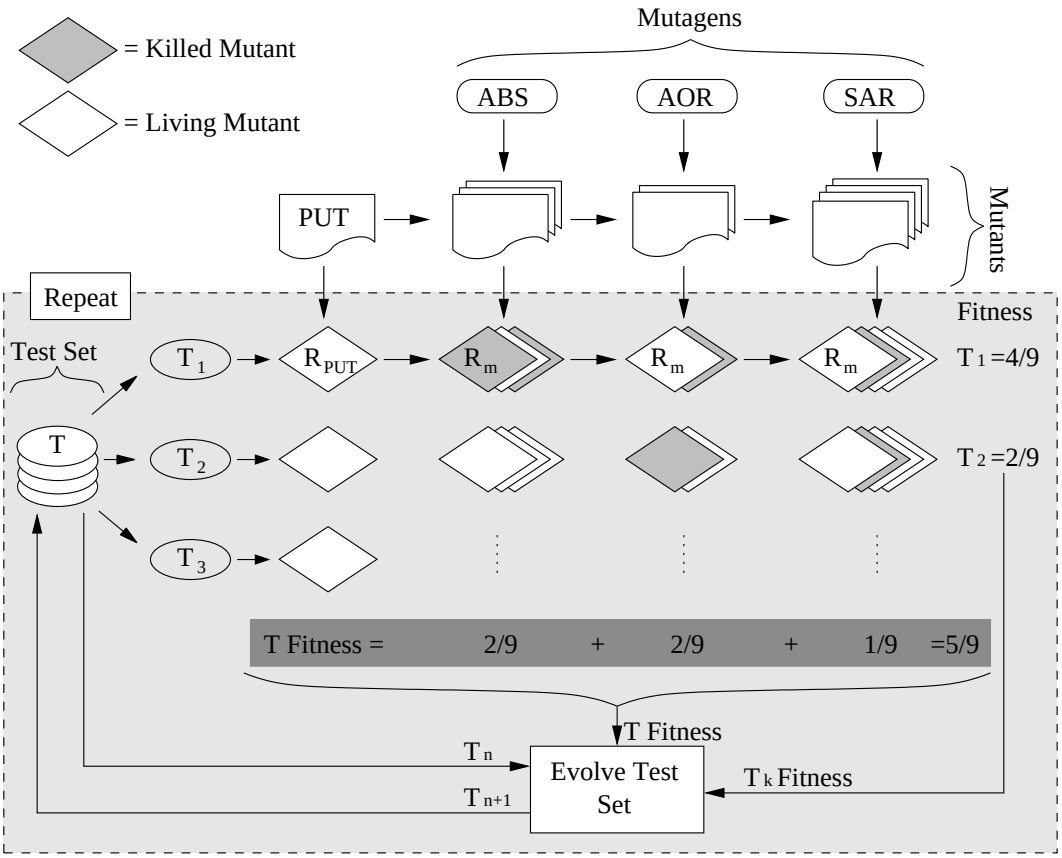

Figure 4.12: Test data evolution using mutation testing.

kills all the mutants.

Both algorithms have subtly different ways of evolving the population of tests. Understanding these differences will be useful in explaining any observed differences in the results. To this end, the algorithms used are described below in sections 4.5 and 4.6. These are explained using an evolutionary algorithm engineering framework (see section 4.4), adapted from [25], although other similar frameworks also exist, such as MAD as used by Watkins [104].

Before describing details of how each algorithm operates however, it is worth discussing in greater depth how the developed mutation system operates. Given some of the technical solutions to certain aspects of the system can impact the mutation scores achieved, particularly how tests are created and modified, knowledge of how the system works is important to fully understanding the experimental

results obtained in Chapters 5 and 6.

4.3 The Mutation Testing System

The mutation testing system developed for this research, shown in figure 4.13, operates in two conceptual stages: *initialisation* and *operation*. During initialisation, a PUT and a set of mutagens are used to generate the mutant programs. These are then used during the operation phase to automatically evolve tests. Eight operators have been defined: AAR; ABS; AOR; ASR; CNR; ROR; SAR; SVR (see table 2.1 for details). The system has been written using Java J2SE 5.0 [99].

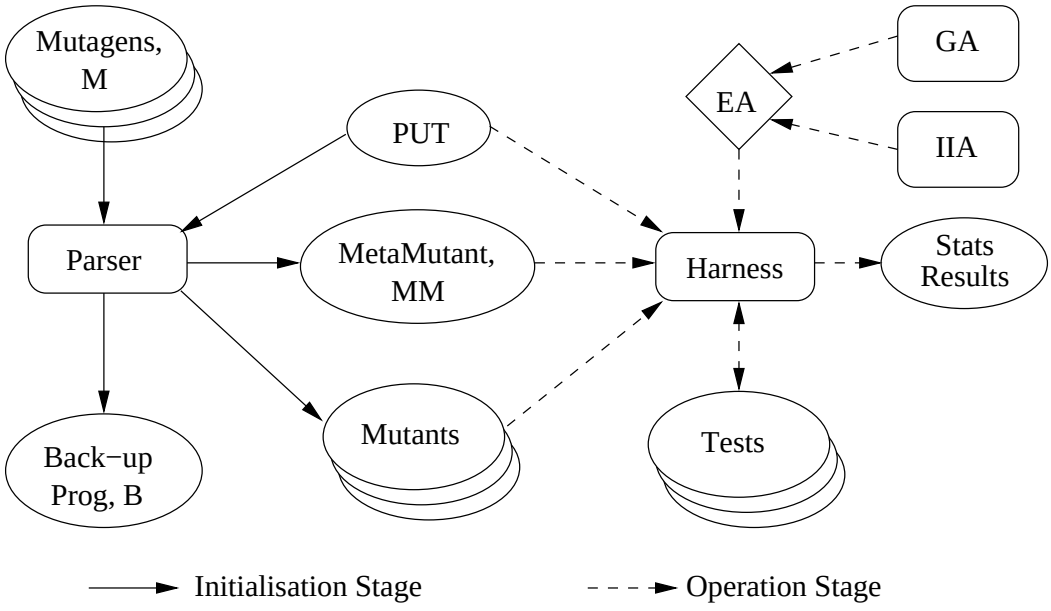

Figure 4.13: Outline of the MTAIS process engineered for this research. The process operates in two stages indicated by the different arrows.

4.3.1 Initialisation

Inputs:	Program-under-test	**Outputs:**	Mutants
	Mutagens		MetaMutant Program
			(Backup Program)

Initialisation of the system happens via two sequential processes: *Parsing* - produces a MetaMutant program; *Mutant generation* - produces all mutants from the MetaMutant. A backup of the PUT is also made.

Parsing

Parsing of the PUT occurs using ANTLR (version 2.7.2) which is a tool for generating compilers and translators from grammatical descriptions of a programming language [3]. In essence, an AST (Abstract Syntax Tree) is generated for the PUT, which is then modified by traversing it using a series of rule-based grammars (one for each mutagen). Each grammar replaces an operator, function or variable use (mutagen dependent) with an appropriate metaprocedure. The result is a syntactically correct metamutant AST which can be converted back to a compilable metamutant program that is able to be executed as any mutant.

MetaProcedures A metaprocedure is a mutagen specific function call that can represent one or more mutants. When executed, a supplied argument determines which mutant the metaprocedure should represent, and the appropriate action is taken. For example, the arithmetic mutagen for the code `x=y+z` will generate 6 mutants: `x=y-z`; `x=y*z`; `x=y/z`; `x=y%z`; `x=y`; `x=z`. These can all be replaced by a single metaprocedure, with the appropriate mutant selected at runtime by way of the last argument:

```
x=AOR.op(y, z, 0)
```

The last argument to any metaprocedure is always an integer, and represents an index into a "mutant array" (see the Mutant Generation section, following).

For every new metaprocedure used, this argument is incremented. The number of preceding arguments is dependent on the mutagen.

Mutant Generation

A mutant is represented by an integer array. Each index of this array refers to the same numbered metaprocedure. For example, index 0 refers to the metaprocedure whose last argument is 0; index 1 refers to metaprocedure 1, and so on.

The value of the integer at any index means different things depending on the mutagen. As a rule, if a mutagen replaces non-user-defined symbols (e.g. relational operators, arithmetic operators), the value in the mutant array represents one of these symbols. For example, for the arithmetic operator, the following values relate to the following operations:

$+ \rightarrow$	126	$- \rightarrow$	127	$* \rightarrow$	60	$/ \rightarrow$	128
$\% \rightarrow$	129	$L \rightarrow$	200	$R \rightarrow$	201		

During execution, the metaprocedure looks up the value in the mutant array for the associated index (i.e. its metaprocedure number). Based on this value, the appropriate action is executed. For example, if the mutant array value for (an AOR) metaprocedure 0 was 128, the metaprocedure would divide the first two arguments together; if it were 201, it would return the second of the two arguments (the right-hand side of the syntax tree).

If a mutagen replaces user-defined symbols (e.g. variables), then the mutant array index and its associated value are both used to return a symbol from a symbol lookup table. During parsing, each relevant metaprocedure maintains a lookup table of user-defined symbols, representing possible mutations for that variable, indexed by metaprocedure number. The first symbol (for any index in the lookup table) is always the original symbol in the PUT. During execution, the mutant array index (for this metaprocedure) selects the appropriate list of user-defined symbol mutations, and uses the value in the mutant array to make the selection. For example, consider figure 4.14. If metaprocedure 4 was an

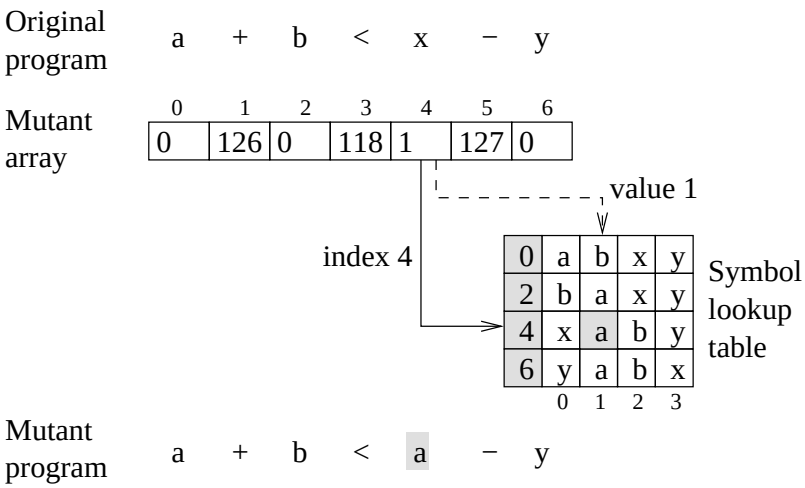

Figure 4.14: Example of how a mutant array is used to access a symbol lookup table.

SVR operator, the metaprocedure would look up the list of possible user-defined variable mutations under index 4, and return the variable indexed by the mutant array value stored at index 4 - i.e. value 1, which returns 'a'.

Having created the metamutant, an integer array for the PUT is created. This contains values for each metaprocedure in order to execute the original program (via the metamutant). A mutant is generated by modifying only a single value in the PUT array. All mutants for a PUT can be generated by creating a number of arrays representing every modification for every index.

4.3.2 Operation

Inputs:	PUT	**Outputs:**	Evolved Test Set
	MetaMutant Program		Statistical Output
	Mutants		
	Initial Test Set		

Operation of the system is a fairly straightforward implementation of the mutation testing process. An abstracted class diagram is shown in figure 4.15[2].

[2]To achieve a high level of abstraction, "classes" shown may refer to groups of classes. For

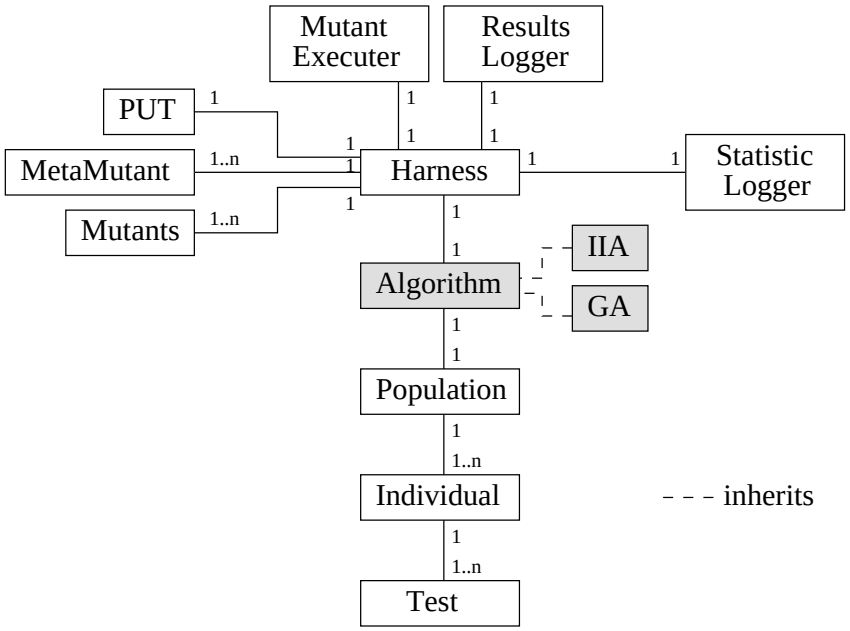

Figure 4.15: Abstract class diagram for the mutation testing system implemented for this research. "Classes" shown may refer to groups of classes. For example, the Harness "class" actually refers to multiple classes necessary to perform all the harness activities.

A test harness is the initialisation and control point of the test evolution. It provides support for the execution of metamutant programs, and as such requires the metamutant and mutants generated from the parsing phase, as well as the PUT. The harness has reference to an evolutionary algorithm (EA) used to evolve the tests. To enable easier code reuse between different EAs, the EA evolves a set of individuals, where each individual may contain one or more tests depending on the specific algorithm employed. Evolution of the tests is described later, in section 4.5 and 4.6, with the resultant evolved test set returned to the tester along with statistics on each iteration from the statistics logger.

---

example, the harness "class" actually refers to multiple classes necessary to control the mutation testing process.

Results Efficiency

Execution of the PUT or a particular mutant with a specific test occurs in the harness. Results from these executions are stored in two tables, indexed by test - one for the PUT results, one for mutant results. Every execution, only mutants that are living and do not have results in the table(s) for the specific test are executed. On execution completion, these tables are consulted to calculate which of the mutants are killed. Unfortunately, as the iterations progress, this table could become rather large and time consuming to search, and so for efficiency reasons unnecessary tests that are removed from the population also have their associated results removed from the tables. This introduces an obvious drawback; should a previously removed test reappear, it will be re-executed on all living mutants. This flaw is negligible however, as the time saved in searching a smaller table makes up for the additional time endured in re-executing tests. Indeed, a test may not even reappear.

Mutant Execution - Infinite Loops

Occasionally, the mutation of the PUT results in a program that will enter an infinite loop for some set of tests. To cater for this, a time limit is imposed (as a multiple of the execution time of the PUT) on the execution time of each mutant, as is standard in other research [7]. Furthermore, a mutant gets a number of attempts at completing execution within this limit. Failing that, it is deemed to have entered an infinite loop and is killed (assuming the PUT did not also enter an infinite loop for that test). Given the danger that a mutant might simply take longer to execute than the PUT however, the mutant is given the extra advantage of an increased time limit each attempt. This caters for instances when the initial time limit is too harsh, or when the processor is occupied with other activities thereby increasing the execution time.

Test Creation and Modification

Generating and modifying tests is a complex process in itself which more than likely affects the ability of the algorithms. This research does not look at the most appropriate method to alter and create tests, although this should be considered when designing such systems. Instead the method adopted for this research attempts to encourage the generation of legal input values (for a program) where this makes sense (e.g. a day value between 1 and 31), or failing that, encourage human-readable values (e.g. 20 for a triangle side length, as opposed to 3.54E5). Obviously, this is entirely program dependent (3.54E5 may be entirely appropriate), and so each test program has to have such appropriate ranges defined for it. It should be noted that this approach is for the creation of random tests only; although mutating tests uses a similar approach (described below), the values achievable are dependent on their original value, and not some pre-described range.

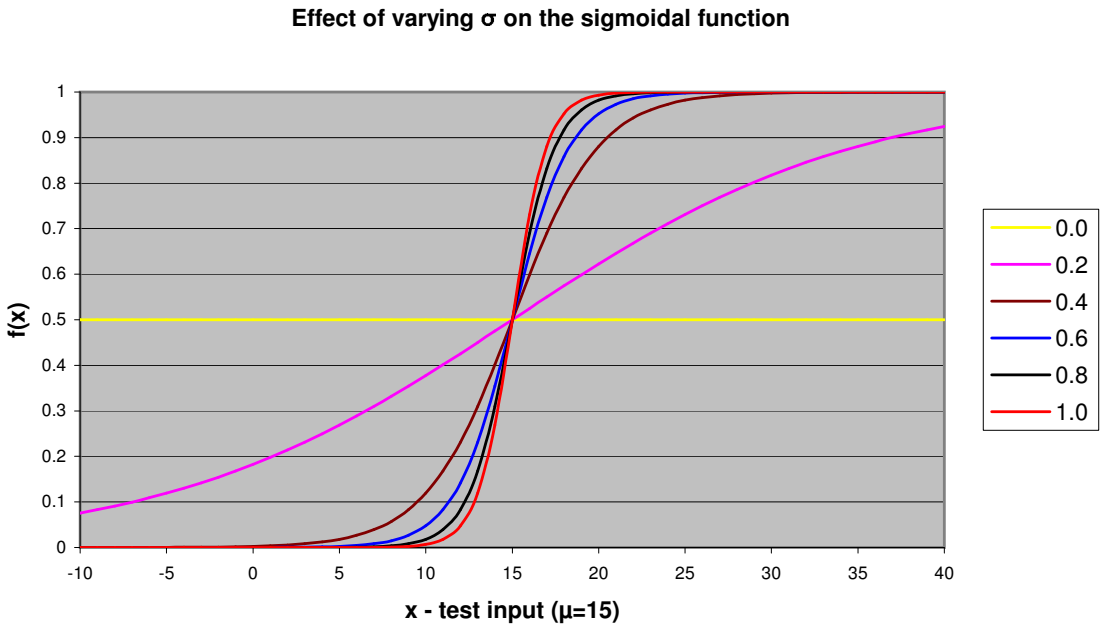

Figure 4.16: The sigmoidal graph and the effect of varying the spread (σ).

Whether randomly creating a test or mutating an existing one, the process is

similar. Each test input is calculated based on a sigmoidal function, as indicated in figure 4.16. Such a function produces an 'S' shaped curve that asymptotically approaches zero and one, making it ideal for adding nonlinearity to the input value selection based on normalised probabilities. The sigmoidal function is calculated (for this work by):

$$f(x) = \frac{1}{1 + \exp^{(-(x-\mu)*\sigma))}} \tag{4.2}$$

where x is an input value; μ is the mean input value; and σ is a normalised probability that alters the spread of the curve (figure 4.16) - smaller rates increase the spread, larger rates decrease it. By controlling the mean value and the σ probability it is possible to control the test values (by specifying a probable range of values) that are likely to be randomly generated.

How are test values determined? Essentially a reverse sigmoidal function is performed. A random probability is chosen, representing a $f(x)$ value on the sigmoidal function, and is used to derive an $f^{-1}(f(x))$ value (the new test input). The following equation is used to do this:

$$x = \mu - \left(\frac{\ln((\frac{1}{p}) - 1)}{\sigma} \right) \tag{4.3}$$

where p is a random probability, μ is the current or default test value, and σ is the rate of mutation which affects the spread of the curve.

When generating a new test, a default mean (μ) and σ probability are used for each program input to encourage legal or human-readable values - these are shown in table 4.5. The same values are used regardless of the algorithm employed. For example, when creating a new test for the *CalDay* program, the first input is calculated using a mean of 15 and a σ of 0.2. These values encourage the first input to be a value approximately within the range (but not limited to) of 0 to 30. The other inputs have there own mean and σ values.

When mutating a test however, the mean is the original test's value, and σ is the mutation rate. For example, if a test for the *CalDay* program had the first

	μ	σ	approximate input values[3]
CD			
day	15	0.2	$0 < x < 30$
month	6	0.5	$0 < x < 12$
year	0	0.002	$-2000 < x < 2000$
DR			
day(1|2)	15	0.2	$0 < x < 30$
month(1|2)	6	0.5	$0 < x < 12$
year(1|2)	0	0.002	$-2000 < x < 2000$
SEL			
index	4	0.5	$-2 < x < 12$
array size	8	0.5	$0 < x < 16$
array element	20	0.1	$-10 < x < 50$
TRI			
x|y|z	15	0.1	$-25 < x < 55$

Table 4.5: Mean and σ values for the inputs to the four tested programs, along with an approximate range of input values that will most likely be generated.

input of 25, then the mean for the reverse sigmoidal function would be 25 and the σ value would be the mutation rate (the mutation rate differs depending on the algorithm used and the ability of the test - see sections 4.5 and 4.6 for a discussion on each algorithm's approach). Ultimately, performing mutation in this manner encourages a test to mutate about its original value, rather than simply picking any value from the input domain.

4.4 Engineering Framework

The defined system requires an evolutionary algorithm to evolve the tests. Such algorithms operate in subtly different ways, but are typically based on a common approach. The engineering framework tries to capture this approach as three abstract layers.

[3]Approximate input values are presented to give an approximate indication of the range of input values most likely produced by the sigmoidal function for the μ and σ values. Actual input values may be outside of these ranges.

Although originally promoted for the design of Artificial Immune Systems, the three layered framework described in [25] can be applied to other biologically inspired techniques (albeit with slight modification to the terminology). Indeed, this structure was conceived by examining how other biologically inspired algorithms work, e.g. neural networks.

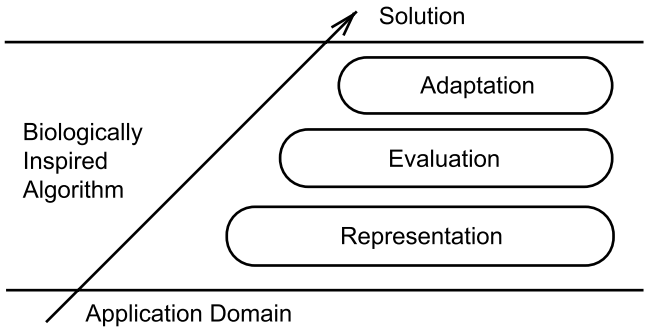

Figure 4.17: Engineering Framework for Evolutionary Algorithms. *Diagram adapted from [25]*

The framework builds a bridge from the application domain to a solution, as shown in figure 4.17. This requires at least the following three elements in order to provide a structurally sound engineering solution:

- *Representation*: A representation for the system's components;
- *Evaluation*: A means to measure the quality/ability of an individual within the environment or compared to other individuals;
- *Adaptation*: Functions that detail the system's dynamics, usually based on the 'evaluation' results.

As an example, consider Artificial Neural Networks [36]. Such networks consist of a set of artificial neurons linked together in a network - *Representation*. During training, input neurons are supplied with a stimulatory signal which propagates

through the network to generate an output. The network can be evaluated by comparing the generated output with the expected output - *Evaluation*. Weights on the neuron links can then be updated based on this evaluation in order to improve the network - *Adaptation*.

4.5 Genetic Algorithm

Genetic algorithms are inspired by Darwinian theories of species evolution, as described in chapter 3. This algorithm iteratively evolves a population of individuals - each individual being **a set** of tests - in an attempt to find a good solution. A good solution is an individual (set of tests) that kills the most non-equivalent mutants. At the end of the process, the best individual is returned to the tester. Figure 4.18 illustrates a single iteration of the GA/MT algorithm, with the pseudocode shown in figure 4.19.

Representation

Darwinian evolution of a population is based on survival of the fittest and genetic variation in response to environmental pressures. Each member of the population is defined by a set of genes, called a chromosome. Therefore, using a genetic algorithm, a good member for mutation testing is a set of tests that kill the most mutants:

- A **chromosome** is a **set of tests**;
- The **environmental pressure** is the **set of mutants generated by all the mutagens**.

As an example, for the *TriangleSort* program, a chromosome would be a set of m tests, such as:

[<1,2,3>, <4,5,6>, <-1,5,7>, ..., <99,42,8>]

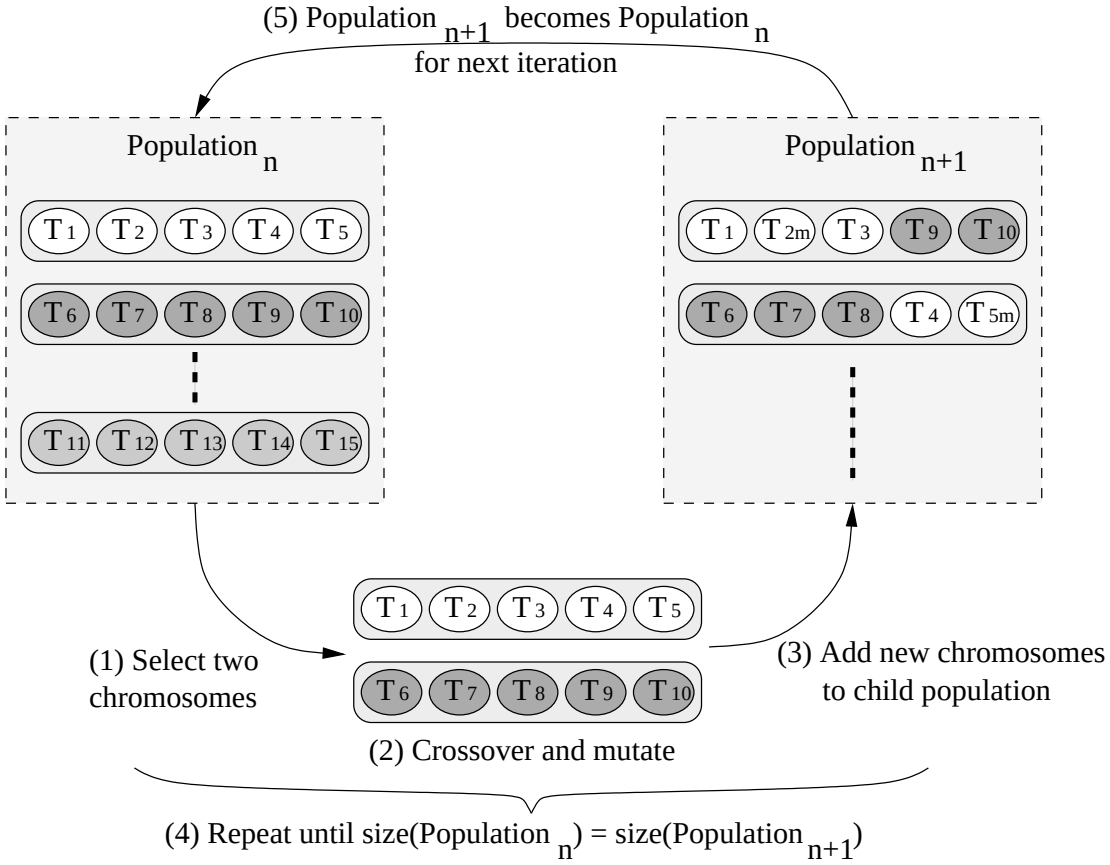

Figure 4.18: Single iteration of the Genetic Algorithm process.

Evaluation

Population members whose chromosome makeup is best adapted for their environment have the greatest chance of survival. This fitness level also affects their likelihood of reproduction. Therefore, to use a GA for mutation testing:

- **Fitness** is the **overall mutation score of all tests in the chromosome**.

The fitness of a single test is the mutation score, calculated as described in section 2.2. However, a chromosome contains m tests, and so the fitness must be calculated by the total number of non-equivalent mutants killed by all the tests combined (**note:** this is not simply the sum of each test's mutation score).

```
            n           the number of iterations to perform
            s           size of main population
  inputs:   m           the number of tests in each individual
            crossRate   the rate of crossover amongst individuals
            mutRate     the rate of mutation
 1  begin
 2      i ← 0
 3      P_i ← initPop(s)
 4      while i < n do
 5          Ch ← {}
 6          calculateAffinity(P_i)
 7          total ← sumAffinities(P_i)
 8          normaliseAffinities(P_i, total)
 9          B ← getBestTest(P_i)                    // get best Test
10          Ch ← combine(Ch, B)    // add best to child population
11          while size(Ch)< s do
12              I_1 ← rouletteSelection(P_i)
13              I_2 ← rouletteSelection(P_i)
14              Q_1,2 ← singlePointCrossover(I_1, I_2, crossRate)
15              Q_1 ← mutateChild(Q_1, mutRate)
16              Q_2 ← mutateChild(Q_2, mutRate)
17              Ch ← combine(Ch, Q_1,2)
18          end
19          P_i+1 ← Ch
20      end
21      return getBestTest(P_i)        // return the highest fitness
                                                    // individual
22  end
```

Figure 4.19: Genetic Algorithm for Mutation Testing

Adaptation

Reproduction in a population is based on the fitness of the members. Fitter members are more likely to survive, and therefore more likely to reproduce than less fit members. Reproduction is a recombination of two members' chromosomes along with mutation of random genes, as described in section 3.2.2. This process is used for a GA approach to mutation testing:

- Let the population in the i-th iteration be called P_i. This is initialised to contain s individuals, each containing m tests, that are either randomly generated or specified by the tester. n iterations are performed. At the beginning of an iteration, every individual in P_i has its affinity calculated. An individual's affinity is the combined mutation score of all the tests in that individual - i.e. the normalised percentage of all the mutants killed by all the tests in that individual (*line 6*). Each affinity is then normalised (*line 8*) against the sum of all the affinities (*line 7*) in preparation for roulette wheel selection. Next, to prevent the population mutation score from dropping between iterations, the best individual is selected and added as the first member of the child population (*lines 9-10*). This copied individual does not undergo crossover and mutation, however the original also remains in the main population for the remaining steps. Further members are added to the child population through a three step process of: selecting two parents using roulette wheel selection (higher affinity individuals have a higher probability of being selected - *lines 12-13*); performing single point crossover (the tails of each parent chromosome, from a randomly chosen point, are swapped - *line 14*); and finally, performing mutation on randomly chosen tests within each crossed-over individual (*lines 15-16*). These children are added to the child population (*line 17*), and the process repeats until it contains s individuals. The child population then becomes the next iteration's parent population, T_{i+1}.

4.5.1 Variables

n The number of iterations to perform.

s The number of individuals in the population.

m The number of tests in each individual.

crossRate The probability of crossover occuring.

mutRate The probability of a test being mutated.

4.6 Immune Inspired Algorithm

The Immune Inspired Algorithm for mutation testing is based on the Clonal Selection theory, described in chapter 3, primarily because of its ability to evolve a set of useful receptors and its similarities with traditional Genetic Algorithms. More specifically, the IIA/MT was loosely based on the CLONALG implementation [24], although changes were made to focus the algorithm on the mutation testing problem domain, in particular removing the concept of a memory individual per antigen[4], and instead allowing many individuals to contribute to an antigen's recognition.

The algorithm iteratively evolves a population of individuals - an individual being a test - searching for useful solutions. A useful solution is a test which kills at least one mutant program not already killed by any other test found so far. Those tests that are found to be useful are placed into a memory set to be returned to the tester at the end of the process. A single iteration of this process is illustrated in Figure 4.20.

Pseudocode for the IIA/MT is shown in figure 4.21; references to line numbers in the following descriptions refer to this code. Details of the methods used in this code are described in Appendix A.

[4]In CLONALG, the highest-affinity antibody is stored in the memory layer for each antigen applied to the algorithm.

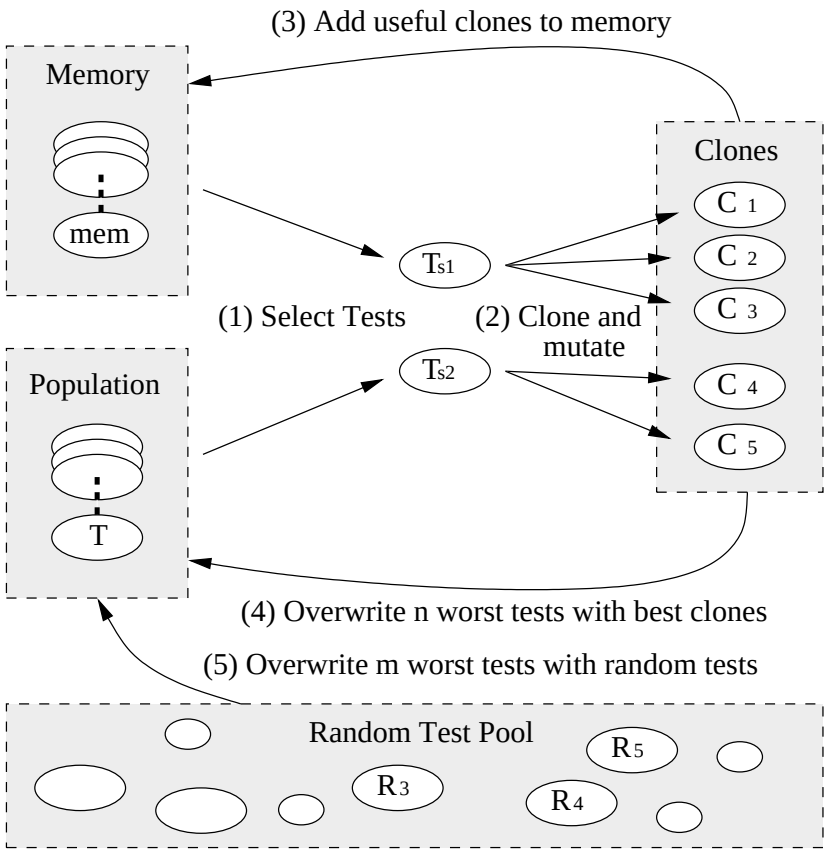

Figure 4.20: Single iteration of the 'Immune Inspired Algorithm for Mutation Testing' process.

Representation

The immune system develops a population of antibodies to ward off various antigens that invade the body. Any algorithm inspired by the immune system metaphorically needs a population of artificial antibodies to ward of artificial antigens. Therefore in this Immune Inspired Algorithm:

- An **antibody** is a **test** - e.g. <1,2,3> (i,j,k) for the *TriangleSort* program;
- An **antigen** is the **set of mutants generated by all the mutagens**.

	n	the number of iterations to perform
	s	size of main population
inputs:	*nFittest*	number of highest affinity Individuals to select
	nWorst	number of worst affinity Individuals to select
	cloneRate	affinity multiplier affecting the number of clones

```
1  begin
2     i ← 0
3     T_i ← initPop(s)
4     M ← {}                                  // reset memory set, M
5     while i < n do
6        calculateAffinity(T_i)
7        L ← addToMemory(T_i, M)          // L = useful individuals
8        B ← selectFittest(T_i, nFittest)       // added to memory
9        R ← randomSelection(M, nFittest)
10       B ← combine(B, R)
11       B_1 ← randomSelection(B, nFittest)
12       L ← combine(L, B_1)
13       C ← clonalSelection(L, cloneRate)
14       addToMemory(C, M)                  // result not needed
15       T_{i+1} ← metadynamics(T_i, C, nFittest, nWorst)
16    end
17    return M                                // return memory set
18 end
```

Figure 4.21: Immune Inspired Algorithm for Mutation Testing

Evaluation

Recognition in the immune system occurs with a complimentary match between an antibody and antigen's shape[5]. The degree on complementariness defines the strength of the binding, or *affinity*, and affects the amount of adaptation of the antibody population. In this algorithm:

- **Affinity** is the **mutation score**.

The mutation score of an antibody is calculated as described in section 2.2, as the number of non-equivalent mutants killed by that test.

[5]Shape is a term for the receptor's structure which is defined by properties such as electrostatic charge. See section 3.3.1.

Adaptation

Antibodies within the immune system undergo clonal selection when they bind with an antigen - this is affinity proportional cloning and mutation as described in section 3.3.1. A similar process occurs in this Immune Inspired Algorithm:

- Let the population in the i-th iteration be called T_i. This is initialised to contain s tests, either by randomly generating tests or by specifying each one. The memory set is initially empty. At the beginning of an iteration, every test in T_i has its affinity calculated (i.e. its mutation score) with a record kept of which mutants each test kills (*line 6*). Useful tests (those that kill at least one mutant not killed by any test in the memory set) are added to the memory set, M (*line 7*). `nFittest` highest affinity tests are then selected from T_i (*line 8*). These selected tests are combined with `nFittest` random[6] tests selected from the memory population (*line 9*). In neither case are the selected tests removed from their respective populations. `nFittest` tests are chosen from this combined set and together with the useful tests selected earlier, undergo clonal selection - i.e. affinity proportional cloning and mutation (*lines 10-13*). A minimum of 1 cloned test per parent test is created, and all clones undergo mutation inversely proportional to their affinity (mutation score). Useful cloned[7] tests are added to the memory set (*line 14*). `nFittest` cloned tests are added to T_i, and a number of worst affinity tests in T_i are removed until T_i size equals s. Finally, `nWorst` tests in T_i are replaced by new, randomly generated tests (*line 15*). This process repeats for n iterations.

[6] Tests are selected at random from the memory population because a "useful" test does not necessarily have a high mutation score (e.g. it may only kill a single mutant). Selecting only high affinity memory tests would restrict local searches from occurring around these low scoring tests.

[7] 'Cloned' tests are clones that may have also been mutated.

4.6.1 Variables

n The number of iterations to perform.

s The initial number of individuals in the population.

nFittest Defines the number of test to be selected (from either the memory set or the current population) to undergo clonal selection. Also used in the metadynamics phase to replace the `nFittest`[8] worst tests in the current population by `nFittest` best clones.

nWorst Defines the number of worst (lowest affinity) tests to remove from the current population and replace with randomly generated ones during metadynamics.

cloneRate A multiplication factor of an individual's affinity defining the number of clones to produce during clonal selection. A minimum of 1 clone is produced during cloning.

4.7 Differences in the Algorithms

Both algorithms evolve a population of individuals in an attempt to find a good solution to the problem they encode. However, a fundamental difference is that, in general, a GA aims to find the best individual to encompass the problem domain as opposed to an Immune Inspired Algorithm which evolves a set of specialist individuals to encompass the problem. This has an impact on an individual's representation. In mutation testing, it is unlikely that a single test will kill all mutants. As such, the best GA individual needs to possess enough tests to kill all mutants. *But how many is enough?*[9] An Immune Inspired Algorithm, on the other hand, naturally evolves a dynamic number of specialist individuals, each killing

[8] It helps to think of `nFittest` purely as a number in this case, and not as "n fittest".
[9] This is discussed more in section 6.3.1

at least one mutant not killed by anything else. There is no need to predefine how many tests will be needed.

The evaluation mechanism are the same between the algorithms - the number of mutants an individual kills. Further differences however, occur in the adaptation mechanisms of the algorithms themselves. Whilst there is an argument that immune algorithms are effectively GAs without crossover, there is an important difference between the purposes of their respective selection and mutation methods. GAs *select* high fitness individuals because they are good solutions to the problem domain. Disregarding crossover, they randomly *mutate* child individuals to add diversity to the population - to search for new solutions or to escape local maxima. Immune algorithms on the other hand, evolve a set of specialist solutions. Their form of *selection*, as part of cloning, produces local populations around potentially good solutions. These clones are *mutated* to search around these, already good, solutions in an attempt to find better (higher affinity) solutions - local searches. Diversity - breadth search - is added in a later stage called metadynamics, by the death of individuals and the introduction of new, random individuals. Consequently, for an Immune Inspired Algorithm, cloning and mutation are proportional to affinity (high affinity individuals undergo higher cloning rates and less mutation than low affinity ones), whereas for a GA, although selection is proportional to fitness, mutation is usually at a fixed rate (typically $< 10\%$).

4.8 Programs Under Test

The following two chapters compare both algorithms and analyse the effects of parameter variability on four test programs. The code for these programs is shown in Appendix B. To help understand the results in the next two chapters, it is useful to have knowledge of the programs-under-test themselves. This section provides high-level specifications of the programs, indicating the functionality of each program.

4.8.1 CalDay

This program converts the specified date (given as three arguments: `day`, `month`, `year`) into a Julian date (not including the fractional time part). The first three lines (of the specification) account for negative years and the rearrangement of the year to begin in March (dates in January and February are moved to the end of the previous year). The Julian date is then calculated as the number of days from the 1st January, 4713 B.C. (1720995). If the date to be converted is after 15th October 1582, then the Julian date is adjusted to suit (this is the date of the introduction of the Gregorian calendar, designed to correct inaccuracies in the length of the year calculations). The Julian date is then returned.

toJulian

$$
\begin{array}{l}
day? : \mathbb{Z} \\
month? : \mathbb{Z} \\
year? : \mathbb{Z} \\
result! : \mathbb{R} \\
julian : \mathbb{R} \\
julianMonth : \mathbb{R} \\
julianYear : \mathbb{R} \\
number_days_since_base : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
date : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow DATE \\
change_to_gregorian_calendar : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R} \\
\hline
(year? < 0) \Rightarrow (julianYear = year? + 1) \\
(month? > 2) \Rightarrow (julianMonth = month? + 1) \\
(month? \leq 2) \Rightarrow ((julianMonth = month? + 13) \wedge \\
\qquad\qquad\qquad\qquad (julianYear = julianYear - 1)) \\
julian = number_days_since_base(julianMonth, julianYear, 1720995.0) \\
(date(day?, month?, year?) \geq date(15, 10, 1582)) \Rightarrow \\
\qquad julian = change_to_gregorian_calendar(julian, julianYear) \\
result! = julian
\end{array}
$$

4.8.2 DateRange

This program calculates the number of days between two dates. `numberDays` has 6 inputs, representing the day, month and year of the two dates. Initially the dates are analysed for validity by the `goodDate` function. If valid, the number of

days between the two dates is returned (note, this is always a positive number). If either of the dates are invalid, -1 is returned to indicate an error.

numberDays

$$
\begin{array}{l}
day1? : \mathbb{Z} \\
month1? : \mathbb{Z} \\
year1? : \mathbb{Z} \\
day2? : \mathbb{Z} \\
month2? : \mathbb{Z} \\
year2? : \mathbb{Z} \\
result! : \mathbb{Z} \\
difference_in_days : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
(goodDate(day1?, month1?, year1?) \land \\
\quad goodDate(day2?, month2?, year2?) \land \\
\quad result! = difference_in_days(day1?, month1?, year1?, \\
\qquad\qquad\qquad\qquad day2?, month2?, year2?)) \lor \\
(\neg\ goodDate(day1?, month1?, year1?) \lor \\
\quad \neg\ goodDate(day2?, month2?, year2?) \land \\
\quad result! = -1)
\end{array}
$$

The validity of a date is determined by the `goodDate` function. This checks that the `month` argument is within the range 0 to 11 inclusively, and that the `day` argument is between 1 and the number of days in the month. This function returns a boolean.

$$BOOL ::= True \mid False$$

goodDate

$$
\begin{array}{l}
day? : \mathbb{Z} \\
month? : \mathbb{Z} \\
year? : \mathbb{Z} \\
result! : BOOL \\
days_in_month : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{N} \\
\hline
((0 \leq month? \leq 11) \land \\
\quad (1 \leq day? \leq days_in_month(month?, year?)) \land \\
\quad result! = True) \lor \\
(result! = False)
\end{array}
$$

4.8.3 Select

The *Select* program returns the `k`-th largest element from the un-ordered array `arr`, without affecting the ordering of this array. `k` must be in the range of the number of elements in `arr`, or else an exception is thrown.

selip

$$
\begin{array}{l}
k? : \mathbb{Z} \\
arr? : seq\mathbb{R} \\
result! : \mathbb{R} \mid Exception \\
temp : seq\mathbb{R} \\
\hline
((k? < 1) \lor (k? > \#arr?)) \Rightarrow (result! = Exception) \\
temp = arr? \\
temp' = shell(\#temp, temp) \\
result! = temp?(k?)
\end{array}
$$

The `shell` function sorts the first `num` elements of the `a` array. The '$for : (seqT) \rightarrow \mathbb{Z}$' and '$after : (seqT) \rightarrow \mathbb{Z}$' functions are defined in [15] and return the subsequence of elements (from the first sequence argument) from the start to the second-argument element, or from the second-argument element to the end, respectively. The remainder of the original sequence, `a`, is catenated on to the end of the sorted section.

shell

$$
\begin{array}{l}
num? : \mathbb{Z} \\
a? : seq\mathbb{R} \\
sort : seq\mathbb{R} \rightarrow \mathrm{seq}\,\mathbb{R} \\
\hline
a?' = (sort(a?\ for\ num?)) \frown (a?\ after\ num?)
\end{array}
$$

4.8.4 TriangleSort

This program classifies triangles into either scalene, isosceles, equilateral or invalid based on three side length arguments. Preconditions are that all side lengths must be greater than 0 and that each side must be less than the sum of the other two sides. If either of the preconditions are false, then the triangle is invalid. If the

preconditions are true, then if all sides are equal, the triangle is equilateral; if two sides are equal, the triangle is an isosceles; and if all sides are different lengths, then the triangle is a scalene triangle.

$$RESULT ::= Scalene \mid Isosceles \mid Equilateral \mid Invalid$$

triang

$$
\begin{array}{l}
i? : \mathbb{Z} \\
j? : \mathbb{Z} \\
k? : \mathbb{Z} \\
result! : RESULT \\
\hline
((((i? > 0) \land (j? > 0) \land (k? > 0)) \land \\
((i? + j? > k?) \land (i? + k? > j?) \land (j? + k? > i?)) \land \\
\quad ((i? == j? \land j? == k?) \Rightarrow (result! = Equilateral)) \lor \\
\quad ((i? == j? \land j? \neq k?) \lor (i? == k? \land k? \neq j?) \\
\qquad \lor (j? == k? \land k? \neq i?) \Rightarrow (result! = Isosceles)) \lor \\
\quad (result! = Scalene)) \\
\lor (result! = Invalid))
\end{array}
$$

4.9 Summary

The main aim of this thesis is to test whether an Immune Inspired Algorithm approach consistently generates test data using mutation testing that is at least as effective as tests generated using a Genetic Algorithm. To be *consistent*, the tests generated must be at least as effective over multiple runs. *Effectiveness* however, can be measured in three ways: primarily by the number of mutant executions required to achieve a particular mutation score; also, by how many hard-to-kill mutants the tests kill; and finally, by the size of the generated test set.

This chapter described and compared the two algorithms, with reference to the engineering framework outlined in section 4.4. Both these algorithms will be used in the following chapter to test the main hypothesis.

Chapter 5

Algorithm Comparison

5.1 Introduction

Genetic Algorithms (GAs) have previously been used to automatically generate test data using mutation testing [7, 13, 46], achieving good results. This chapter compares a GA approach with the Immune Inspired Algorithm (IIA) approach presented in chapter 4, asking whether this new technique is at least as effective (see Chapter 4 for a definition of effectiveness). To answer this hypothesis (H1), 6 sub-hypotheses are first described (H2-7) and then fairly investigated within the constraints imposed in section 5.3. Section 5.3.1 describes the statistics used throughout this chapter.

5.2 Hypotheses

The primary hypothesis is simply:

H 1 *An Immune Inspired Algorithm is consistently, at least as effective as a Genetic Algorithm for evolving test data through mutation testing.*

To verify this hypothesis, it needs to be split into component hypotheses defining effectiveness. As identified, effectiveness relates to: an improved mutation

score in (possibly) less time; finding tests for hard-to-kill mutants; and, generating a smaller sized test set. In the first instance however, it may be pertinent to determine whether an IIA/MT is capable of evolving tests:

H 2 *An Immune Inspired Algorithm for Mutation Testing (IIA/MT) is capable of improving the mutation score for a given program by automatically evolving the test data.*

Given other evolutionary algorithms have been applied to test evolution, and the similarities between these and AIS's, it may seem unnecessary to question the use of Immune Inspired Algorithms in this context. For completeness, the hypothesis still needs to be asked. There is little point continuing if it is found to be false.

As previously stated, the most relevant measure, in terms of effectiveness, is determined by the time taken to generate a test set and its associated mutation score. Evolutionary algorithms evolve a population of individuals over a series of discrete intervals, usually called *iterations*. Comparing algorithms by measuring mutation scores on a per iteration basis would seem logical, but the amount of work each algorithm does per iteration is unlikely to be the same. Instead, the number of mutants executed forms a better basis on which to compare algorithms as the work involved in executing a mutant can be considered constant irrespective of the algorithm. This leads to:

H 3 *Test data evolution using the IIA/MT requires fewer mutant program executions than the GA/MT to achieve at least the same mutation score.*

A second important measure of algorithm effectiveness is directly related to an improved mutation score. Algorithms can often be used to evolve test sets achieving high (>90%) mutation scores; it is the last remaining few mutants that often prove difficult to kill as they require very specific tests. An algorithm which can consistently produce tests to kill such mutants is comparatively more effective than one which cannot (see Section 5.7 for details about hard-to-kill mutants).

H 4 *The final test set generated by the IIA/MT consistently identifies more hard-to-kill mutants than one created by the GA/MT.*

Finally, and more an ideal feature than a necessity, effectiveness can be measured by the size of the generated test set. An algorithm that generates less tests to produce an equal mutation score is more effective than one which creates more tests. This reflects the idea that a smaller test set is functionally easier (i.e. it requires less time to test with) than a larger equivalent one.

H 5 *Test data evolution using the IIA/MT results in a smaller average test set size than from using the GA/MT.*

The four hypotheses (H2-H5) should present enough evidence to clarify hypothesis H1. How these results should be combined is by no means straightforward however. In particular, *what emphasis should be placed on each of the three 'effectiveness' components?* The answer to this is likely to depend on the tester's requirements. For example, maybe a tester has killed a large percentage of mutants and only has hard-to-kill (HTK) mutants left - his choice of algorithm will probably favour ability to kill HTK mutants rather than speed of operation.

Besides these hypotheses though, there are other hypotheses that, whilst not aiding clarification of H1, are useful to study in their own right. As previously stated, the time taken to achieve a certain mutation score has highest importance in terms of effectiveness. Hypotheses H2 and H3 use the number of mutant program executions as a more consistent measure of execution time. For completeness, a comparison based on the number of iterations may also be useful.

H 6 *Test data evolution using the IIA/MT achieves a higher mutation score by a specific number of iterations than by using the GA/MT.*

Lastly, with regards the final test set, Immune Inspired Algorithms allow tests to be introduced through two main behaviours: cloning and mutation of individual tests; or, introduction of novel tests (tests introduced at random, not via cloning).

Immune Algorithms use both techniques to search for good solutions to a problem - cloning and mutation acts as a local search; introducing novel tests acts as a global search. Both behaviours give indications as to how the algorithm performs. If the final test set solely contains novel tests then a similar functionality could be obtained through many iterations of random test generation. If, on the other hand, the test set contains cloned and mutated tests, the local search behaviour of the algorithm has an effect.

H 7 *The resultant test set achieved with the IIA/MT comprises of a mixture of cloned tests and randomly introduced tests.*

5.3 Qualifiers

The experiments to test the hypotheses make numerous comparisons between two algorithms, each of which contain a number of variable settings (see sections 4.5 and 4.6). To keep the comparisons fair, the settings applied to each algorithm should remain constant across the various hypotheses, as well as representing fair values (i.e. values not considered extreme so as to cause unnecessary bias in the behaviour of one algorithm over another). For the purpose of comparing the algorithms, the following variable settings and assumptions have been made.

Four programs have been tested (detailed in Appendix B): *DateRange* (**DR**); *TriangleSort* (**TRI**); *CalDay* (**CD**); and, *Select* (**SEL**). Each program was executed at least 30 times per algorithm (*DateRange*: IIA-30, GA-34; *TriangleSort*: IIA-34, GA-30; *CalDay*: IIA-33, GA-31; and *Select*: IIA-32, GA-31). The initial population of either algorithm contains 300 tests (300 1-test individuals for the IIA/MT, 15 20-test individuals for the GA/MT), each initialised to the same value (see Appendix B for the values used for each program). Test values were randomly chosen that produced a low starting mutation score, and these same values (for a given program) were used for each experiment. This method was chosen over the random generation of each test as it allows the effects of each algorithm to

be easily observed, with each algorithm starting from the same state - no bias is given to either algorithm from starting with a different population of tests.

In total, 500 iterations of each algorithm were performed. This number was chosen based on test run results (for 1000 iterations) which indicated that for both algorithms on all four programs, relatively little change occurred in the mutation score after this number of iterations. Based on these results it was deemed that 500 was a reasonable number of iterations to allow each algorithm to reach its plateau, as well as giving each a good chance to improve the mutation score. Obviously however, one more iteration may cause a test to be found to improve the mutation score further, prompting the more general question for testing and evolutionary algorithms alike: *what is a good stopping criteria?*

Each algorithm was executed with the following parameter settings. These values are not optimised for either algorithm, if indeed that is possible (see chapter 6 for an analysis of each parameter's effect which indicates that the PUT is an important factor in determining good parameter settings), but merely represent typical values that may be chosen. No work was undertaken to optimise the parameter values for each PUT. Chapter 6 presents an analysis of other parameter values, to show how each one affects the algorithm's effectiveness. The values chosen for that analysis however, are (rather ironically) limited to prevent excessive execution times.

IIA/MT		**GA/MT**	
Parameter	Value	Parameter	Value
s (initial size of population)	300	s (size of population)	15
nFittest	5	m (#tests per member)	20
nWorst	5	crossRate	0.8
cloneRate	10	mutRate	0.02

Table 5.6: Parameter values used for each algorithm.

5.3.1 Statistics

Because of the small sample sizes, comparisons between the mean values obtained for each algorithm are statistically compared using a T-test for non-independent samples. This calculates two values, T_{obt} and T_{crit}, which can be compared to determine if the differences in means are statistically significant at a given level ($T_{obt} \geq T_{crit}$ implies significance). All comparisons were made at the 0.05 significance level - i.e. there is a 5% chance that the difference between two means happened by chance. Readers interested in the exact details of these methods, and how the T_{obt} and T_{crit} values are calculated, should consult [51] for an introduction.

5.4 H1 - Primary Hypothesis

An Immune Inspired Algorithm is consistently, at least as effective as a Genetic Algorithm for evolving test data through mutation testing.

Being the primary hypothesis of this research, it is unfortunate that it cannot be resolved by a single experiment. To clarify this hypothesis however, this section provides a summary of the results obtained in the subsequent hypotheses. In the context of mutation testing, algorithm effectiveness can be broken into three main areas: improved mutation score in (possibly) less time; finding more hard-to-kill mutants; and, generating smaller test set sizes. Experiments covering these three areas are examined in hypotheses H3-H5. However, not only does the IIA/MT have to be at least as effective, but it also has to be consistent in its efforts too. This must occur on two levels: it must achieve similar results for every mutation testing run using the same program - i.e. the algorithm is capable of reproducing the same level of results; and, it must present similar effectiveness abilities for a variety of programs - i.e. the algorithm is at least as effective irrespective of which program is undergoing testing.

Hypothesis H2 suggests favourable results on both counts of consistency for IIA/MTs. Firstly, the mean average final mutation score achieved after 500 iterations is in excess of 92% for all four programs (table 5.7). Compared with the starting mutation scores, this presents increases of 25.26% to 67.67%, hinting that IIAs are capable of improving mutation scores regardless of the program being tested. Furthermore, the low standard deviations (between 0.05% and 1.84%) on the mean mutation scores achieved after 500 iterations suggest that IIA/MTs present a consistent approach to generating a high mutation scoring test set.

	MS after 1 iteration	MS after 50 iterations	MS after 500 iterations	t_{obt}	t_{crit}	H0
DR	84.85%±2.24%	97.63%±0.91%	99.78%±0.45%	6.26	2.05	Reject
TRI	30.48%±10.30%	95.24%±2.13%	98.15%±0.05%	6.58	2.03	Reject
CD	70.92%±0.34%	91.59%±1.55%	96.18%±1.84%	13.06	2.04	Reject
SEL	42.34%±9.94%	90.23%±1.50%	92.99%±0.85%	5.06	2.04	Reject

Table 5.7: Average mutation scores and standard deviations achieved for each program after 1, 50 and 500 iterations using an Immune Inspired Algorithm. *All results are to 2 decimal places.*

Hypotheses H3-H5 complement this consistency value with an effectiveness comparison between the IIA/MT and the GA/MT. H3 compares the two algorithms using the number of mutant program executions as a realistic (and more comparative) measure of algorithm execution time. For two of the four programs, the IIA/MT generates higher mutation scoring test sets than the GA/MT in significantly fewer mutant executions. For the other two programs the IIA/MT generates higher mutation scoring test sets, however it requires more mutant executions than the GA/MT. This inconclusive (with respect to H3) result is deceiving however. Consider the equally highest mutation score achieved by both algorithms in at least 25 runs, for these latter two programs - *CalDay* achieves an 84.63% score in an average of 27551 mutant executions for the IIA/MT compared to 2341483 for the GA/MT; *Select* achieves 88.97% in an average of 1733911 executions for the IIA/MT compared to 8920678 for the GA/MT. In both cases, the IIA/MT

is capable of achieving the same mutation score in significantly fewer executions than the GA/MT. At this score therefore, the IIA/MT is consistently more effective. Both algorithms are capable of increasing the mutation score beyond this point however, as indicated by table 5.8, only in detriment to their consistency. The IIA/MT is able to achieve higher mutation scores than the GA/MT, and with greater consistency. As fewer GA/MT experiments achieve the high mutation scoring test sets (compared to the IIA/MT), it is likely that the GA/MT will need to execute excessively more mutant programs in order to achieve the same mutation score as the IIA/MT, with the same consistency. Therefore these results can also be seen as favouring the IIA/MT approach, where the local and global search facilities offered by the IIA/MT seem to outperform the adaption techniques of the GA/MT. Furthermore, whilst it may be possible to generate a test set within a specific size that kills all mutants, the IIA/MT's ability to cope with variable sized sets allows for easier generation of tests.

	IIA/MT		GA/MT	
	MS	# executed	MS	# executed
DR	100%	1075779 ± 611909	96.67%	2181424 ± 1062018
TRI	98.16%	1120972 ± 782389	93.42%	3625432 ± 0*
CD	98.17%	4413829 ± 731962	86.47%	3439576 ± 176370
SEL	93.94%	24088117 ± 0*	91.69%	12438537 ± 0*

Table 5.8: Mean number of mutants executed for each algorithm achieving its highest mutation score. The number executed (and any standard deviation) has been rounded up to the nearest whole number.

H4 compares the two algorithms based on their abilities to identify hard-to-kill (HTK) mutants. Test sets can be generated to kill large proportions of mutants, but for the remaining few mutants it can be difficult to find tests capable of killing them. The results from this hypothesis indicate that an IIA/MT is consistently able to generate test sets that kill significantly more hard-to-kill mutants than

*A standard deviation of 0 is recorded for these results because only 1 run (out of 30+) achieved the respective mutation score.

GA/MT produced sets. For example, for the four programs tested, the IIA/MT kills between 75.00% and 97.33% of HTK mutants, whereas the GA/MT kills 27.22% to 71.54% (table 5.9). A likely explanation for this however, is due to the restriction on the number of tests in the final test set that the GA/MT makes - high mutation scoring tests are more likely to be incorporated than low scoring, HTK killing tests. This claim is analysed further in Chapter 6.

	Total	IIA/MT		GA/MT	
	HTK	# HTK	% HTK	# HTK	% HTK
DR	15	14.60 ± 0.81	97.33% ± 5.42%	10.65 ± 0.54	70.98% ± 3.63%
TRI	12	9.00 ± 0.00	75.00% ± 0.00%	3.27 ± 1.14	27.22% ± 9.52%
CD	32	26.12 ± 1.34	81.63% ± 4.19%	11.00 ± 0.00	34.38% ± 0.00%
SEL	365	314.50 ± 17.39	86.16% ± 4.77%	261.13 ± 15.87	71.54% ± 4.35%

Table 5.9: The mean average number of hard-to-kill (HTK) mutants killed by each algorithm, and percentage of the total number of HTK mutants, after 500 iterations. *All results are to 2 decimal places.*

Finally, H5 makes a comparison based on the size of the generated test sets. Fewer tests mean less testing needs to be done to achieve the same results as a larger, equivalent set. Results suggest that whilst there are significant differences in the sizes of the generated test sets (at equally highest mutation scores), it cannot be concluded that an IIA/MT always generates a smaller set - for 2 of the 4 programs, a larger test set is recorded using an IIA/MT (on average, *TriangleSort* requires 24.79 tests for the IIA/MT compared to 20 for the GA/MT, and *Select* needs 39.13 tests compared to 19.96). By allowing an IIA/MT to generate a larger test set when it requires however, does appear to allow it to reach a higher mutation score than the GA/MT. Encouraging larger test sets may therefore be advantageous.

5.4.1 Conclusion

Of the three measures of effectiveness, an IIA/MT is shown to be consistently more effective than a GA/MT in 2 areas: it is capable of generating higher mutation

scoring test sets in significantly fewer mutant executions (i.e. less time); and, it is able to consistently identify more hard-to-kill mutants. The third area - generating smaller test sets - is inconclusive. Whilst differences in test set sizes are significant, it is not always the IIA/MT which returns the smaller set. Primarily however, the flexibility in the IIA/MT's test set size does appear to be a strength rather than a weakness, allowing it to generate a wide variety of tests - some that only kill a single mutant, others which kill a substantial set.

Combining these results then, if one considers the primary aim of an effective algorithm as achieving at least the same mutation score in at least the same time, then an IIA/MT exceeds a GA/MT approach. From these results it appears an IIA/MT approach can generate test sets exceeding a GA/MT's in fewer mutant executions. Based on this viewpoint, these experiments present strong evidence to reject the null hypothesis, and conclude that an IIA/MT is at least as effective as a GA/MT.

5.5 H2 - Usefulness of Immune Inspired Algorithms

An Immune Inspired Algorithm for Mutation Testing (IIA/MT) is capable of improving the mutation score for a given program by automatically evolving the test data.

Null Hypothesis The IIA/MT does not cause a significant difference between the mean mutation scores achieved after 1 and 500 iterations, for any program.

It is initially worthwhile examining whether or not the IIA/MT can be used to improve test data for mutation testing. If not, then further experiments are worthless. Furthermore, the results from this experiment provide an indication as to the consistency of the algorithm over both multiple runs, and over differing programs.

An improvement in test data will be classified by a statistically significant difference between the mean average mutation scores at the start (after 1 iteration[2]) and end (after 500 iterations). Significance will be calculated by a T-test for non-independent samples at the 0.05 level. No other consideration - other than observational - will be given to the size of the difference (if any), or the rate of increase. Additionally, no comparison with a GA/MT will be made at this stage; previous work has already indicated the usefulness of GA's for generating mutation-adequate test data [7, 13, 46].

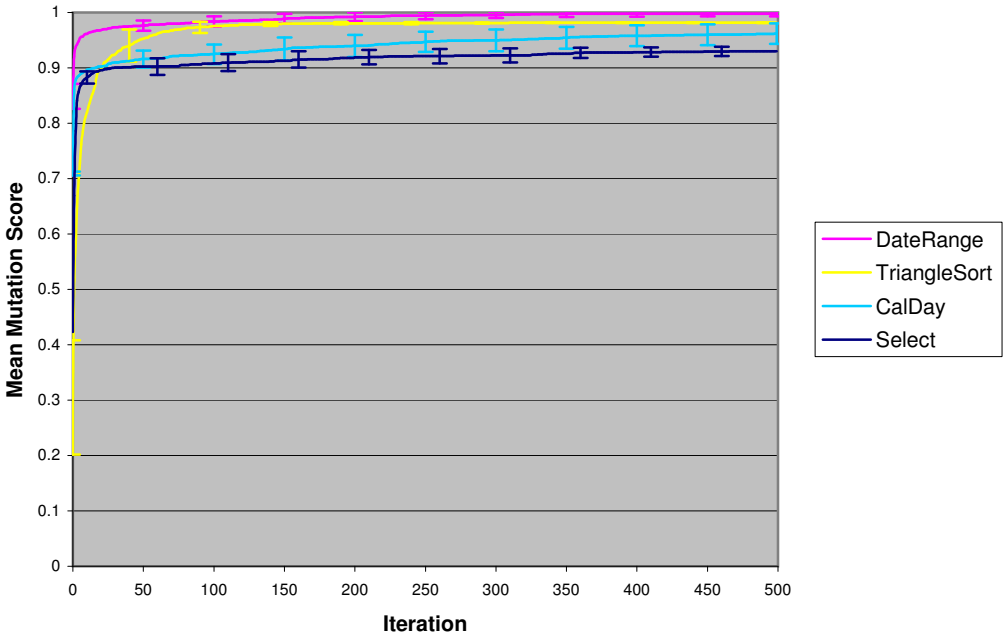

Figure 5.22: Mean mutation scores achieved at each iteration for all four programs using the Immune Inspired Algorithm for Mutation Testing. *Error bars are ± 1 s.d.*

[2]The starting mutation score is measured after 1 iteration (not after 0 iterations) to allow the IIA/MT to be compared equally with the GA/MT. Before any iterations have occurred, the IIA/MT would return an empty memory set, whereas the GA/MT would return its best individual; after 1 iteration, both algorithms would have had the opportunity to evolve their tests once (for better or worse) and the IIA/MT would have been able to add useful tests to its memory, thereby providing a comparable test set with the GA/MT's best individual.

Figure 5.22 presents the mutation scores achieved at each iteration of the immune algorithm for the four programs tested. All four programs demonstrate a similar shaped curve. In the initial 50 iterations, the mutation score's rate of increase is steep before decreasing sharply. Subsequent increases in mutation score per iteration are negligible compared to the start. For all four programs, the final mutation scores exceed 92%. These results suggest that the IIA/MT is capable of increasing the mutation score for any program, however they do not determine if the increase is significant or not.

Table 5.10 indicates the mutation score at the start, after 50 iterations and at the end, for each program, along with the non-independent T-test results to qualify the significance of any differences between 1 and 500 iterations.

	MS after 1 iteration	MS after 50 iterations	MS after 500 iterations	t_{obt}	t_{crit}	H0
DR	84.85%±2.24%	97.63%±0.91%	99.78%±0.45%	6.26	2.05	Reject
TRI	30.48%±10.30%	95.24%±2.13%	98.15%±0.05%	6.58	2.03	Reject
CD	70.92%±0.34%	91.59%±1.55%	96.18%±1.84%	13.06	2.04	Reject
SEL	42.34%±9.94%	90.23%±1.50%	92.99%±0.85%	5.06	2.04	Reject

Table 5.10: Average mutation scores and standard deviations achieved for each program after 1, 50 and 500 iterations using an Immune Inspired Algorithm. *All results are to 2 decimal places.*

A T-test rejects the null hypothesis if the calculated t_{obt} value is greater than or equal to the critical value of t, t_{crit}. As can be seen from table 5.10, the t_{obt} value for each program is greater than its respective t_{crit} value, meaning the null hypothesis (that there is no difference between start and end mean mutation scores for a program) is rejected in all four cases. Additionally, the low standard deviation ($< 2\%$) seen after 500 iterations, and even after 50 ($< 2.13\%$), suggest that IIA/MTs are capable of consistently reproducing the same results over multiple runs for the same program.

5.5.1 Conclusion

The T-test results reject the null hypothesis for all four programs at the 0.05 level - i.e. there is evidence to suggest a significant difference between the mean mutation scores achieved after 1 and 500 iterations, for a program. Figure 5.22 highlights that the mutation score increases over time (rather than decreases), and that this result is reasonably consistent for any program. Coupled with the variance data, this evidence suggests that the specified Immune Inspired Algorithm can reliably be used as an automatic way to significantly increase the mutation score for any programs test data. Further experiments are therefore worthwhile.

5.6 H3 - Number of Program Executions

Test data evolution using the IIA/MT requires fewer mutant program executions than the GA/MT to achieve at least the same mutation score.

Null Hypothesis There is no significant difference between the mean number of mutants executed to achieve at least the specified mutation score for each algorithm.

The most relevant measure of algorithm effectiveness is determined by the time taken to generate a test set and the mutation score achieved; a more effective algorithm will achieve the same mutation score in less time. In this case however, time is best measured by counting the number of mutant program executions.

A T-test comparison, at the 0.05 level, will be made for each of the four programs based on the number of executed mutant programs. The comparison point will be the equally highest mutation score achieved by both algorithms for at least 25 runs[3].

[3]Comparisons are made where at least 25 runs have achieved the result so that reliable mean averages (and T-test results) can be calculated. Obviously a higher mutation score may be possible using an algorithm, just fewer than 25 runs achieve it.

Figures 5.24-5.27 show the mean number of mutants executed to achieve at least a specific mutation score for each of the four programs (Some graphs display drops in mutation score; this is a result of a decrease in the number of runs achieving that mutation score affecting the overall mean). The results are similar for all four programs. During the initial stages the number of mutants executed remains low for both algorithms. As the mutation score increases, the number of mutants needing to be executed to improve the mutation score increases dramatically for the GA/MT before increasing for the IIA/MT. This suggests that in early iterations for both algorithms, weak mutants are killed by the introduction of new, weak tests - i.e. any test is capable of killing a mutant, resulting in a large increase in mutation score with few mutants executed. As the iterations progress however, the weaker mutants are removed leaving only the stronger ones. The tests needed to kill these are harder to generate meaning more mutants are executed before a test is found to improve the mutation score. The graphs show that an IIA/MT is able to generate more of these "harder" tests in fewer executions, leading to higher mutation scores for the same number of executions. Alternatively, an IIA/MT needs fewer executions to achieve the same mutation score.

The GA/MT execution of *CalDay* (Figure 5.26) is distinctive from the other programs. It initially follows the same pattern as the others - low mutation score increasing rapidly, with the number of executions increasing significantly as the mutation score does. However, at a mutation score of approximately 73%, the mutation score suddenly increases rapidly to approximately 82% in relatively few mutant executions. At this point, the mutation score increases at the "standard" rate of mutant executions. This behaviour is different from that exhibited by the other programs, and although the results coincide with a decrease in the number of runs achieving these (and higher) mutation scores (1 run less), it is unlikely this decrease caused the effect seen - if it was the reason, the effect would more likely have been an increase in mutation score but with a noticeable **decrease** in the number of mutants executed (the fewer the runs used to calculate the mean

averages, the larger the variability in the results), followed by further increases in mutation score and mutant executions, as seen in figures 5.25 and 5.27. A reasonable explanation would be that specific tests are needed to kill 73% of mutants. These tests may also kill further mutants, improving upon the mutation score with no extra executions. Furthermore, when these tests are mutated, any slight change may also be enough to kill additional mutants, and so the mutation score increases dramatically, with very little increase in mutant executions. This effect is not apparent for the IIA/MT in the graph shown in figure 5.26, however given the number of executions remains (relatively) low up to approximately 86%, this is unsurprising. A similar effect can be seen in the higher granularity graph of figure 5.23, supporting the notion that this effect is a result of this program's test data development, and not some curiosity from the GA/MT algorithm itself.

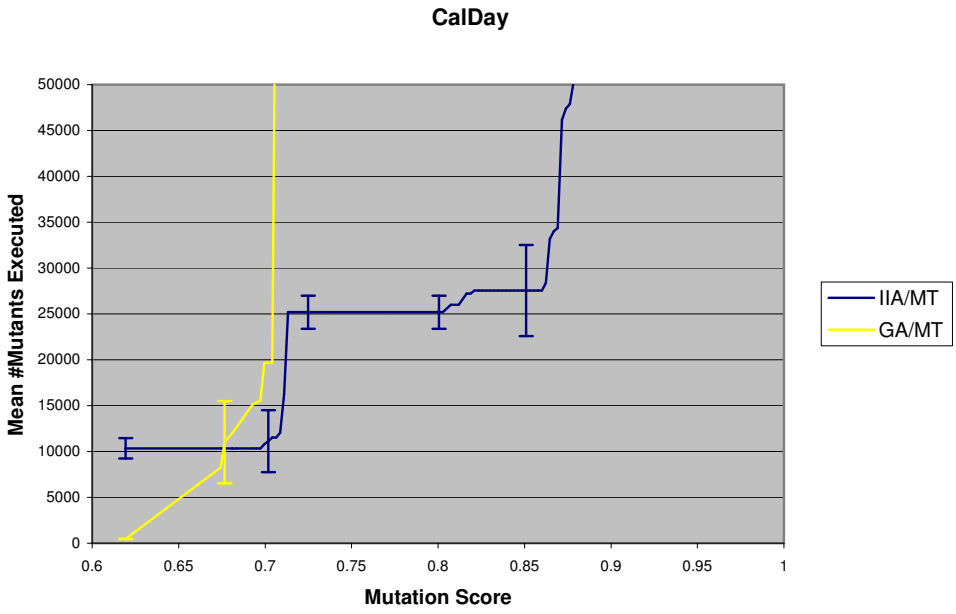

Figure 5.23: Zoom in of figure 5.26. Mean number of mutants executed (up to 50,000) to achieve at least a specific mutation score for the *CalDay* program. *Error bars are ± 1 s.d.*

Ultimately then, it would appear that the IIA/MT's method of generating tests outperforms (in terms of number of executions) the GA/MT's. Foremost, the GA/MT has a finite number of tests (in an individual) with which to generate

a full mutation score. If every one of the 20 tests[4] kills at least one distinct mutant, then all 20 tests are required. If however, the individual's mutation score is less than 100% then a further test is required. Unfortunately there is no room to add it into the individual, and so one of the existing tests must be improved - something which may not be possible. Put another way, let each test in the GA/MT's best individual kill a set K_i of mutants, where i is the index of the test in the individual. To improve the mutation score the GA/MT must mutate (at least) one of these tests so that it now kills a superset of K_i. This could be hard, especially if the mutants are hard to kill and so require very specific tests. The IIA/MT on the other hand, has no set size restriction, and is free to add additional tests that only kill a single mutant. This advantage is complemented by the IIA/MT's local and global search facilities. By cloning and mutating (see section 3.3.3) tests proportional to their mutation scores, a local search effect is produced, looking for similar tests that kill other living mutants - this could be considered as similar to searching around input boundaries. On top of this, the IIA/MT randomly introduces a number of tests each iteration to perform a global search.

T-tests at the 0.05 level, to check the significance of the difference in mean number of mutants executed, are shown in table 5.11. They reject the null hypothesis for all four programs. These tests were performed at the highest mutation score achieved by at least 25 runs of both algorithms. T-tests were also performed at every mutation score achieved. The null hypothesis was rejected at every mutation score for both the *TriangleSort* and *DateRange*. For the *CalDay* program, the null hypothesis was accepted only at mutation scores of 67.66% and 68.12% (2 d.p.) - two scores achieved in the early iterations of the algorithm. Taking the majority result then, this program can also be considered to reject the null hypothesis at every mutation score achieved. The *Select* program indicates less significant results - the number of mutants executed was not significant (at the 0.05 level) for mutation scores between 35.91%-36.74%, 89.36%-89.53%, and 89.96%-91.65%

[4]For these experiments a GA individual contains 20 tests.

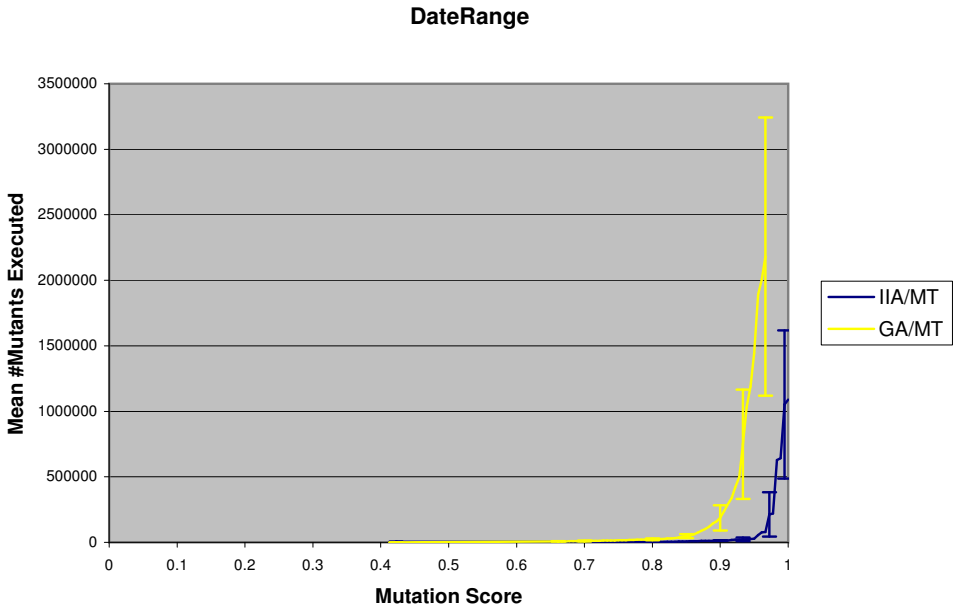

Figure 5.24: Mean number of mutants executed to achieve at least a specific mutation score for the *DateRange* program. *Error bars are ± 1 s.d.*

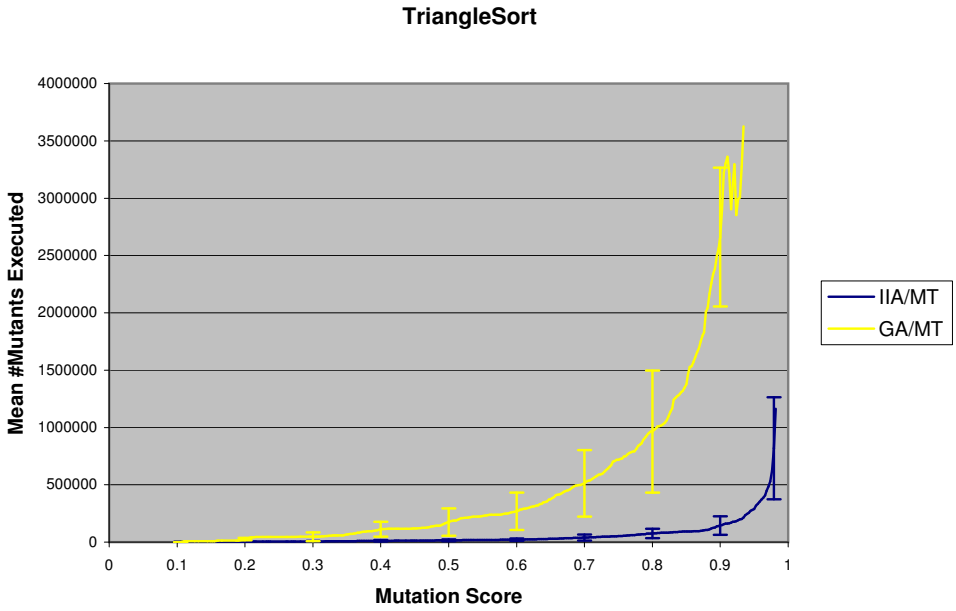

Figure 5.25: Mean number of mutants executed to achieve at least a specific mutation score for the *TriangleSort* program. *Error bars are ± 1 s.d.*

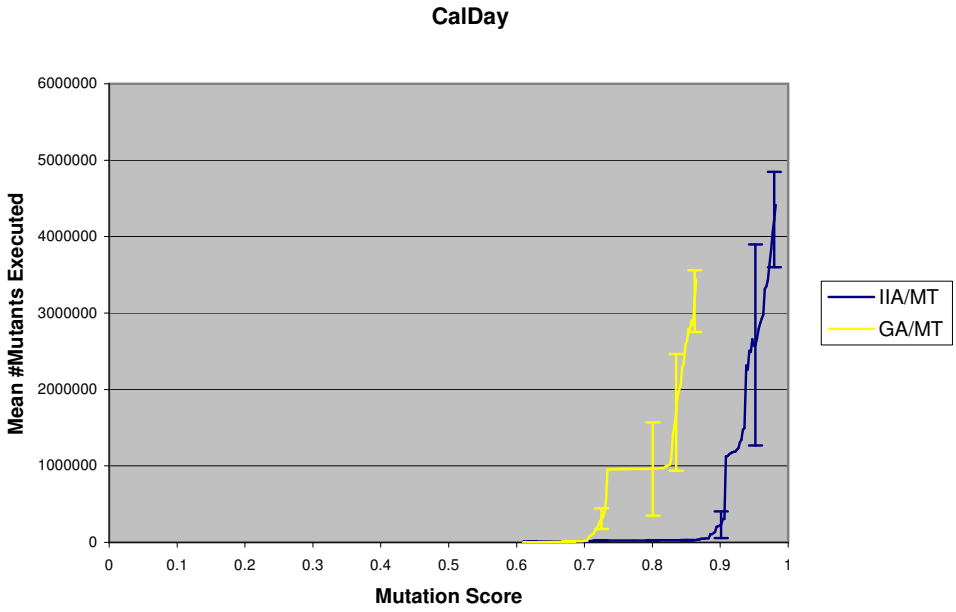

Figure 5.26: Mean number of mutants executed to achieve at least a specific mutation score for the *CalDay* program. *Error bars are ± 1 s.d.*

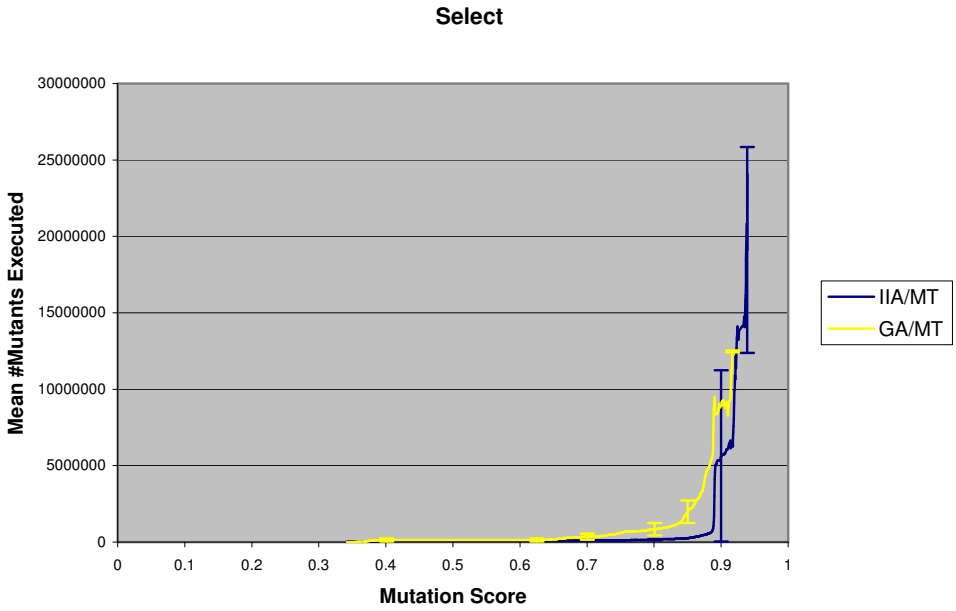

Figure 5.27: Mean number of mutants executed to achieve at least a specific mutation score for the *Select* program. *Error bars are ± 1 s.d.*

(2 d.p.). Despite this, the IIA/MT still achieved these mutation scores in fewer mutant executions and, possibly more importantly, with consistency - 31 runs out of 32 achieved the 91.65% mutation score for the IIA/MT, compared to only 2 runs out of 31 for the GA/MT. Assuming the GA is capable of generating a full mutation score (given its individual size), this drop in consistency can be attributed to the experiments being stopped after 500 iterations. Given additional iterations, more GA/MT runs would be able to succeed at reaching those higher mutation scores, but at a cost of executing more mutant programs. As a result, the difference in mean numbers of mutant executions between the two algorithms would probably be considered as significant, however the IIA/MT would require fewer. On this basis, the results for the *Select* program could also be viewed as supportive of the hypothesis.

	MS	IIA/MT mean	GA/MT mean	t_{obt}	t_{crit}
DR	94.44%	24372 ± 12991	1188082 ± 706193	9.02	2.00
TRI	88.95%	125802 ± 74347	2339044 ± 741550	17.34	2.00
CD	84.63%	27551 ± 4953	2341483 ± 709736	18.78	2.00
SEL	88.97%	1733911 ± 1664843	8920678 ± 2767622	12.16	2.00

Table 5.11: T-test (0.05 level) results for the significance between the mean number of mutants executed for each algorithm, at the highest mutation score obtained by both algorithms in at least 25 runs. The mean number of mutants executed (and the standard deviation) is rounded up to the nearest integer to reflect that a mutant is either executed completely or not at all. *T-test results are to 2 decimal places.*

The results in table 5.11 clearly highlight that to consistently (for more than 25 runs) achieve the same mutation score using both algorithms requires a significantly larger number of mutant executions using the GA/MT than by using the IIA/MT. As the number of executions is proportional to time, the choice of algorithm is therefore crucial to the speed of test generation. Furthermore, if the final mutation score after 500 iterations is considered, the IIA/MT consistently achieves a higher score than the GA/MT, however only in fewer mutant executions for the *DateRange* and *TriangleSort* programs (see table 5.12). Specifically

however, across any of the four programs, at least 30 of the IIA/MT's runs achieve a mutation score greater than the highest score achieved by the GA/MT, again suggesting that the IIA/MT has preferable results based on consistency.

	IIA/MT		GA/MT	
	MS	# executed	MS	# executed
DR	100%	1075779 ± 611909	96.67%	2181424 ± 1062018
TRI	98.16%	1120972 ± 782389	93.42%	3625432 ± 0[§]
CD	98.17%	4413829 ± 731962	86.47%	3439576 ± 176370
SEL	93.94%	24088117 ± 0[§]	91.69%	12438537 ± 0[§]

Table 5.12: Mean number of mutants executed for each algorithm achieving its highest mutation score (i.e. results after 500 iterations). The number executed (and any standard deviation) has been rounded up to the nearest whole number.

Table 5.12 shows the mean number of mutants executed by each algorithm achieving its highest mutation score. T-test comparisons have not been performed on these results because in most cases few runs achieved the highest mutation score, thereby reducing the statistical significance of any calculation.

5.6.1 Conclusion

The number of mutants executed is an important measure as it can accurately be used to compare execution times across various algorithms. Given each algorithm performs differing operations per iteration, counting the number of iterations to achieve a mutation score is not a consistent measure across algorithms. Each algorithm must execute mutants to calculate the mutation score. As long as this done consistently by each algorithm (e.g. only living mutants are executed), then the number of mutants executed reflects the time the algorithm takes.

The T-test results in table 5.11 reject the null hypothesis for the four programs tested, within 500 iterations - i.e. there is evidence to suggest a significant difference in the mean number of mutants executed to achieve at least a specific

[§]A standard deviation of 0 is recorded for these results because only 1 run (out of 30+) achieved the respective mutation score.

mutation score. As can be seen in figures 5.24-5.27, the Immune Inspired Algorithm approach requires fewer executions to achieve at least the same mutation score - a result best highlighted in table 5.11. Combined, this evidence suggests that using the IIA/MT is favourable to the GA/MT, particularly for higher mutation scores, when the difference in execution numbers is even more significant, and certainly in terms of consistency. For lower mutation scores however, the difference is not always significant, as indicated by the *Select* program. Despite this, the IIA/MT still executes fewer mutants, continuing the support towards favouring this algorithm.

5.7 H4 - Number of Hard-To-Kill Mutants Found

The final test set generated by the IIA/MT consistently identifies more hard-to-kill mutants than one created by the GA/MT.

Null Hypothesis There is no significant difference between the number of hard-to-kill mutants identified by the final test sets generated by both algorithms.

Algorithms often generate high mutation scoring test sets ($> 90\%$), but it is the remaining few mutants that are usually difficult to kill. An algorithm which is consistently able to kill a high percentage of these hard-to-kill (HTK) mutants is more effective than an algorithm which cannot.

Hard-to-kill mutants have been manually identified at the same time as mutants were checked for equivalence. As such, whether a mutant is HTK or not is dependent on the tester's definition of hard-to-kill and their enforcement of this rule. For the purposes of this research, a mutant is, in general, defined as hard-to-kill according to the following heuristics:

- a mutant is HTK if it places a restriction on the program's input domain that restrains one or more inputs to a particular value;

- a mutant is HTK if it results in a complex relationship between the inputs that restrict the tests that kill the mutant;
- a mutant is HTK if a test which kills it is difficult to ascertain from static examination of the code, and is instead generated during the course of the program execution[5].

For example, consider the third ABS mutation of the *TriangleSort* program which adjusts line 4:

from: `if((i<=0) || (j<=0) || (k<=0)){`
to: `if(ZP(i)<=0) || (j<=0) || (k<=0)){`

Tests that kill this mutant require the input `i` to be exactly 0, which according to the first heuristic, implies the mutant is HTK.

An example of the second heuristic is another ABS mutation of the *TriangleSort* program, which mutates line 22:

from: `if((i+j<=k) || (j+k<=i) || (i+k<=j)){`
to: `if(ABS(i+j)<=k) || (j+k<=i) || (i+k<=j)){`

For a test to kill this mutant, the original `i+j` must produce a negative number smaller than `k`, with (for the mutant) the absolute value of this negative number being greater than `k`. According to the conditions in line 4 however, `i`, `j`, and `k` must all be positive. To satisfy these conditions requires knowledge that the addition of two large integers in Java will cause an integer overflow, resulting in a negative number, i.e. `i` and `j` need to be large positive numbers, such as 1453635131 and 1722147072 respectively. To get a differing output between the original and the mutant would then simply require setting the value of `k` appropriately, e.g. 414811834.

As a final example, the third heuristic is demonstrated by an ABS mutation of line 138 of the *Select* program:

[5]This heuristic is employed for the *Select* program, where the complexity of the program means statically identifying tests to kill mutants is difficult.

from: `a[l] = a[l-inc];`

to: `a[l] = ABS(a[l-inc]);`

The test that identifies this mutation is $k = 4$ and $arr = [-5.7110815, -14.880631, 41.726444, -37.05031, 23.39116, -22.205894]$. The complex nature of the *Select* program means statically identifying tests to kill mutants is not always trivial, with, in some cases, tests being found during execution of the program instead. This generates a concern with determining the equivalence of mutants, as identified by Acree [1] - non-equivalents are sometimes marked as equivalent, which are difficult to correct. To combat this, mutants that cannot be determined to be equivalent are left as non-equivalent (as these can be corrected in later iterations). This may result in a reduced overall mutation score, but is favourable when compared to indicating a higher mutation score than has actually been achieved. Improving equivalence detection would help solve this problem (which, it should be noted, is a concern for all mutation testing research).

Similar to equivalence, if there is any doubt to the nature of the mutant, it is **not** marked as hard-to-kill. The results that follow are therefore minimal figures; there could be further HTK mutants that have been ignored.

Table 5.13 shows the mean average results obtained after 500 iterations of each algorithm executing each program. For each algorithm, the mean number of HTK mutants killed is given, along with this figure as a proportion of the total number of HTK mutants for that program.

	Total	IIA/MT		GA/MT	
	HTK	# HTK	% HTK	# HTK	% HTK
DR	15	14.60 ± 0.81	97.33% ± 5.42%	10.65 ± 0.54	70.98% ± 3.63%
TRI	12	9.00 ± 0.00	75.00% ± 0.00%	3.27 ± 1.14	27.22% ± 9.52%
CD	32	26.12 ± 1.34	81.63% ± 4.19%	11.00 ± 0.00	34.38% ± 0.00%
SEL	365	314.50 ± 17.39	86.16% ± 4.77%	261.13 ± 15.87	71.54% ± 4.35%

Table 5.13: The mean average number of hard-to-kill (HTK) mutants killed by each algorithm, and percentage of the total number of HTK mutants, after 500 iterations. *All results are to 2 decimal places.*

Observationally, the IIA/MT generates test sets killing a large percentage of HTK mutants on all four programs - between 75.00% and 97.33% - with reasonably consistency - standard deviation is at most 5.42%. The results for the GA/MT are similarly consistent - standard deviation at most 9.52% - although not as high - the percentage of HTK mutants killed is between 27.22% and 71.54%.

A T-test for independent samples (0.05 level) on these results will highlight the significance of any differences in the number of HTK mutants killed by each algorithm (the null hypothesis - there is no significant difference - is rejected if t_{obt} exceeds t_{crit}). These results are shown in table 5.14.

	t_{obt}	t_{crit}	H0
DR	23.09	2.00	Reject
TRI	29.29	2.00	Reject
CD	62.77	2.00	Reject
SEL	12.71	2.00	Reject

Table 5.14: T-test results (0.05 level) for the difference in the mean average percentage of HTK mutants killed by each algorithm after 500 iterations. *All results are to 2 decimal places.*

These results clearly imply that the IIA/MT is capable of killing significantly more HTK mutants than the GA/MT; table 5.14 indicates that the difference in number of HTK mutants killed is significant, with table 5.13 showing that it is the IIA/MT that kills the most in all cases. A possible explanation for this lies in the different representations of an individual used by each algorithm. For a GA/MT, an individual represents a finite number of tests that will hopefully kill all mutants (in this case, 20 tests). These individuals evolve over many iterations, reaching a reasonably high mutation score by killing weaker mutants using the majority of the tests in each individual; each test is therefore likely to kill a high percentage of mutants. At this point, a number of HTK mutants remain living. To kill these requires niche tests that are not only difficult to generate, but may only kill a few mutants anyway. If a high-percentage killing test in an individual is evolved to one of these specific tests, despite killing a HTK mutant, it is likely to

reduce the overall mutation score for that individual. If this occurs, this individual will probably die out when compared against its fitter siblings. In contrast, the IIA/MT does not restrict the number of tests that can enter its memory set, nor restrain them based on ability. A test only killing a single HTK mutant can still form part of the solution memory set, thereby allowing it to make a valid contribution without any detriment to the overall mutation score of the final test set. The validity of this explanation can be tested by varying the number of tests in a GA/MT individual - this is tested in section 6.3.1.

5.7.1 Conclusion

The results in tables 5.13 and 5.14 provide evidence to support rejecting the null hypothesis - i.e. using the IIA/MT kills a statistically significant larger proportion of hard-to-kill mutants than the GA/MT, after 500 iterations. A possible reason for this is because the GA/MT restricts the number of tests forming the final test set. The IIA/MT does not have this restriction, enabling it to kill more HTK mutants and ultimately achieve a higher mutation score. This explanation is explored in Chapter 6.

5.8 H5 - Test Set Size

Test data evolution using the IIA/MT results in a smaller average test set size than from using the GA/MT.

Null Hypothesis There is no significant difference in the average test set sizes generated by both algorithms achieving at least the same mutation score.

A smaller test set is functionally easier (i.e. requires less time to test with) than a larger, equivalent set. On this basis, an algorithm that returns smaller sets can be considered more effective than one which returns larger equivalent sets.

A T-test comparison, at the 0.05 level, will be used to determine whether the

difference in average test set sizes is significant or not for the highest mutation score equally achieved by at least 25 runs of both algorithms. The results of this experiment are shown in table 5.15. As an aside, due to the structure of the Genetic Algorithm's individual, some tests in the returned individual (for the GA/MT) may be duplicated because they have not evolved. Because of this, the test set size for the GA/MT is the number of *distinct* tests returned to the tester.

	MS	IIA/MT # tests	GA/MT # tests	T_{obt}	T_{crit}	H0
DR	94.44%	14.40 ± 1.33	20 ± 0.00	23.47	2.00	Reject
TRI	88.95%	24.79 ± 2.27	20 ± 0.00	10.55	2.00	Reject
CD	84.63%	9.21 ± 0.96	20 ± 0.00	56.04	2.00	Reject
SEL	88.97%	39.13 ± 4.31	19.96 ± 0.20	22.18	2.00	Reject

Table 5.15: T-test results (0.05 level) for the significance between the average test set sizes generated by each algorithm, at the highest mutation score achieved by both algorithms in at least 25 runs. *All results are to 2 decimal places.*

The results from this experiment are inconclusive. Whilst the difference in average test set size for each program is significant at the 0.05 level for all four programs, it was not always the IIA/MT that produces the fewest tests. For the *Select* program in particular, the IIA/MT generates nearly double the number of tests than the GA/MT; for the *TriangleSort* program the IIA/MT generates approximately 5 more tests.

It is worth noting that the GA/MT imposes a restriction on how many tests it can generate. Each individual within the GA/MT population is a fixed number of tests (in this case, 20). The individual with the highest mutation score after 500 iterations is the final test set, and so at most then, this final test set can have 20 distinctive tests, with each one being useful (i.e. killing at least one distinct mutant). Restricting the number of tests in this manner could be, at least in part, responsible for the lower mutation scores achieved by the GA/MT, implying more than 20 tests are required to kill all mutants. If this is true, the IIA/MT has an advantage with its variable size memory set. *DateRange* and *CalDay* highlight

the exception to this train of thought though - both programs show that it is possible to achieve a higher mutation score using fewer than 20 tests. Restricting the number of tests therefore, does not always restrict the achievable mutation score.

As an alternative, the poor GA/MT performance for these two programs could be explained by how the number of tests is calculated. Not all distinct tests in the individual may kill mutants not killed by other tests. These are therefore surplus to requirement and yet they are still counted because they are distinct. In other words, the GA/MT's test set size is greater than it should be. This however, may also be true for the IIA/MT. Consider a memory set with only two tests in it. For these to be in the memory set, each test must kill something the other test does not. To add a third test to this memory set, it must also kill a mutant not already killed; it could however, do this by killing a superset of mutants killed by the current two tests, making the original two tests redundant - the true set size is 1 and not 3. Given that both approaches possess a similar problem then, making comparisons on the results in table 5.15 seems fair.

Table 5.16 shows the test set sizes returned for the highest mutation scores achieved by each program for each algorithm. The results for all four programs provide additional support to the notion that restricting the number of tests restricts the achievable mutation score (albeit with the previously mentioned measuring problems in mind). In three of the four cases, the IIA/MT generates a higher final mutation score by generating a test set greater than or equal to the GA/MT's test set size. For the fourth, the *DateRange* program, the average test set size is only just smaller than the GA/MT's. To improve its mutation score therefore, it is likely that the GA/MT will need to increase the size of its individuals - something which is difficult to judge. A tester would need to know, *a priori* to testing, the number of tests required to achieve a full mutation score in order to set an appropriate size for the GA/MT's individuals. In contrast, the IIA/MT algorithm does not have this restriction, and is able to return test sets capable of achieving higher mutation scores. Appropriate GA/MT individual sizes are

investigated in chapter 6.

	IIA/MT		GA/MT	
	Mutation Score	# Tests	Mutation Score	# Tests
DR	100%	19.38 ± 1.69	96.67%	20.00 ± 0.00
TRI	98.16%	35.33 ± 2.48	93.42%	20.00 ± 0.00¶
CD	98.17%	27.25 ± 1.50	86.47%	20.00 ± 0.00
SEL	93.94%	60.00 ± 0.00¶	91.70%	20.00 ± 0.00¶

Table 5.16: Final test set sizes generated for the highest mutation score achieved by each program for each algorithm. *All results are to 2 decimal places.*

5.8.1 Conclusion

The results presented in table 5.15 reject the null hypothesis for all 4 programs, suggesting there is a significant difference in the average test set size generated by each algorithm. However, it cannot be concluded that an IIA/MT algorithm generates a smaller test set than a GA/MT; only that an IIA/MT does not have its achievable mutation score restricted by the number of tests it can generate. Furthermore, it is noted that it is possible for both algorithms to suffer from returning larger test sets than necessary. Whilst this has implications from a usability point of view, the detriment to the tester is outweighed by the higher mutation scores achieved for the IIA/MT, and limited by the size of the individual for the GA/MT.

5.9 H6 - Mutation Score per Iteration

Test data evolution using the IIA/MT achieves a higher mutation score by a specific number of iterations than by using the GA/MT.

¶A standard deviation of 0 is recorded for these results because only 1 run (out of 30+) achieved the respective mutation score.

Null Hypothesis There is no significant difference in the average mutation scores achieved by either algorithm after a specific number of iterations.

Whilst not an entirely accurate measure of time, it may be useful to consider how the mutation score varies with each iteration. T-test comparisons, at the 0.05 level, will be made at every iteration to check the significance of any mutation score differences. For simplicity however, only the results after 500 iterations will be presented - see table 5.17. Graphical results of the mutation scores achieved at every iteration for the four programs are presented in figures 5.28, 5.29, 5.30 and 5.31.

The four graphs show similar results for each of the programs. For both the GA/MT and the IIA/MT, the rate of increase for the mutation score is initially high, before decreasing towards zero. Within roughly the first 30 iterations, this rate of increase is larger for the IIA/MT, indicated by the IIA/MT quickly reaching higher mutation scores. However, the IIA/MT's rate of increase also tends to drop quicker than the GA/MT's, prompting a sharper bend in the graph. After this, both algorithms start to reach similar rates of increase, with the IIA/MT having attained a higher mutation score.

What explanation can be given for the IIA/MT's ability to quickly increase the mutation score? From these results it appears that during the first few iterations the IIA/MT generates tests which kill more mutants per iteration than the GA/MT. Consider the two algorithms presented in figures 4.21 and 4.19 (Chapter 4). Every iteration the GA/MT modifies each individual with fixed probabilities, therefore only a certain proportion of new tests will be created which only vary by a fixed amount. The IIA/MT on the other hand, varies the number of new child tests created and the amount they are modified based on each parent test's ability. In the early iterations, there are a likely to be a lot of easy-to-kill mutants, giving some parents relatively high mutation scores. The higher the scores, the more new tests created (through cloning and mutation), and the higher the probability that these will be able to kill the remaining tests. Conjointly, parents

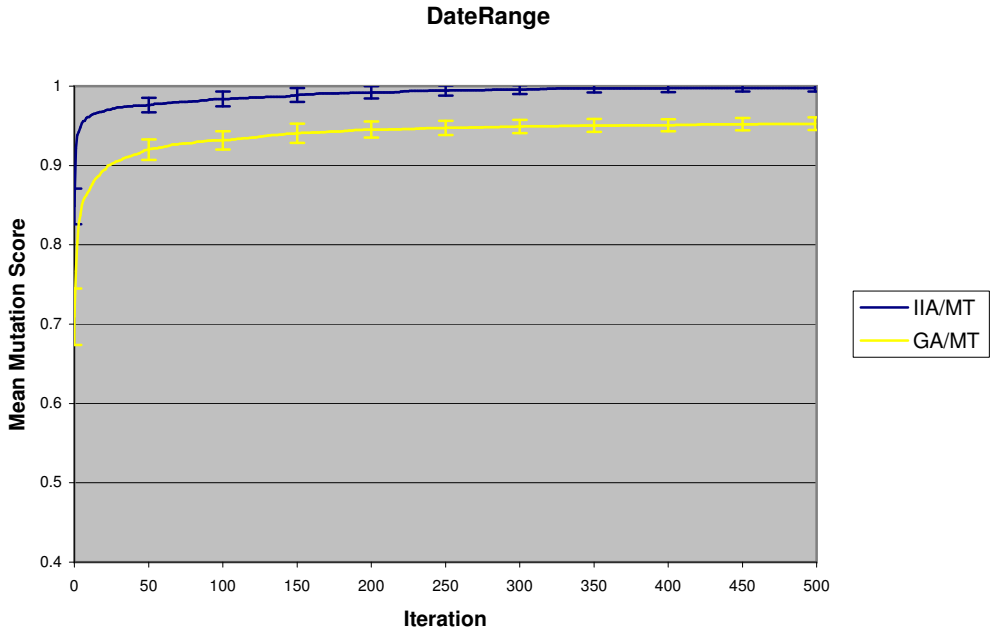

Figure 5.28: Mean mutation score per iteration for the *DateRange* program. *Error bars are ± 1 s.d.*

Figure 5.29: Mean mutation score per iteration for the *TriangleSort* program. *Error bars are ± 1 s.d.*

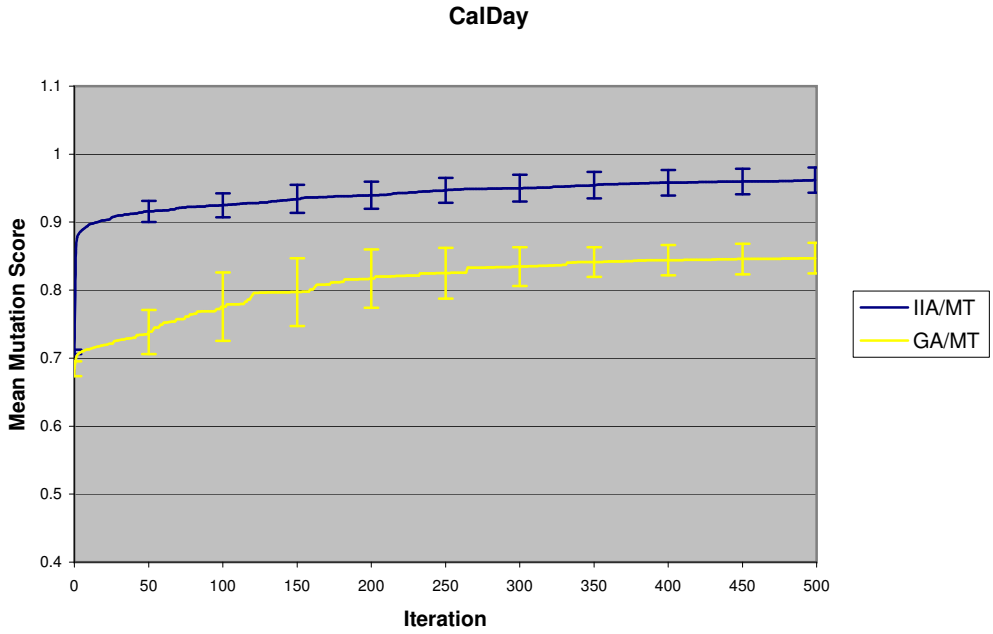

Figure 5.30: Mean mutation score per iteration for the *CalDay* program. *Error bars are ± 1 s.d.*

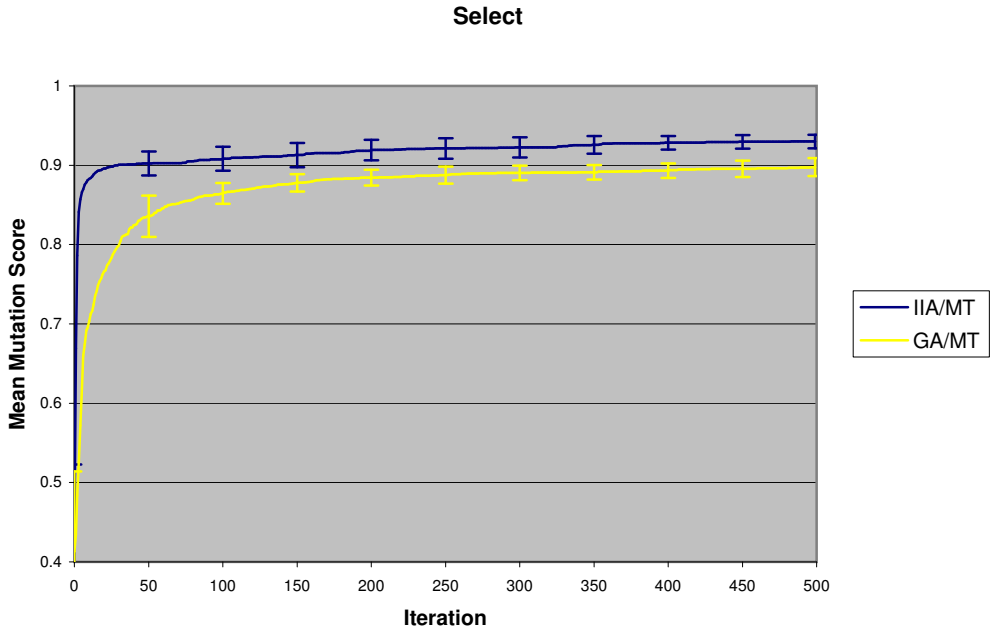

Figure 5.31: Mean mutation score per iteration for *Select* program. *Error bars are ± 1 s.d.*

with lower mutation scores will undergo a higher amount of mutation, effectively producing a wider search of the shape-space. Furthermore, the IIA/MT also adds extra diversity to its population by incorporating random tests. These too will widen the shape-space search, increasing the probability of identifying new tests.

	IIA/MT MS	GA/MT MS	T_{obt}	T_{crit}	H0
DR	99.78% ± 0.45%	95.26% ± 0.80%	27.29	2.00	Reject
TRI	98.15% ± 0.05%	90.05% ± 1.40%	33.82	2.00	Reject
CD	96.18% ± 1.84%	84.70% ± 2.24%	22.41	2.00	Reject
SEL	92.99% ± 0.85%	89.76% ± 1.12%	12.91	2.00	Reject

Table 5.17: T-test results for the significance between the average mutation scores for each algorithm after 500 iterations. *Plus/minus figures are 1 standard deviation. All results are to 2 decimal places.*

The results after 500 iterations are shown in table 5.17. As can be seen, the mutation score achieved for each program by the IIA/MT was greater than that reached by the GA/MT. Despite this, the GA/MT did achieve good mutation scores - greater than 84% for all programs, with *DateRange* achieving more than 95%. In addition, the low standard deviations of both algorithms suggest the test sets they generate are consistent over many runs. The T-tests (table 5.17) between these means show the differences to be significant at the 0.05 level. Furthermore, the T-tests performed after every iteration show that the difference in mutation scores achieved by each algorithm at any iteration is also significant. This evidence suggests a rejection of the null hypothesis; there is a significant difference in the mutation scores achieved by each algorithm after any number of iterations.

5.9.1 Conclusion

The results from the T-tests for every iteration reject the null hypothesis in every case; ultimately there is a significant difference between the mutation scores achieved by both algorithms for the first 500 iterations. Figures 5.28-5.31 also provide additional graphical evidence to support this hypothesis, showing higher

mutation score results for the IIA/MT in all four programs. A plausible explanation for the higher results achieved by the IIA/MT is due to the higher rates of new test adoption (either parent test modification or random test introduction) by the IIA/MT algorithm, and by its targeted localised search (mutating tests inversely proportional to mutation score as opposed to the GA/MT's static probability). These two effects encourage the generation of new tests to kill living mutants. In particular, the local search generates tests relative to existing useful tests, insinuating both border conditions on inputs and the relation between input values are important to test data.

5.10 H7 - Number of Cloned and Random Tests

The resultant test set achieved with the IIA/MT comprises of a mixture of cloned tests and randomly introduced tests.

Null Hypothesis The resultant test set achieved with the IIA/MT comprises solely of randomly introduced tests.

Immune Inspired Algorithms allow for new tests to be added to the population through two methods: cloning (and mutation), and random generation. Cloning and mutation perform a local search around good tests; random generation performs a global search of the search-space. Both techniques are useful for evolving a population, therefore if the resultant test set only contains randomly introduced tests, the local search is not having an impact and a similar functionality could be obtained through many iterations using a random test generator. If, on the other hand, the resultant test set only contains cloned tests, the global search is not having an impact, and the tests required to kill all mutants are confined to a small partition of the input space. This latter case is more unlikely however.

Table 5.18 shows the mean average number of cloned and randomly generated tests in the memory set after 500 iterations. As can be seen, all four programs are comprised of a mixture of test types, implying that a local search adds value to

	Mean # Clones	Mean # Random
DR	11.30 ± 1.76	7.90 ± 1.47
TRI	22.03 ± 3.33	13.21 ± 2.14
CD	16.36 ± 4.48	7.36 ± 1.17
SEL	42.94 ± 4.38	9.47 ± 2.11

Table 5.18: The mean average number of tests produced by cloning (and mutating) existing tests or by random generation after 500 iterations

just pure random generation of tests. It is difficult to conclude from these results what impact the program itself has (if any) on the numbers of each test type generated.

5.10.1 Conclusion

Results in table 5.18 indicate that the local search function encapsulated by cloning and mutating tests is advantageous to the evolution of test data in this application. Furthermore, the global search produced by random generation complements this local search by adding diversity to the solution set.

5.11 Summary

The primary aim of this thesis is to test whether the designed Immune Inspired Algorithm is consistently at least as effective as the Genetic Algorithm for evolving test data through mutation testing. Effectiveness was determined to relate to: an improved mutation score in (possibly) less time; finding tests for hard-to-kill mutants; and, generating a smaller sized test set. Consistency was measured in two ways: repeatable results over multiple runs of the same experiment; and similar results through using different programs.

Experiments were performed to compare the effectiveness and consistency of using the new Immune Inspired Algorithm approach against an already tried Genetic Algorithm technique. The evidence from these investigations suggest that an IIA/MT approach is capable of generating higher mutation scoring test sets

in significantly fewer mutant executions (i.e. less time). One problem with mutation testing, which partly affects its uptake by industry, is its lengthy execution time. The results from these experiments are therefore important, indicating an improvement over existing automatic techniques (e.g. using a GA/MT). Furthermore, these results were consistent over multiple runs and for all four programs - the IIA/MT took fewer executions to achieve at least the same highest mutation score across (at least) 25 runs of both algorithms.

Algorithms can often generate test sets capable of killing a large proportion of mutants ($> 90\%$), especially considering the initial test set can easily kill between 50-70% of mutants ([6] and the first iteration's results in table 5.10). It is however, the last few remaining mutants that are often hardest to kill, usually requiring very specific tests to do so. The percentage of these hard-to-kill mutants consistently killed by an algorithm is a useful metric for determining effectiveness. These experiments suggest that an IIA/MT is able to generate test sets that kill more HTK mutants than a GA/MT (75.00-97.33% compared with 27.22-71.54%). The most plausible explanation for the GA/MT's inadequacy comes from the limitation posed on the size of final test set (in this case 20), and highlights a further advantage of using the more dynamic IIA/MT (this will be examined in the next chapter). In terms of consistency, similar results were seen for all four programs (larger number of HTK killed for the IIA/MT) and for all runs - a low standard deviation from the mean for all runs (0.00-5.42% for IIA/MT, and 0.00-9.52% for GA/MT).

Finally, algorithm effectiveness can be measured on the size of the final test set. A small test set is more manageable than a larger, equivalent[6] one. From these experiments however, it cannot be concluded that an IIA/MT will always generate a smaller test set - a larger final set size is generated for 3 out of the 4 programs (table 5.16). Despite this, there is evidence to suggest that the lack of restrictions on the test set size, as also indicated by the HTK experiments, is advantageous to the IIA/MT in allowing test sets with higher mutation scores.

[6] An equivalent test set generates the same mutation score as the comparison set.

This will be researched in the next chapter.

If the three metrics for effectiveness are combined and a majority weighting given to improving the mutation score in less time, then the IIA/MT approach can be viewed as consistently more effective than the GA/MT. The outcomes from these experiments however, whilst suggestive for a larger domain, are only applicable to the four programs tested. Furthermore, they are currently only applicable to the variable settings chosen in table 5.6. The following chapter examines how a range of different variable settings affects the results for each algorithm.

Chapter 6

Parameter Analysis

6.1 Introduction

The IIA/MT and GA/MT algorithms, detailed in figures 4.21 and 4.19 respectively (Chapter 4), define a number of parameters which may affect each algorithm's performance. The previous chapter compared the two algorithms based on specific, consistent values for each parameter, and concluded that the IIA/MT approach is consistently more effective than the GA/MT. However, the values chosen were part of a much larger input space; this needs exploring to allow practitioners to choose optimal values for their experiments, as well as to validate the previous chapter's results for other parameter value combinations.

6.2 Methodology

A single experimental run is a single execution of the mutation testing system against a specific program, with a particular algorithm and parameter settings. Whilst a single experiment's results are important however, they are not statistically significant. Instead, the average of multiple runs should be used. For this research, at least 30 runs of each experiment were performed, where each experiment lasted for 500 iterations and was initially populated with 300 same-valued tests - this forced both algorithms to start with the same poor test set. However,

because of the number of repeats and the long duration of the *Select* program (in excess of 16 hours per experiment, depending on the parameter values), only three programs were used to analyse the effect of each parameter: *CalDay*, *DateRange* and *TriangleSort*.

Chapter 5 compared the algorithms based on their effectiveness. This was measured in three ways: time taken to evolve a high mutation scoring test set (measured by the number of mutants executed); the number of hard-to-kill mutants identified; and, the size of the generated test set. This chapter will use these same measures to determine the effect an algorithm's parameter has on its effectiveness. A fourth measure based on the mutation score achieved at each iteration will also be analysed for completeness. Only one parameter's value will be changed (from the default values used in chapter 5) per experiment. All graph results appear in Appendix C, with some repeated in this chapter for clarity.

6.2.1 Statistics

Comparisons between the mean values obtained using different parameter settings are compared using the ANalysis Of VAriance (ANOVA) statistical technique. This calculates two values, F_{obt} and F_{crit}, which can be compared to determine if at least one pair of parameter values results in significantly different means ($F_{obt} \geq F_{crit}$). To determine exactly which pairs of parameter values have significantly different means, the Scheffé method of post-hoc analysis is used. This operates in a similar vain to the ANOVA calculations, creating C_{obt} values for each pair of means, and a C_{crit} value to compare them against ($C_{obt} \geq C_{crit}$ implies significance). All comparisons were made at the 0.05 significance level - i.e. there is a 5% chance that the difference in means happened by chance. Readers interested in the exact details of these methods should consult [51] for an introduction.

6.3 Genetic Algorithm for Mutation Testing

The Genetic Algorithm, as defined in figure 4.19, has two parameters in common with the IIA/MT, namely the number of iterations and the population size (number of individuals). Iterations remain constant (at 500) throughout all these experiments, the population size however, is subtly different. To keep a fair comparison between algorithms, the number of **tests** within the population must remain constant, not the number of individuals. For the IIA/MT, the number of tests and number of individuals are the same (there is 1 test per individual) - this prevents any changes in population size; at least without affecting the total number of tests. For the GA/MT however, each individual contains many tests meaning that the number of individuals can be varied whilst retaining the same total number of tests. For the purpose of these experiments, the population size (number of individuals) shall be considered dependent on the individual size (number of tests per individual), in order to maintain a total of 300 tests.

With this in mind, the GA/MT has 3 user-definable parameters: `indSize`; `crossRate`; and `mutRate`. Table 6.19 shows the values assignable to each parameter, with the default options highlighted in **bold**. Unfortunately there is a tradeoff between exercising a full range of parameter values and the execution time. To allow statistically significant results, the values chosen were deliberately kept low to maintain reasonably low execution times, allowing more runs of each experiment. Given industrial use will generally also rate speed as important, the values chosen will probably be typical of those chosen in practical use. As such, the trends these parameter settings produce, within this range, are also important, but they do not necessarily reflect the trends across the entire range of parameter values. In any experiment only one parameter is modified, with the other parameters assuming their default values. As such, there are 13 possible combinations: the default plus: 3 for `indSize`, 5 for `crossRate` and 4 for `mutRate`.

Parameter	Values
`indSize`	5, 10, **20**, 30
`crossRate`	0.0, 0.2, 0.4, 0.6, **0.8**, 1.0
`mutRate`	0.01, **0.02**, 0.04, 0.06, 0.08

Table 6.19: Possible parameter values for the Genetic Algorithm. Default values are shown in **bold** font.

6.3.1 indSize

Chapter 5 suggested that the number of tests in a GA/MT individual may play an important part in determining the highest mutation score achieved by the algorithm. Essentially, each individual must contain enough tests to permit obtaining a full mutation score without adding unnecessary expense - too few tests and the achievable mutation score will be limited; too many and the algorithm may be costly to run[1]. Unfortunately, knowing the ideal number of tests per individual before testing is difficult, and forms a large part of the disadvantage of using a GA. Results from this research however, suggests that an appropriate size is related to the program's complexity (calculated using the McCabe Complexity [70]).

This research keeps the total number of tests in the population constant (to aid comparisons between algorithms), and instead opts to vary the population size (number of individuals) in response to a change in individual size (number of tests per individual). Doing this means any change in experiment outputs (from Chapter 5) can be attributed to the change in individual size, and not in response to a changed total number of tests. Unfortunately, this approach is not perfect. To maintain the total number of mutants, both the individual size **and** the population size must change, either of which could ultimately be responsible for altering the experiment outputs - this cannot be avoided. The advantage of using this approach however, lies in what it implies about the cause of the experiment

[1]For these experiments, the total number of tests (in the population) remains constant by adjusting the number of individuals to suit. If instead, the number of individuals in the population were fixed, increasing the individual size would increase the number of tests in the population. More tests increases execution time.

outcomes: if the total number of tests varies then the number of executions would also have to vary (each test needs to be executed on every mutant; more tests means more executions), regardless of the change (if any) in each individual's size; by keeping the total number of tests constant however, varying the number of tests can safely be ignored from causing any variation in execution numbers.

It is expected that this parameter will affect the execution time of the algorithm, with execution times decreasing as the individual size increases. This parameter is also expected to affect the mutation score per iteration and the number of HTK mutants identified, with both increasing as the individual size does. Finally, and rather obviously, the individual size is expected to affect the number of tests in the test set. Reasoning behind these expectations will be discussed in the following subsections.

Effect on the Number of Mutant Executions

Figure 6.32 (*DateRange* program) clearly indicates that the number of mutant executions is inversely proportional to the individual's size. A similar effect is also observed by the *TriangleSort* program (figure 6.33), although not for the *CalDay* program (figure 6.34).

In theory, a constant total number of tests and a static mutation rate should imply that the number of mutant executions per iteration will remain approximately constant, regardless of the individual size; or rather, that the number of executions to achieve a mutation score will remain approximately equal, regardless of the individual size. This is not necessarily the case however. Low individual sizes mean there are few tests per individual with which to generate a high mutation score. To improve an individual's mutation score, one of its tests must be improved via mutation - i.e. the mutated test must kill a greater number of mutants than the un-mutated test; something that increases in difficulty as the remaining living mutants become fewer (i.e. the remaining mutants are harder to kill, often requiring specific tests that generate low mutation scores). In order to find this "better" mutated test, it is likely the original test will have to be

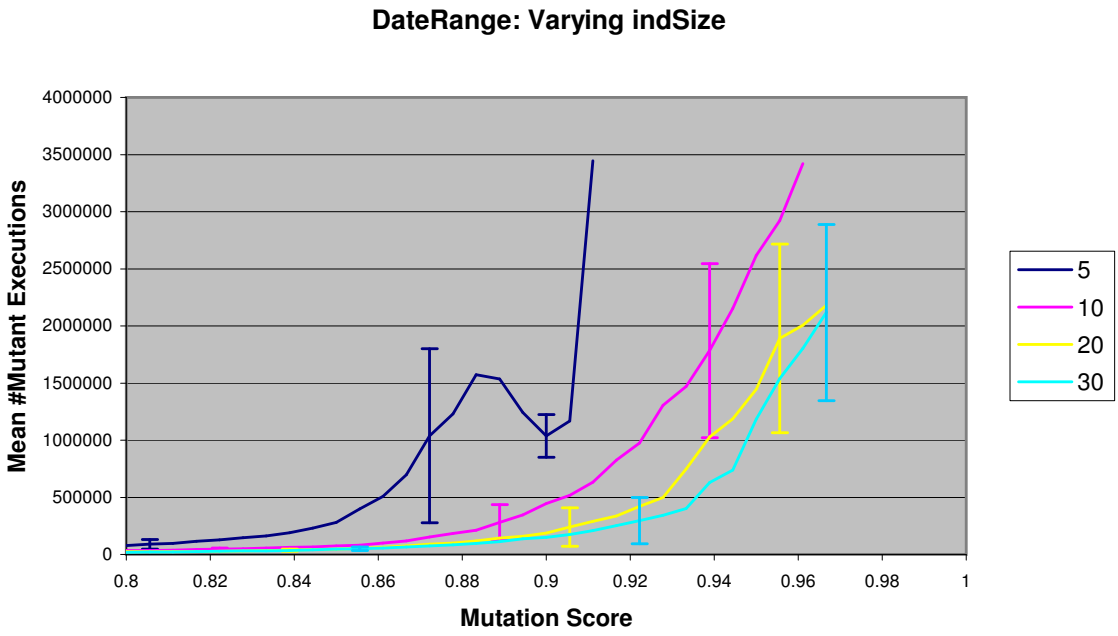

Figure 6.32: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

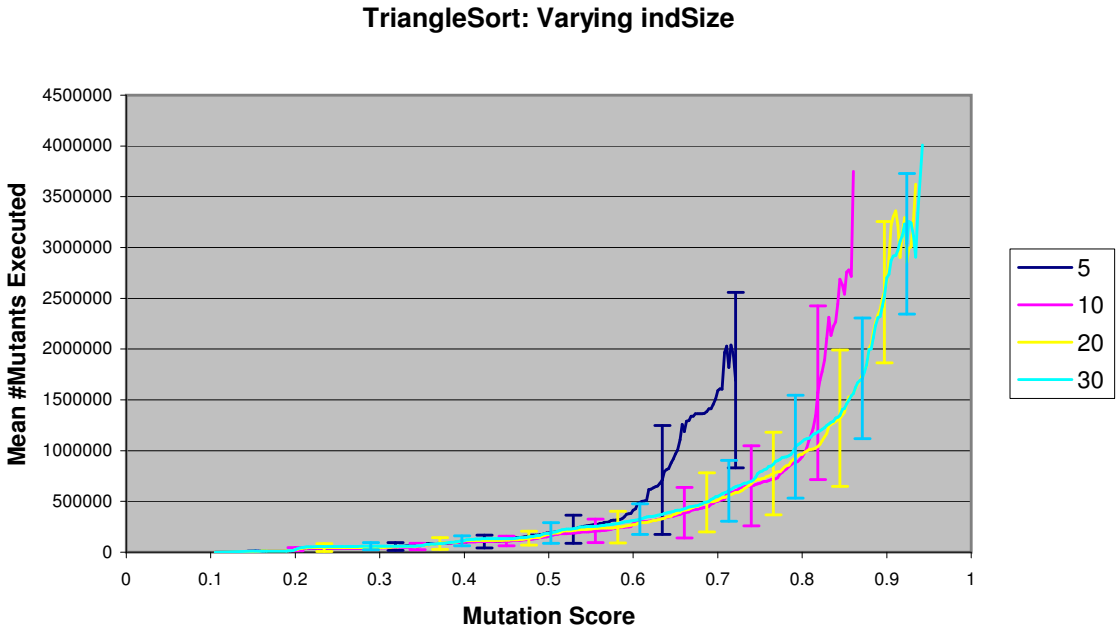

Figure 6.33: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

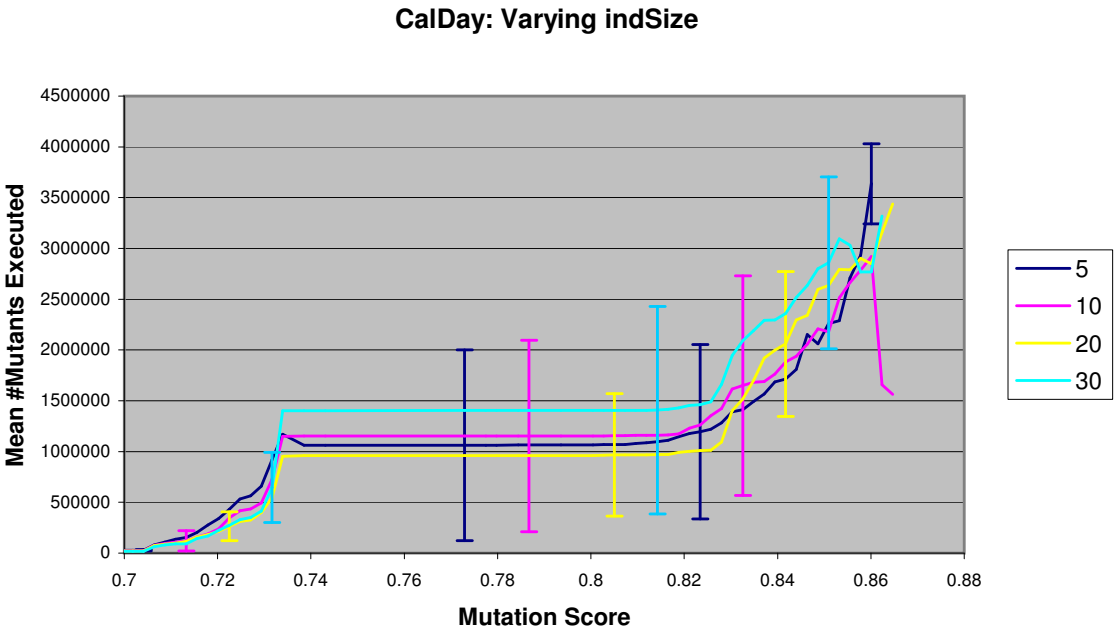

Figure 6.34: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

mutated more than once. Therefore, as every new mutated test will need to be executed on all mutants, reducing the individual's size will increase the number of mutant executions needed to achieve a particular mutation score. On the other hand, by increasing the individual's size allows the GA to be more relaxed about improving tests. New tests can be incorporated into the test set that only kill a few mutants, and yet help improve the overall mutation score - the number of mutant executions will decrease as the individual size increases, approaching a minimum value determined by the approximate number of mutated tests per iteration[2].

For each program, table 6.20 indicates the mean number of mutants executed to achieve the equally highest mutation score obtained by at least 25 runs for each of the four individual sizes. The results for the *DateRange* program clearly show the mean number of mutant executions decreases as the individual size increases.

[2]The estimated number of mutated tests per iteration is normally distributed around: $constant_mutation_rate * constant_number_of_tests$. Also, see the future work section in Chapter 7 for a brief discussion on alternative experiments.

Similarly for the *TriangleSort* program, the mean number of executions decreases dramatically when the individual size increases from 5 to 10; after this however, the mean number of executions increases slightly instead. The *CalDay* program offers no clarity in the matter, showing only an oscillation in the mean number of executions as the individual size increases. ANOVA and Scheffé calculations (also in table 6.20) show whether these observed differences in mean number of executions, caused by the varying individual sizes, are statistically significant (highlighted in **bold**).

indSize	DR MS: 87.78%	TRI MS: 70.00%	CD MS: 82.80%
5	1229827 ± 860997	1552438 ± 917277	1283966 ± 823223
10	182389 ± 99665	500556 ± 274995	1496200 ± 978277
20	101071 ± 43448	513033 ± 290777	1091833 ± 554958
30	82942 ± 36405	549471 ± 281399	1571294 ± 964933
ANOVA			
f_{obt}	**53.26**	**28.43**	1.88
f_{crit}	2.68	2.69	2.69
Scheffé			
c_{obt}			
5-10	**9.75**	**7.86**	-
5-20	**10.82**	**7.76**	-
5-30	**10.92**	**7.49**	-
10-20	0.80	0.10	-
10-30	0.97	0.38	-
20-30	0.18	0.29	-
c_{crit}	2.84	2.84	-

Table 6.20: The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean numbers of executions (in **bold**). The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. Values replaced with '-' are not important as the ANOVA results are not significant. *Mutation scores and ANOVA/Scheffé results are to 2 decimal places.*

As can be seen, the ANOVA tests indicate a significant difference between the

mean values for each individual size for the *DateRange* and *TriangleSort* programs. The Scheffé post-hoc analysis method indicates that for both of these programs, the mean number of mutant executions differs significantly when using an individual size of 5. In neither case is there a significant difference between using an individual size of 10, 20 or 30 though. Figures 6.32 and 6.33 however, hint that this significance should change with higher mutation scores. Scores greater than approximately 90% for *DateRange* and 81% for *TriangleSort* suggest that an individual size of 10 should also result in a significantly different mean number of mutant executions. To clarify this, ANOVA and Scheffé calculations were made based on mean execution numbers at the highest mutation score obtained by at least 25 runs for individual sizes of 10, 20 and 30 (individual size of 5 was ignored as this has already been identified as significantly different). Results are shown in table 6.21. Again, the ANOVA results indicate a significant difference for both *DateRange* and *TriangleSort*, with the Scheffé tests suggesting that using an individual size of 10 results in significantly different means from using sizes of 20 or 30. In no case does using an individual size of 20 or 30 result in a significantly different number of mutant executions, suggesting that no more than 20 tests are required per individual, at least for these three programs. None of the ANOVA results, for the *CalDay* program, indicate a significant difference amongst the mean number of executions from using any of the four individual sizes - a result also echoed in figure 6.34.

Primarily, the individual's size does appear to have an effect on the number of mutant executions, with 2 out of the 3 programs requiring significantly more executions to achieve the same mutation score when using a lower individual size. Because the *CalDay* program does not exhibit the same results however, it is difficult to ascertain whether there is an inversely proportional relationship with the individual's size, or not. The other two programs do hint at this though.

More importantly, the results support the notion of a threshold on the number of tests required to achieve a high mutation score - individual sizes larger than this threshold (i.e. 20 and 30) have little impact on the number of executions needed;

indSize	DR MS: 92.78%	TRI MS: 82.63%	CD MS: 82.80%
5	-	-	1283966 ± 823223
10	1306656 ± 676469	1891191 ± 1002279	1496200 ± 978277
20	498526 ± 302169	1136750 ± 589230	1091833 ± 554958
30	342144 ± 217220	1225014 ± 296488	1571294 ± 964933
ANOVA			
f_{obt}	**43.28**	**8.31**	1.88
f_{crit}	3.10	3.11	2.69
Scheffé			
c_{obt}			
10-20	**7.39**	**3.78**	-
10-30	**8.76**	**3.34**	-
20-30	1.49	0.46	-
c_{crit}	3.05	3.05	-

Table 6.21: The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the three individual sizes: 10, 20, 30 (individual size of 5 ignored). ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean numbers of executions (in **bold**). The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. Values replaced with '-' are not important as either the ANOVA results are not significant or Scheffé results are not calculated because the individual size=5 experiment only achieved a lower mutation score. *Mutation scores and ANOVA/Scheffé results are to 2 decimal places.*

sizes lower that this (i.e. 5 and 10) restrict the achievable mutation score. This opens a number of questions: *what is the threshold value for a program? Is the threshold value different for different programs?* And, *is it possible to determine this threshold a priori to testing.* The last question is important as it means testing can commence using appropriately sized individuals, without the need to first determine the size by performing multiple tests using various sized individuals.

The McCabe complexity of a program indicates the number of paths through that program, and is therefore often used as a useful measure of the number of tests required to test each branch condition [70]. On this basis, *could it also provide an indication of the threshold value for a program?* Assuming that it can

implies that *CalDay*'s threshold is approximately 4 tests, *DateRange*'s threshold is approximately 6 tests and *TriangleSort*'s threshold is 11. *How do these threshold values compare with the results already observed?*

For *CalDay*, all four individual sizes tested are greater than its threshold, implying that none of these four sizes should cause a significant difference in the number of mutant executions; figure 6.34 shows this is indeed the case. A similar effect is also seen for the *DateRange* program (figure 6.32) using individual sizes of 10, 20 and 30 (i.e. sizes greater than its threshold). Using an individual size lower than its threshold (i.e. 5) results in attaining a lower mutation score in considerably more mutant executions. Interestingly, as the mutation score increases, an individual size of 10 also starts to execute significantly more mutants (than either 20 or 30). This is either a failing of using the McCabe complexity as an approximate threshold measure, or indicates the difficulty of improving the mutation score of a test set when its size is close to the minimum threshold. For individual sizes close to the threshold, it is foreseeable that they will attain a reasonable mutation score, after which point it will become increasingly difficult to mutate a test so that it improves its mutation score - the individual size means the GA does not have room to incorporate tests that only kill a few specific mutants, and yet still improve an individual's overall mutation score. This latter explanation is in many ways what figure 6.32 suggests, as the results for an individual size of 10 do not become significant until a reasonably high mutation score is achieved. For *TriangleSort* (figure 6.33), the lowest two individual sizes are less than the threshold, implying that these should achieve lower mutation scores than the larger sized individuals - the results show this to be true. Furthermore, the results also indicate that as the individual size decreases below the threshold, the mutation score attained also drops, and the number of executions needed to achieve a specific mutation score increases.

Whilst not presenting a conclusive outcome, the results observed do suggest that the number of mutant executions varies inversely proportional to the individual's size. Furthermore, the results are suggestive of a minimum threshold value

surrounding the number of tests in an individual - sizes larger than this have no impact on mutant executions; sizes lower than this require significantly more executions. The McCabe complexity measure was suggested as an approximate value for this threshold, and coincides with the results, making it possible to determine an appropriate individual size *a priori* to testing.

Effect on the Mutation Score

Figures 6.35 (*DateRange*) and 6.36 (*TriangleSort*) suggest the mean average mutation score after 500 iterations is proportional to the individual size. This result is not demonstrated by the *CalDay* program however (figure 6.37), which actually hints at the opposite (the final mutation score is inversely proportional to individual size). Table 6.22 details the final mutation scores obtained and indicates which combinations of individual sizes result in significantly different mean mutation scores. In particular, they show that for *DateRange* and *TriangleSort*, the differences in mutation scores achieved for each individual size are significant (at the .05 level).

The previous subsection suggested that limiting the individual size increases the difficulty in improving the overall mutation score. An improvement in mutation score can only occur in a GA by modifying the tests in an individual. With fewer tests, smaller sized individuals have to kill more mutants with each test - something that becomes harder as the mutation score increases, leaving only HTK mutants. As a consequence, more mutations will occur in order to improve the mutation score, which ultimately requires more iterations. This is evidenced by figures 6.35 and 6.36. *CalDay* however, seems to detract from this theory, although its results can be explained if the previously mentioned notion of a minimum threshold for an individual's size is taken into account.

The effect the individual's size has on the number of mutant executions encouraged the idea of a minimum threshold on the individual's size in order to achieve a high mutation score. This threshold was approximated using a program's McCabe complexity, with promising results. Individual sizes greater than

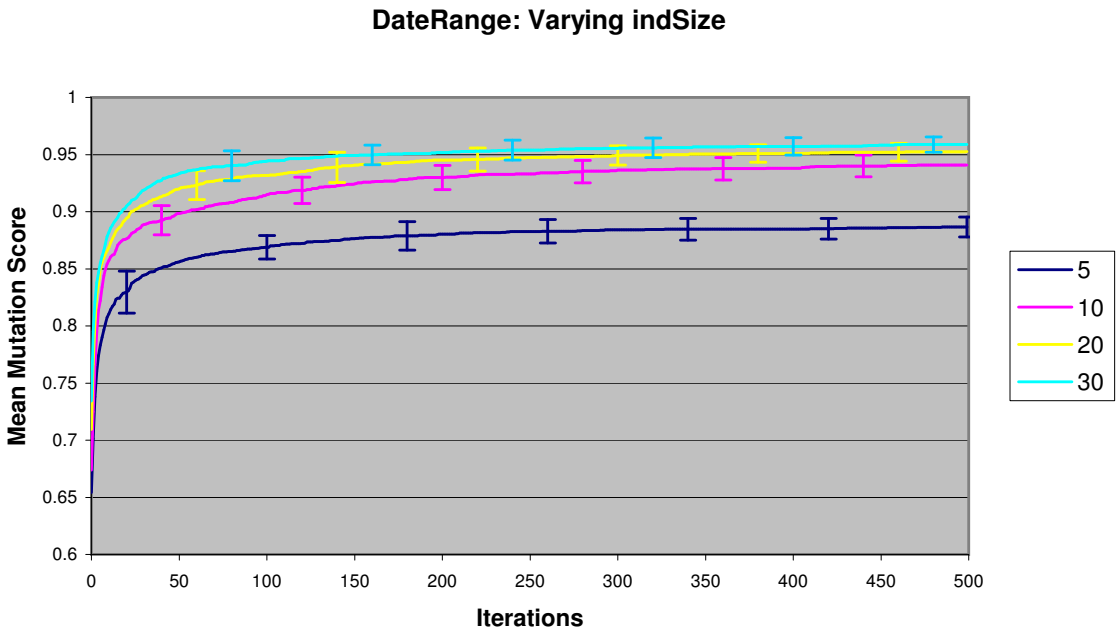

Figure 6.35: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

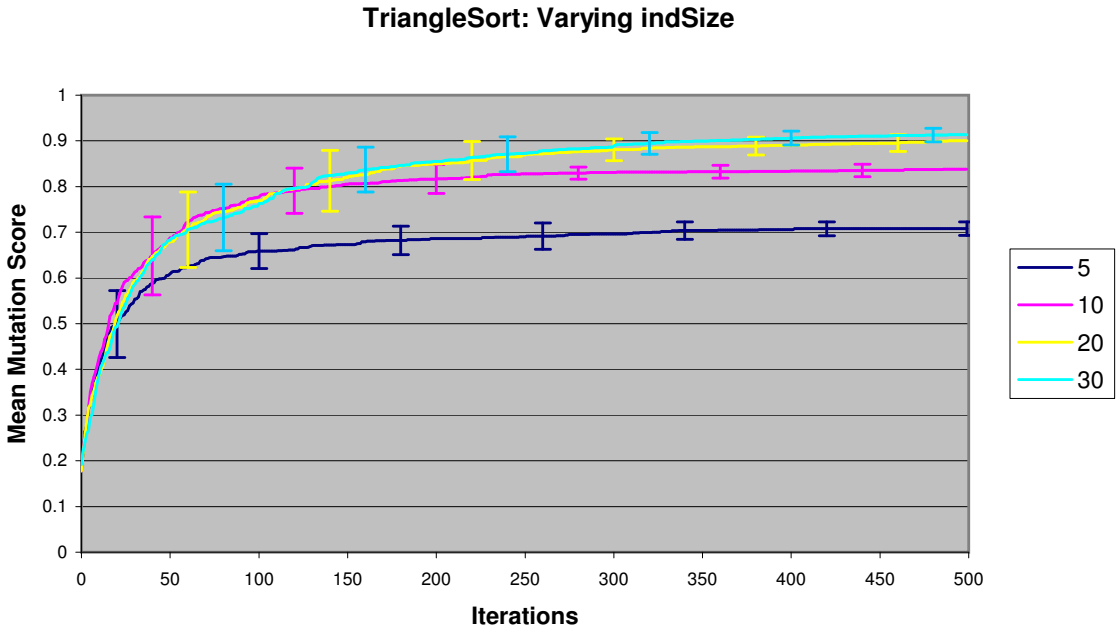

Figure 6.36: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

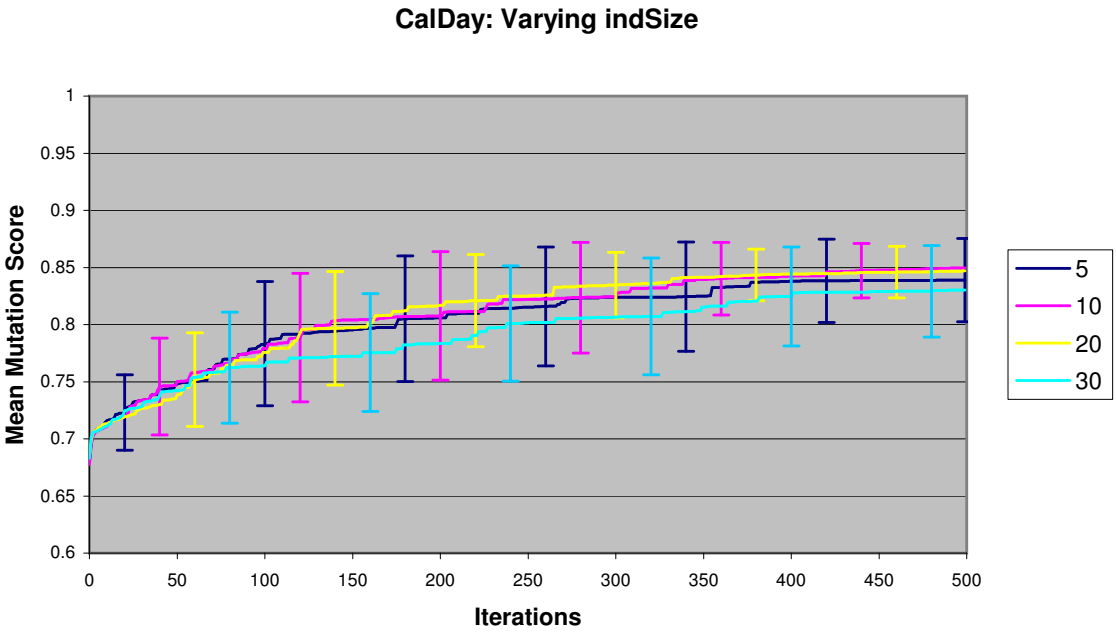

Figure 6.37: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

this threshold should not significantly improve the mutation score; sizes smaller than it should restrict the achievable mutation score. This theory also corresponds with the results observed for the effect on the mutation score. The approximated threshold for *DateRange* is 6 - comparisons between sizes above this (i.e. 10, 20 and 30) result in relatively insignificant[3] differences amongst the final mutation scores (the Scheffé c_{obt} scores for comparisons between individual sizes of 10, 20 and 30 are only just larger than the c_{crit} value, especially in relation to the scores for comparisons with an individual size of 5), suggesting that these sizes have little effect. The threshold for *TriangleSort* is 11 - comparisons between sizes 20 and 30 also result in relatively insignificant differences amongst final mutation scores; comparisons with sizes below this threshold (i.e. 5 and 10) exhibit relatively significant differences. *CalDay*'s threshold is 4 - all comparisons between individual sizes are not significant, as all individual sizes are larger than this threshold.

[3]Significance in this sense is measured relatively to how much larger c_{obt} is to c_{crit}; only just larger can be described as "relatively insignificant".

`indSize`	DR	TRI	CD
5	88.65% ± 0.87	70.79% ± 1.54	83.90% ± 3.64
10	94.07% ± 1.04	83.82% ± 1.27	85.02% ± 2.39
20	95.26% ± 0.80	90.05% ± 1.40	84.70% ± 2.24
30	95.89% ± 0.65	91.37% ± 1.48	83.01% ± 4.05
ANOVA			
f_{obt}	**471.59**	**1253.42**	2.42
f_{crit}	2.68	2.68	2.68
Scheffé			
c_{obt}			
5-10	**24.88**	**34.85**	-
5-20	**31.25**	**51.53**	-
5-30	**33.99**	**55.05**	-
10-20	**5.61**	**16.98**	-
10-30	**8.53**	**20.56**	-
20-30	**3.06**	**3.58**	-
c_{crit}	2.83	2.84	-

Table 6.22: The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean mutation scores (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

Overall, the results observed are suggestive that the mutation score is proportional to the individual's size, with a minimum size existing in order to achieve a high mutation score. Individuals smaller than this threshold achieve significantly lower mutation scores, whilst those dramatically larger offer no significant increase. Ultimately this is to be expected, as a certain number of tests will be required to achieve a full mutation score. Less than this number will restrict the mutation score; greater than it will not allow any more mutants to be killed. It is interesting that this number can be approximated by the McCabe complexity of the programs under test.

Effect on the Number of HTK identified

Logically it follows that, if the mutation score is proportional to the individual's size, so must the number of hard-to-kill mutants identified - more HTK mutants must have been killed in order to achieve a higher mutation score. Similarly to the results obtained for the mutation score then (previous subsection), this expected result is demonstrated by both *DateRange* and *TriangleSort*, and not by *CalDay*. Figure 6.38 shows the percentage of HTK mutants killed by each individual size, for each program. Table 3.35 in Appendix C.1.3 (page 245) indicates which differences between individual sizes are significant.

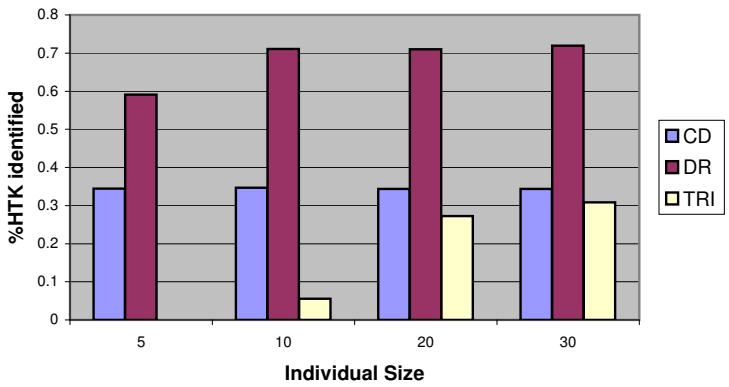

Figure 6.38: Effect of varying the individual size on the mean number of HTK mutants identified for all three programs.

If the individual size threshold is taken into account (mentioned in the previous subsection), then the differing results between the *CalDay* program and the others can be explained. Improving a mutation score means killing more HTK mutants (for higher scores at least) and so: individual sizes less the threshold kill fewer HTK mutants, whereas larger sizes have little effect. This idea is demonstrated in the results. *CalDay* has a McCabe complexity of 4, meaning all individual sizes are larger than the threshold. As expected, the percentage of HTK killed differs insignificantly between each `indSize` (at the 0.05 level). *DateRange* has a McCabe complexity of 6 - only one individual size is smaller than the threshold (`indSize` 5), and it is this size that kills significantly fewer HTK mutants; the other three sizes

kill approximately the same percentage. *TriangleSort* has a McCabe complexity of 11 - two individual sizes are smaller than the threshold (`indSize` 5 and 10), and both of these kill significantly fewer HTK mutants (`indSize` 5 kills 0%); the other two larger sizes kill approximately the same percentage.

The results suggest that the number of hard-to-kill mutants identified relates to the individual's size (albeit through reasoning about the mutation score attained), and in particular to the minimum size threshold needed to achieve a high mutation score. Individual sizes less than the threshold kill fewer HTK mutants; larger sizes have no effect. For killing HTK mutants, an individual size slightly larger than the threshold value would be preferential.

Effect on the Number of Tests

As the individual's size imposes a limit on the number of tests, it will have an obvious (limiting) effect on the number of tests created. This is shown for the *DateRange* program in the figure 6.39 (the other two programs have similar graphs, shown in Appendix C.1.4). Table 3.36 in Appendix C.1.4 (page 248) indicates the mean number of distinct tests obtained by at least 25 runs for each of the individual sizes, as well as the significance of any differences.

There are two possible causes as to why the individual's size should affect the number of tests in the way seen: counting the number of 'distinct' tests, rather than the number of 'useful' tests; and, how mutation generates new tests. The first possibility implies that by counting tests based on whether they are different from any other test in the individual, and not whether they are the sole killer of at least one mutant, means that the number of tests counted could be greater than the number actually needed. Tests could be counted that add nothing to the mutation score. Finding the minimum subset of tests needed to kill the maximum number of mutants is a difficult task however, and beyond the scope of this work. Additionally though, the number of distinct tests in an individual is heavily influenced by the processes of crossover and mutation. Each iteration, these mechanisms are responsible for generating new tests which ultimately result

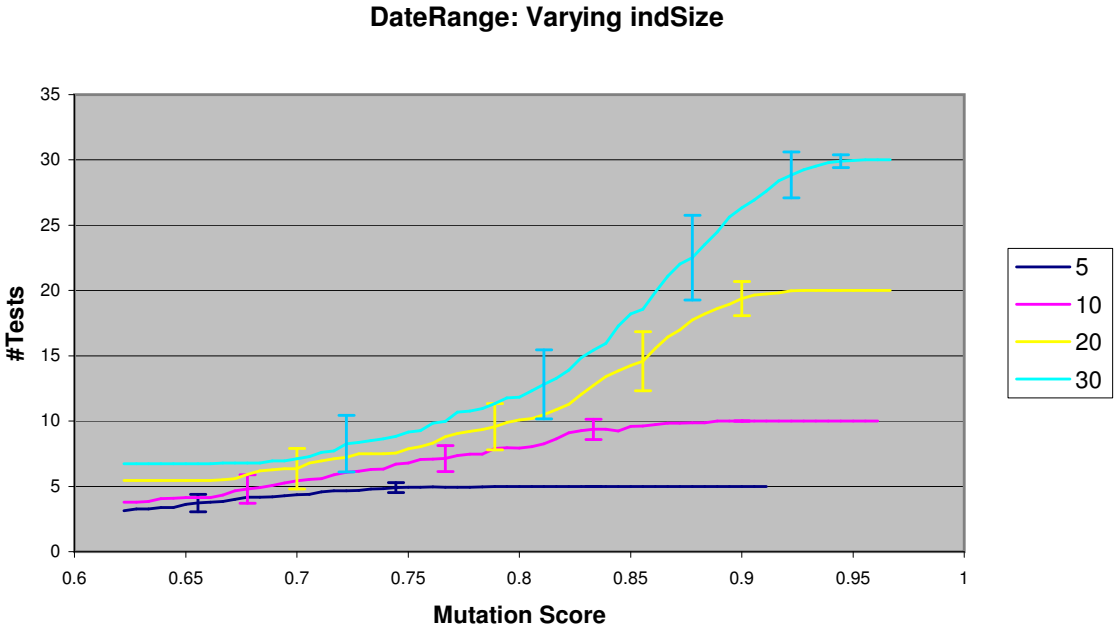

Figure 6.39: Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program.

in more distinct tests, regardless of whether they are useful or not.

The number of tests mutated per iteration can be approximated based on the mutation rate probability. For simplicity assume the average number of tests mutated is calculated by: `mutRate*indSize`[4]. Assuming a constant mutation rate (`mutRate`) of 0.02: roughly 0.1 tests will mutate per iteration for an `indSize` of 5; 0.2 tests will mutate for an `indSize` of 10; 0.4 tests for an `indSize` of 20; and, 0.6 tests will mutate for an `indSize` of 30. Despite the constant mutation rate, the individual's size has a definite impact on the number of mutated tests per iteration, and consequently, the number of distinct tests created. Furthermore, as the iterations progress, killing mutants becomes harder (specific tests are required) and so more iterations are needed to increase the mutation score. The more iterations per mutation score increase, the more tests that are mutated, and the larger the test set size becomes. This effectively means that the differences in

[4]The actual mean number of tests is calculated slightly differently as the mutation rate refers to the probability of mutating a value within a test, not the probability of mutating the test itself. For this example however, the true values are unimportant.

the number of tests mutated per iteration (between the various individual sizes) is multiplied by the number of iterations needed, causing a divergence of test set sizes as seen in the figures. The number of tests is only limited by the fixed size of the individuals - a bound set by the individual's size!

Finally, from these results it would appear that the minimum individual threshold size has little impact on the test set size. Ultimately it is a combination of the mutation rate and the individual size that indicates the test set size, with the maximum being limited by the individual's size. On the basis of set size, a smaller individual size is better, although this will probably inhibit the overall mutation score achieved.

6.3.2 crossRate

Pseudocode in figure 4.19 indicates that the `crossRate` parameter is used numerous times each iteration to swap the tail portions of two randomly selected individuals (single-point crossover). The purpose of this is to recombine tests from both individuals in an attempt to generate a fitter (higher mutation scoring) individual. No mutation occurs during this process however, meaning there are no new tests created that need to be executed against all mutants (all tests in the population have their results cached until they are removed from the population). For this reason, it is expected that `crossRate` will have no effect on the number of mutant executions needed to achieve a certain mutation score, nor on the number of HTK mutants identified or the number of iterations required to achieve a specific mutation score. Finally, it is also expected that `crossRate` will not affect the test set size. These expectations will be discussed more in the following subsections.

Effect on the Number of Mutant Executions

Figures 3.80-3.82 in Appendix C.1.5 clearly demonstrate that the crossover rate has little effect on the number of mutant executions. To confirm these results

however, ANOVA calculations were performed on the mean number of mutants executed at every mutation score achieved to determine if there is any significant difference in the results for each crossover rate. For completeness, the results of these calculations at the equally highest mutation score achieved by at least 25 runs for each of the six crossover rates, are shown in table 3.37 (Appendix C.1.5, page 251).

All three programs exhibit similar results - the null hypothesis (i.e. no significant difference between any of the mean numbers of mutants executed) was accepted at every mutation score achieved, except for a few high mutation score values (at most, for all three programs, the null hypothesis was rejected at 5 mutation score values). These rare exceptions are due to subtle differences in the mean execution numbers, which most likely result from fewer experimental runs achieving the higher mutation scores.

Overall, these results confirm that the crossover rate causes no significant difference in the mean number of mutants executed. As no test mutation occurs during crossover (i.e. no new tests are created), this results is to be expected. Based on this evidence, and in terms of the number of mutant executions, a crossover rate of 0 may as well be employed as this would save the unnecessary computational expense of creating crossed-over offspring individuals. This highlights the fact that it is the combination of tests in an individual which is most important rather than particular tests - a test that is useful in one individual may not be in another.

Effect on the Mutation Score

Figures 3.83-3.85 in Appendix C.1.6 (pages 252-253) show that the crossover rate has no effect on the number of iterations either. To confirm this, ANOVA calculations were performed on the mean average mutation score achieved at every iteration. For the majority of iterations, the null hypothesis was accepted (at the 0.05 level), suggesting no significant differences amongst the mean mutation scores achieved at each iteration.

To summarise the results, the mutation scores achieved after 500 iterations are presented in table 3.38 (Appendix C, page 254). ANOVA calculations for these figures are also indicated, and show the f_{obt} value for the *TriangleSort* program to be greater than the f_{crit} value, suggesting a significant difference between mutation scores achieved for at least two crossover rates. Interestingly, the Scheffé values do not indicate any such significance for any pair of results. As the f_{obt} value (2.36 [2 d.p.]) is only just larger than the f_{crit} value (2.27 [2 d.p.]), it is safer to assume these results are not significant.

In general, these results indicate that the crossover rate has no effect on the mean mutation score achieved at each iteration. This is to be expected. Crossover swaps the tail portions of two randomly selected individuals in an attempt to combine the best parts of two parent individuals. Given many tests are likely to kill the same mutants however and the fact that the whole population is generally being improved (in terms of each individual's mutation score), swapping any number of tests is unlikely to have much effect on the mutation scores of those individuals. Furthermore, the results recorded represent the best individual in the population in any given iteration, with this being kept (in its original form, i.e. no crossover or mutation applied) in the population to the next iteration at least. Because of this, the mutation score can never decrease, effectively implying the crossover rate has no effect on improving the mutation score at any iteration, and that a `crossRate` of 0 may as well be used.

Effect on the Number of HTK identified

Figure 6.40 indicates the results on the percentage of HTK identified from varying the crossover rate. These show that the crossover rate has no significant effect, which is to be expected as `crossRate` has previously been shown not to affect the achievable mutation score. Table 3.39 in Appendix C.1.7 (page 255) shows the insignificance of the results. A crossover value of 0 may as well be used.

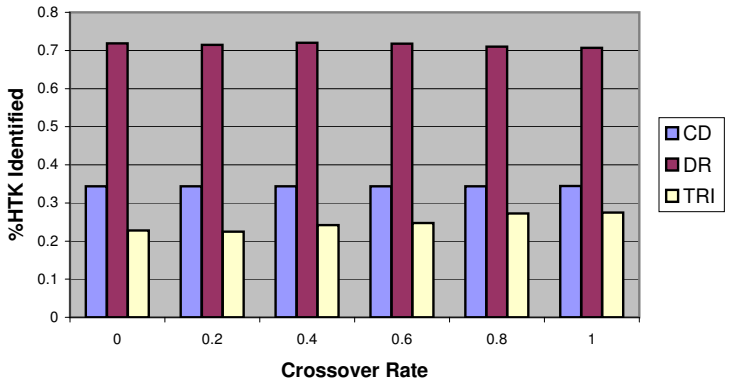

Figure 6.40: Effect of varying the crossover rate on the mean number of HTK mutants identified for all three programs.

Effect on the Number of Tests

Figures 3.86-3.88 in Appendix C.1.8 show that the crossover rate has no effect on the mean number of distinct tests needed to achieve a specific mutation score. To confirm this, ANOVA calculations were performed on the mean number of distinct tests needed at each mutation score obtained. The majority of the results accepted the null hypothesis - i.e. there is no evidence to suggest the crossover rate significantly affects the test set size needed to achieve specific mutation scores. Those results where the null hypothesis was rejected occurred mainly at low mutation scores - between 20 and 30% for the *TriangleSort* program - with only a few sporadic rejections at other mutation score values; all results for *CalDay* accepted the null hypothesis.

To summarise the results, the mean test set sizes needed to achieve the highest mutation score, obtained by at least 25 runs of each crossover rate, are shown in table 3.40 (Appendix C.1.8, page 258). The table also presents the ANOVA calculations of these results, indicating that none of the test set sizes (for a given crossover rate) are significantly different from the others.

The results indicate that the crossover rate does not impact on the test set size for a specific mutation score. Crossover does not mutate any tests, but merely

swaps over the tail portions of two individuals. Each individual will undergo mutation which is capable of creating distinct tests. However, because all individuals undergo mutation at the same rate, the number of distinct tests in each individual is likely to be roughly equal (see section 6.3.3). In this case, the number of distinct tests in an individual will remain roughly constant irrespective of how many are crossed over (i.e. irrespective of the crossover rate). To save expense, a crossover rate of 0 could be used.

6.3.3 mutRate

Each iteration the `mutRate` parameter directs the amount of mutation each child individual undergoes. The higher the mutation rate, the greater the chance that one or more of an individual's tests will be mutated. It is expected that the number of mutant executions will be proportional to the mutation rate, as will the mutation score for any given iteration. Increases in mutation score suggest increases in the number of HTK mutants identified, so it is expected these will also be proportional to `mutRate`. Finally, given the test set size is calculated by the number of distinct tests in an individual, a parameter that can increase this number (by mutating existing tests) should increase the set size. However, as more tests in an individual are mutated, the number of distinct tests will approach the number of tests within the individual. The expectation, therefore, is that the test set size for relatively low mutation scores (i.e. the first few iterations) is proportional to the mutation rate, after that, test set size is unaffected by mutation rate and approaches the individual size as the iterations increase.

Effect on the Number of Mutant Executions

Figures 6.41 and 6.42 indicate the expected effect the mutation rate has on the number of mutant executions - as the mutation rate increases, so does the number of executed mutants. This expectation is also demonstrated in the results for the *CalDay* program (figure 6.43), except that the mutation score achieved by the

higher rates (0.06 and 0.08) has been severely limited within the 500 iterations (if it were allowed more iterations, it would most likely execute more mutants than the lower `mutRate`s to achieve the higher mutation scores). A similar mutation score limitation is also seen with the *TriangleSort* program, except the results are not as noticeable.

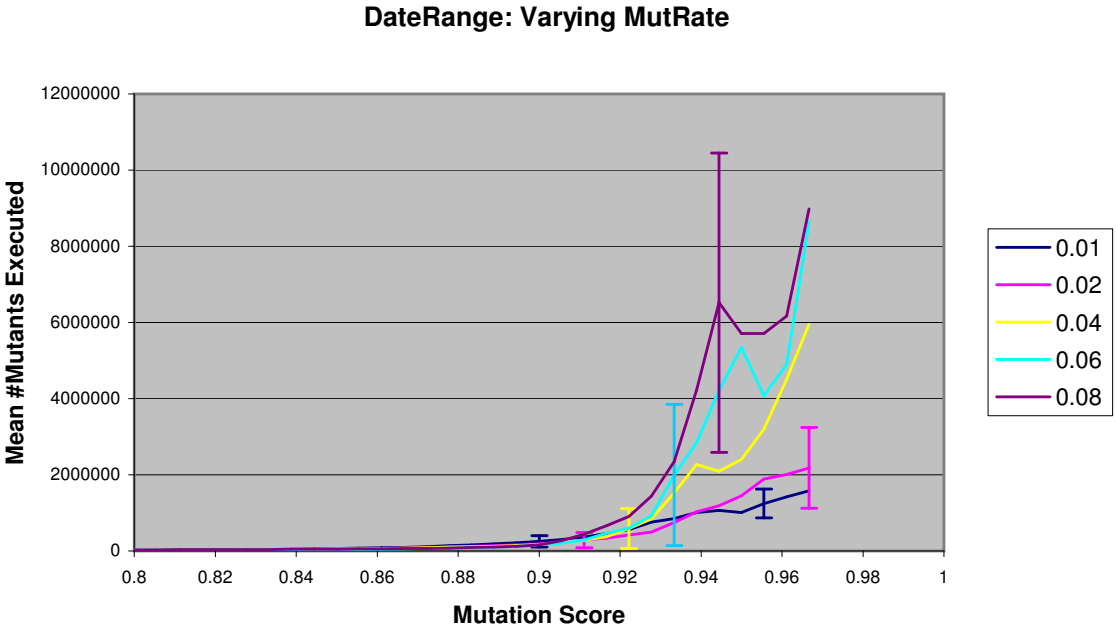

Figure 6.41: Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

Table 3.41 (Appendix C.1.9, page 261) indicates the mean number of mutant executions at the equally highest mutation score achieved by at least 25 runs across each mutation rate. Unfortunately these results are not as conclusive as the graphs suggest, primarily because the comparison point - the equally highest mutation score achieved in at least 25 runs across all mutation rates - is just before the divergence seen in the figures. For at least one mutation rate per program, fewer than 25 runs achieve a higher mutation score than the comparison point. These results however, should not necessarily be seen as indicative of a lack in significance (between the mean number of executions) at higher mutation scores. For example, the low comparison point for *CalDay* occurs because neither

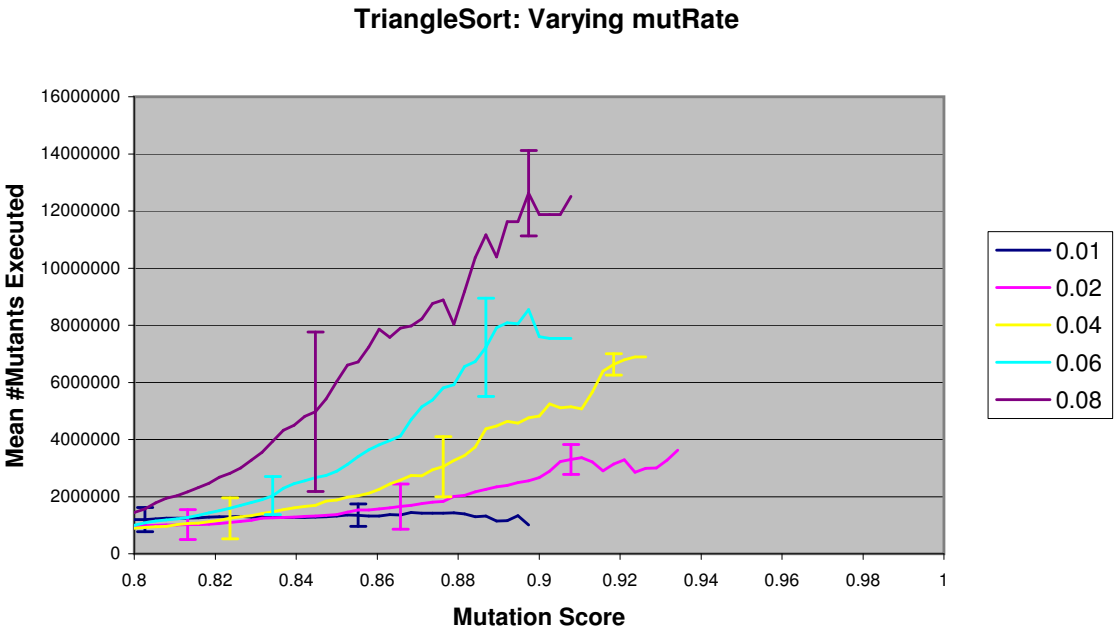

Figure 6.42: Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

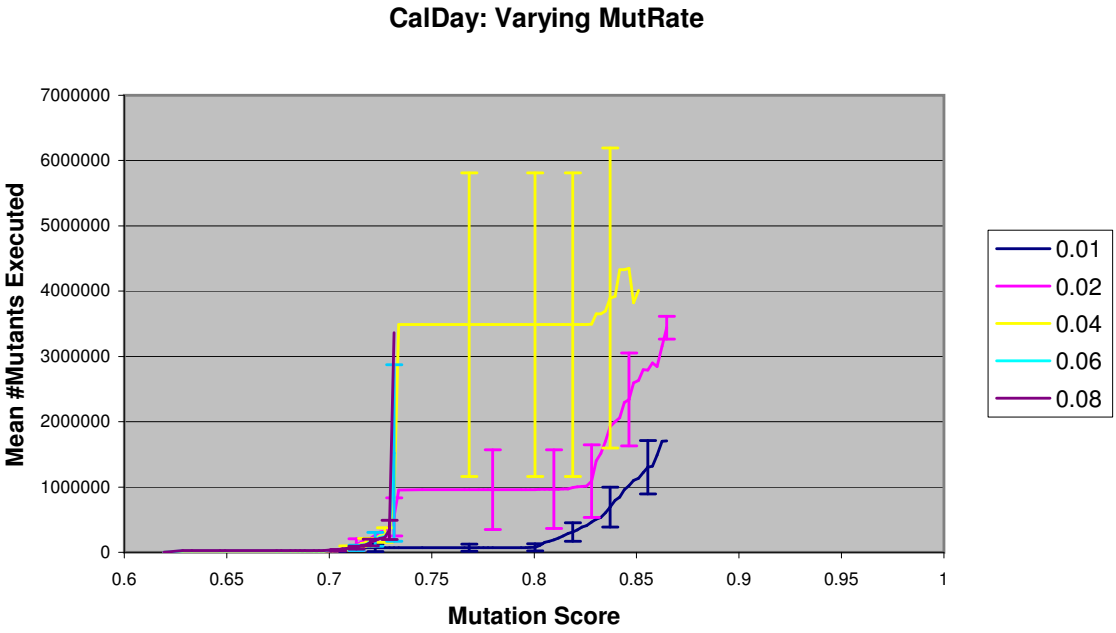

Figure 6.43: Effect of varying the mutation rate on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

a mutation rate of 0.06 or 0.08 achieve greater than 73.17%. This means that, if given an infinite number of iterations, these mutation rates would require a much larger number of mutant executions to achieve higher mutation scores, causing a significant difference at higher mutation rates.

The proportionality of these results is to be expected. Every iteration each offspring individual (besides the highest scoring individual that is kept from the previous iteration) undergoes mutation at a constant rate. This mutation rate is the probability of a test input being mutated, and the degree to which it is mutated. Clearly, the higher the mutation rate is, the more likely a test is to be mutated, and therefore the more new tests that will be generated per iteration; each of these will need to be executed against the mutant programs (in order to determine their mutation scores). Obviously then, a higher `mutRate` results in a higher number of mutant executions. *But why the mutation score limitation?* This limitation happens because higher mutation rates are also responsible for causing greater degrees of mutation to each test input. Instead of subtly improving a test through a slight mutation, much more variation is likely to occur, resulting in drastic changes to the tests and their mutation scores. Essentially, as the mutation rate increases, the local search tends towards a randomised search.

In terms of the number of mutant executions, these results suggest that the lower the mutation rate the fewer mutant executions that are needed to achieve a specific mutation score. If the mutation rate is too low however, the achievable mutation score will be limited as not enough searching will be performed. The results suggest that a `mutRate` of 0.02 is likely to achieve a higher overall mutation score (albeit in slightly more executions) than the lowest rate (tested) of 0.01. Using a mutation rate of 0.02 is therefore the preferential choice, despite the slightly higher execution costs.

Effect on the Mutation Score

The effect `mutRate` has on the mutation score per iteration is not always the same for the various programs tested. *DateRange* and *TriangleSort* (figures 6.44 and

6.45 respectively) suggest that the higher mutation rates (0.04, 0.06, and 0.08) initially achieve higher mutation scores in fewer iterations than the lower rates; this is not indicated by *CalDay* (figure 6.46). All three figures do however, suggest that these higher `mutRate`s achieve a lower mutation score after 500 iterations, than a rate of 0.02. Only in one case (for the *CalDay* program) does a `mutRate` of 0.01 achieve the highest mutation score after 500 iterations. Furthermore, all three programs indicate that the lower mutation rates present a more leisurely gain in mutation score per iteration.

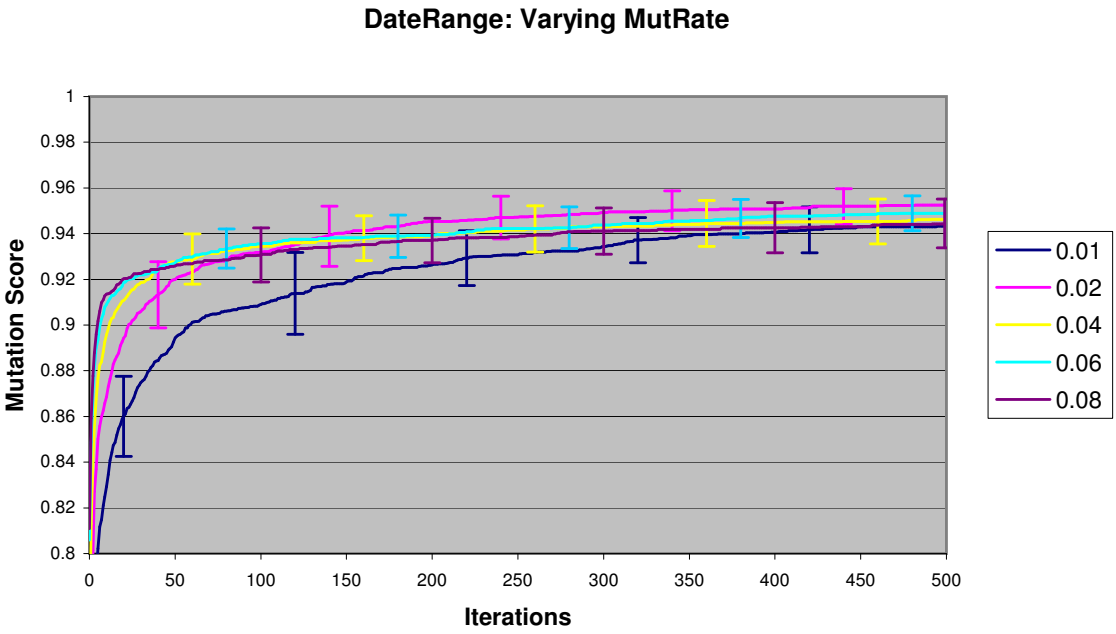

Figure 6.44: Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *DateRange* program.

Table 3.42 (Appendix C.1.10, page 264) indicates the mean mutation scores after 500 iterations and which pairs of mutation rates give significantly different results. As can be seen, for all three programs a mutation rate of either 0.01 or 0.02 achieves the highest mutation score after 500 iterations. Despite this evidence, and the overall results presented in the three figures, only the *CalDay* program strongly indicates a significant difference between the mutation scores achieved with rates of 0.01/0.02 and 0.04-0.08. This is the least complex program

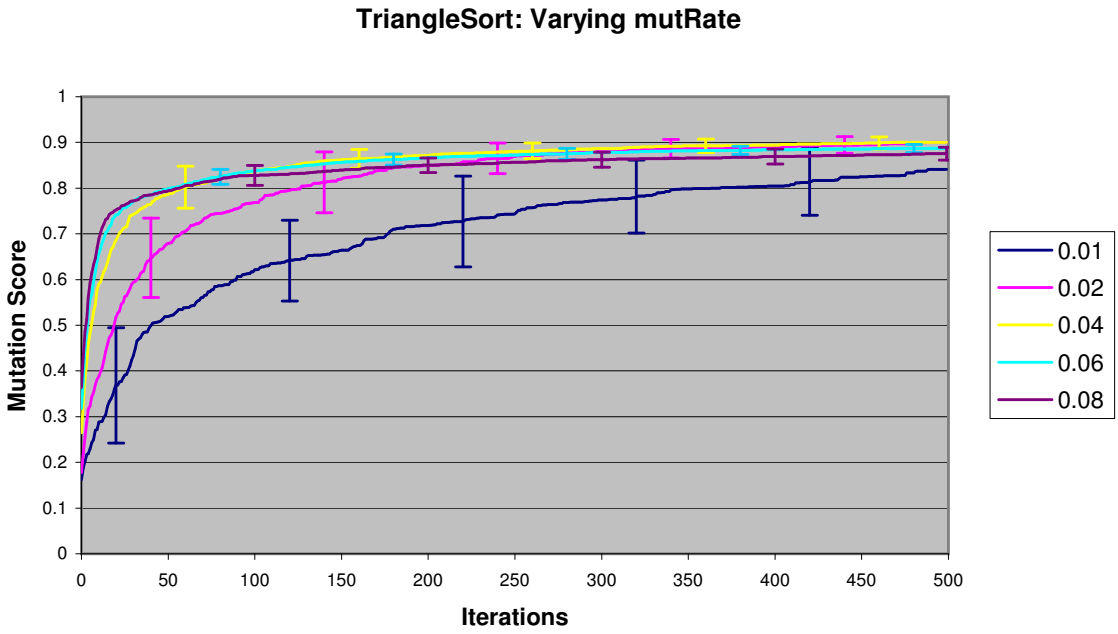

Figure 6.45: Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *TriangleSort* program.

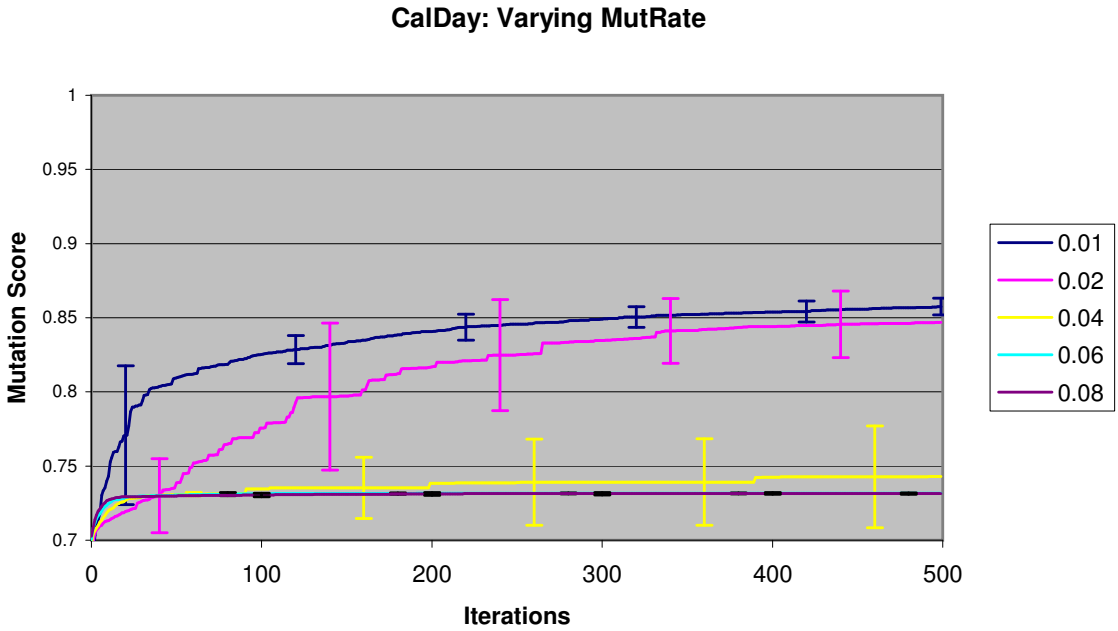

Figure 6.46: Effect of varying the mutation rate on the mean mutation score achieved per iteration for the *CalDay* program.

of the three (McCabe complexity of 4) possibly hinting at a relationship between the mutation scores achievable with various mutation rates and the program's complexity. Unfortunately there is not enough evidence from these experiments to formulate any claim.

It was expected that the mutation score achieved for any given iteration would be proportional to the mutation rate. Higher mutation rates would cause more tests to change per iteration, increasing the probability of killing living mutants (i.e. improving the mutation score). This expectation has not been realised according to the three programs tested, except maybe for the first 30-40 iterations of the *DateRange* and *TriangleSort* programs. It is in these iterations that the test set is improved from a weak set (each individual contains 20 of the same low mutation scoring tests), and so mutation rates that cause the most variation are more likely to improve these tests quickly. After the mutation score has increased to a certain level however, large changes in the tests (caused by higher mutation rates) will make it more difficult to improve the mutation score - at this point each test's search needs to be directed implying lower mutation rates; higher mutation rates tend to make the search more random. This is the results seen in *DateRange*, *TriangleSort*, and even *CalDay* to a degree. Lower mutation rates, on the other hand, would cause less test mutation each iteration and so not increase the mutation score as quickly. However, because they cause less variation in each test, they present a more directed search for killing living mutants, and therefore a better chance at generating a higher mutation score overall. Too low a mutation score though, and not enough change will occur in order to find 'stronger' tests, with the result being a lower mutation score again.

Overall then, lower mutation rates exhibit a more gradual increase in mutation score, but keep increasing the mutation score over more iterations. The more directed search (through lower test variation) offered by these lower rates also improves the likelihood of achieving a higher mutation score. A mutation rate of 0.02 offers the best results for these three programs, killing the most mutants (out of the rates tested) for all three programs.

Effect on the Number of HTK identified

Figure 6.47 details the effect of varying the mutation rate on the percentage of hard-to-kill mutants killed, for each of the three programs. As can be observed, they do not present any clear, consistent correlation between two. *CalDay*'s results are fairly constant across all mutation rates, as are *DateRange*'s results, bar a significantly lower result for a `mutRate` of 0.01. *TriangleSort*'s results are also fairly consistent except that a `mutRate` of 0.02 kills a significantly larger number of HTK mutants than 0.04, 0.06 and 0.08.

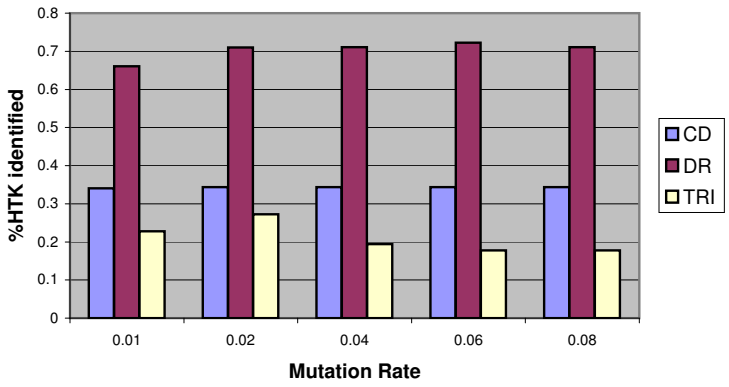

Figure 6.47: Effect of varying the mutation rate on the mean number of HTK mutants identified for all three programs.

Theoretically, the number of HTK mutants killed is related to the overall mutation score achieved - a high mutation score is likely to kill more HTK mutants than a low mutation score. It is interesting to see then, that this does not reflect in *CalDay*'s results. *CalDay* achieves a significantly lower mutation score in the higher (0.04, 0.06, 0.08) mutation rates than the lower rates, and yet the number of HTK mutants killed remains constant (only differing slightly for `mutRate` 0.01). It is probable that some HTK mutants are more easier to identify than others, and these are being killed in earlier iterations by all mutation rates, with the harder ones remaining alive. Furthermore, as mentioned in section 5.7, the hard-to-kill mutants were manually identified and therefore, potentially only represent a minimum number of HTK mutants. It is possible then, that the extra mutants

killed by the lower mutation rates should be marked as HTK also. If this were the case, then the percentage of HTK identified for `mutRate`s 0.01 and 0.02 would probably be greater than for the higher mutation rates, and reflect the original relationship between mutation score and HTK identified.

A further interesting result is those for *TriangleSort*. A mutation rate of 0.01 achieves the lowest mutation score after 500 iterations, however it identifies the second largest percentage of HTK mutants (0.02 killing the most). In general, a high mutation score implies more HTK mutants must have been killed. A low mutation score however, does not have to imply a low number of HTK were killed. A test that kills a HTK mutant may only kill a few non-HTK mutants, emphasising the speciality of test required to kill that HTK mutant. In general then, a test set focussed towards killing HTK mutants will kill many HTK, but may only achieve a low overall mutation score. It is likely then, that for *TriangleSort* at least, the mutation rate of 0.01 is focussed more towards killing HTK mutants. This is not to be unexpected, as a low mutation score will lessen the amount of test mutation occurring each iteration, thereby directing the search for niche tests.

In general though, the results suggest a mutation rate of 0.02 is preferential, coinciding with the preferred rate for achieving a higher mutation score.

Effect on the Number of Tests

All three figures (3.95-3.97 in Appendix C.1.12) exhibit interesting qualities for the initial mutation score increases - they show increases in mutation score without any increase in the number of tests (indicated by the horizontal lines). This is a limitation in the way the results are calculated. Higher values of `mutRate` generate more new tests per iteration than lower rates, and therefore are more likely to achieve a higher initial mutation score increase - they effectively "skip" over the lower mutation score increments achieved by lower valued `mutRate` experiments. The number of tests in the memory set are plotted on the graphs at every mutation score across all `mutRate` valued experiments, however it is the number of tests needed to achieve *at least* that mutation score that are plotted. For the larger

`mutRate` values, this usually means (for the first few points, at least) plotting the number of tests to achieve some higher mutation score at a lower one - hence a horizontal line.

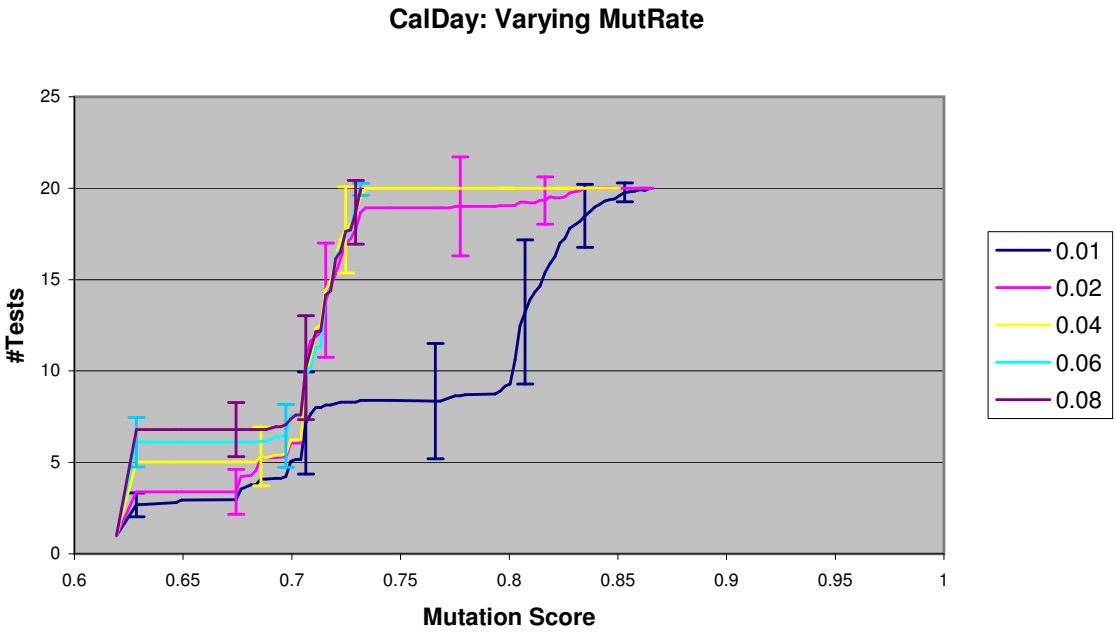

Figure 6.48: Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *CalDay* program.

After these initial discrepancies however, *DateRange* and *TriangleSort* generally show no significant difference in the number of distinct tests needed for a specific mutation score. As the number of distinct tests in an individual increases, modifying these tests is unlikely to change their distinctiveness. Higher mutation rates, although more likely to mutate tests, no longer significantly increase the number of distinct tests. On the other hand, *CalDay*'s results (figure 6.48) indicate discrepancies in the number of tests produced for different rates - the biggest difference occurring between the rate of 0.01 and the others. It would appear that this low rate is capable of finding tests to increase the mutation score from approximately 70 to 80%, without creating a large number of distinct tests to do so. Or alternately, the higher mutation rates generate more distinct tests before they are able to perform the same mutation score increase. This effect is, more

than likely, peculiar to this particular program, highlighting the fact that this automated technique is dependent on the program being tested, and the methods used to generate/mutate tests. A reasonable conclusion would be to assume that the mutation rate has no effect on the number of tests needed, especially considering the fact that the number of tests are limited by the individual's size anyway.

6.4 Immune Inspired Algorithm for Mutation Testing

The Immune Inspired Algorithm for Mutation Testing, as defined in figure 4.21, has 3 user-definable parameters (excluding population size and the number of iterations to perform): `nFittest`; `nWorst`; and `cloneRate`. Table 6.23 shows the values assignable to each parameter, with the default options highlighted in **bold**. There are 15 possible combinations of parameters, and therefore 15 experiments: the default plus: 4 for `nFittest`, 5 for `nWorst`, and 5 for `cloneRate`. As with the GA/MT, these values are chosen with execution times in mind.

Parameter	Values
`nFittest`	1, 3, **5**, 7, 9
`nWorst`	0, 1, 3, **5**, 7, 9
`cloneRate`	0, 1, 5, **10**, 15, 20

Table 6.23: Possible parameter values for the Immune Inspired Algorithm. Default values are shown in bold font.

6.4.1 nFittest

The code in figure 4.21 (Chapter 4) indicates that the `nFittest` parameter is used for two reasons. Predominantly it is used to define the number of tests to be selected (from either the memory set or the current population) to undergo clonal selection. The other use is in the metadynamics method at the end of

every iteration - the `nFittest`[5] worst tests (i.e. `nFittest` tests that kill the least mutants) are replaced by the `nFittest` best tests from the clones. Primarily because of the first use, it is expected that this parameter will affect the execution time of the algorithm, with execution times increasing as `nFittest` does. It is also expected to affect the mutation score achieved after a specific number of iterations and the number of hard-to-kill mutants identified, with both increasing as `nFittest` does. No effect is expected in the test set size. Reasoning behind these expectations will be discussed more in the following subsections.

Effect on the Number of Mutant Executions

The metadynamics phase replaces the `nFittest` worst tests with the `nFittest` best tests from the cloned and mutated test set. As such, it does not cause the creation (either through mutation or otherwise) of any new tests, and so will not affect the number of mutant executions. Instead, the use of `nFittest` for defining how many tests are selected for clonal selection is more important - the more clones generated, the more mutated tests are likely to occur (assuming a specific mutation rate). Theoretically then, as every mutated test will need to be executed on all mutant programs in order to determine its affinity, the higher the value of `nFittest`, the more executions that will be needed to achieve a certain mutation score - the longer the execution time.

Figures 6.49-6.51 plot the mean number of mutants executed to achieve at least the specified mutation score. The *DateRange* program, and to a lesser degree the *CalDay* program, both demonstrate the proportional relationship between `nFittest` and the number of mutant executions at high mutation scores. This relationship is not seen in the *TriangleSort* program however, which instead hints that the number of mutant executions is inversely proportional to `nFittest`. It should be noted though, that the difference in mutant executions between the highest and lowest `nFittest` values (at approximately the highest mutation score)

[5]It helps to think of `nFittest` purely as a number in this case and not as "n fittest".

is relatively small for *TriangleSort* (≈ 0.5-0.6 million) when compared with *DateRange* (≈ 1.5 million) or *CalDay* (≈ 4 million), reducing the significance that should be placed on trends derived from *TriangleSort*'s results.

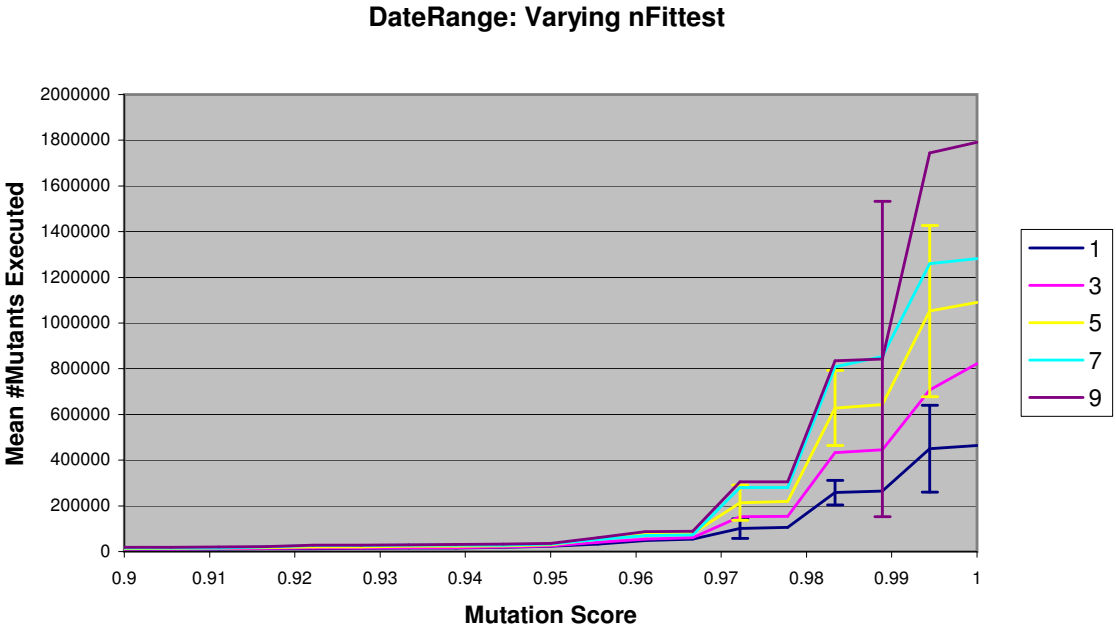

Figure 6.49: Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *DateRange* program.

Table 6.24 indicates, for each program, the mean number of mutant executions (and standard deviation) needed to achieve (at least) the highest mutation score obtained by at least 25 runs, for each of the five `nFittest` values. Analysis of variable (ANOVA) tests were calculated on these values to indicate whether any of the `nFittest` values result in significantly different means. Which specific combinations of `nFittest` values have significantly different means is analysed by way of Scheffé post-hoc analysis.

All three programs exhibit signs of significantly different mean numbers of executions, although *TriangleSort*'s result is only based on a significant difference between an `nFittest` pair of 3-9. *DateRange* on the other hand, shows a significance between `nFittest` pairs of 1-7, 1-9 and 3-7, whilst *CalDay* shows a significance between pairs of 1-7, 1-9, 3-7, 3-9 and 5-7. Despite these results, there

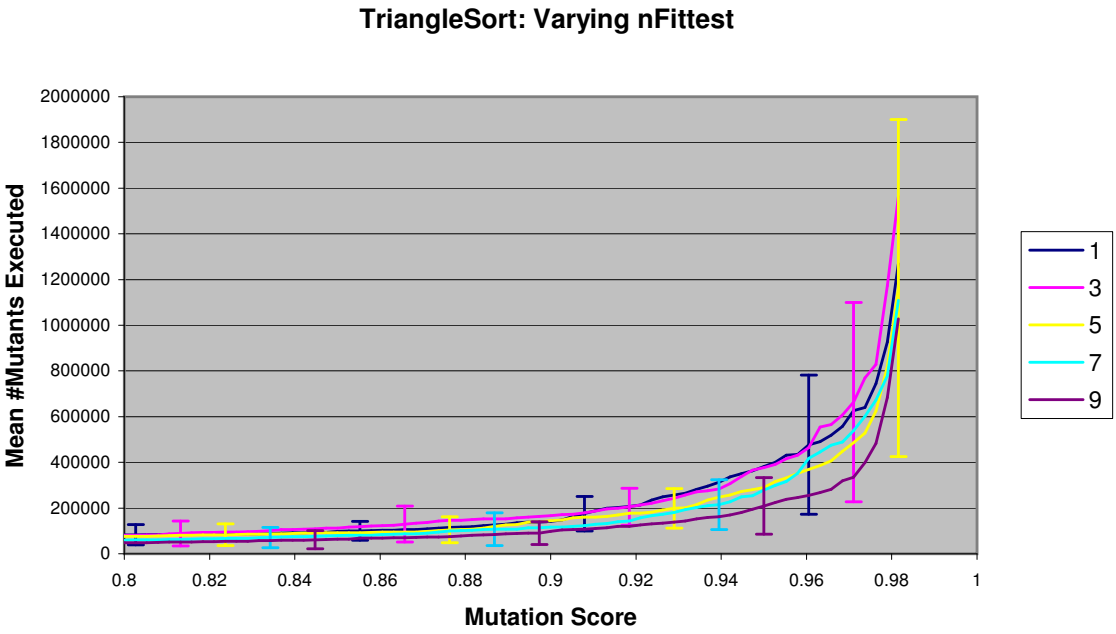

Figure 6.50: Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *TriangleSort* program.

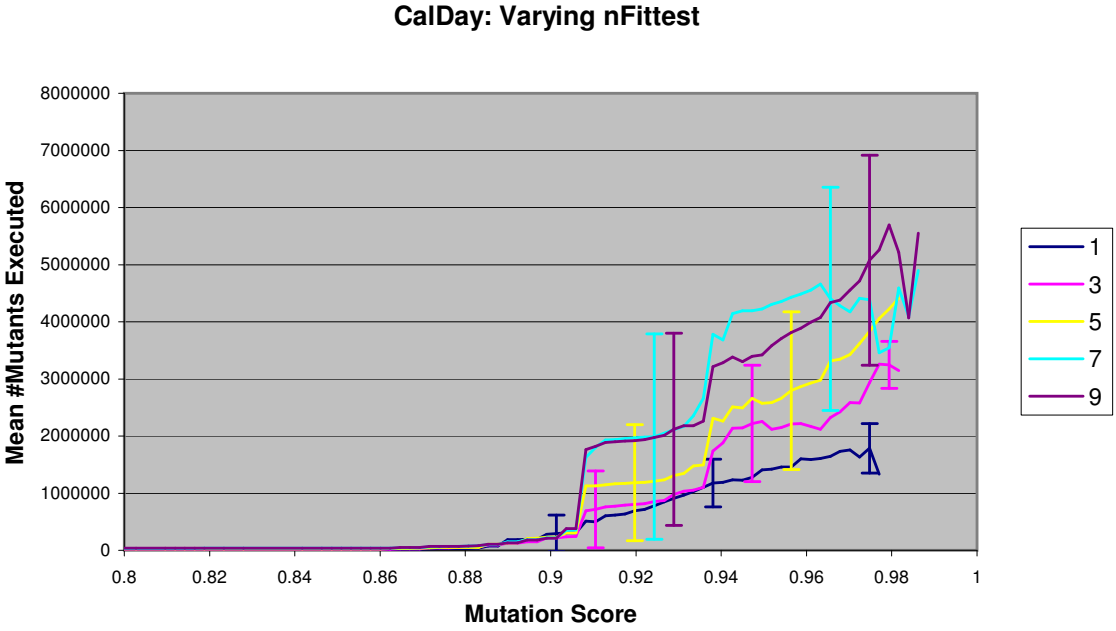

Figure 6.51: Effect of varying `nFittest` on the mean number of mutant executions to achieve a specific mutation score for the *CalDay* program.

`nFittest`	DR MS: 98.89%	TRI MS: 97.63%	CD MS: 93.58%
1	265388 ± 71.55%	744779 ± 44.02%	1104351 ± 43.33%
3	444814 ± 68.51%	829200 ± 55.26%	1102104 ± 50.60%
5	642955 ± 58.23%	624068 ± 51.53%	1493328 ± 70.65%
7	853806 ± 86.82%	673662 ± 71.74%	2658207 ± 67.37%
9	842544 ± 82.66%	482196 ± 63.11%	2261889 ± 66.10%
ANOVA			
f_{obt}	**7.56**	**3.76**	**10.94**
f_{crit}	2.43	2.43	2.43
Scheffé			
c_{obt}			
1-3	1.37	0.82	0.01
1-5	2.86	1.18	1.32
1-7	**4.46**	0.69	**5.15**
1-9	**4.41**	2.59	**3.83**
3-5	1.51	2.15	1.34
3-7	**3.13**	1.61	**5.19**
3-9	3.06	**3.67**	**3.87**
5-7	1.60	0.52	**3.92**
5-9	1.53	1.52	2.58
7-9	0.09	2.02	1.30
c_{crit}	3.12	3.12	3.12

Table 6.24: The mean number of mutants executed (and standard deviation as a percentage of the mean) at the highest mutation score obtained by at least 25 runs for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nFittest` values result in significantly different (at the 0.05 level) mutant execution numbers (in **bold**). *The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. All other results are to 2 decimal places.*

are two points which should be considered when drawing conclusions. Firstly, the Scheffé c_{obt} values are not much larger than the c_{crit} values - this is particularly the case for the *TriangleSort*'s results which could be just peculiar to this program. Secondly, all three programs indicate a large standard deviation in the mean values, indicating a large spread in the number of mutant programs executed (and therefore the execution time) to achieve the specified mutation score. Both points place doubt as to the significance of the results.

Overall, and considering how the results presented in the graphs extend beyond the mutation scores in table 6.24, *DateRange* and *CalDay* suggest a proportional relationship between the number of mutant executions and `nFittest`. This is particularly noticeable between larger differences in `nFittest` values, e.g. between 1 and 9. *TriangleSort* however, hints at an inversely proportional relationship although the result is barely significant at the 0.05 level. If it is significant, then *why should this program present different results?* Given the few programs tested it is impossible to ascertain a definitive reason; one possible explanation however relates to the average affinities of the tests added to the memory set. Figures 6.52-6.54 indicate the average test affinities generated for each mutation operator and the test set size, at each `nFittest` value, for each program. Figure 6.55 shows the same for the *Select* program, but only for the single value of `nFittest` (=1) tested (included for the discussion to follow). Considering only the main three programs being examined, these graphs indicate that *TriangleSort* has a relatively larger test set size than the other two programs, but that the average test affinities (per mutation operator) are smaller. They also indicate that the set sizes and average affinities are approximately constant across all `nFittest` values.

The number of clones generated per parent is proportional to the parent's affinity. Low affinity tests, such as those in *TriangleSort* will produce few clones, however a high percentage of these will mutate to produce new tests (test mutation is inversely proportional to affinity). As `nFittest` does not appear to affect the average affinities, it will not affect the number of new tests generated per parent. It will however, affect the number of parents generated (number proportional to

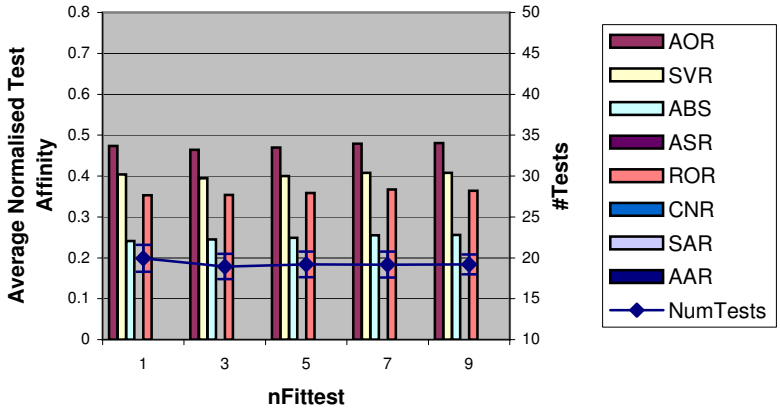

Figure 6.52: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program.

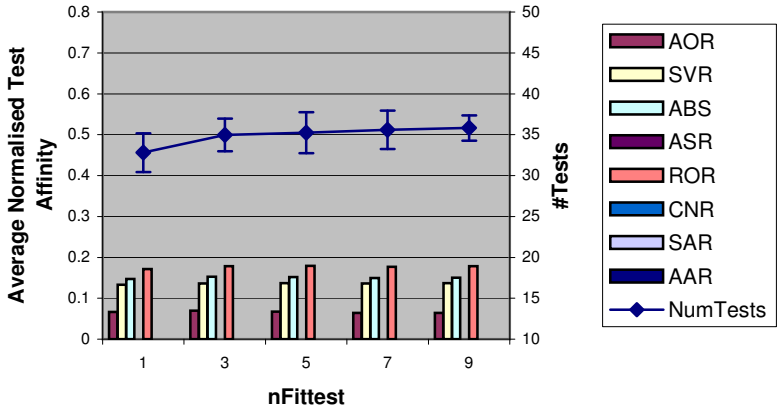

Figure 6.53: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program.

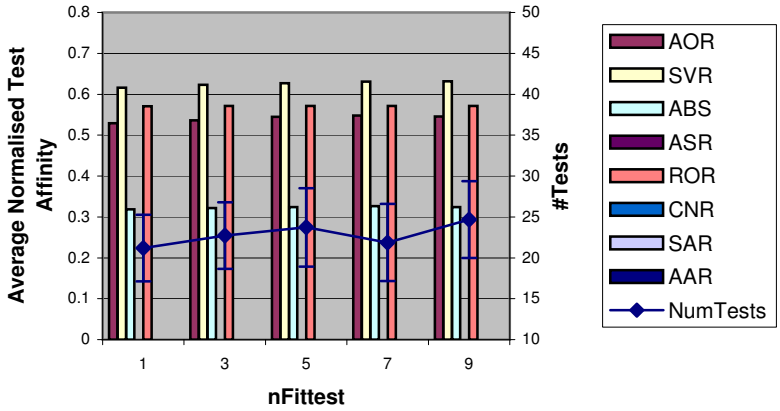

Figure 6.54: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program.

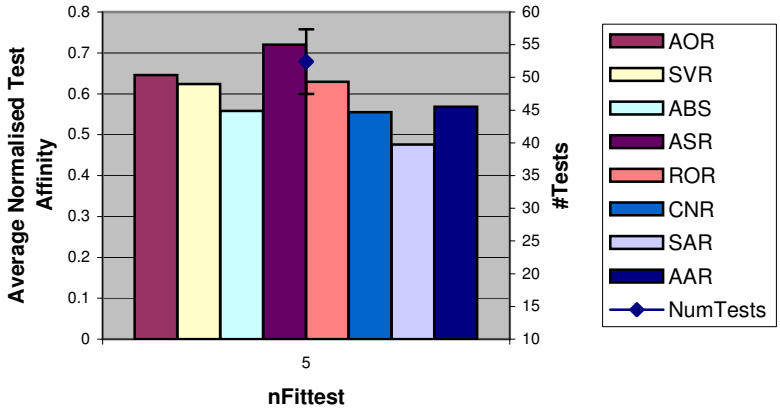

Figure 6.55: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *Select* program.

`nFittest`), and thus affect the **total** number of new tests created per iteration. For *TriangleSort*, an `nFittest` of 1 will generate very few new tests, implying more iterations will be needed (and therefore more mutant executions) in order to improve the mutation score. Increasing `nFittest` will increase the number of new tests per iteration, decreasing the number of iterations (and therefore the number of executions) needed to achieve a mutation score. In effect, increasing `nFittest` means increasing the coverage of the localised shape-space per iteration.

If, on the other hand, *TriangleSort*'s results are not considered significant due to the large variance (and this is entirely reasonable given how little the ANOVA F_{obt} result is greater than the F_{crit} value - table 6.24), then the above explanation could still be true except that the parameter values tested do not cause a large enough difference in execution numbers to warrant a significant result. Larger `nFittest` values may cause a significant result, however it is envisaged that there will be a turning point where values larger than this point will not offer any improvement in mutation score and instead significantly increase the number of mutant executions needed - the process will effectively become similar to what happens with *DateRange* and *CalDay*. For these latter two programs, the higher average affinities will cause a significant number of new tests to be generated with an `nFittest` of 1, which are adequate enough for generating a high mutation score. If `nFittest` is increased, the number of new tests generated also increases, but these appear to have little effect on improving the high mutation scores, and instead only increase the number of executions.

The question that this poses is *why does TriangleSort have lower average affinity tests than the other programs?* With few programs tested, it cannot be determined if *TriangleSort* is just an exception to the rule. Assuming it is not, then one hypothesis would be that programs with high complexities create a larger number of mutants that require more subtle variations of tests to identify them. These tests will probably only kill a few mutants and so more will be needed, resulting in larger test sets with lower average affinities. Typically, high complexity programs often posses a large number of branch conditions that serve to alter the program's

control flow. Mutant variations of these conditions often need slightly differing inputs to detect each one, for example, consider the *TriangleSort* program (in Appendix B.4). The first two relationship operator (ROR) mutations adjust line (line number: 4) `if((i<=0)||(j<=0)||(k<=0)){` to:

```
if((i<0)||(j<=0)||(k<=0)){
if((i==0)||(j<=0)||(k<=0)){
```

The first mutation can be identified by the inputs: $i = 0$; $j = 1$; and $k = 1$ (original returns 4, mutation returns 2). These values do not work for the second mutation however, but a small variation of them does: $i < 0$; $j = 1$; and $k = 1$ (original returns 4, mutation returns 2). In particular, this highlights the importance of the relationships between the program's inputs. Generating a random test in this instance, is likely to lose the important equality between j and k, meaning even if i is created as a negative number, the test will still not help. In these cases then, there is little need for random mutation, with the localised search being more beneficial.

By requiring specific, subtle tests for certain mutants, the final test set is likely to be larger but consisting of similarly structured, lower mutation scoring tests. Each test kills a small, subtly different set of mutants, meaning more tests are needed to kill all the mutants generated for a program. A benefit of this hypothesis is that a program's complexity can be calculated using the McCabe cyclomatic complexity measure [70], and used to indicate the appropriate `nFittest` value to test the program with - specifically, higher McCabe complexities require larger `nFittest` values.

Select's results (figure 6.55) seemingly contradict this hypothesis though. *Select* has the highest McCabe complexity (19) of all the programs tested and so, as hypothesised, has a high number of memory tests. However, it also has a high average mutation score per test (for each mutation operator) implying that high complexity programs do not always need a lot of *low mutation scoring tests* to achieve a high overall mutation score. This evidence does not necessarily mean

that the hypothesis is wrong however, only that the McCabe complexity may not be a solely reliable indicator for an appropriate `nFittest` value. The results show that, as expected by the theory, *Select* still requires a large number of tests to achieve a high overall mutation score, the majority of which are generated through cloning rather than random introduction (evidenced in table 5.18) - most tests are therefore variations of other useful tests. If a larger number of "subtle mutants"[6] are still created for the *Select* program, then this suggests that each test only kills a few subtle mutants, therefore requiring the larger test set to achieve a high mutation score. Because the total number of mutants is greater for *Select* however (see table 2.30 in Appendix B), the proportion of subtle mutants to "non-subtle" mutants is lower. Each test, needed to kill particular subtle mutants, is also capable of killing a large number of non-subtle mutants, giving each test a high average mutation score. These results suggest that it is the proportion of subtle mutants created that is important for determining the average test affinity, and that this is evidenced by a program's complexity and the number of mutants it creates.

As an aside, only one *Select* experiment (repeated 39 times) was performed[7], with an `nFittest` of 1. If the above results hold true for this program, and the average test mutation score remains at the same high level for all `nFittest` values, then it would be expected that `nFittest` should affect the number of mutant executions in a manner similar to *DateRange* or *CalDay* - the number of executions should be proportional to `nFittest`.

It should be noted that this expectation on the *Select* program and the overall hypothesis cannot be verified given the number of programs tested. Further results are needed, particularly from high complexity (McCabe complexity > 10) programs. This unfortunately requires a significant amount of time to test. On the basis of the observed results however, a smaller `nFittest` value seems preferential.

[6]Subtle mutants are mutants that can be identified though subtle changes to tests.

[7]Only one Select experiment was performed because of time constraints - each run of the Select program takes approximately 4 days on a state-of-the-art PC (Intel P4 2.2GHz, 1Gb RAM).

Effect on the Mutation Score

Figures 3.105-3.107 in Appendix C.2.2 plot the mutation scores achieved at every iteration. Figure 6.56 (repeated from the appendix) gives the clearest indication of a trend between the mutation score per iteration and the `nFittest` value - a large rate of increase in the mutation score, proportional to `nFittest`, during the initial iterations, before tapering off to a small rate of increase in the later iterations. Whilst the other two programs demonstrate the large mutation score increase to start followed by a tapering off, neither conclusively indicate a clear relationship with `nFittest`. Both hint at a proportional relationship, at least in the first 50 iterations, with the mutation scores achieved with an `nFittest` of 9 being greater than those for an `nFittest` of 1, but these results do not have the clarity of *TriangleSort*'s results.

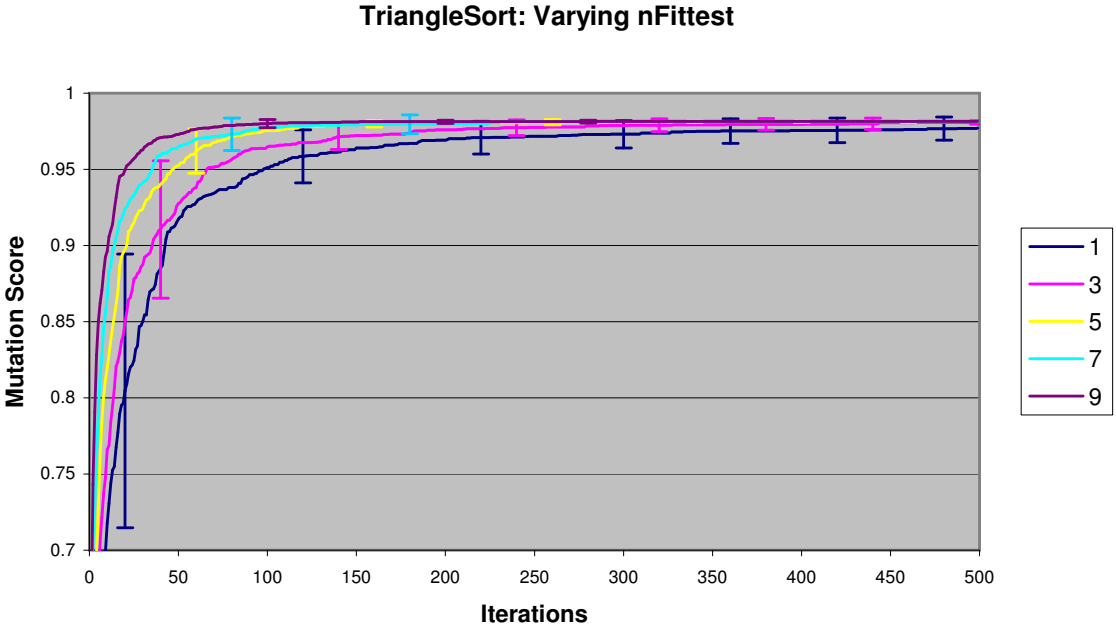

Figure 6.56: Effect of varying `nFittest` on the mean mutation score achieved per iteration for the *TriangleSort* program.

Table 6.25 indicates, for each program, the final mutation score attained after 500 iterations using each of the five `nFittest` values. As can be seen, all values result in very high mutation scores, especially *DateRange* which achieves in excess of

99.5% for all settings. Furthermore, the standard deviation is low ($<\approx 2\%$) across all programs indicating that the majority of the runs achieve these high scores after 500 iterations. The ANOVA results suggest that each program demonstrates significant differences amongst at least one pair of mean results, with *TriangleSort* providing the strongest significance. Scheffé calculations indicate that, for *TriangleSort* and *CalDay* at least, an `nFittest` of 1 generates a mean mutation score significantly different from an `nFittest` of 9. When the mean mutation scores for these `nFittest` values are compared, a setting of 9 achieves a higher score for all three programs, offering evidence that a proportional relationship (between `nFittest` and mutation score) continues after 500 iterations. As the difference in mutation scores between these two values is so low however, and if the variance is taken into consideration, then the claim that this relationship continues for 500 iterations becomes weaker.

What explanation is there for these observations? As mentioned in the previous subsection (effect on the number of mutant executions), the `nFittest` parameter directly affects the number of tests selected for clonal selection - i.e. increasing it will increase the number of tests and therefore the number of clones. A proportion of these clones will undergo mutation at a rate inversely proportional to each clone's affinity, resulting in a number of new tests each iteration; the number of new tests can be estimated by: $(1 - \mathit{clone_affinity}) * \#\mathit{clones}$. If the average clone's affinity is assumed constant for any given program across varying values of `nFittest` (as indicated by figures 6.52-6.54), then increasing the number of clones (by increasing the number of selected parent tests - achieved by increasing `nFittest`) will increase the number of new tests. As each new test per iteration essentially increases the chance of killing further mutants, increasing `nFittest` will therefore improve the chance of killing more mutants in any given iteration.

In early iterations there are a large number of easily killed mutants still living. The larger number of new tests produced by larger `nFittest` values are likely to kill more mutants than the smaller number from lower `nFittest` values, resulting in a higher mutation score. This can easily be observed in the early iterations

nFittest	DR	TRI	CD
1	99.89% ± 0.34%	97.70% ± 0.78%	94.90% ± 1.67%
3	99.57% ± 0.55%	98.12% ± 0.17%	95.78% ± 1.75%
5	99.78% ± 0.45%	98.15% ± 0.00%	96.18% ± 1.84%
7	99.89% ± 0.34%	98.16% ± 0.00%	95.41% ± 2.02%
9	99.96% ± 0.20%	98.16% ± 0.00%	96.56% ± 1.87%
ANOVA			
f_{obt}	**4.69**	**10.55**	**4.07**
f_{crit}	2.43	2.43	2.43
Scheffé			
c_{obt}			
1-3	**3.16**	**4.53**	1.97
1-5	1.09	**5.28**	2.86
1-7	0.00	**5.30**	1.13
1-9	0.75	**5.41**	**3.65**
3-5	2.06	0.69	0.88
3-7	**3.16**	0.77	0.81
3-9	**3.94**	0.78	1.70
5-7	1.09	0.09	1.67
5-9	1.85	0.09	0.84
7-9	0.75	0.00	2.47
c_{crit}	3.12	3.12	3.12

Table 6.25: The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nFittest` values (in **bold**) result in significantly different mean mutation scores. *All results are to 2 decimal places.*

of figure 6.56. As the easy tests are killed though, improving the mutation score becomes harder. As before though, a larger number of new tests per iteration are more likely to find a test to kill some living mutants than a smaller number - a higher `nFittest` is preferential.

Effect on the Number of HTK identified

Figure 6.57 indicates the number of hard-to-kill mutants identified after 500 iterations using each `nFittest` value, for each of the 3 programs. These results suggest that, unlike expected, the percentage of HTK mutants identified is not proportional to `nFittest`.

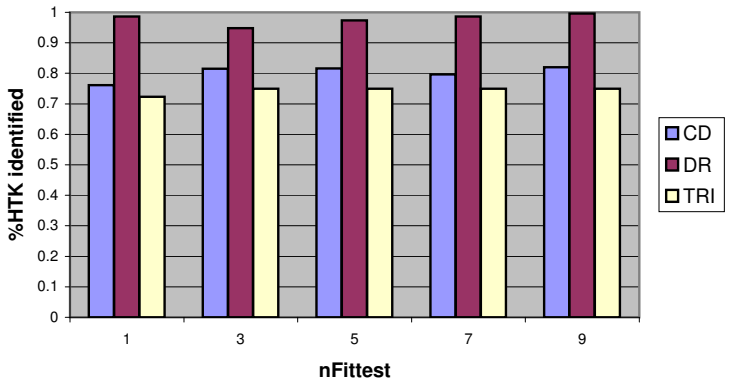

Figure 6.57: Effect of varying `nFittest` on the mean number of HTK mutants identified for all three programs after 500 iterations.

It was expected that the mutation score would be proportional to `nFittest`, and in the early iterations this was shown to be the case. High mutation scores imply more HTK mutants must have been killed, and so this mutation score expectation suggested that the percentage of HTK identified should also be proportional to `nFittest`. The previous subsection however, showed that the mutation score after 500 iterations was not significantly proportional to `nFittest`, implying that neither will the number of HTK identified - this is the observed outcome. Furthermore, the previous section indicated that, although not significant for all programs, the mutation score achieved for an `nFittest` of 9 was always larger than for an `nFittest of 1`. This results is also reflected in the percentage of HTK killed, with all three programs killing more HTK mutants for an `nFittest` of 9 than for 1. Similar to the effect on the mutation score however, this result is only (just!) statistically significant for the *CalDay* and *TriangleSort* program (indicated by the **bold** results in table 3.44). On this basis, a higher `nFittest` would be preferential, although possibly offering little in regards HTK mutant identification.

Effect on the Number of Tests

Tests are added to the IIA/MT's memory set if they kill a mutant not already killed by the current memory set; no tests are ever removed. To affect the number of tests then, a parameter would have to alter the 'quality' (measured by mutation score) of the tests being added, e.g. to increase the set size, all tests would have to kill only a few mutants each, requiring more tests to kill the same total number of mutants. As shown by the results in figures 6.52-6.54, the *nFittest* parameter does not do this. Instead, it appears that the program itself is a deciding factor in the average mutation score of each memory test, with the hypothesis being that higher complexity programs have larger memory sets with the average test mutation score depending (inversely) on the number of mutants.

The results in figure 6.58 summarise the results for the other two programs (see figures 3.108-3.110 in Appendix C.2.4). All three programs clearly show that the number of tests in the IIA/MT's memory set is unaffected by the value of `nFittest`, and it is instead the programs themselves that result in different sized test sets.

As an aside, the graph for *DateRange*, and to a lesser extent the one for *CalDay* too (see figures 3.108 and 3.110 in Appendix C.2.4), indicate the same initial quality, discussed in the `mutRate` subsection - an increase in mutation score without any increase in the number of tests (indicated by the horizontal lines). This happens for two reasons. Firstly, a higher `nFittest` typically presents a larger increase in mutation score for the first few tests added (to the memory set). Secondly, it is the number of tests needed to achieve *at least* a particular mutation score, for all mutation scores achieved by all `nFittest` values that are plotted - i.e. results are still plotted for low mutation scores even though the higher `nFittest` valued experiments "skip" over them.

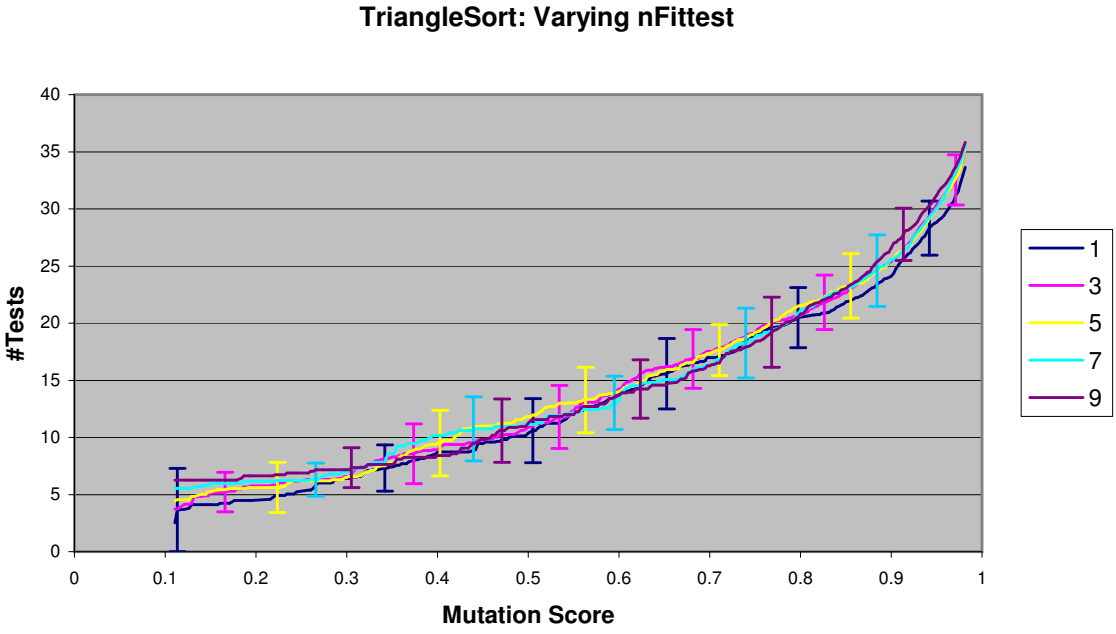

Figure 6.58: Effect of varying `nFittest` on the mean number of tests needed to achieve a specific mutation score for the *TriangleSort* program.

6.4.2 nWorst

The `nWorst` parameter is used solely in the final metadynamics phase, to remove the `nWorst` individuals (tests) who have the lowest mutation score and replace them with randomly generated individuals. Larger values of `nWorst` therefore replace a larger number of poor individuals with completely new ones than lower `nWorst` values. It is this metadynamics process that acts as a global search of the shape-space, and should be encouraged (to a degree) in order to find new tests to kill living mutants. Too much encouragement however (i.e. by replacing a large proportion of the population with random ones), would be detrimental to the local search - the best clones from one generation are kept for the next; replacing a large proportion of the population with random ones may overwrite these descendants (especially if they have a low affinity), destroying a possibly valuable search path; this should be avoided. Because of this parameter's influence on the number of random tests introduced, it is to be expected that `nWorst` will affect the number of mutant executions. Further impact is also expected on the mutation score per

iteration and the number of HTK mutants identified (both metrics increasing as `nWorst` does). No effect is expected on the test set size. These expectations will be discussed more in the following subsections.

Effect on the Number of Mutant Executions

Each iteration, the metadynamics phase replaces the `nWorst` tests in the population with randomly generated ones. This serves two purposes. Firstly, it enables the global search for new tests that kill living mutants, allowing a multimodal shape-space to be searched. Secondly, it helps prevent the search becoming isolated in a local maxima - randomly introducing tests will encourage new areas of the shape-space to be searched which may outperform this current area.

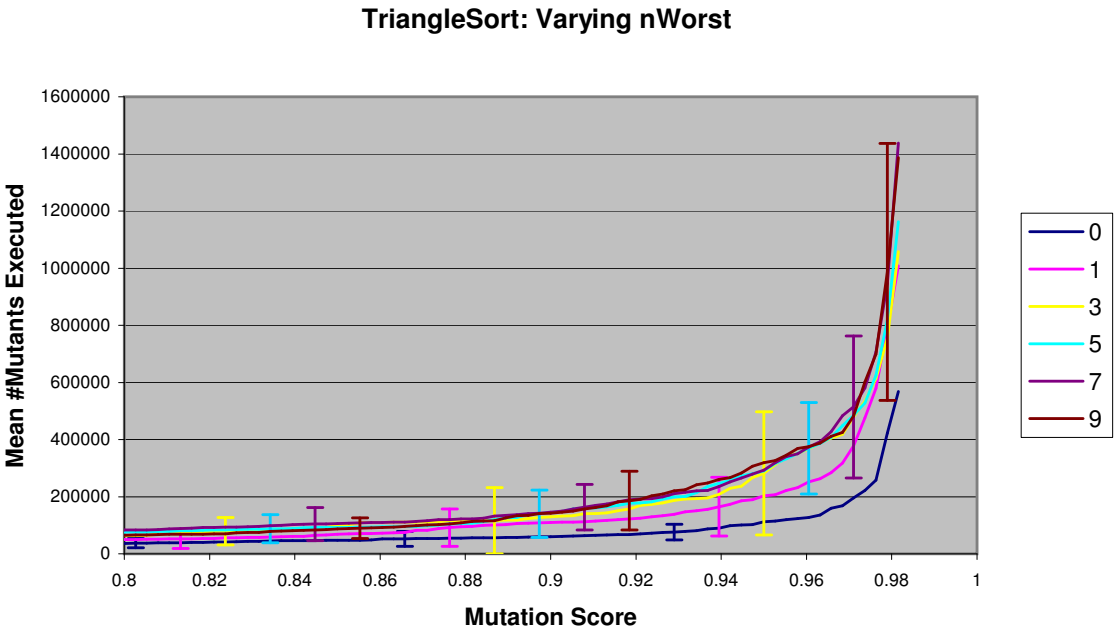

Figure 6.59: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

Figure 6.59 plots the mean number of mutants executed to achieve at least the specified mutation scores for *TriangleSort*, and suggests that the number of executions is roughly proportional to the value of `nWorst`. Given `nWorst` directly

affects the number of new, randomly created tests, it is unsurprising that this relationship should exist. As the value of `nWorst` is increased towards the population size, the amount of local searching would be reduced. Each generation's population would effectively consist of larger numbers of randomly generated tests, and so the algorithm would be limited to performing only a random search of the shape-space. For the sake of improving the mutation score, this is something that should be avoided, although for the low values of `nWorst` chosen for these experiments, this is not an issue.

If, going the other direction, the value of `nWorst` was reduced to 0, then no random tests would be introduced each iteration. No global search would be conducted, and instead improvements in mutation score would rely on finding new tests solely through the local search mechanisms instigated by cloning and mutation (see section 3.3.3). The likelihood is an `nWorst` of 0 will restrict the mutation score achievable within 500 iterations; an effect that can clearly been seen in the *DateRange* and *CalDay* results, shown in figures 6.60 and 6.61 respectively. At best (i.e. in all experiment runs), *DateRange* achieves a mutation score of 95.56% (2 d.p.) for an `nWorst` of 0, compared with a full mutation score for the other `nWorst` values. *CalDay* achieves, at best, a mutation score of 72.94% (2 d.p.) for an `nWorst` of 0, compared with at least 97.48% (2 d.p.) for the other values.

Interestingly, this effect is not observed in *TriangleSort*, with all `nWorst` values achieving a mean mutation score of 98.16% (2 d.p.). This implies that the random introduction of new tests serves no effect in improving the mutation score, but instead only increases the execution time of the algorithm (via increased mutant executions), as demonstrated clearly in figure 6.59. The observed effect also agrees with the results from `nFittest`, which suggested that the localised search is sufficient for improving *TriangleSort*'s mutation score. The same hypothesis as presented for `nFittest` could therefore be used to explain these results. The proportion of subtle mutants (influenced by the program's complexity and number of mutants) determines the appropriate balance of localised to global searching. Programs with a high proportion of subtle mutants (i.e. high complexity, low number

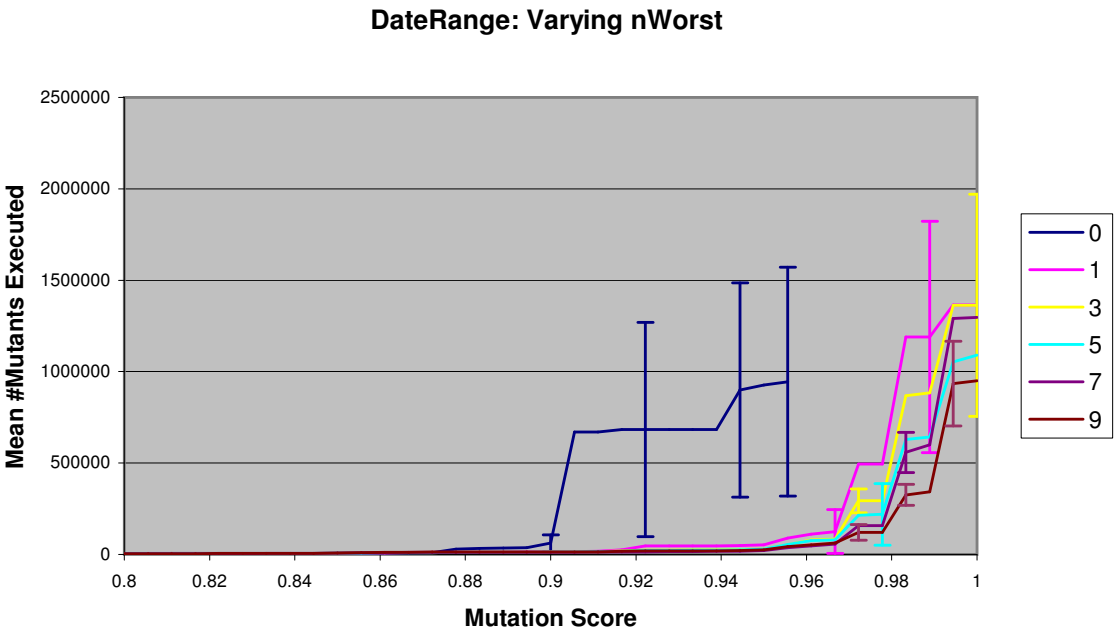

Figure 6.60: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

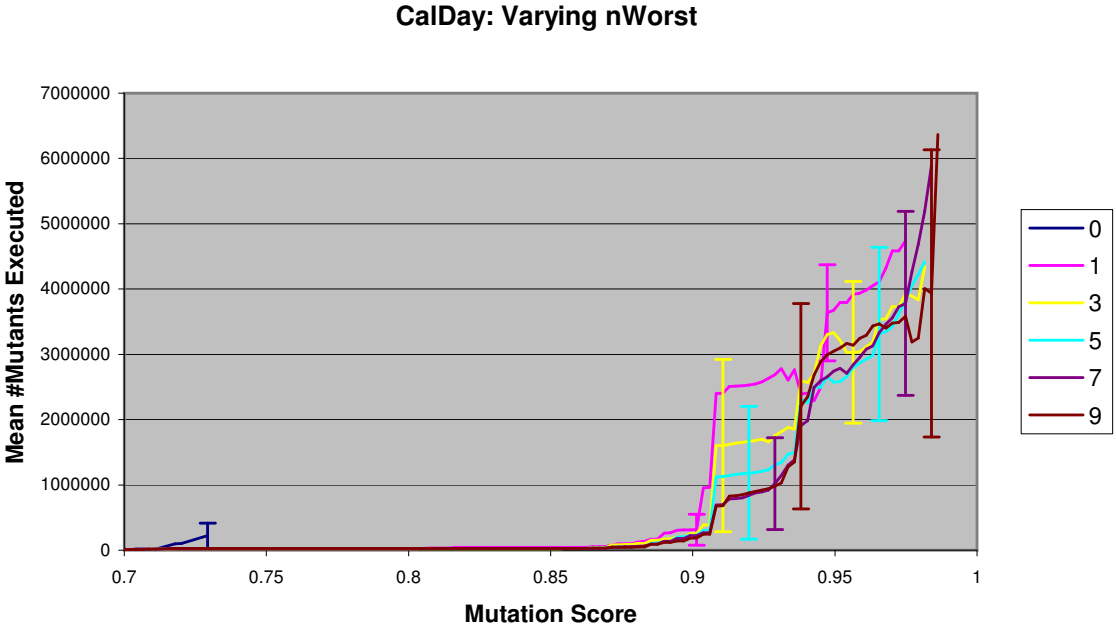

Figure 6.61: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

of mutants) rely primarily on local searching; programs with a lower proportion (i.e. low complexity or high number of mutants) require more global searching.

As `nWorst` affects the amount of global searching that occurs, this parameter will have most influence on the lower proportioned programs. For these programs, an `nWorst` of 0 will cause no global searching and will limit the mutation score achieved. Increasing the value of `nWorst`, at least within the range tested for these experiments, hints that for these programs a higher mutation score will be obtained in fewer executions. This is to be expected as such programs rely on global searching for finding tests to kill mutants, meaning that the more global searching that occurs, the more likely the mutation score is to improve. This result is not statistically significant however, with both *CalDay* and *DateRange* showing oscillations in the ordering of the results (e.g. for *DateRange*, an `nWorst` of 7 executes more mutants than a value of 5, to achieve mutation scores greater than 98.89% (2 d.p.)), as well as each value's results only differing by a relatively small amount (e.g. *CalDay*'s differ by approximately 1 million executions, compared with the 6 million required to achieve the highest mutation score).

The results from the higher proportioned *TriangleSort* suggest however, that a proportional relationship exists. As such programs place little emphasis on global searches, this result is also to be expected. Increasing `nWorst` does not improve the amount of local searching that occurs (only the shape-space [see Section 3.3.1] areas that are locally searched), and so will not improve the overall mutation score. Instead, by increasing the number of new tests added to each generation, increasing `nWorst` will only increase the number of mutant executions.

Interestingly, all the results obtained show that the mutation score achieved with an `nWorst` of 0 is proportional to the McCabe complexity of the program. *CalDay* (McCabe complexity 4) achieves a mutation score of at best, 72.94% (2 d.p.), *DateRange* (McCabe complexity 6) achieves 95.56% (2 d.p.), and *TriangleSort* (McCabe complexity 11) obtains 98.16% (2 d.p.). Extrapolating a conclusion from these results would be dangerous however, where instead more research is required with additional programs.

On the basis of the hypothesis suggested in the `nFittest` section, the appropriate value of `nWorst` seemingly depends on the program being tested. Programs with a low proportion of subtle mutants (low complexity or high number of mutants) hint that a higher `nWorst` value is able to achieve higher mutation scores in fewer executions. This result is not conclusive however, with more programs and a larger range of `nWorst` values requiring testing. On the other hand, programs with a high proportion of subtle mutants (high complexity and a low number of mutants) hint that lower `nWorst` values can achieve the same mutation scores as higher values, except that they execute fewer mutants. Given the uncertainty involved in these results however, the suggested parameter value is one greater than 0, with the default used in these experiments (5) seeming reasonable.

Effect on the Mutation Score

Increasing `nWorst` increases the number of new, random tests introduced each iteration. For programs with no strong reliance on localised searching, the more of these tests that are executed every iteration, the greater the chance of finding a test to kill living mutants, therefore the mutation score achieved per iteration should be proportional to `nWorst`. This effect can clearly be seen for the *DateRange* program in figure 6.62, and the *CalDay* program in figure 3.116 (Appendix C.2.6). Table 6.26 presents the mean mutation scores achieved after 500 iterations for each of the `nWorst` values, and an indication of which `nWorst` values result in significantly different scores (by way of Scheffé tests; significant results are in **bold**).

This effect is only common for two out of the three programs tested however. Figure 6.63 indicates the results for the *TriangleSort* program. This clearly shows that an `nWorst` of 0 achieves a high mutation score quicker than the other `nWorst` values, although overall the differences (between each value) are relatively small when compared against those for the other two programs. Having previously identified that the introduction of random tests has little effect on the number of mutant executions for this program however, it is no surprise that `nWorst` has little effect on the mutation score per iteration.

`nWorst`	DR	TRI	CD
0	93.17% ± 2.76%	98.16% ± 0.00%	72.94% ± 0.00%
1	98.70% ± 0.59%	98.14% ± 0.07%	93.00% ± 1.92%
3	99.50% ± 0.63%	98.16% ± 0.00%	95.47% ± 1.85%
5	99.78% ± 0.45%	98.15% ± 0.05%	96.18% ± 1.84%
7	99.85% ± 0.38%	98.15% ± 0.05%	96.91% ± 1.53%
9	99.93% ± 0.28%	98.15% ± 0.05%	96.78% ± 1.62%
ANOVA			
f_{obt}	**144.42**	0.70	**1186.46**
f_{crit}	2.27	2.26	2.26
Scheffé			
c_{obt}			
0-1	**17.69**	-	**52.09**
0-3	**20.40**	-	**58.06**
0-5	**21.12**	-	**59.88**
0-7	**21.36**	-	**61.28**
0-9	**21.60**	-	**60.44**
1-3	2.54	-	**6.32**
1-5	**3.41**	-	**8.13**
1-7	**3.64**	-	**9.93**
1-9	**3.88**	-	**9.53**
3-5	0.89	-	1.80
3-7	1.13	-	**3.63**
3-9	1.37	-	3.28
5-7	0.23	-	1.85
5-9	0.47	-	1.51
7-9	0.23	-	0.32
c_{crit}	3.37	-	3.36

Table 6.26: The mean mutation scores (and standard deviation) achieved after 500 iterations for each of the six `nWorst` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations show the which pairs of `nWorst` values (in **bold**) result in significantly different mean mutation scores. *All results are to 2 decimal places.*

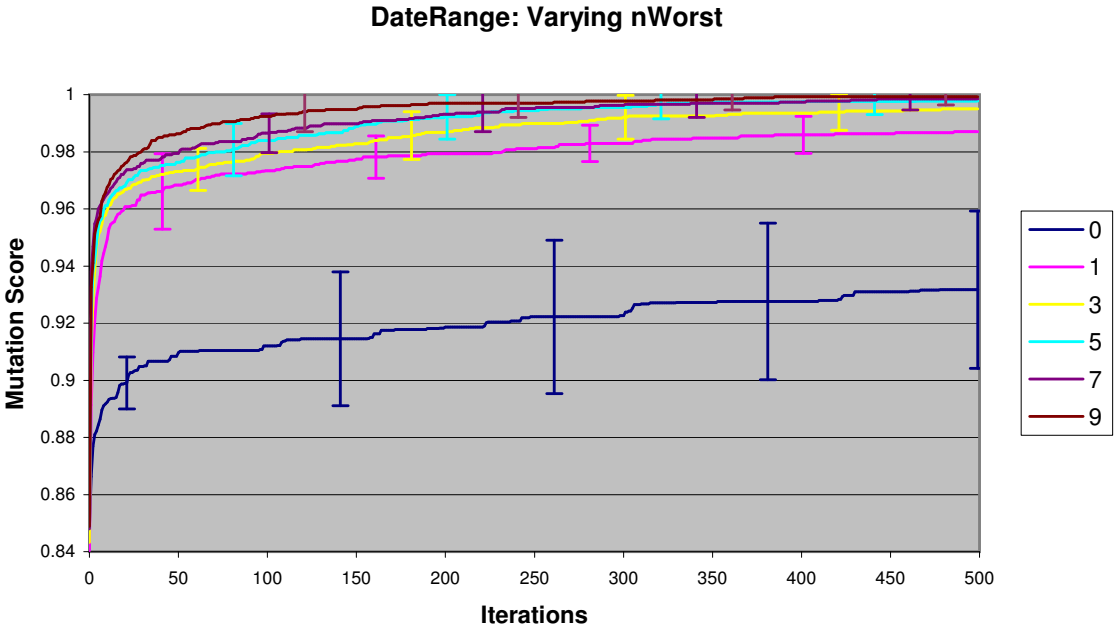

Figure 6.62: Effect of varying `nWorst` on the mean mutation score per iteration for the *DateRange* program.

As suggested in the previous subsection, an appropriate value for `nWorst` would be one greater than 0. In terms of the mutation score achieved per iteration, it would seem that for programs focussed on global searches, the mutation score per iteration is proportional to `nWorst`, and so a higher value should be favoured. For programs focussed on local searches (such as *TriangleSort*), `nWorst` appears to have little impact. Overall then, a higher `nWorst` value should be favoured. As previously mentioned however, increasing `nWorst` towards the population size will reduce the amount of local searching occurring. Excessively high `nWorst` values should therefore be avoided.

Effect on the Number of HTK identified

The results in figure 6.64 indicate, for *CalDay* and *DateRange*, the expected relationship between `nWorst` and the number of HTK identified - the number of HTK mutants killed is proportional to `nWorst`. Table 3.45 in Appendix C.2.7 (page 281) shows that the differences between `nWorst` values of 0, 1 and the others are

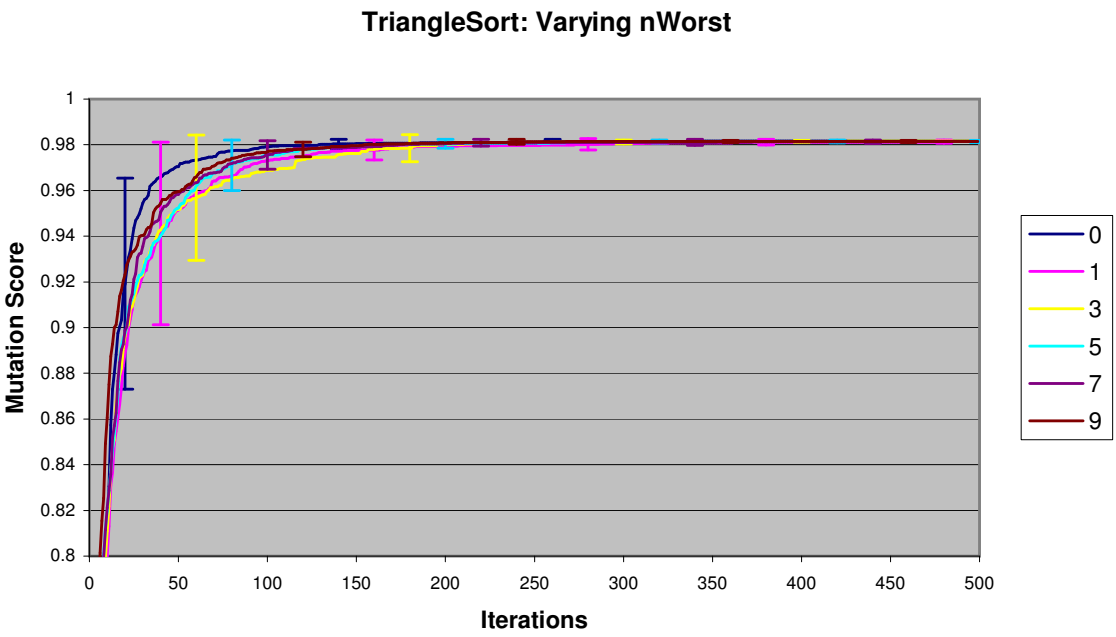

Figure 6.63: Effect of varying `nWorst` on the mean mutation score per iteration for the *TriangleSort* program.

significant (at the 0.05 level), giving support to this expected relationship. Figure 6.64 however, shows that this relationship does not hold for the *TriangleSort* program (with all values achieving the same result).

Given the relationship between mutation score and the number of HTK mutants identified (more HTK have to be killed to obtain a higher mutation score), it is not surprising that *CalDay* and *DateRange* have these results - in both cases the mutation score per iteration is proportional to `nWorst`. Neither are the results from *TriangleSort* unexpected; *TriangleSort* achieves approximately the same mutation score after 500 iterations, regardless of the `nWorst` value, and so, the number of HTK identified is also equal, as indicated by the relationship between mutation score and HTK identified.

Essentially then, according to the hypothesis from `nFittest`, programs requiring more global searches will achieve higher mutation scores with higher `nWorst` values, and so kill more HTK mutants. Programs focussed on local searches are unaffected by `nWorst` (in terms of mutation score achieved) and so the number of

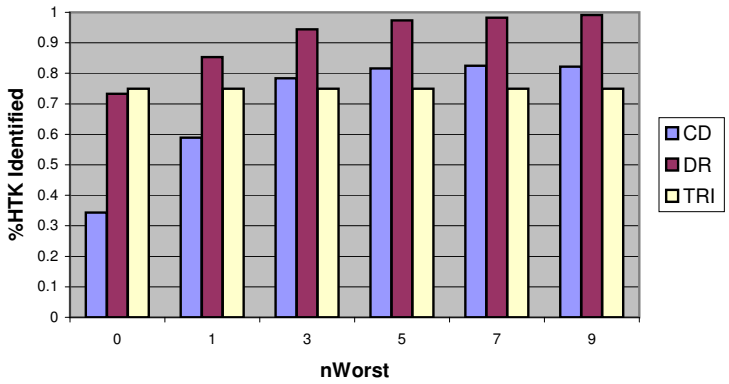

Figure 6.64: Effect of varying `nWorst` on the mean number of HTK mutants identified for all three programs after 500 iterations.

HTK is also unaffected. An `nWorst` greater than 0 is therefore generally preferred, with the default used in these experiments (5) being adequate.

Effect on the Number of Tests

Figures 3.117-3.119 in Appendix C.2.8 clearly indicate no effect on the test set size in relation to a varying `nWorst` parameter. As previously mentioned in the `nFittest` subsection, the number of tests in the IIA/MT's memory set will only be affected if the 'quality' of the tests is reduced. The `nWorst` parameter only affects the number of randomly generated individuals in the next iteration's population. It serves purely as a method to diversify the search results. Consequently, `nWorst` has no effect on the IIA/MT's memory set size.

6.4.3 cloneRate

The `cloneRate` parameter is used in the cloning phase to determine the number of clones produced per parent (with a minimum of 1 clone). It also indicates the maximum number of clones produced when the parent has an affinity of 1, e.g. the default `cloneRate` of 10 will produce 10 clones for any parent test with a full mutation score. Obviously then, the mutation score of the parent tests has an important influence on the number of clones, however this is not a user-definable

parameter[8].

The purpose of the clones is to produce a local search of the shape-space to try to improve upon the parent tests. More clones (i.e. tests) mean a greater chance of improving the overall mutation score, however they will also increase the number of mutant executions that occur. The `cloneRate` parameter could therefore be expected to increase the number of mutant executions to achieve a certain mutation score as it increases, although as `nFittest` highlighted, the average affinity (mutation score) of the tests seems to be an important factor in determining parameter effects. This is also true for `cloneRate`. Because of this, it is expected that the specified variations to this parameter will not significantly impact on the number of mutant executions for programs that generate low affinity tests, however they will cause a larger proportional difference for programs creating higher affinity tests. For similar reasons, it is expected higher affinity tests will also cause the mutation score per iteration to be proportional to `cloneRate`, as well as the number of HTK mutants identified. Little effect is expected on the test set size. Further reasoning will be given in the following sections.

Effect on the Number of Mutant Executions

The number of clones produced for any given test is calculated by: $\#clones_per_test = test_mutation_score * \texttt{cloneRate}$. Assuming a constant mutation score then, the number of clones generated will be proportional to the `cloneRate`. However, as theorised for `nFittest`, the program being tested affects the average mutation score of its tests. Programs with a low proportion of subtle mutants seem to generate tests with higher mutation scores, meaning the number of clones generated per test will be larger (for any given `cloneRate`). Programs with higher proportions of subtle mutants generate tests with lower mutation scores, resulting in a smaller number of clones per test.

[8]The results of section 6.4.1 and in particular figures 6.52-6.54 suggest that the program does influence the ability of tests generated. Some programs create low mutation scoring tests that combine to produce a high overall mutation score; other programs create high scoring tests. As this affects the number of clones generated, the program itself should be taken into consideration.

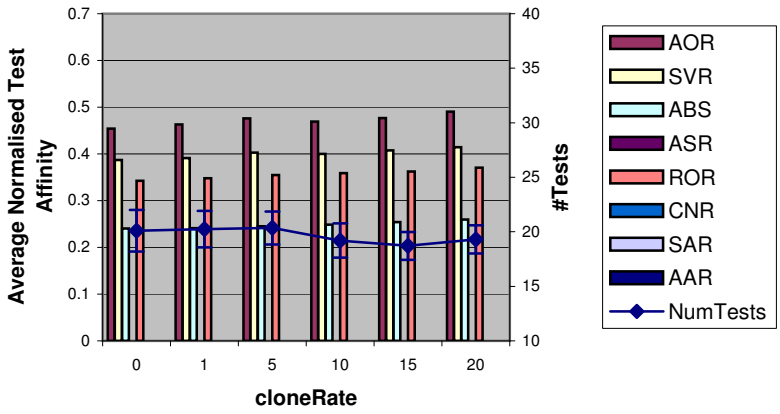

Figure 6.65: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program.

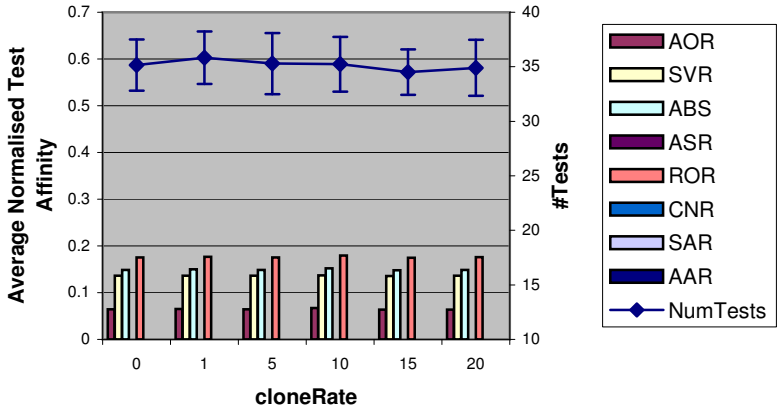

Figure 6.66: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program.

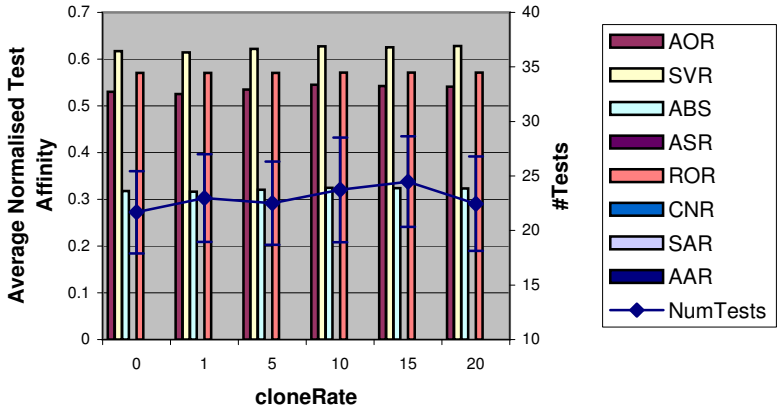

Figure 6.67: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program.

Figures 6.65-6.67 indicate the average memory set size after 500 iterations, along with the average test affinity for each mutation operator, for the three programs tested. The results are similar to `nFittest`'s, with *TriangleSort* generating a larger number of lower affinity tests compared to the other two programs.

The *DateRange* program (figure 6.68), and to a lesser extent the *CalDay* program (figure 3.122 in Appendix C.2.9), indicate the expected results from varying the `cloneRate` parameter, for programs with larger average test mutation scores. Both programs imply that the number of mutant executions to achieve a high mutation score is proportional to the `cloneRate`. Given they both generate tests with larger average mutation scores, the number of clones produced per test will also be larger, causing the difference in mutant executions to be bigger as the `cloneRate` increases.

TriangleSort (figure 3.121, Appendix C.2.9) also hints at this relationship, with the mean number of mutants increasing very slightly as `cloneRate` does. However the differences are small and so its results are not significant (at the 0.05 level). This is expected as this program generates tests with a lower average mutation score than the others, meaning fewer clones are created per test, and so fewer mutant executions occur. Increasing the `cloneRate` will increase the number of clones, thereby retaining the proportional relationship.

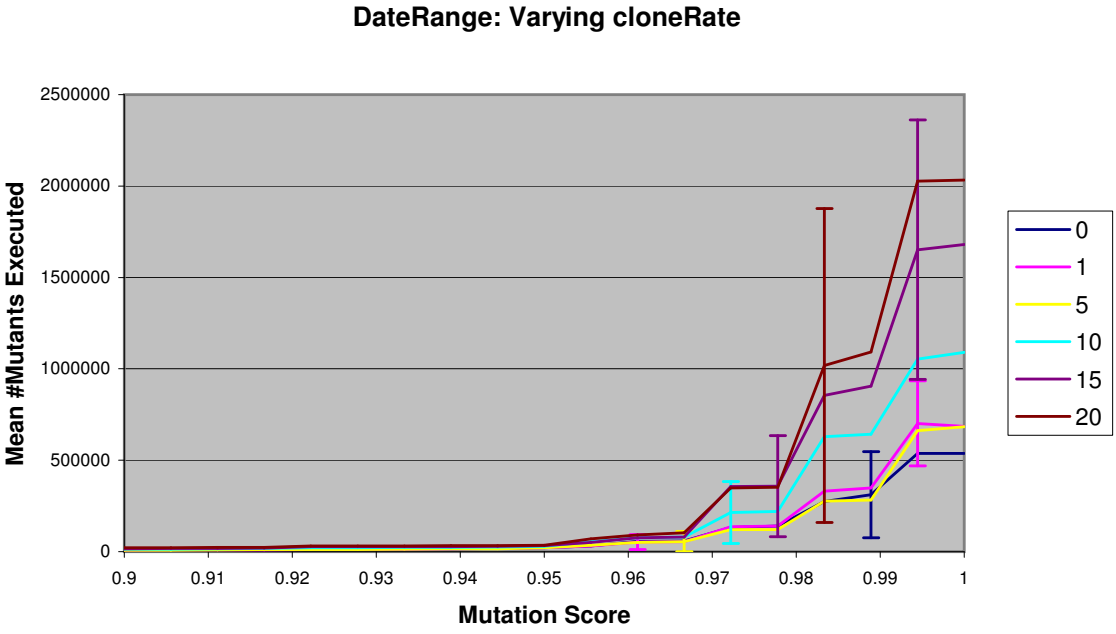

Figure 6.68: Effect of varying the `cloneRate` on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

The reasoning for why different programs generate tests with different average affinities is hypothesised in the `nFittest` section. Ultimately though, all programs do well with a low `cloneRate`. Observant readers will notice that a `cloneRate` of 0 still achieves a high mutation score, even for the *TriangleSort* program (which is supposedly more reliant on local searching). A `cloneRate` of 0 would imply no clones being created for any test; this is a little misleading however, as the IIA/MT in fact generates a minimum of 1 clone per test (see section 4.6.1). These results suggest that generating a single clone per test is enough, implying that the `cloneRate` variable is of little use, at least in terms of the number of mutant executions. Consider a single iteration; `nFittest` individuals are selected and cloned. With a `cloneRate` of 0, there will be `nFittest` clones produced (i.e. 1 clone per parent), with the most useful (i.e. tests which kill at least one mutant not already killed by the memory set) added to the memory set. Higher `cloneRate` values will generate more clones, causing more divergence in the localised searching and

requiring more mutant executions. Lower `cloneRate` values however, whilst causing less divergence and fewer mutant executions, appear capable of achieving the same mutation scores. A possible explanation may be that because the memory set retains useful tests, the portion of the shape-space needing to be searched is effectively reduced with each test added. Consequently, as the iterations progress, less divergence of the localised search is needed. In contrast, in early iterations weaker mutants are living which are killed by easier-to-find tests - again less divergence in the localised search path is needed. Both stages can be achieved with fewer clones, saving on mutant executions. The IIA/MT could therefore theoretically be simplified to only generating one clone per parent test, which would then be independent of the proportion of subtle mutants the program generates.

Effect on the Mutation Score

Figure 6.69 shows that `cloneRate` has little effect on the mean mutation score per iteration for the *DateRange* program. This result is common to the other two programs except *CalDay* hints at a proportional relationship (see figure 3.128 in Appendix C.2.10) with a slightly greater variation between the various `cloneRate` values.

These results can be explained by considering the average test affinity generated by each program, shown in figure 6.65-6.67. *TriangleSort* creates low affinity tests, each of which ultimately produce a low number of clones, regardless of the `cloneRate`. *DateRange* and *CalDay* however, generate higher affinity tests causing a larger increase in the number of clones produced as the `cloneRate` value increases. More clones per iteration would theoretically improve the chance of killing new mutants, and so the mutation score achieved per iteration should be proportional to the `cloneRate` value, with the results more obvious in higher affinity tests (e.g. *CalDay*). The reason for the difference in average test affinity is described in the `nFittest` section.

Two of the three programs present no obvious choice for a beneficial `cloneRate` value, out of the range of values chosen. The *CalDay* program however hints that

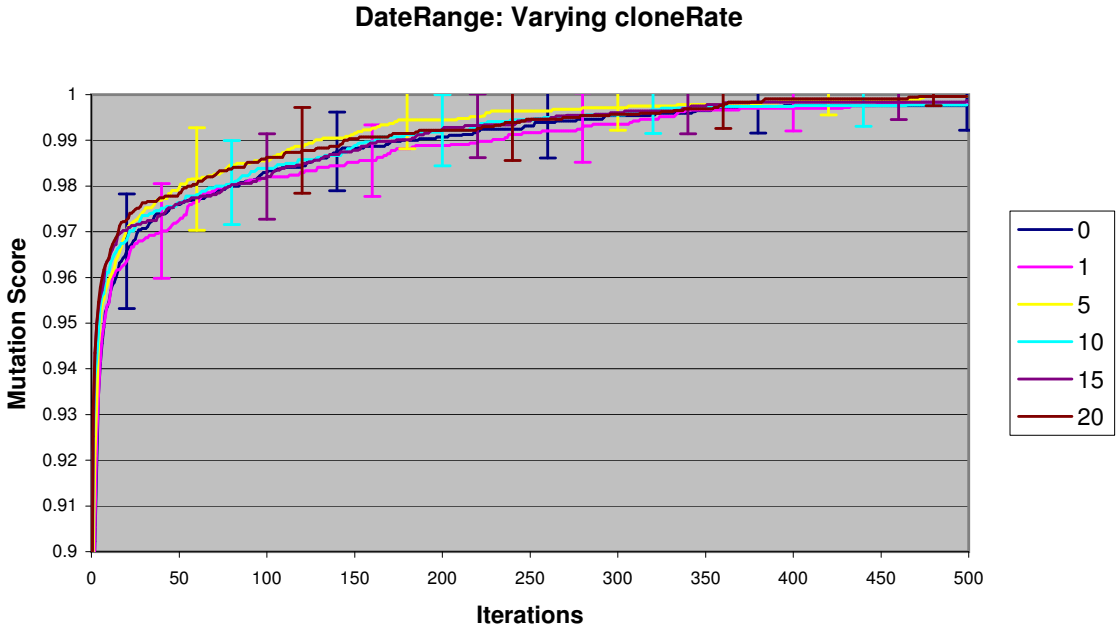

Figure 6.69: Effect of varying the `cloneRate` on the mean mutation scores per iteration for the *DateRange* program.

a higher `cloneRate` generates a higher mutation score per iteration. Further research using larger `cloneRate` values may confirm the proportional relationship between this parameter and the mutation score per iteration. Given these results, especially that a high `cloneRate` makes little difference for two programs whilst improving the third, a high `cloneRate` parameter is preferential.

Effect on the Number of HTK identified

Figure 6.70 suggests that the `cloneRate` parameter has no significant effect on the number of hard-to-kill mutants identified after 500 iterations. This result is not unsurprising given the fact that `cloneRate` has little impact on the mutation score achieved per iteration (at least within the range of values tested in this research).

Table 3.46 in Appendix C.2.11 (page 290) indicate the results for each parameter value, along with the ANOVA calculations. Apart from the *CalDay* program, which demonstrates a minor significant difference in the HTK count

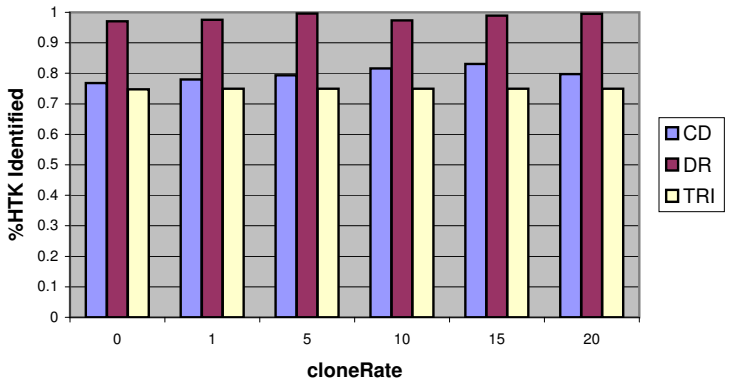

Figure 6.70: Effect of varying `cloneRate` on the mean number of HTK mutants identified for all three programs.

between `cloneRate`s of 0 and 15, all other differences are not significant (i.e. they probably occur by natural variation). As far as identifying hard-to-kill mutants is concerned, the value of the `cloneRate` parameter is not important, at least within the parameter value range tested.

Effect on the Number of Tests

Figures 3.129-3.131 in Appendix C.2.12 (pages 291-292) show that varying the `cloneRate` has no effect on the number of tests in the IIA/MT's memory set. This is to be expected, as this parameter does not affect the 'quality' of the tests, just the number of clones produced. Incidently, the graphs for *CalDay* and *DateRange* also exhibit the same limitation as discussed in section 6.4.1 (increase in mutation score without any increase in the number of tests).

6.5 Summary

This chapter is aimed at discerning useful parameter settings to improve the overall effectiveness of both the Genetic Algorithm and the Immune Inspired Algorithm. Ultimately, such improvements are measured by increased mutation scores and reduced number of mutant executions (i.e. reduced execution time). To

determine the significance of these results, at least 30 runs of each variable setting were performed. Only one variable was modified from the default values in any single experiment. The following summaries first describe the user-definable parameters, followed by the primary hypotheses relating to mutation score and mutant executions for each, before summarising the results.

6.5.1 Genetic Algorithm for Mutation Testing

The GA/MT has three user-definable parameters:

`indSize` This parameter defines the number of tests within each GA/MT individual. To maintain a fair comparison with the IIA/MT and other GA/MT `indSize`s, the total number of tests within the population is kept constant. Modifying the individual size means the number of individuals within the main population must also change.

`mutRate` This parameter defines the probability that a test input will be mutated (section 3.2.3), and is also used to control the amount of mutation that occurs (Appendix 4.3.2).

`crossRate` This parameter defines the probability of recombination occuring (section 3.2.3), and is used to identify the point at which crossover shall occur.

indSize

H 1 *The achieved mutation score is proportional to* `indSize`*.*

H 2 *The number of mutant executions to achieve a specific mutation score is inversely proportional to* `indSize`*.*

Using a Genetic Algorithm for mutation testing has a drawback in that the number of tests needed to achieve a full, or at least a high, mutation score needs to be known *a priori* to testing. Too many tests and there will be more executions than needed, with no apparent improvement in mutation score. Too few, and

a high mutation score will not be possible. Unfortunately the appropriate number is dependent on the program being tested and is difficult to ascertain before testing. Results from the `indSize` analysis however, hint at a useful metric for determining an individual's size - the minimum size is approximately equal to the McCabe complexity of the program. Given this complexity measure is indicative of the number of paths through a program, this observation implies that there is a relationship between these paths and the mutations applied to a program - a result possibly having connotations with the constraint-based testing (CBT) approach discussed in Section 2.5.1. In particular, CBT uses three conditions to develop tests to identify a mutant: a test **must** cause the mutated statement to be reached, generate an incorrect state immediately after execution, and propagate this through to a failure upon termination. The first and last of these conditions relates strongly to the path through a program.

The observed results suggest that the minimum individual size (threshold), identified by the complexity, has an impact on the number of mutant executions. For `indSize`s less than the threshold, the achievable mutation score appears restricted. If it is possible to achieve higher mutation scores, a considerably larger number of mutant executions would be needed. On the other hand, `indSize`s larger than the threshold have little difference on the achieved mutation score or the number of executions required to ascertain this affinity. From these results however, it does not appear that the McCabe complexity is an exact threshold boundary, and so it should be regarded as the minimum GA/MT individual size needed to obtain reasonably high mutation scores.

mutRate

H 1 *The achieved mutation score is inversely proportional to* `mutRate`*.*

H 2 *The number of mutant executions to achieve a specific mutation score is proportional to* `mutRate`*.*

Mutating tests creates new tests which need to be executed against all mutants, therefore the number of executions needed to achieve high mutation scores will increase as the `mutRate` does. This is worsened as higher rates imply more change to each individual which seems to restrict the mutation scores achieved. Lower mutation rates are therefore favourable and present a more gradual growth in mutation score (over the 500 iterations). If the `mutRate` is too small however, not enough new tests are created to encourage a high mutation score within 500 iterations. A value of 0.02 is recommended based on the results observed.

crossRate

H 1 *The achieved mutation score is not affected by* `crossRate`.

H 2 *The number of mutant executions to achieve a specific mutation score is not affected by* `crossRate`.

Finally, from the obtained results it appears that the crossover rate has little effect on either the number of mutant executions or the mutation scores achieved. This is to be expected. It is the combination of tests within a GA/MT individual that is important - what may be useful in one individual, may not in another, irrespective of whether it has a high mutation score or not. Furthermore, tests are not ordered within an individual. For these two reasons, swapping larger tail portions is unlikely to cause any benefit over smaller portions. Crossover may have more of an impact if tests were ordered (possibly by mutation score) within each individual; further experiments would be required to verify this though. Ultimately, on the basis of these results, single-point crossover should be removed from the algorithm.

6.5.2 Immune Inspired Algorithm for Mutation Testing

The IIA/MT has three user-definable parameters:

`nFittest` This parameter has two purposes: it defines the number of tests selected to undergo the clonal selection process; and it defines the number of tests

in the main population that are replaced by the highest mutation scoring cloned tests.

`cloneRate` This parameter determines the number of clones produced from a parent test using the following function:

$$\#clones = maximum \begin{cases} parent_affinity * \texttt{cloneRate} \\ 1 \end{cases} \quad (6.4)$$

`nWorst` This parameter is used in the metadynamics phase to replace the `nWorst` lowest mutation scoring tests with randomly generated ones.

nFittest

H 1 *The achieved mutation score is proportional to* `nFittest`.

H 2 *The number of mutant executions to achieve a specific mutation score is proportional to* `nFittest`.

Foremost, the results from `nFittest` suggest that the appropriate value of `nFittest` (for reducing the number of mutant executions) is inversely proportional to the average affinity (mutation score) of the memory tests. Low affinity tests warrant higher `nFittest` values; high affinity tests prompt for lower `nFittest` values.

The average memory test affinity is however, dependent on the program being tested. More interestingly, the results suggest that the average memory test's affinity is dependent on the program's complexity and the number of mutants it generates. *TriangleSort* implies that programs with high complexities and low numbers of generated mutants produce memory tests with relatively lower mutation scores than either lower complexity programs (e.g. *CalDay* or *DateRange*) or programs with high complexities and large numbers of generated mutants (e.g. *Select*).

A possible explanation for this difference is theorised as the proportion of "subtle" mutants generated, where a "subtle" mutant is one which can be identified by a small change to a test that identifies another mutant. Low complexity

programs generate few "subtle" mutants to start with; high complexity programs generate relatively more, except the proportion of these (to "non-subtle" mutants) is dependent on the population size. These kind of mutants are likely to require specialised tests to identify them, which are unlikely to kill a large number of mutants. Therefore, the higher the proportion of "subtle" mutants, the lower the average test affinity will probably be. Unfortunately (and rather ironically), due to time constraints no results for `nFittest`'s effect on the *Select* program have been obtained. If the hypothesised explanation is true however, it would be expected that because of *Select*s low proportion of subtle mutants, `nFittest` would have a similar effect on the number of mutant executions as for *CalDay* and *DateRange* - i.e. the number of executions is proportional to `nFittest`. Considering only the results obtained though (i.e. ignoring the hypothesis generated), a smaller `nFittest` value is recommended - e.g. 1.

This result is slightly contradicted by the effect on the mutation score per iteration however, which suggests that a higher parameter value offers an improved mutation score per iteration. This effect is mainly relevant to approximately the first 50 iterations; after 500 iterations, the improvement in mutation score from using a higher `nFittest` is barely significant (at the 0.05 level). Ultimately then, an `nFittest` of 1 would be acceptable.

cloneRate

H 1 *The achieved mutation score is proportional to* `cloneRate`*.*

H 2 *The number of mutant executions to achieve a specific mutation score is proportional to* `cloneRate`*.*

The 'memory test affinity' explanation derived from `nFittest` can also be used to explain the results from the `cloneRate` parameter. In general, the number of mutant executions is proportional to `cloneRate`, however the difference in numbers between the various parameter values is seemingly dependent on the average test affinity. Lower affinity (mutation scoring) tests generate fewer clones

(see equation 6.4, page 203) and so the small changes in the tested `cloneRate` values will not produce such a large difference in the number of executions. Higher affinity tests however, generate many more clones, causing a larger difference between the various `cloneRate`s. On the basis of these results, a low `cloneRate` would be preferable, e.g. 0 or 1. Interestingly a value of 0 achieves as high a mutation score as the other values, indicating that a single clone (per parent test) is adequate for improving the mutation score, and that additional clones have no effect. The `cloneRate` parameter could therefore be removed, and replaced with a single clone generation each instance instead.

Removing `cloneRate` is not necessarily suggested when examining the mutation score achieved per iteration however. Although the results for *DateRange* and *TriangleSort* show very little difference between the various `cloneRate` values, *CalDay* hints at a proportional relationship. This would suggest that a high `cloneRate` is best. The reason for the observed results can once again be explained by considering the average test affinity - higher affinity tests produce more clones which increase the chance of improving the mutation score. Despite these results favouring a higher `cloneRate` however, simplifying the algorithm by removing variable rate cloning is appealing, and therefore recommended.

nWorst

H 1 *The achieved mutation score is proportional to* `nWorst`.

H 2 *The number of mutant executions to achieve a specific mutation score is proportional to* `nWorst`.

For two of the programs tested (*CalDay* and *DateRange*), an `nWorst` of zero restricts the mutation score achieved (at least within the 500 iterations). Using `nWorst` values greater than 0 allows these programs to achieve higher mutation scores, although there does not appear to be a consistent relationship between `nWorst` and the number of executions. Higher `nWorst` values do achieve higher mutation scores per iteration however. *TriangleSort* is different from *CalDay* and

DateRange in that it does not restrict the mutation score with an `nWorst` of zero, and instead hints at a proportional relationship between `nWorst` and the number of executions. Furthermore, `nWorst` does not seem to affect the mutation score per iteration, suggesting that global searching (produced by the introduction of new randomly generated tests) is not important to this program - something that is also hinted at by the `nFittest` and `cloneRate` parameters.

Considering the proposed affinity explanation (see `nFittest` above), these results would suggest that the proportion of "subtle" mutants is an indicator of the balance between localised and global searching. High proportioned programs need more subtle changes to tests to identify the "subtle" mutants - more localised searching. Lower proportioned programs need less subtle test changes - less localised searching. The most appropriate `nWorst` value will therefore depend on the complexity and number of mutant programs generated, in accordance with the theory explained in section 6.4.1. In order not to restrict the mutation score however, an `nWorst` greater than 0 should be used, with the default of 5 seeming appropriate.

Chapter 7

Conclusion

7.1 Introduction

Software needs testing. New bugs must be prevented from infecting code, and current infections must be eradicated. To this end, this thesis turned towards biology, in particular the human immune system, as inspiration for a novel approach to vaccinating software against faults. The vaccine itself consists of a number of tests, improved over numerous iterations, with its ability measured by the mutation testing adequacy criteria. This system has been compared against a traditional evolutionary approach - an elitist Genetic Algorithm.

7.2 Revisiting the Problem Domain

Mutation testing suffers from a number of problems, as described in chapter 2, to which a number of solutions have been developed. These problems can broadly be classified into two groups: computational expense; and lack of automation. Expense occurs primarily because of the need to execute vast numbers of mutated programs in order to determine the adequacy of a test set. Automation difficulties on the other hand, increase the input necessary by a tester. Both of these difficulties ultimately prevent mutation testing being adopted into mainstream industry, regardless of it being widely accepted as a powerful unit testing technique

[48, 82, 91, 103].

Current solutions to the expense problems focus on three areas: do fewer; do faster; and do smarter. Do fewer techniques reduce the actual number of variant program executions that occur (e.g. by selecting a random sample of variants). Do faster techniques focus on quicker execution of variant programs (usually through compilation rather than interpretation). And do smarter techniques typically use knowledge about mutation testing (e.g. the difference between a variant and the Program-Under-Test [PUT]) to improve performance. All these techniques operate with good results, often achieving mutation scores in the high 90's (percentage), combined with appreciable savings in the number of variant program executions (often saving more than 50% of the variant executions). As a reminder, the mutation score quantifies the correctness of the PUT as a percentage of the variant programs that are proven incorrect.

Despite these achievements however, mutation testing still suffers from difficulties in automation, primarily in three areas: determining the correctness of an output; detecting equivalent mutant programs; and, generating tests. The correctness of a program's output in response to some input is determined by an 'oracle' (e.g. a human, or another piece of software), however defining automated versions is not simple. This research does not concern itself with oracles though, and instead assumes that appropriate ones already exist. Testing practitioners should be aware of the limitation this imposes.

Equivalent mutant programs are also problematic as they exhibit the same output as the original program for every input; they are equivalent in semantics to the original program, and so serve only to unnecessarily reduce confidence in the correctness of a program - *are the remaining living mutants equivalent, or can they be identified by some, yet-to-be-found, test?* Unfortunately, determining equivalence is undecidable [17], although a number of heuristics do exist [39, 89, 83]. This research manually identifies equivalent mutants before testing commences, bypassing this issue.

Generating tests poses the final automation difficulty, and is the subject of this

thesis. Whilst randomly generating tests may identify some mutant programs as incorrect, it may not quickly and easily generate a high mutation score. Instead, a more targeted search for tests is required. Current solutions can be split into traditional and evolutionary approaches, with the former typically using knowledge of mutation testing (i.e. that a variant program differs from the original by only one line) to generate algebraic constraints which are solved to generate tests (constraint based testing [CBT]) [80]. This technique offers promising results, identifying around 97% of non-equivalent variants [28].

An alternative to a CBT approach are evolutionary techniques that draw inspiration from nature. These have been applied to test generation, viewing the task as an optimisation problem, although many approaches have used other adequacy metrics besides mutation testing to determine a test's ability. In particular however, Genetic Algorithms (GAs) have been used for mutation testing, generating test sets with good results ($> 90\%$ mutation scores). More recently, Bacteriological Algorithms (BAs), a subtle variation on GAs, have been applied to mutation testing with apparently better results (see the Evolutionary Techniques subsection in section 2.5.1) [7].

In terms of algorithm design, BAs form a bridge between GAs and the Immune Inspired Algorithm (IIA) developed for this research. GAs evolve a population of individuals (each consisting of multiple tests) using processes of crossover and mutation (both at static rates), with the best individual in any generation being the test set returned to the tester. BAs however, reduce each GA individual to a single test, evolve these through mutation alone (again with a static rate), and store the useful tests (as long as they exceed a threshold) in a memory set. The IIA/MT approach works in a similar fashion to the BA, except that it does not impose a threshold for introduction to the memory set, and it performs affinity (i.e. mutation score) proportional cloning and mutation. The IIA/MT's enhancements should encourage a higher overall mutation score; tests are not being restricted (and therefore lost) from the memory set, and the test generation is being focussed depending on the ability of the previous generation (as opposed to receiving the

same amount of mutation regardless). However, due to time constraints and the commonality of GAs as an evolutionary approach, BAs have not been used as a comparison algorithm for this research - only GAs.

7.3 Evolutionary Approaches

The overarching hypothesis of this research is:

"*An Immune Inspired Algorithm is consistently, at least as effective as a Genetic Algorithm for evolving test data through mutation testing.*"

This led to a number of further hypotheses, described in chapter 5, based on the notion of *effectiveness*. Effectiveness is determined to relate to three areas (in order of importance): an improved mutation score in (possibly) less time; finding tests for hard-to-kill mutants; and, generating a smaller sized test set. Consistency is defined as: repeatable results over multiple runs of the same experiment; and similar results through using different programs (see section 4 for a discussion on these explanations).

The results in chapter 5 conclude that the IIA/MT consistently presents a more effective approach to generating tests than the GA/MT approach. Importantly, the IIA/MT demonstrates the ability to consistently generate a higher mutation score in significantly fewer mutant program executions (at the 0.05 level). As the number of mutant executions is a direct measure of the overall execution time, this result is beneficial for encouraging mutation testing to be adopted by industry.

But why should the IIA/MT outperform the GA/MT? Foremost, and probably the largest contributor to the improved mutation score is the memory set capability of the IIA/MT. This in itself provides two important features missing from the GA/MT approach: the lack of a size restriction on the number of tests; and, no modification of good, useful tests. The former allows ('necessary') tests which may only kill a single mutant program to enter the memory set, thereby increasing the overall memory set mutation score with a poor scoring test. Because of its

size restriction however, the GA/MT has no room for such low scoring tests. Each individual is being optimised to have the highest mutation score possible, and so the GA/MT favours high scoring tests instead. If it is assumed that the individual size is large enough to obtain a full mutation score however, the GA/MT must then modify a worthless test (one that does not identify any unidentified mutant program) into a necessary test, without modifying any of the other useful tests. With no bias on the mutation and crossover mechanisms, and no memory set capable of retaining useful tests in this manner, the GA/MT has no way to ensure that this happens. Useful tests could, and indeed do, get modified. Even if the necessary test is generated, if useful tests are modified so that the individual's overall mutation score is reduced below the best score (for that generation), then the population as a whole may not capitalise on the extra knowledge gained from the necessary test. In essence, the GA/MT is not designed to make use of low mutation scoring tests, and instead places greater focus on generating a specific sized mutation adequate test set.

Besides achieving a higher mutation score though, the IIA/MT typically executes significantly fewer mutants (in cases when it does not, it still achieves a higher mutation score). This is, in part, encouraged by the memory set which stops all useful tests from being lost (and having to be regenerated), allowing the IIA/MT to execute fewer mutant programs to obtain the same mutation score as a GA/MT. However, attention must not be drawn away from the processes used by the IIA/MT to adapt its population; these are advantageous too. In particular, the IIA/MT allows (to a degree), the selection of which tests are mutated. A number (`nFittest`) of the highest scoring tests are taken from the main population, and the same number are randomly taken from the memory set - this new set represents useful tests, of which half undergo affinity (i.e. mutation score) proportional cloning and mutation. By focussing cloning and mutation on these useful tests, combined with the memory set, allows the IIA/MT to perform a local search around key input domains (i.e. input values where subtle variations cause the identification of further mutant programs). In addition to this,

the IIA/MT also introduces a number (`nWorst`) of new, randomly generated tests each iteration. These serve to globally search out new areas on the input domain for useful tests, performing a complementary search to that provided by cloning and mutation.

In comparison, the GA/MT also performs both the global and localised searching, except that these occur through a single process - mutation. This happens at a predefined, fixed rate (`mutRate`), that defines how much mutation an individual suffers, as well as how much each test changes. A problem with having a static rate is that it offers no variation in searching depending on how the population is doing. If the test population is poor (i.e. has a low mutation score), then it is likely that more global searching is required to find areas of interest in the input domain - i.e. a higher mutation rate is required. However, if the test population is strong, then it is likely that more localised searching is required to improve the mutation score further - a lower mutation rate is needed. Having only a single process, there is a fine trade off between the searching trends (i.e. global or local). Too much emphasis in either direction and the overall mutation score is reduced.

Returning to the IIA/MT's increased effectiveness over the GA/MT, the results in chapter 5 also demonstrate that the IIA/MT is capable of identifying a larger number of hard-to-kill mutants. Given that the IIA/MT generates a test set with a higher mutation score than the GA/MT, and that in order to do this requires identifying more HTK mutants, this result is unsurprising[1].

On the basis of this comparative study, the evidence suggest that the IIA/MT approach is more effective than the GA/MT. As suggested, this is more than likely due to the introduction of the memory set, and the benefits this enables. That said, the processes of cloning and mutation also play important roles, the extents of which can be hinted at through studying how the parameter values affect the algorithm's capabilities.

[1] As an aside, it should be noted that hard-to-kill mutants were manually identified for this research, and as such represent a minimum number of HTK mutants. There could be more, but it is expected that a similar relationship would exist, with the IIA/MT identifying more.

7.3.1 Parameter Values

Each algorithm has a number of different parameters which the user can adjust. Comparisons between the two algorithms were made using particular parameter values as mentioned in chapter 5. These were not optimised for either algorithm however, and instead were set to values that were thought to reduce the number of mutant executions whilst still achieving high mutation scores. To determine the appropriateness of these values, chapter 6 performed a basic analysis on different parameter settings for each algorithm, in order to determine their effect.

The results from the GA/MT experiments hint that appropriate parameter values are dependent on the PUT. This is to be somewhat expected if one considers that the program itself dictates what input domains will be useful for identifying mutants. In particular this is important for the individual size, which seems to bear a relation to the program's complexity; it is hypothesised that the program's complexity is an approximate measure of the minimum number of tests needed to achieve a high mutation score. Further experiments are required to test this theory. Based on the results obtained, and primarily focussing on the number of mutant program executions, the following suggestions are made for GA/MT approaches:

indSize

- *Use an individual size larger than the McCabe complexity of the program being tested.*

mutRate

- *Use a mutation rate of 0.02.*

crossRate

- *Remove the crossover process altogether.*

Results for the IIA/MT approach also suggest that the PUT plays a critical

role in determining the appropriate parameter settings. Rather more interestingly though, there is evidence to suggest a correlation between the average memory test mutation scores and the parameter values. It is hypothesised that the average memory test mutation scores are related both to the program's complexity and the number of mutants it generated. Further experimentation is needed to determine if this hypothesis is indeed true, or just a peculiarity of the three programs tested. From the results observed however, and again focussing on mutant executions, the following suggestions are made for IIA/MT approaches:

nFittest

Programs with:-

- High McCabe complexities and a low numbers of mutants
  should use higher `nFittest` *values, e.g. 5+.*

- Low McCabe complexities OR high complexities and a high number of mutants
  should use lower `nFittest` *values, e.g. 1.*

cloneRate

- *Replace the* `cloneRate` *parameter, and instead generate a single clone for each parent test.*

nWorst

Programs with:-

- High complexities and a low numbers of mutants
  should use a low `nWorst` *value, e.g. 0 or 1.*

- Low complexities OR high complexities and a high number of mutants
  should use higher `nWorst` *values, e.g. 5+.*

7.4 Further Work

Whilst this research presents an initial comparison between two evolutionary algorithm approaches to test data generation, considerably more work is required to verify these results. For a start, this work is limited to only a handful of programs and parameter settings due to the long execution times - ironic as reducing execution costs is the main aim of this research. On this basis, a good starting point for future work would be to test other programs, as well as to try a wider range of parameter settings. It is expected that this will verify the results presented in this work. A friendly warning however, execution times can be extremely lengthy - 500 iterations of the IIA/MT (`nFittest`=5, `nWorst`=5, `cloneRate`=10) against the *Select* program takes approximately 4 days on a state-of-the-art PC (Intel P4 2.2GHz, 1Gb RAM).

Alternative experiments could also be performed. For example, one complementary experiment would be to vary the population size, rather than the individual size, of the GA/MT. The GA/MT's individual size could be fixed at the same number of tests needed for the IIA/MT to generate its best score. A single GA/MT individual is now capable of generating the same high mutation score as the IIA/MT. It could be expected that the probability of finding this high scoring individual would be proportional to the population size - increasing the number of individuals, increases the probability of generating a high scoring one, and vice versa. However, the total number of tests (and therefore the number of mutant executions) will also be proportional to the population size, meaning that increasing the population size may result in a less effective algorithm (i.e. it requires more mutant executions to achieve a high mutation score). These expectations are theoretical and need investigating.

There are of course, other interesting areas to research. Building upon the work presented in [67, 68], a co-evolutionary approach to mutation testing could be employed, that evolves a set of "useful" mutation operators as well as the test set. A "useful" mutation operator is one that produces mutant programs that often

find mistakes introduced in a particular programming environment (i.e. programming team, programming language, problem); for example, a programmer may constantly make the same relational operator mistake and so the ROR mutagen will be more useful than the others. Obviously if the programmer learns to correct this mistake, then this mutagen will no longer be as useful. Co-evolving the tests and the set of mutagens to the most useful, reduces the total number of mutant programs generated, and therefore the execution time of algorithm, whilst hopefully retaining a good strong-mutation adequate test set. Furthermore, should the programming environment change, the system is also able to change, evolving the useful mutagens and the test set together. In essence this co-evolutionary system will provide a do-fewer, do-faster, do-smarter approach to mutation testing.

Evolving a set of "useful" mutation operators will occur on a larger timescale than for evolving mutation-adequate tests. Whereas evolving tests occurs over the lifetime of a single PUT, evolving mutagens happens across many programs. The system should be developed with this in mind. One possible starting point would be to develop, independent of test evolution, a GA or AIS based algorithm that accomplishes mutagen evolution. Once completed, the two 'halves' could be joined to develop a competitive co-evolutionary approach. Using the framework described in chapter 4, the following solution could be employed for an AIS based approach:

- *Representation*: *each individual is a mutation operator*;
- *Evaluation*: a 'useful' individual is one which identifies errors. Errors will be indicated by un-equivalent mutant programs that remain living after testing, and so *an inverse mutation score should be used*;
- *Adaptation*: In the first instance, *a dynamic population adapted through clonal selection principles should be employed*.

There are however two points to consider when developing the mutagen evolution half. Firstly, if an individual in the population is a mutagen, then there

is little point in having more than one of the same mutagen in the population. If this were to happen, the same mutant programs would be executed more than once, resulting in an increased execution time. Instead a better approach would be to assign each individual a 'resources' or 'concentration' level - good, useful individuals have their concentration level increased in some manner (possibly related to their affinity); poor individuals have their's decreased. The second point to consider is how the affinity is calculated and its effect on a co-evolutionary approach. With each population (test and mutagen) evolving, the sets used to measure an individuals affinity will alter from one generation to the next (e.g. mutagen population evolution will affect the mutagens used in the mutation score calculation for each test, making it difficult to compare affinity values between tests - *i.e. is the mutation score calculated on the same set of mutagens?*). To compensate for this, each test should store results for all mutagens it has ever been executed against, regardless of whether these exist in the mutagen population. Each test's mutation score can then be calculated, when needed, based on the current set of mutagens, allowing for fairer comparisons. It is expected that by continually evolving both sets, 'good' (high mutation scoring) test data can be generated for each PUT, as well as an appropriate set of mutation operators that reflect common mistakes in the development environment.

Continuing with the singular evolution of tests in this research, further enhancements could be made to the Immune Inspired Algorithm. Firstly, it could be modified to only measure tests against the remaining living mutants, similar to the method employed by the BA. This was not originally done in order to present a fairer comparison with the GA/MT, as well as to allow easier preliminary development of the above co-evolutionary approach (changing mutagen sets will affect the mutation score affinity measure, and so it is simpler to have a consistent calculation for mutation score, rather than one which also varies dynamically with the memory set). If this was employed, it is expected that the IIA/MT would produce even greater savings on the number of mutants executed, whilst still attaining the high mutation scores. A further enhancement would also be to create a more

dynamic memory set. Every memory test could be given a time duration, which when expired, returns that test to the main population (possibly for a minimum number of iterations). If this test is still useful (i.e. it kills something not already identified), then it will return to the memory set. If it is no longer useful, it will remain in the memory set until it is naturally replaced. The advantage of this is that the algorithm will attempt to reduce the memory set's size, without losing important tests. Furthermore, because of the caching of current tests results, swapping a test between the main population and the memory set will not incur any extra mutant executions (they have already happened). A foreseeable problem would be if the extra living mutants, caused by downgrading a test, are killed by more tests in the main population than just the downgraded test. *Which tests should be added to kill the most living mutants without dramatically increasing the memory set size?* This problem could become worse if more tests are downgraded each iteration.

Other interesting areas revolve around how best to generate and modify the tests. Evolutionary algorithms are useful at optimising when there is a relationship between an individual's fitness measure and the structure of the individual itself. For example, evolving the highest value for a 2-dimensional function, $y = f(x)$, can be achieved because there is a relationship between the 'output' y value, which provides the fitness, and the 'input' x value (e.g. a y value close to a maxima probably only requires a small change in x to find it). This relationship is not apparent for mutation testing - given a good mutation scoring test, *how should the test mutate in order to improve the mutation score?* The method employed in this research used a cumulative Gaussian distribution (described in Appendix 4.3.2) to affect the amount a test input was changed using the test's value as the mean and the "mutation" rate as the spread (this rate is inversely proportional to the test's affinity for the IIA/MT, and static for the GA/MT). The benefit of this approach is that the "mutation" rate can be used to set the range of possible test mutations, with the new value being chosen at random from this range (see Appendix 4.3.2 for a more detailed discussion). Whilst this offers some direction (and limitation)

to the amount of mutation a test incurs, it is not the only possible approach to achieving this. More sophisticated techniques may use knowledge of the program to direct test mutation, possibly looking at the range of possible values an input can take, or the relationship between input parameters.

One possible starting point would be to consider the relationships between tests that kill similar mutant programs. For example, *what commonality is there between the tests that kill mutant variants of* `if(x>y)`*? Is it possible to infer a relationship between these tests, that can be used to direct mutation? Are there generic rules for common programming expressions, or are they specific to each program?* This also introduces the idea of partitioning the input domain. Tests can be classified (in some program dependent fashion) into groups depending on the output the program responds with. Each group could then be represented as an algebraic constraint describing what test inputs belong to that set. *Can these constraints be determined through dynamic execution of many mutant variations of a program? What information does each variant provide to the domain constraints?*

Finally, there is the matter of detecting equivalent mutants and the oracle problem to contend with. Neither problem has been tackled in this research, with the assumption being that appropriate methods exist to solve these. In order to facilitate mutation testing's adoption by industry however, feasible solutions are needed to these tasks. Detecting equivalent mutants may simply involve ignoring them and using the reduced mutation score to evolve tests - if test evolution is constantly occurring, the best set of tests to date can be taken. The disadvantage of this is that it distracts the focus of evolving tests - e.g. *should test evolution focus on this living mutant, or is it equivalent?* The oracle problem is also extremely important, however its solution will probably depend upon the program being tested. For this reason, solutions to the oracle problem should probably focus on deriving information from earlier stages of software development, e.g. specifications, and use this to determine output correctness.

Appendix A

Complementary Functions

Function	Description
$A \leftarrow$ `initPop`(a)	Returns the set A consisting of a initialised (random or otherwise) individuals.
`calculateAffinity`(A)	Calculates the affinity (normalised mutation score) of every individual within the set A.
$B \leftarrow$ `combine`(B, R)	Returns the set B combined with all individuals in set R. Duplicates remain.

Table 1.27: Functions used by both the Immune Inspired Algorithm and the Genetic Algorithm.

Function	Description
$L \leftarrow$ `addToMemory`(A, M)	Returns the subset L of individuals from the set A which kill at least one mutant program not killed by the memory set M. Individuals in L are added to the memory set before this function returns.
$B \leftarrow$ `selectFittest`(A, b)	Returns the set B of b highest affinity (normalised mutation score) individuals from A.
$R \leftarrow$ `randomSelection`(S, b)	Returns the set R of b randomly selected individuals from the set S.
$C \leftarrow$ `clonalSelection`(L, cR)	Returns the set of cloned and mutated individuals C, derived from the set L, using a clone rate of cR. The number of clones produced for each individual in L is calculated by: $individual_affinity * cR$, with a minimum of 1 clone. Mutation occurs at a rate inversely proportional to the individual's affinity (normalise mutation score).
$D \leftarrow$ `metadynamics`(A, C, b, c)	Returns the set D of individuals based on the following steps: The set A is reduced in size (by removing the lowest affinity individuals) so that $/A/ + /M/ = initialpopulationsize$, where M is the memory set; b lowest affinity (normalised mutation score) individuals from A are been replaced by b highest affinity clones from set C; c lowest affinity (normalised mutation score) individuals from this new set are replaced by c randomly created new individuals.

Table 1.28: Complementary functions used by the Immune Inspired Algorithm only.

Function	Description
$t \leftarrow$ `sumAffinities`(A)	Returns the sum of the affinities (normalised mutation scores) of each member of the set A.
`normaliseAffinities`(A, t)	Calculates each individual's (in A) proportion of the total affinity costs, t (obtained from `sumAffinities`.
$b \leftarrow$ `getBestTest`(A)	Returns the highest affinity (normalised mutation score) individual from the set A.
$i \leftarrow$ `rouletteSelection`(A)	Returns an individual from set A selected using roulette wheel selection, as described in section 3.2.3.
$q_{1,2} \leftarrow$ `singlePointCrossover`(i_1, i_2, cR)	Returns two individuals i_1 and i_2 which are individuals i_1 and i_2 with their tail portions swapped over, as described in section 3.2.3. cR is the probability of crossover occuring.
$q_m \leftarrow$ `mutateChild`(q, mR)	Returns q_m, the individual q mutated using the mutation rate mR, in accordance to section 3.2.3 and Appendix 4.3.2.

Table 1.29: Complementary functions used by the Genetic Algorithm only.

Appendix B

Test Programs

Program	Mutation Operators								Total
	AAR	ABS	AOR	ASR	CNR	ROR	SAR	SVR	
CD	-	155	132	-	-	21	-	168	476
DR	-	80	30	-	-	28	-	86	224
SEL	269	540	222	510	22	168	130	925	2786
TRI	-	158	54	-	-	119	-	141	472

Table 2.30: The number of mutant programs created by each mutation operator for each program.

B.1 CalDay

B.1.1 Example Mutation Adequate Test Set

#	`day`	`month`	`year`
1	0	0	0
2	0	-13	1
3	1	1	-1
4	1	1	-2004
5	1	1	-4713
6	1	1	-5000

#	day	month	year
7	1	2	2004
8	1	-20	2004
9	14	20194	-100
10	-15	10	1582
11	15	-10	1582
12	15	138994	-10000

Table 2.31: A Mutation adequate test set for the *CalDay* program based on the manually determined equivalent mutants.

B.1.2 Initial Test Values

`day` = 1 `month` = 1 `year` = 2004

B.1.3 Code

```
1public class CalDay{
2  public double toJulian(int day, int month, int year) {
3    int JGREG = 15+31*(10+12*1582);
4    double HALFSECOND = 0.5;

6    int julianYear = year;
7    if(year<0){
8      julianYear = julianYear+1;
9    }
10   int julianMonth = month;
11   if(month>2){
12     julianMonth = julianMonth+1;
13   } else {
14     julianYear = julianYear-1;
```

```
15        julianMonth = julianMonth+13;
16      }

18      double t = Math.floor(365.25*julianYear);
19      double s = Math.floor(30.6001*julianMonth);
20      double julian = t+s+day+1720995.0;

22      int temp = day+31*(month+12*year);
23      if(temp>=JGREG){
24        // change over to Gregorian calendar
25        int ja = (int) (0.01*julianYear);
26        julian = julian+2-ja+(0.25*ja);
27      }
28      return Math.floor(julian);
29    }
30}
```

B.2 DateRange

B.2.1 Example Mutation Adequate Test Set

#	day1	month1	year1	day2	month2	year2
1	0	1	2004	1	1	2004
2	1	1	0	1	1	2004
3	1	1	2004	0	1	2004
4	1	1	2004	1	1	0
5	1	1	2004	1	1	2004
6	-1	1	2004	1	1	2004
7	1	-1	2004	1	1	2004
8	1	1	-2004	1	1	2004
9	1	1	2004	-1	1	2004
10	1	1	2004	1	-1	2004
11	1	1	2004	1	1	-2004
12	1	11	2004	1	1	2004
13	1	12	2004	1	1	2004
14	2	1	2004	31	0	2000
15	31	0	2004	2	1	2000
16	35	1	2004	1	1	2004

Table 2.32: A Mutation adequate test set for the *DateRange* program based on the manually determined equivalent mutants.

B.2.2 Initial Test Values

`day1` = 1 `month1` = 1 `year1` = 2004 `day2` = 1 `month2` = 1 `year2` = 2004

B.2.3 Code

```
1import java.util.GregorianCalendar;
2import java.util.Calendar;

4public class DateRange{
5   public long numberDays(int day1,
```

```
6                               int month1,
7                               int yr1,
8                               int day2,
9                               int month2,
10                              int yr2){

12      date1 = new GregorianCalendar(yr1, month1, day1);
13      date2 = new GregorianCalendar(yr2, month2, day2);

15      // check both dates are valid
16      if(goodDate(day1, month1, yr1, date1)
17       &&goodDate(day2, month2, yr2, date2)){

19        // get difference in milliseconds
20        long diffMillis = date1.getTimeInMillis()
21                          -date2.getTimeInMillis();

23        // get difference in days
24        long diffDays = diffMillis/(24*60*60*1000);

26        // return absolute difference in days
27        return Math.abs(diffDays);
28      } else {
29          // invalid date(s)
30        return -1;
31      }
32  }

34  public boolean goodDate(int day,
35                          int month,
```

```
36                              int yr,
37                              GregorianCalendar date){
38          // check month is in range
39      if(month<0||month>11){
40        return false;
41      }

43      // check that the day is not negative or 0
44      if(day<1){
45        return false;
46      }

48      // max number days in the specified month
49      int days=Date.getActualMaximum(Calendar.DAY_OF_MONTH);

51      // check that the day is less than max number
52      // of days in the month
53      if(day>days){
54        return false;
55      }

57      // date is good
58      return true;
59    }

61    private GregorianCalendar date1, date2;
62}
```

B.3 Select

B.3.1 Example Mutation Adequate Test Set

#	k	arr
1	-1	[1.0,2.0,3.0]
2	4	[1.0,2.0,3.0]
3	0	[3.0,2.0,1.0]
4	1	[3.0,2.0,1.0,0.0]
5	3	[3.4028235E38,2.0,1.0]
6	12	[0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0]
7	1	[10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0]
8	1	[12.0,11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,0.0]
9	1	[12.0,11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,-3.0,2.0,0.0]
10	1	[12.0,11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0]
11	11	[13.0,12.0,11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0]
12	24	[24.0,23.0,22.0,21.0,20.0,19.0,18.0,17.0,16.0,15.0,14.0,13.0,12.0,11.0, 10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0]
13	25	[25.0,24.0,23.0,22.0,21.0,20.0,19.0,18.0,17.0,16.0,15.0,14.0,13.0,12.0, 11.0,10.0,9.0,8.0,7.0,6.0,5.0,4.0,3.0,2.0,1.0]
14	1	[1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0, 1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0, 1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0, 1.0,1.0,1.0,1.0,1.0,1.0]
15	11	[-3.4028235E38,-3.4028235E38,-3.4028235E38,-3.4028235E38, -3.4028235E38,-3.4028235E38,-3.4028235E38,-3.4028235E38, -3.4028235E38,-3.4028235E38,4.0]
16	12	[-3.4028235E38,-3.4028235E38,-3.4028235E38,-3.4028235E38, -3.4028235E38,-3.4028235E38,-3.4028235E38,-3.4028235E38, -3.4028235E38,-3.4028235E38,-3.4028235E38,4.0]

#	k	arr
17	4	[-5.7110815,-14.880631,41.726444,-37.05031,23.39116,-22.205894]
18	15	[22.91872,24.014654,16.52458,4.6267824,19.871696,35.34549, 19.553448,22.565273,20.515776,17.790375,18.116589,24.03556, 17.774513,23.742893,12.443327,17.50667,3.757459,21.281898, 12.450602,22.943853,18.411558,10.76836,7.145749,4.577767, 35.56168,24.001644,-13.539833,13.960985,12.586792,21.717928, 7.761176,8.833367,14.195336,14.055513,21.987413,18.113111, 12.938291,13.2787075,15.46505,18.904062,20.587824,23.622593, 16.107018,20.29628,19.784386,21.078314,18.142649,25.601658]
19	16	[20.965218,11.247814,19.660957,17.865246,30.035995,30.31636, 34.6558,12.30845,17.90089,-0.24872124,17.934645,39.49695, 34.90146,26.921118,14.216193,27.267248,21.75644,12.971771, 23.01032,12.747556,11.393146,28.675354,-0.29288453,15.202724, 29.410002,14.395665,21.874817,13.654877,27.974867,28.11235, 19.745798,24.722267,23.58682,22.133926,-1.0012056,5.991516, 20.963291,16.190794,17.92023,15.606408]
20	16	[21.211082,13.3827505,28.704947,20.423193,18.478935,19.798641, 33.41666,16.89122,17.503212,27.283127,17.133081,28.681181, 20.566145,20.738182,19.500904,23.316935,8.847331,19.898027, 18.710484,21.299479,15.791643,19.590424,19.387543,23.456232, 25.559513,27.029621,19.739862,22.203028,27.522963,30.827885, 24.291775,26.60021,25.632788,26.708681,16.145826,18.9434, 10.237217,20.575977,26.921618,24.054678,23.123625,21.173822, 24.092585,16.622248,22.918564,24.800436,9.994295,9.547069, 20.953508,21.698324,16.136946,20.752983]
21	2	[36.67169,0.18202215,3.7204676,26.743032,50.007263,23.060598, -7.3350587,-2.071009,27.593918,16.692753,15.6716585,51.59463, 4.1991477,24.188866,13.893451,34.6822]
22	3	[32.54497,20.653757,27.500864,1.4117161,16.599707,23.285288,

#	k	arr
		13.773586,19.632303,23.628345,24.256357,29.246836,21.632448, 3.9006517,16.344898,18.12186,16.10166,21.70255,23.902906, 18.161541,15.266052,25.368382,23.896017,26.12119,16.776855, 11.766517,28.264322,18.368097,27.923717,28.025234,23.35657, 7.3589807,11.21263,18.385324,17.468771,26.152277,28.176958, 17.781507,22.106623,17.698368,20.381802,29.969366,24.65536, 9.973658,15.389518,15.873089,24.56311,18.611092]
23	4	[10.776883,4.957707,16.834284,20.962439,26.447643,22.036127, 18.3438,19.022291,21.708933,20.875889,16.250843,15.94117, 19.76901,26.397362,25.785505,11.486564,16.50855,20.222694, 18.04481,29.199928,18.741806,29.043098,25.806753,34.75311, 16.585566,17.433296,11.505542,19.649649,10.952237,19.102888, 19.009598,11.70182,25.604712,15.161908,23.278542,20.385696, 14.487988,24.755348,17.398462,32.123825,12.949027,17.183187, 23.180548,16.903484,25.1901,17.589226,19.308855,13.419565, 27.371662,20.203625,16.178308,15.879096,24.235569,22.925442]
24	4	[27.37919,27.933828,22.265245,0.43366367,21.55346,39.393154, 20.399475,11.402519,21.237303,21.879461,25.532736,49.89668, 13.343876,16.584328,25.221535,15.887644,22.10793,9.031165, 19.894829,8.398689,19.42205,21.705402,17.140053,12.158006, 7.6979003,27.403908,-15.802394,3.1856997,21.665276,28.869164, 21.105743,31.420324,15.180638,21.909044,21.999601,21.504887, 16.60092,17.469921,18.780449,22.932419,25.47064,18.119394, 12.737636,15.346484,27.745052,20.894445,17.531084,19.641977, 10.003345,17.325233]

Table 2.33: A Mutation adequate test set for the *Select* program based on the manually determined equivalent mutants.

B.3.2 Initial Test Values

$$\texttt{k} = 1 \quad \texttt{arr} = [6, 3, 99, 31, 1]$$

B.3.3 Code

```
1public class Select{
2  public float selip(int k, float[] arr)
3                                throws SystemExitException {
4    int M = 10;
5    float BIG = Float.MAX_VALUE;
6    int i;
7    int j;
8    int n = arr.length;

10   // k must be an index within the array length
11   if(k<1 || k>n){
12     throw new SystemExitException("Bad Input to Selip");
13   }

15   int[] isel = new int[M+2];
16   float[] sel = new float[M+2];
17   int kk = k;
18   float ahi = BIG;
19   float alo = -BIG;

21   while(true){
22     // iterate main loop until desired element is found
23     int mm = 0;
24     int number_lo=0;
25     float sum = 0f;
```

```
26        int nxtmm = M+1;

28        for(i=0; i<n; i=i+1){
29          // go through each element in the array
30          if((arr[i]>=alo) && (arr[i]<=ahi)){
31            // consider only elements within the current
32            // lo-hi range
33            mm=mm+1;
34            // if the element equals the lowest value
35            if(arr[i]==alo){
36              // increment the number lo elements counter
37              number_lo = number_lo+1;
38            }

40            // select M elements with equal probability
41            if(mm<=M){
42              // if the number of elements considered
43              // so far is <= M.
44              // mm-1 as our arrays start at index=0
45              sel[(mm-1)]=arr[i];
46            } else if (mm==nxtmm){
47              nxtmm = mm+mm/M;
48              int index= i+mm+kk;
49              index = index % M;
50              sel[index]=arr[i];
51            }
52            sum = sum+arr[i];
53          }
54        }
```

```
56        if(kk<=number_lo){
57          // then kth element must be equal to alo
58          return alo;
59        } else if(mm<=M){
60          // if the number of elements stored is less than M
61          // sort and select kth
62          shell(mm,sel);
63          return sel[kk-1];
64        }

66        sel[M] = sum/mm;
67        shell(M+1,sel);
68        sel[M+1]=ahi;

70        for(j=0; j<M+1; j=j+1){
71          // zero count isel array
72          isel[j]=0;
73        }

75        for(i=0; i<n; i=i+1){
76          // go through array again
77          if((arr[i]>=alo) && (arr[i]<=ahi)){
78            //for each in range element...
79            int j_lo=0;
80            int j_up=M+1;
81            while(j_up-j_lo>1){
82              // ... find its position amongst the
83              //      selection by bisection

85              // j_mid = mid point
```

```
86              int j_mid=(j_up+j_lo)/2;
87              if(arr[i] >= sel[j_mid]){
88                // element is >= this midpoint and so j_lo
89                // becomes the midpoint
90                j_lo=j_mid;
91              } else {
92                // element is < midpoint and so j_up
93                // becomes midpoint
94                j_up = j_mid;
95              }
96            }

98            // add in negative number catch check
99            // that arr[i]>j_lo
100           if((j_lo==0)&&(arr[i]<sel[j_lo])){
101             j_up = j_lo;
102           }

104           isel[j_up] = isel[j_up]+1;
105         }
106       }

108       // narrow the bounds
109       j=0;
110       while(kk>isel[j]){
111         alo = sel[j];
112         kk = kk-isel[j];
113         j = j+1;
114       }
115       ahi = sel[j];
```

```
116     }
117   }

119   // sorts the first n elements of arr
120   public static void shell(int num, float[] a){
121     int inc = 1;

123     while (((3*inc) + 1) < num) {
124       inc = (3*inc) + 1;
125     }

127     while( inc > 0 ) {
128       // for each set of elements (there are inc sets)
129       for (int k = inc - 1; k < num; k=k+1) {
130         // pick the last element in the set
131         float v = a[k];
132         int l = k;
133         // compare the element at v to the one before it
134         // in the set. If they are out of order continue
135         // this loop, moving elements "back" to make room
136         // for v to be inserted.
137         for(l=k; (l >= inc) && (a[l-inc] > v); l=l-inc){
138           a[l] = a[l-inc];
139         }
140         // insert v into the correct place
141         a[l] = v;
142       }
143       //all sets inc-sorted, now decrease set size
144       inc = inc/3;
145     }
```

```
146     }
147 }
```

B.4 TriangleSort

B.4.1 Example Mutation Adequate Test Set

#	i	j	k
1	0	2	2
2	2	0	2
3	2	2	0
4	2	2	2
5	-2	2	2
6	2	-2	2
7	2	2	-2
8	2	2	3
9	2	2	4
10	2	3	4
11	2	3	5
12	2	4	2
13	2	4	3
14	2	5	2
15	3	2	2
16	3	3	2
17	3	3	7
18	3	4	2
19	3	5	3
20	3	7	3
21	3	7	4
22	4	2	2
23	4	2	3
24	7	3	4
25	24	5	5
26	414811834	1453635131	1722147072
27	1453635131	414811834	1722147072
28	1453635131	1722147072	414811834

Table 2.34: A Mutation adequate test set for the *DateRange* program based on the manually determined equivalent mutants.

B.4.2 Initial Test Values

$\texttt{i} = 1 \quad \texttt{j} = 2 \quad \texttt{k} = 3$

B.4.3 Code

```
1public class TriangleSort {
2  public int triang(int i, int j, int k){
3        // check there are no negative length sides
4    if((i<=0) || (j<=0) || (k<=0)){
5      return 4;
6    }

8    // check for equal length sides
9    int tri = 0;
10   if(i==j){
11     tri = tri+1;
12   }
13   if(i==k){
14     tri = tri+2;
15   }
16   if(j==k){
17     tri = tri+3;
18   }

20   if(tri==0){
21       // if there are no equal sides
22     if((i+j<=k) || (j+k<=i) || (i+k<=j)){
23       // triangle is invalid
24       tri = 4;
25     } else {
26       // triangle is scalene
27       tri = 1;
28     }
```

```
29        return tri;
30      }

32      if(tri>3){
33          // triangle is equilateral
34        tri = 3;
35      } else if((tri==1) && (i+j>k)){
36          // triangle is isosceles (i==j)
37        tri = 2;
38      } else if((tri==2) && (i+k>j)){
39          // triangle is isosceles (i==k)
40        tri = 2;
41      } else if((tri==3) && (j+k>i)){
42          // triangle is isosceles (j==k)
43        tri = 2;
44      } else {
45          // triangle is invalid
46        tri = 4;
47      }

49      return tri;
50    }
51}
```

Appendix C

Parameter Analysis Results

C.1 Genetic Algorithm for Mutation Testing

C.1.1 indSize: Effect on Number of Mutant Executions

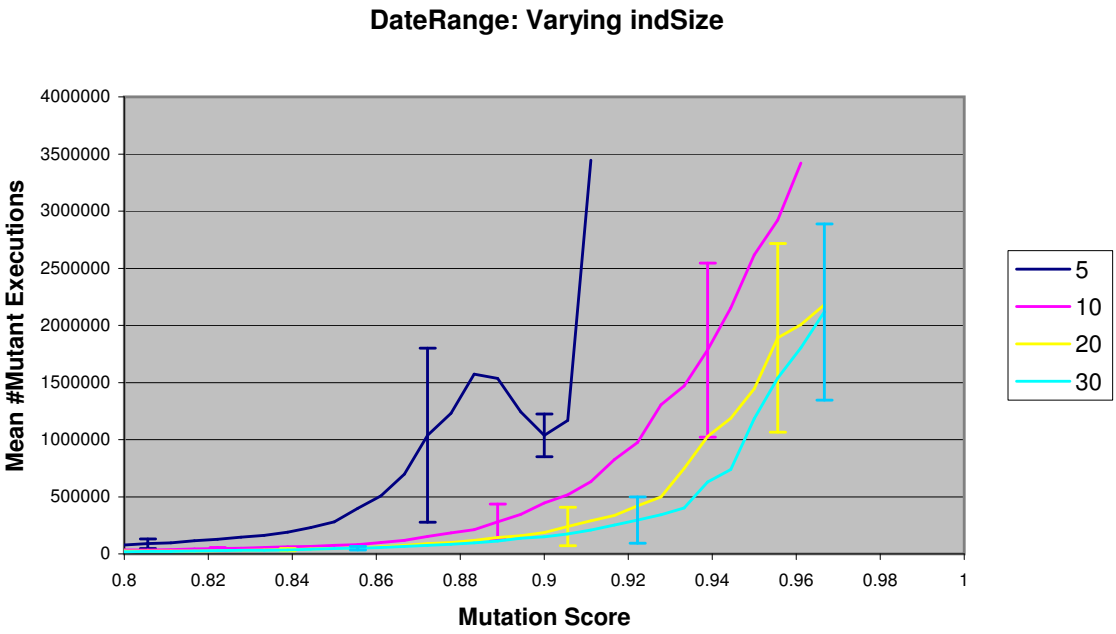

Figure 3.71: Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

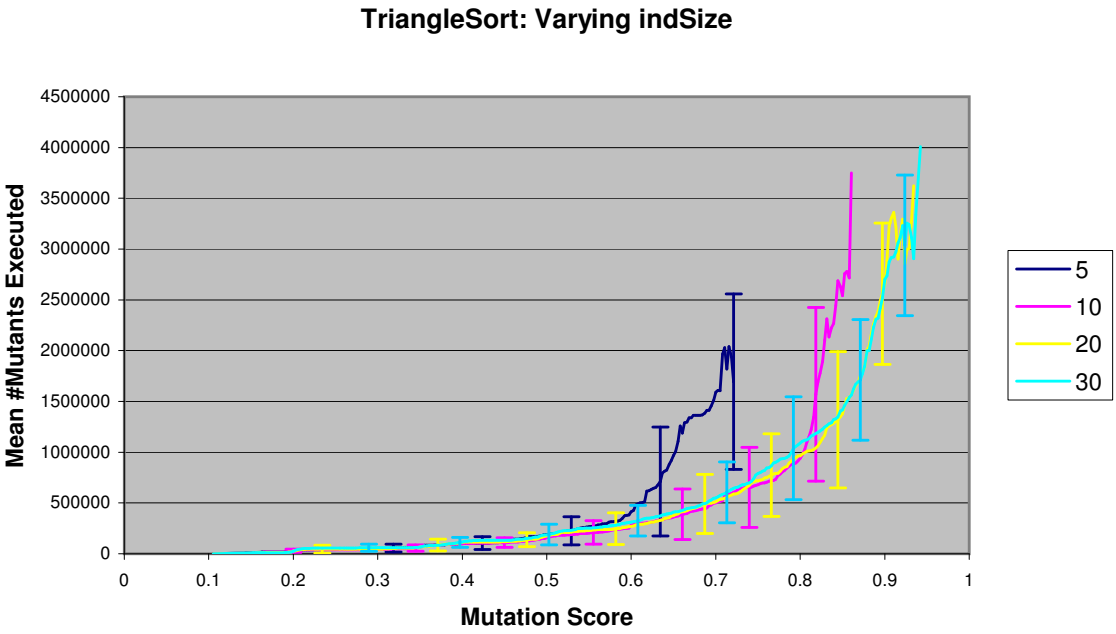

Figure 3.72: Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

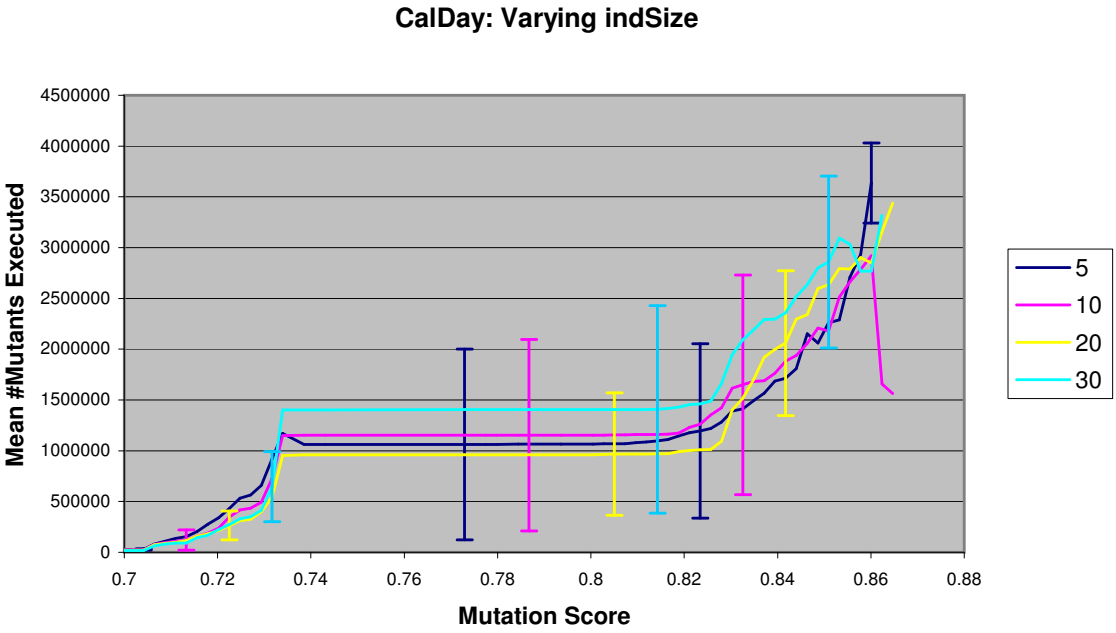

Figure 3.73: Effect of varying indSize on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

C.1.2 indSize: Effect on Mutation Score per Iteration

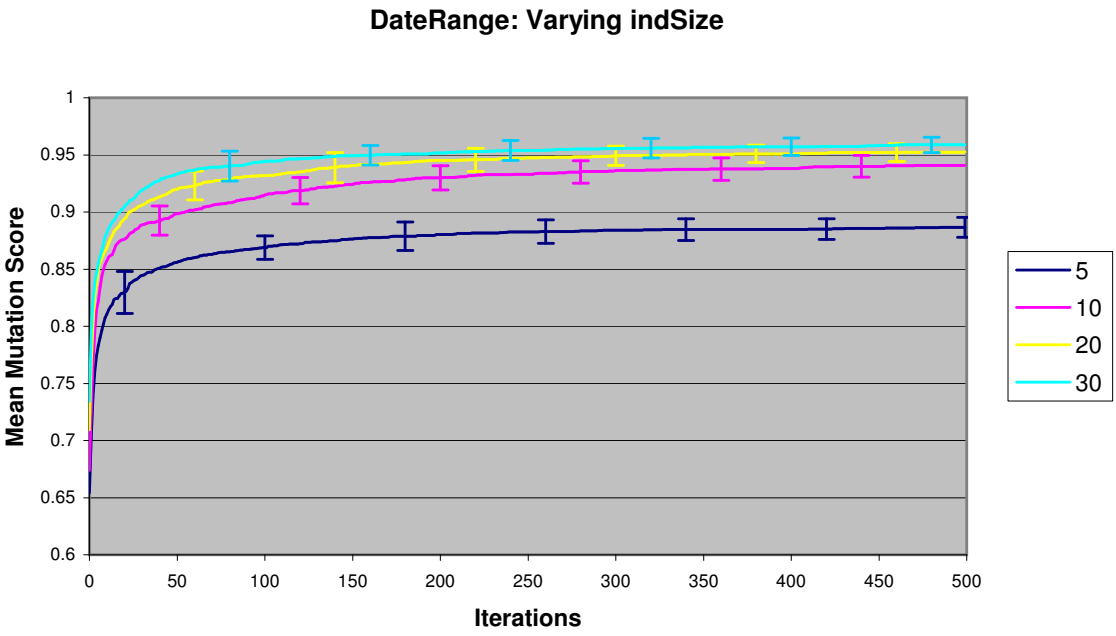

Figure 3.74: Effect of varying indSize on the mean mutation score per iteration for the *DateRange* program.

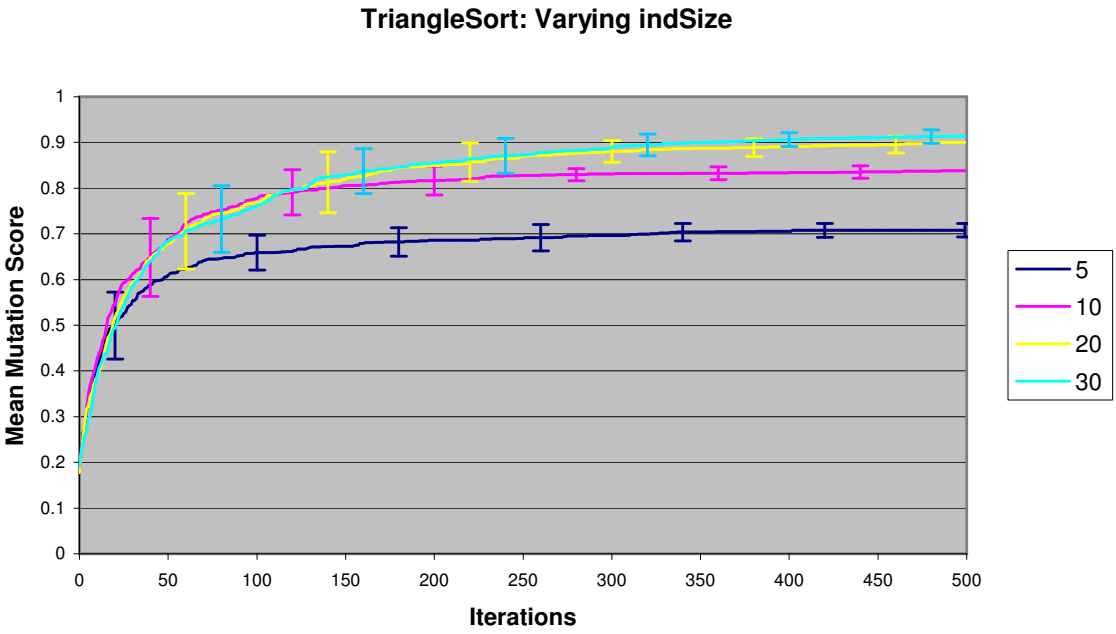

Figure 3.75: Effect of varying indSize on the mean mutation score per iteration for the *TriangleSort* program.

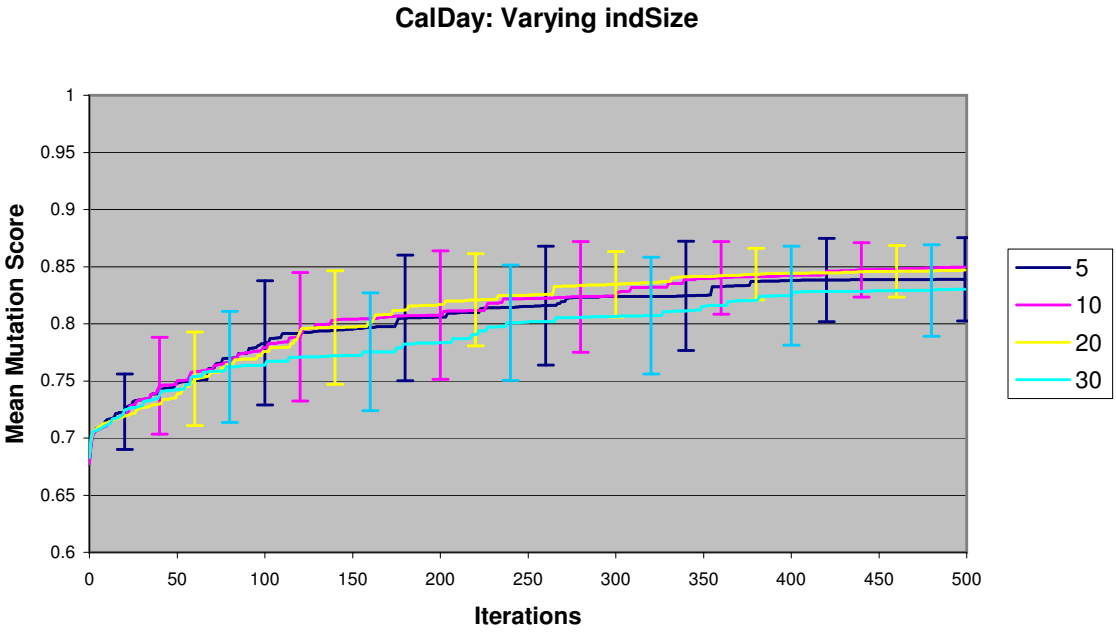

Figure 3.76: Effect of varying indSize on the mean mutation score per iteration for the *CalDay* program.

C.1.3 indSize: Effect on Number of HTK identified

indSize	DR	TRI	CD
5	59.11% ± 7.37%	0.00% ± 0.00%	34.48% ± 0.57%
10	71.11% ± 3.20%	5.56% ± 5.05%	34.69% ± 0.95%
20	70.98% ± 3.63%	27.22% ± 9.52%	34.38% ± 0.00%
30	71.91% ± 3.23%	30.83% ± 8.52%	34.38% ± 0.00%
ANOVA			
f_{obt}	**53.47**	**150.66**	2.14
f_{crit}	2.68	2.68	2.68
Scheffé			
c_{obt}			
5-10	**10.03**	**3.13**	-
5-20	**10.23**	**15.35**	-
5-30	**10.96**	**17.38**	-
10-20	0.11	**12.21**	-
10-30	0.69	**14.25**	-
20-30	0.83	2.04	-
c_{crit}	2.83	2.84	-

Table 3.35: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the four individual sizes: 5, 10, 20, 30. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of individual sizes result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.1.4 indSize: Effect on Number of Tests

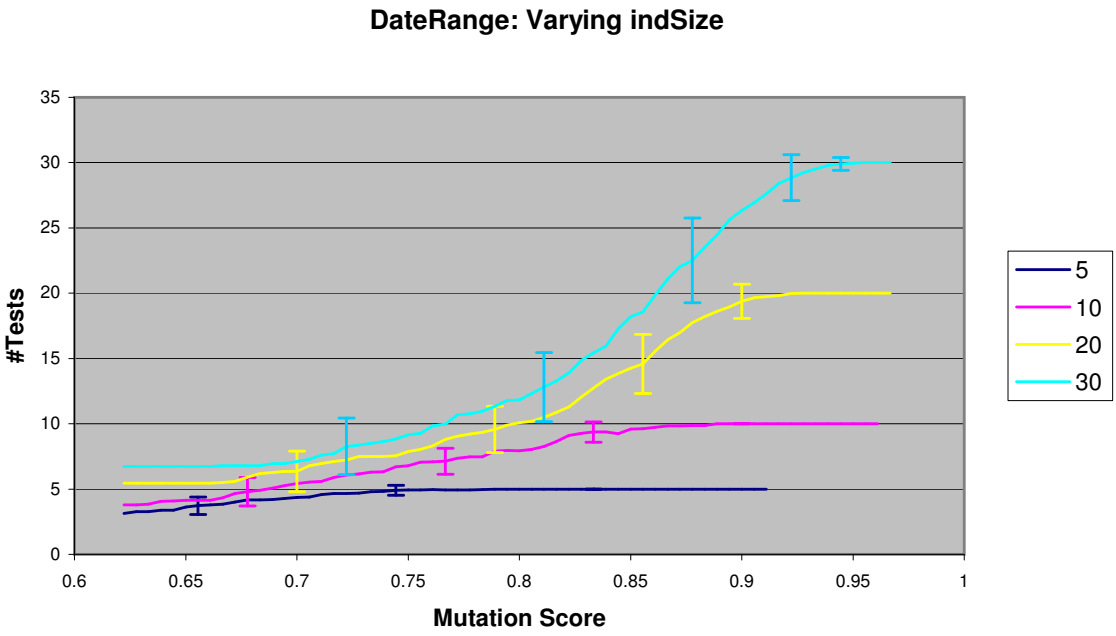

Figure 3.77: Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program.

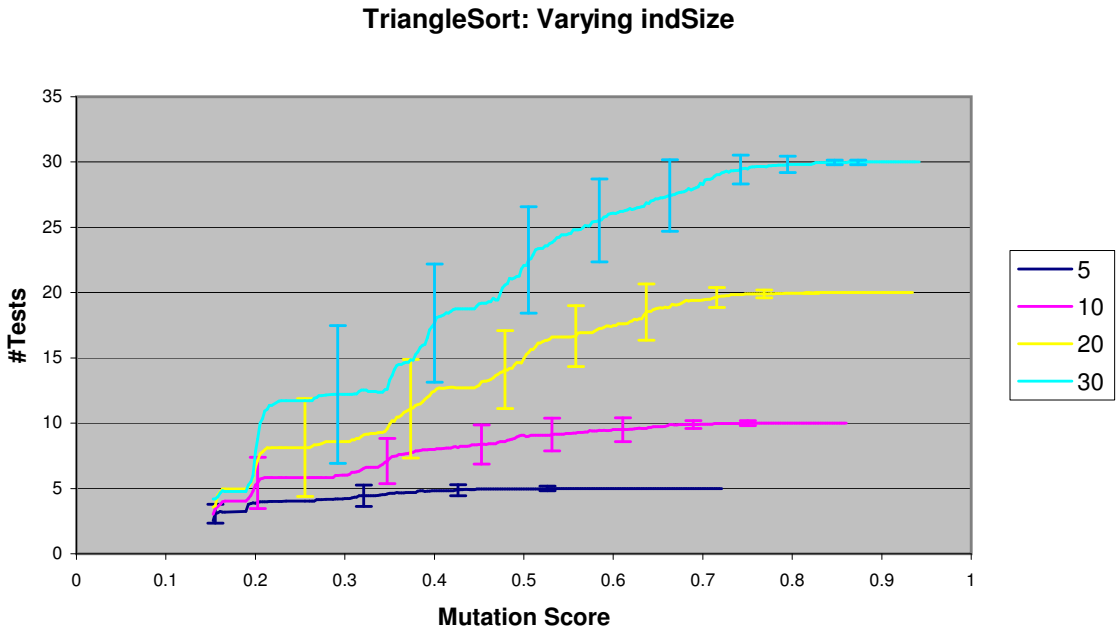

Figure 3.78: Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program.

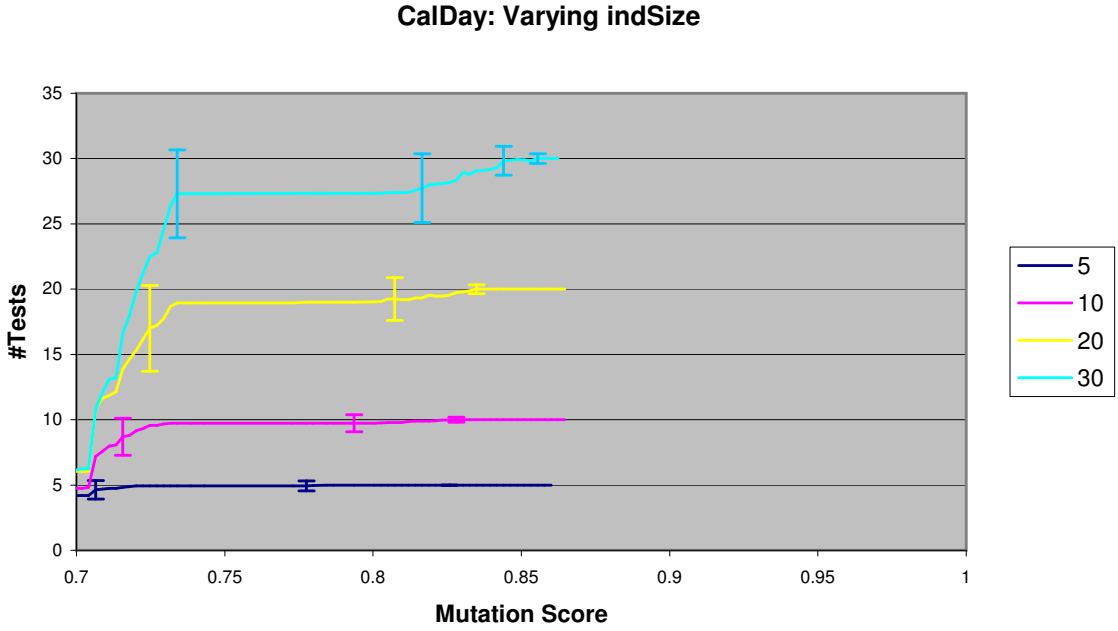

Figure 3.79: Effect of varying the individual size on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program.

`indSize`	DR MS: 87.78%	TRI MS: 70.00%	CD MS: 82.80%
5	5 ± 0.00	5 ± 0.00	5 ± 0
10	9.87 ± 0.35	9.93 ± 0.25	10.00 ± 0.00
20	17.74 ± 2.09	19.43 ± 0.94	19.73 ± 0.58
30	22.52 ± 3.36	28.27 ± 2.50	28.27 ± 2.13
ANOVA			
f_{obt}	**436.97**	**1591.69**	**2525.85**
f_{crit}	2.68	2.69	2.69
Scheffé			
c_{obt}			
5-10	**8.91**	**13.27**	**17.52**
5-20	**24.00**	**38.83**	**52.05**
5-30	**32.79**	**62.60**	**79.36**
10-20	**15.26**	**26.81**	**35.03**
10-30	**24.36**	**51.73**	**63.39**
20-30	**9.50**	**24.93**	**29.85**
c_{crit}	2.83	2.84	2.84

Table 3.36: The mean number of distinct tests needed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the four individual sizes: 5, 10, 20, and 30. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean number of tests produced by each individual size. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *All results are to 2 decimal places.*

C.1.5 crossRate: Effect on Number of Mutant Executions

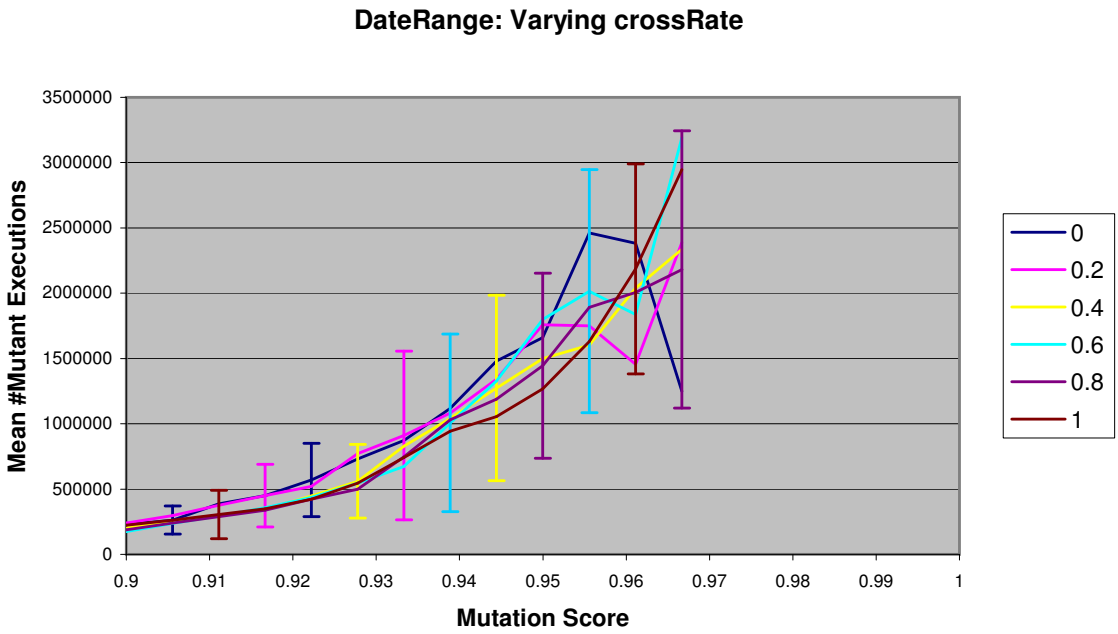

Figure 3.80: Effect of varying the crossover rate on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

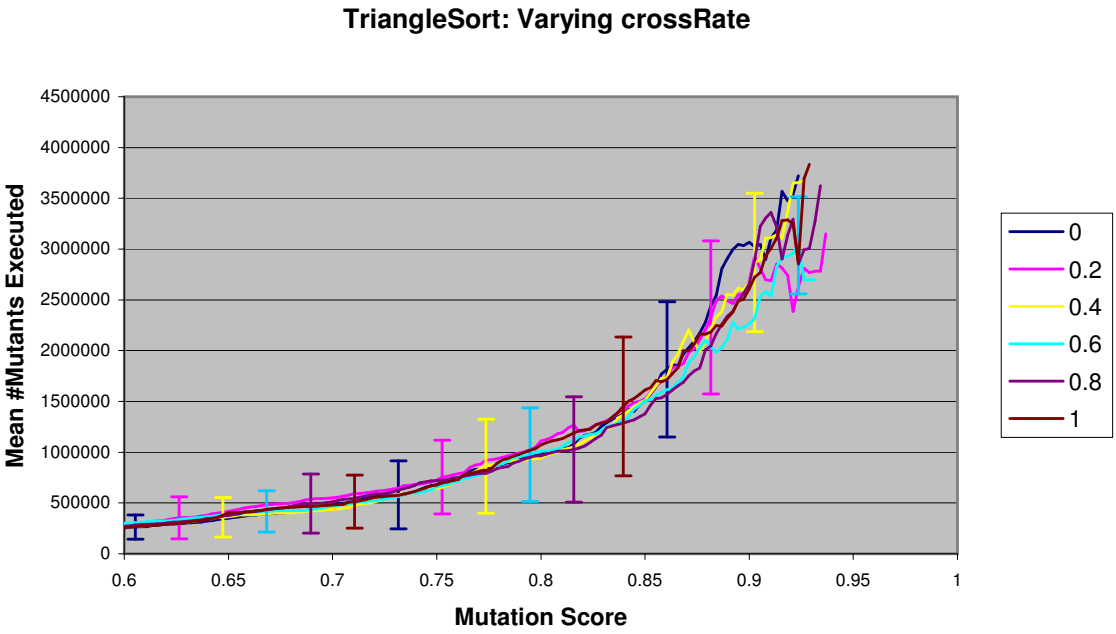

Figure 3.81: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

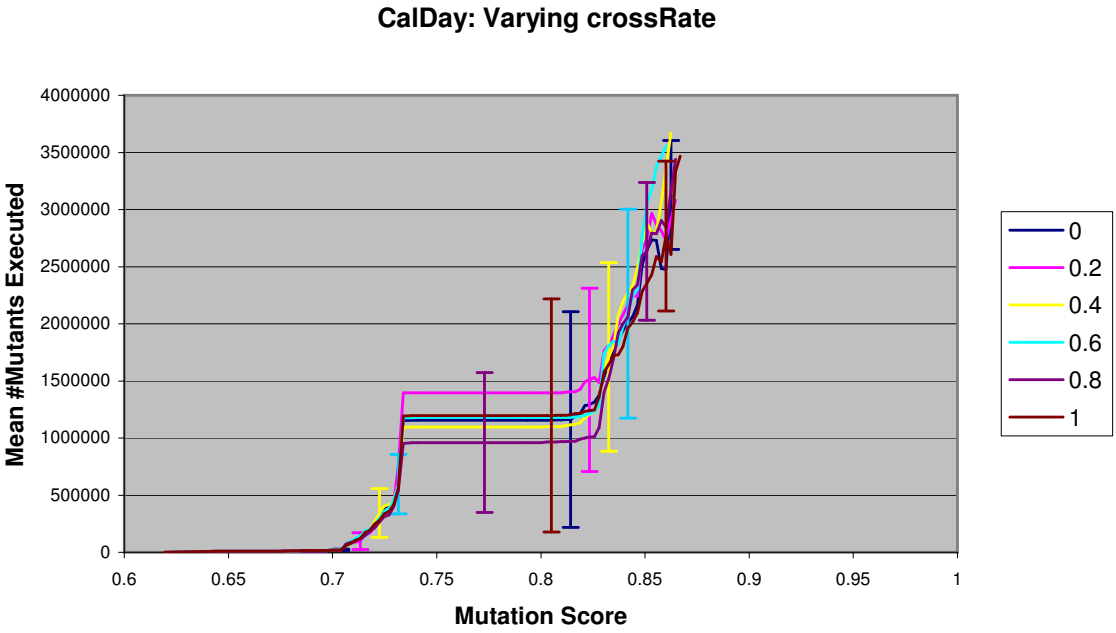

Figure 3.82: Effect of varying the individual size on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

crossRate	DR MS: 94.44%	TRI MS: 82.63%	CD MS: 83.48%
0	1479683 ± 692353	1194916 ± 563506	1755577 ± 761952
0.2	1341053 ± 737400	1273229 ± 732111	1898955 ± 769834
0.4	1274224 ± 710960	1172388 ± 481370	1808078 ± 821265
0.6	1327316 ± 713296	1179859 ± 562372	1847668 ± 1001720
0.8	1188082 ± 706193	1136750 ± 589230	1700971 ± 761583
1.0	1053885 ± 426374	1269004 ± 642899	1722854 ± 927764
ANOVA			
f_{obt}	1.25	0.25	0.22
f_{crit}	2.27	2.27	2.27

Table 3.37: The mean number of mutants executed (and standard deviation) at the highest mutation score, obtained by at least 25 runs for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, 1. The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean values of each rate. *Mutation scores and ANOVA values are to 2 decimal places.*

C.1.6 crossRate: Effect on Mutation Score per Iteration

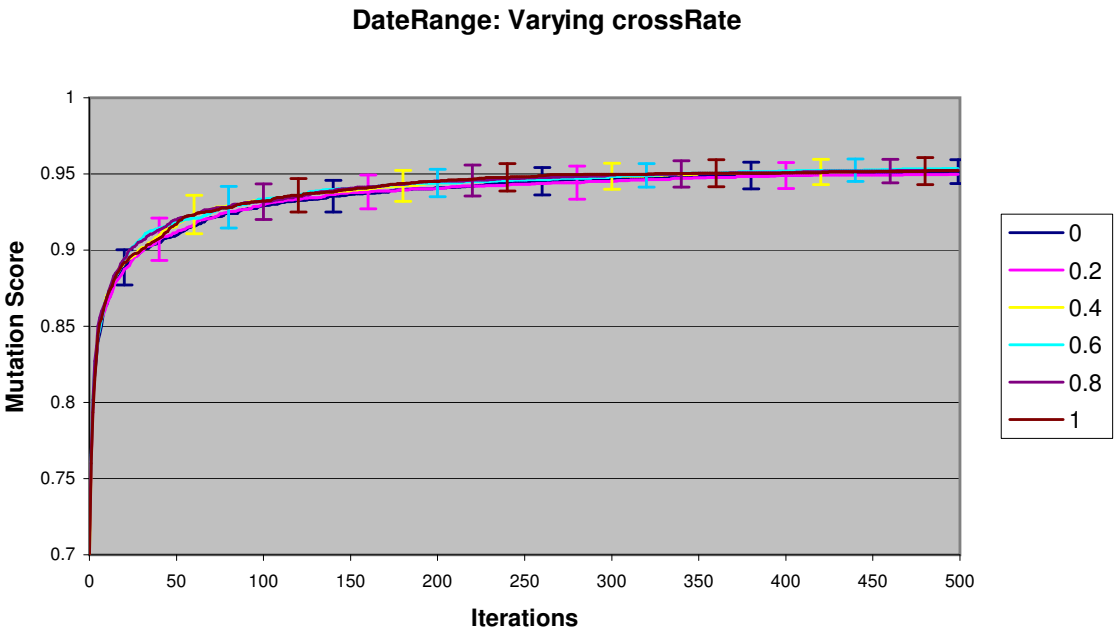

Figure 3.83: Effect of varying the crossover rate on the mean mutation score per iteration for the *DateRange* program.

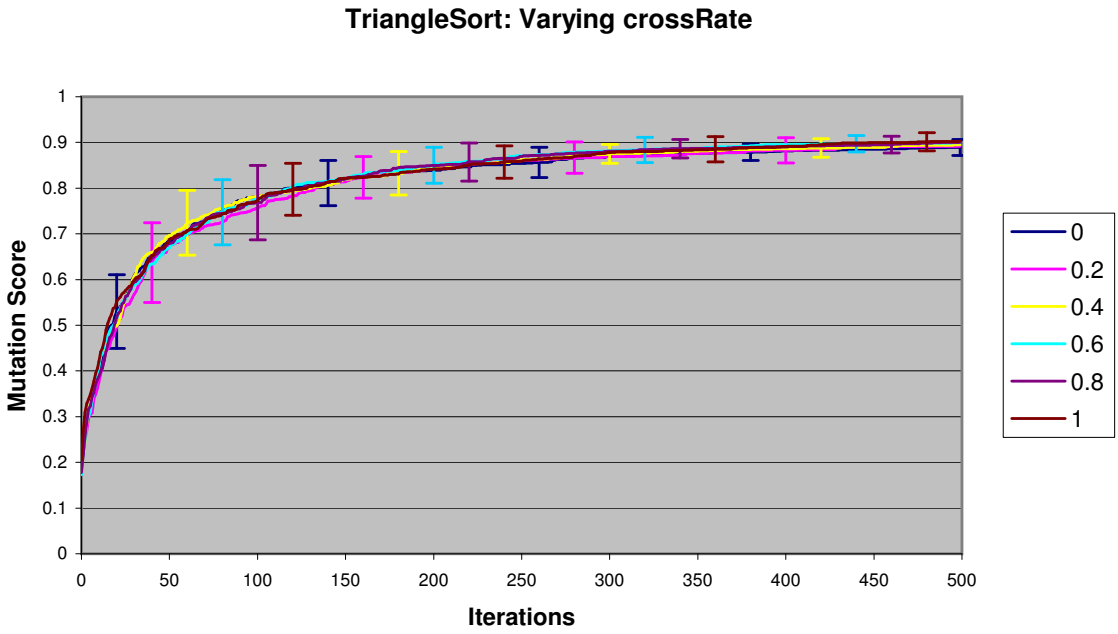

Figure 3.84: Effect of varying the individual size on the mean mutation score per iteration for the *TriangleSort* program.

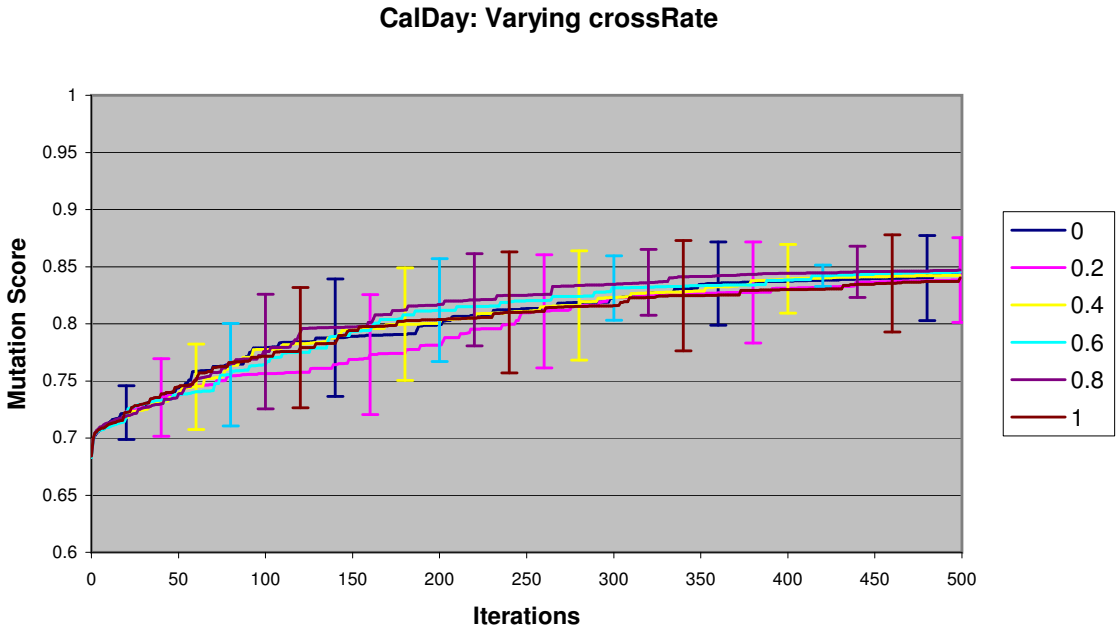

Figure 3.85: Effect of varying the individual size on the mean mutation score per iteration for the *CalDay* program.

`crossRate`	DR	TRI	CD
0	95.16% ± 0.79	88.90% ± 1.79	84.40% ± 3.15
0.2	94.97% ± 0.87	89.21% ± 2.37	83.83% ± 3.71
0.4	95.22% ± 0.75	89.50% ± 1.92	84.24% ± 3.09
0.6	95.35% ± 0.68	89.94% ± 1.67	84.50% ± 0.88
0.8	95.26% ± 0.80	90.05% ± 1.40	84.70% ± 2.24
1.0	95.20% ± 0.91	90.23% ± 1.84	84.02% ± 3.81
ANOVA			
f_{obt}	0.82	**2.36**	0.35
f_{crit}	2.26	2.27	2.27
Scheffé			
c_{obt}			
0-0.2			
⋮	-	all $< c_{crit}$	-
0.8-1			
c_{crit}	-	3.37	-

Table 3.38: Mean mutation scores (and standard deviation) achieved by each crossover rate after 500 iterations. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean mutation scores achieved by each crossover rate. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.1.7 crossRate: Effect on Number of HTK identified

`crossRate`	DR	TRI	CD
0	71.88% ± 3.27%	22.78% ± 7.87%	34.38% ± 0.00%
0.2	71.46% ± 3.87%	22.50% ± 9.32%	34.38% ± 0.00%
0.4	72.00% ± 2.71%	24.19% ± 8.96%	34.38% ± 0.00%
0.6	71.78% ± 3.79%	24.72% ± 9.15%	34.38% ± 0.00%
0.8	70.98% ± 3.63%	27.22% ± 9.52%	34.38% ± 0.00%
1	70.67% ± 4.83%	27.50% ± 10.53%	34.48% ± 0.57%
ANOVA			
f_{obt}	0.63	1.60	1.01
f_{crit}	2.26	2.27	2.27

Table 3.39: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, 1. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of crossover rates result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.1.8 crossRate: Effect on Number of Tests

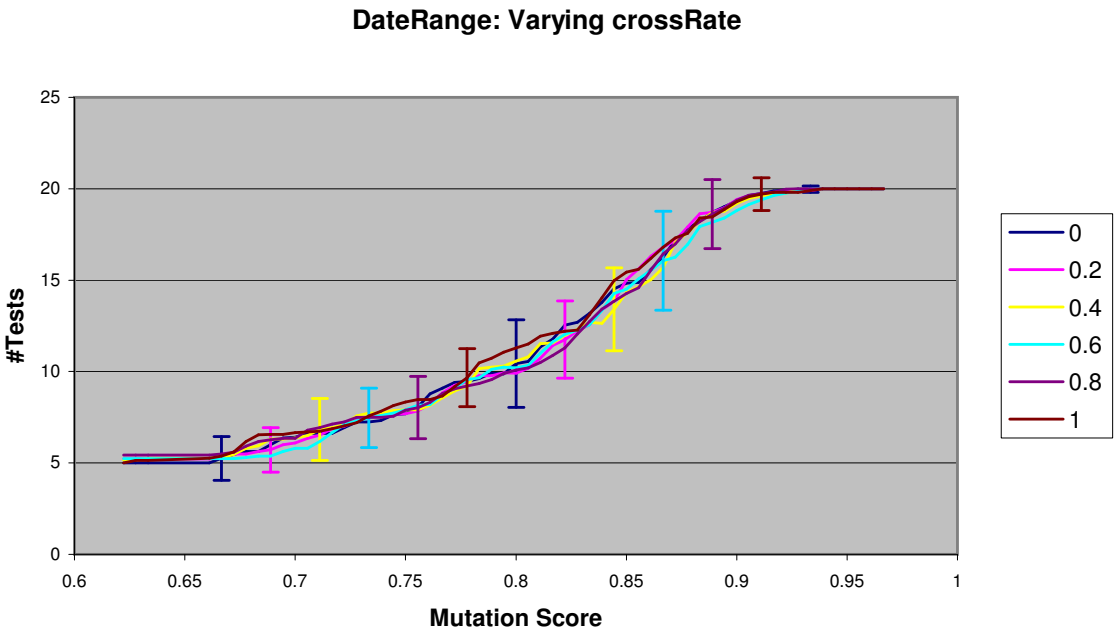

Figure 3.86: Effect of varying the crossover rate on the mean number of Tests generated to achieve specific mutation scores for the *DateRange* program.

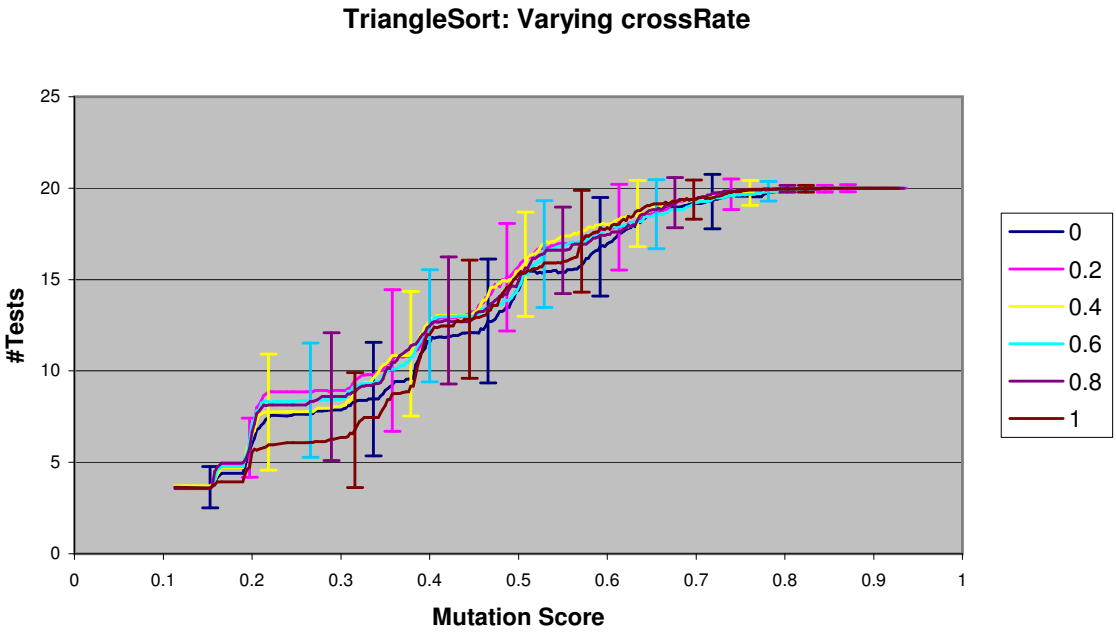

Figure 3.87: Effect of varying the individual size on the mean number of Tests generated to achieve specific mutation scores for the *TriangleSort* program.

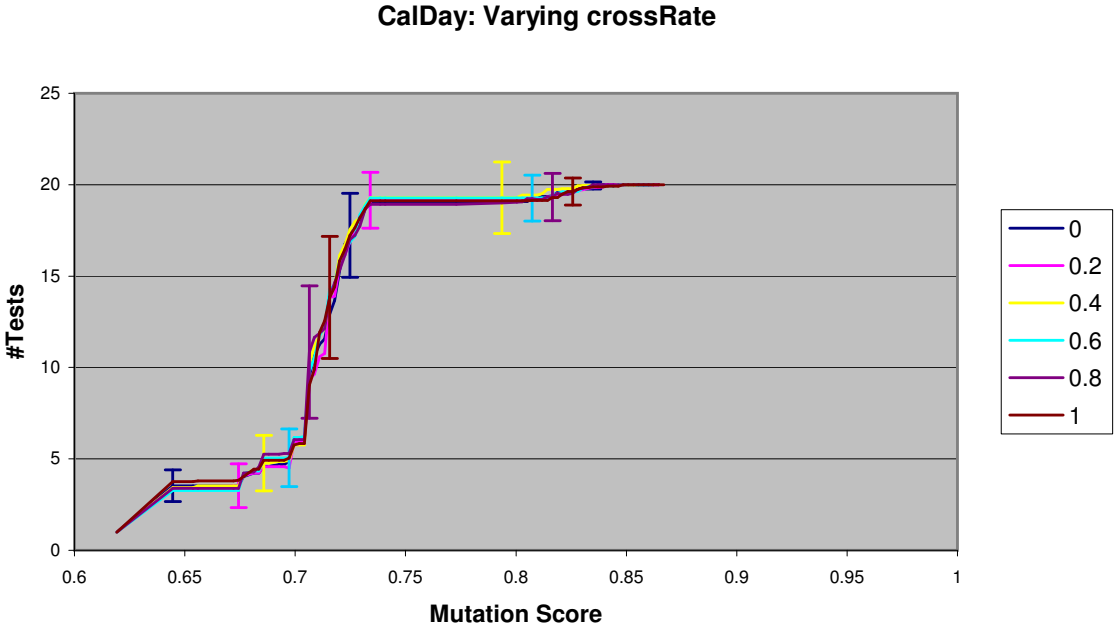

Figure 3.88: Effect of varying the individual size on the mean number of Tests generated to achieve specific mutation scores for the *CalDay* program.

crossRate	DR MS: 94.44%	TRI MS: 82.63%	CD MS: 83.48%
0	20.00 ± 0.00	20.00 ± 0.00	19.96 ± 0.19
0.2	20.00 ± 0.00	19.93 ± 0.26	19.84 ± 0.47
0.4	20.00 ± 0.00	19.94 ± 0.25	19.96 ± 0.19
0.6	20.00 ± 0.00	19.93 ± 0.25	19.88 ± 0.43
0.8	20.00 ± 0.00	20.00 ± 0.00	20.00 ± 0.00
1.0	20.00 ± 0.00	19.97 ± 0.18	19.88 ± 0.44
ANOVA			
f_{obt}	-	0.86	0.97
f_{crit}	2.27	2.27	2.27

Table 3.40: Mean test set sizes (and standard deviations) at the highest mutation score, obtained by at least 25 runs for each of the six crossover rates: 0, 0.2, 0.4, 0.6, 0.8, and 1. ANOVA f_{obt} and f_{crit} values (at the 0.05 level) are given to indicate whether there is any significant difference between the mean values of each rate; a f_{obt} value on "-" indicates that the value could not be calculated as all six crossover rates exhibited the same test set size (with no variation). *All results are to 2 decimal places.*

C.1.9 mutRate: Effect on Number of Mutant Executions

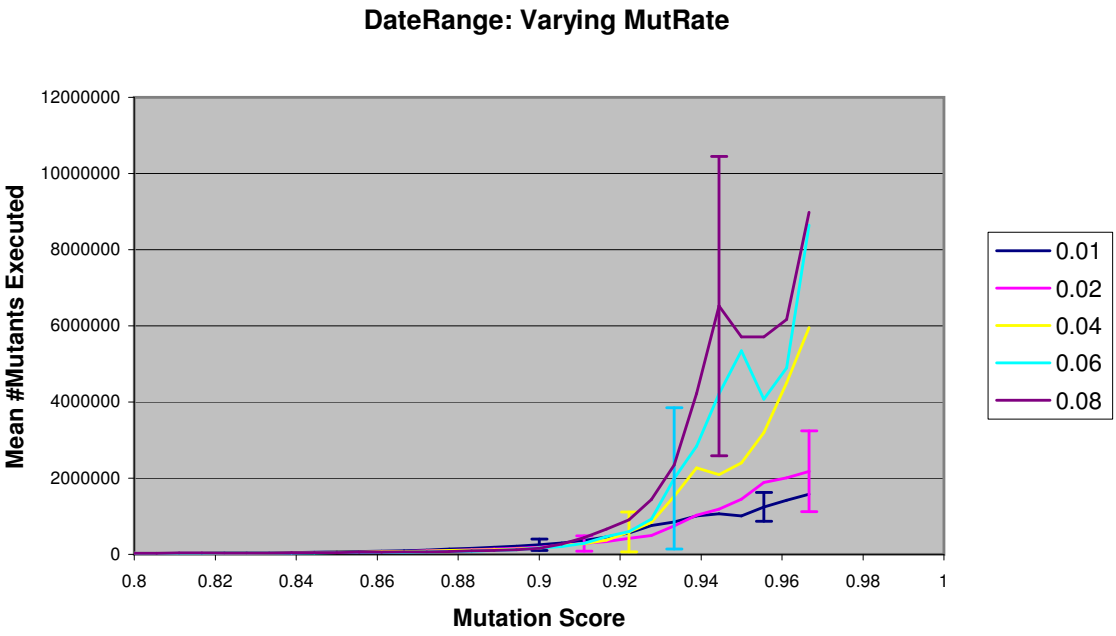

Figure 3.89: Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

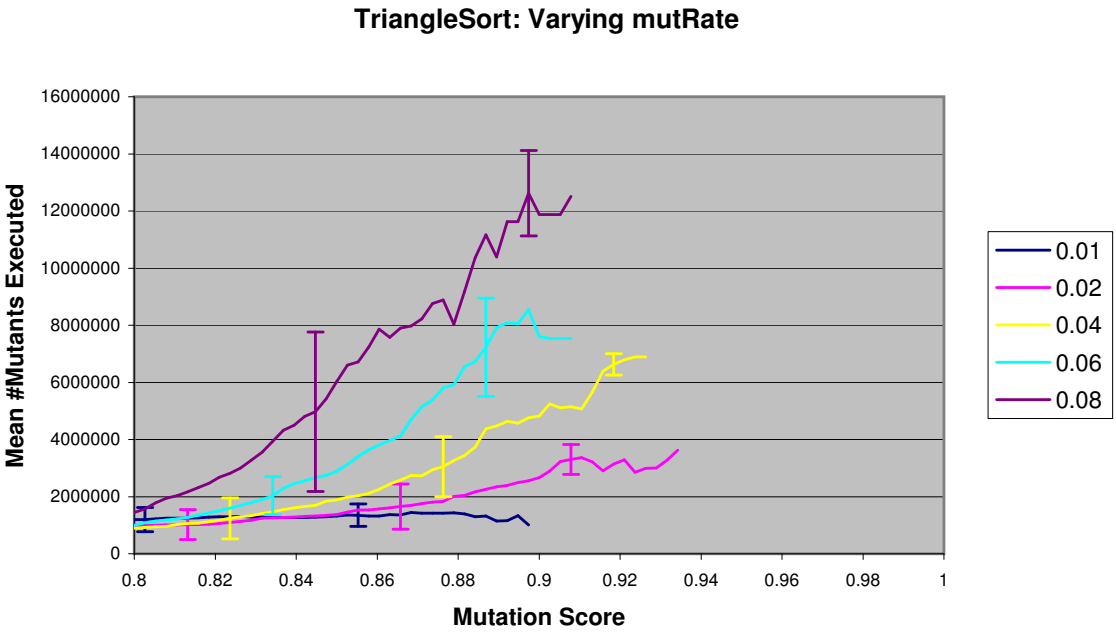

Figure 3.90: Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

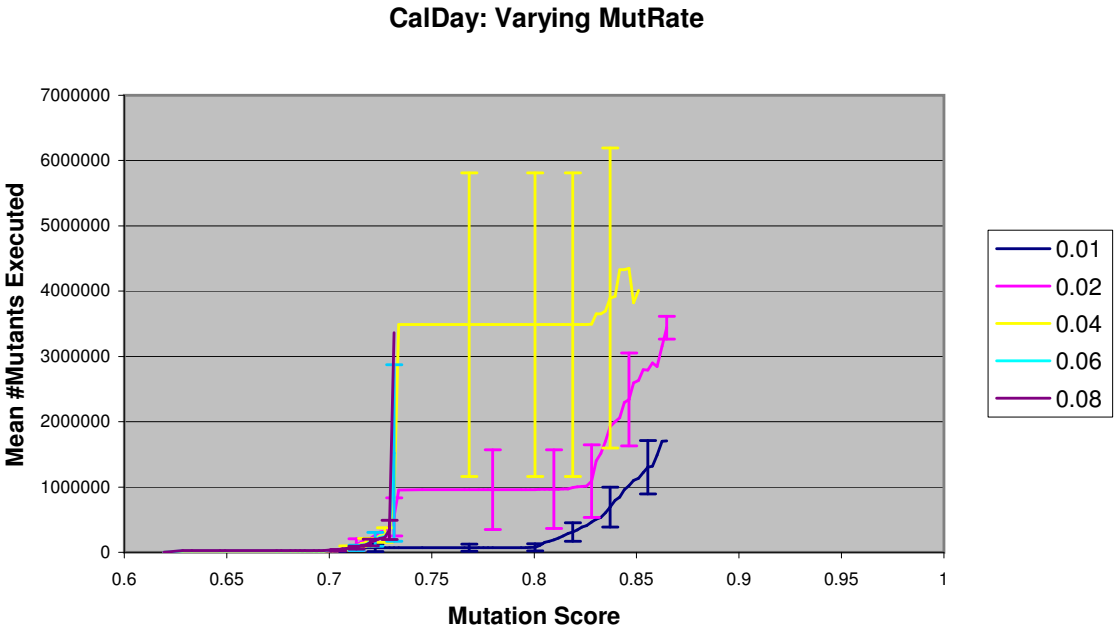

Figure 3.91: Effect of varying mutRate on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

mutRate	DR MS: 93.33%	TRI MS: 78.95%	CD MS: 73.17%
0.01	854944 ± 365255	1183267 ± 439927	69865 ± 56654
0.02	747627 ± 417325	910284 ± 533408	542702 ± 290922
0.04	1521722 ± 1209835	762485 ± 385845	815527 ± 592762
0.06	1995207 ± 1850647	845636 ± 435658	1522942 ± 1352530
0.08	2344008 ± 2050275	1212915 ± 829261	3364980 ± 2586160
ANOVA			
f_{obt}	**7.97**	**3.91**	**29.88**
f_{crit}	2.43	2.44	2.42
Scheffé			
c_{obt}			
0.01-0.02	0.31	1.83	1.28
0.01-0.04	1.88	2.81	2.03
0.01-0.06	**3.20**	2.26	**4.31**
0.01-0.08	**4.14**	0.20	**9.60**
0.02-0.04	2.29	1.04	0.74
0.02-0.06	**3.66**	0.45	2.91
0.02-0.08	**4.64**	2.12	**8.22**
0.04-0.06	1.35	0.58	2.10
0.04-0.08	2.32	**3.16**	**7.43**
0.06-0.08	0.98	2.58	**5.96**
c_{crit}	3.12	3.12	3.11

Table 3.41: The mean number of mutants executed (and standard deviation) at the highest mutation score obtained by at least 25 runs for each of the five mutation rates: 0.01, 0.02, 0.04, 0.06, 0.08. ANOVA and Scheffé values (at the 0.05 level) indicating whether differences between the means obtained using different pairs of mutation rates are significant. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *The number executed (and any standard deviation) has been rounded up to the nearest whole number to reflect that a mutant is either executed completely or not at all. All other results are to 2 decimal places.*

C.1.10 mutRate: Effect on Mutation Score per Iteration

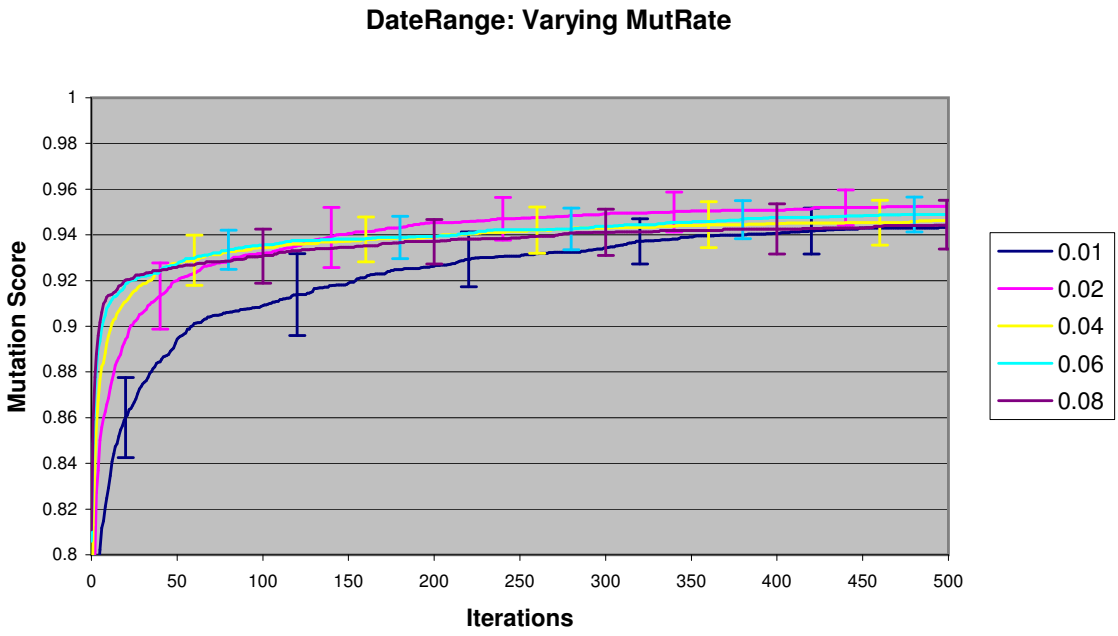

Figure 3.92: Effect of varying mutRate on the mean mutation score per iteration for the *DateRange* program.

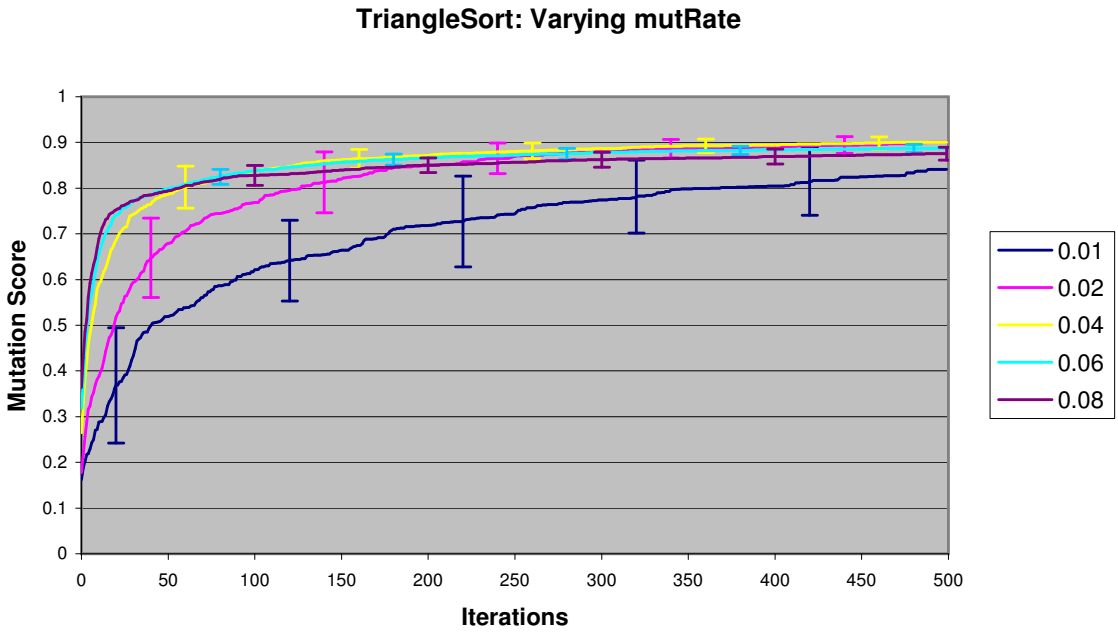

Figure 3.93: Effect of varying mutRate on the mean mutation score per iteration for the *TriangleSort* program.

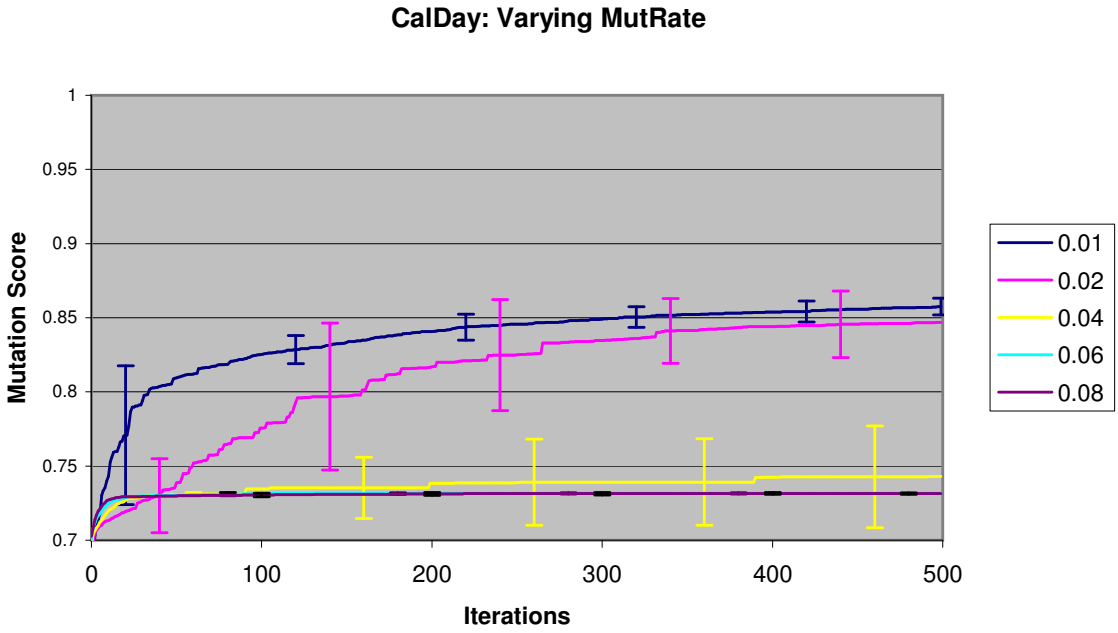

Figure 3.94: Effect of varying mutRate on the mean mutation score per iteration for the *CalDay* program.

`mutRate`	DR	TRI	CD
0.01	94.32% ± 1.08%	84.15% ± 5.45%	85.75% ± 0.56%
0.02	95.26% ± 0.80%	90.05% ± 1.40%	84.70% ± 2.24%
0.04	94.63% ± 1.02%	90.04% ± 1.33%	74.29% ± 3.49%
0.06	94.89% ± 0.76%	88.70% ± 0.87%	73.17% ± 0.00%
0.08	94.44% ± 1.07%	87.56% ± 1.40%	73.15% ± 0.00%
ANOVA			
f_{obt}	**5.03**	**24.77**	**484.28**
f_{crit}	2.43	2.43	2.42
Scheffé			
c_{obt}			
0.01-0.02	**3.98**	**8.51**	2.40
0.01-0.04	1.26	**8.50**	**26.23**
0.01-0.06	2.33	**6.56**	**31.49**
0.01-0.08	0.50	**4.92**	**31.23**
0.02-0.04	2.64	0.01	**23.83**
0.02-0.06	1.56	1.95	**28.86**
0.02-0.08	**3.41**	**3.59**	**28.62**
0.04-0.06	1.05	1.93	2.81
0.04-0.08	0.75	**3.58**	2.81
0.06-0.08	1.80	1.64	0.03
c_{crit}	3.12	3.12	3.11

Table 3.42: The mean mutation score (and standard deviation) achieved after 500 iterations for each of the five mutation rates (0.01, 0.02, 0.04, 0.06, 0.08). ANOVA and Scheffé values (at the 0.05 level) indicating whether differences between the means obtained using different pairs of mutation rates are significant. **Bold** values indicate a significant result ($obt \geq crit$) - i.e. the null hypothesis should be rejected. *All results are to 2 decimal places.*

C.1.11 mutRate: Effect on Number of HTK identified

mutRate	DR	TRI	CD
0.01	66.04% ± 6.64	22.78% ± 6.90	34.07% ± 2.33
0.02	70.98% ± 3.63	27.22% ± 9.52	34.38% ± 0.00
0.04	71.11% ± 4.74	19.44% ± 7.37	34.38% ± 0.00
0.06	72.22% ± 3.07	17.78% ± 6.83	34.38% ± 0.00
0.08	71.11% ± 3.64	17.78% ± 6.47	34.38% ± 0.00
ANOVA			
f_{obt}	**9.05**	**8.68**	0.64
f_{crit}	2.43	2.43	2.42
Scheffé			
c_{obt}			
0.01-0.02	**4.42**	2.30	-
0.01-0.04	**4.40**	1.72	-
0.01-0.06	**5.36**	2.58	-
0.01-0.08	**4.40**	2.58	-
0.02-0.04	0.11	**4.02**	-
0.02-0.06	1.09	**4.88**	-
0.02-0.08	0.11	**4.88**	-
0.04-0.06	0.95	0.86	-
0.04-0.08	0.00	0.86	-
0.06-0.08	0.95	0.00	-
c_{crit}	3.12	3.12	-

Table 3.43: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the five mutation rates: 0.01, 0.02, 0.04, 0.06, 0.08. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of mutation rates result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.1.12 mutRate: Effect on the Number of Tests

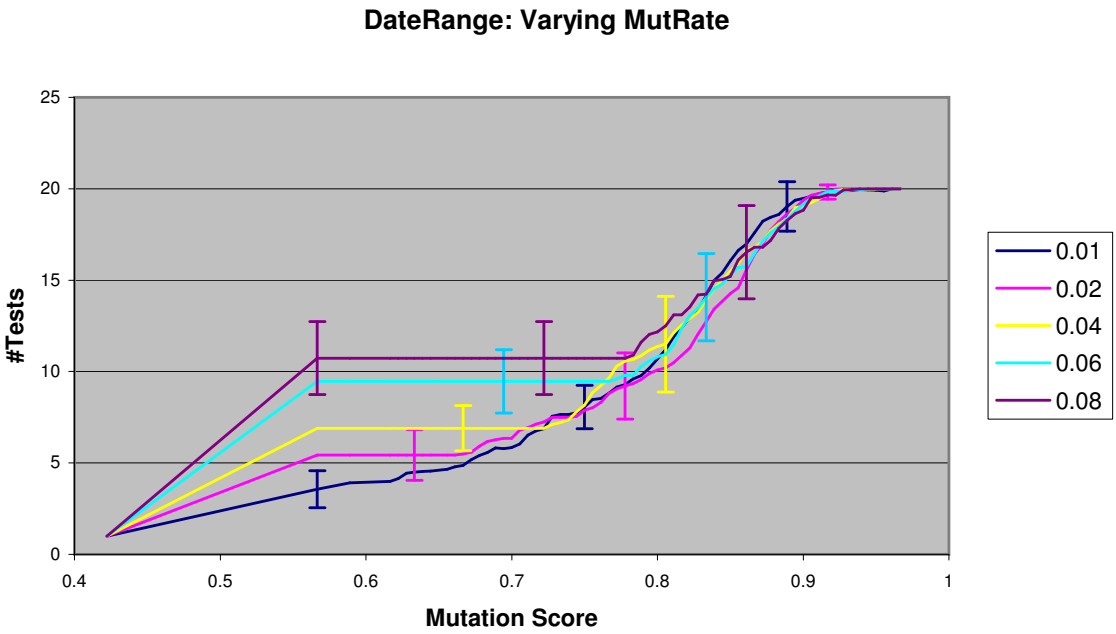

Figure 3.95: Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *DateRange* program.

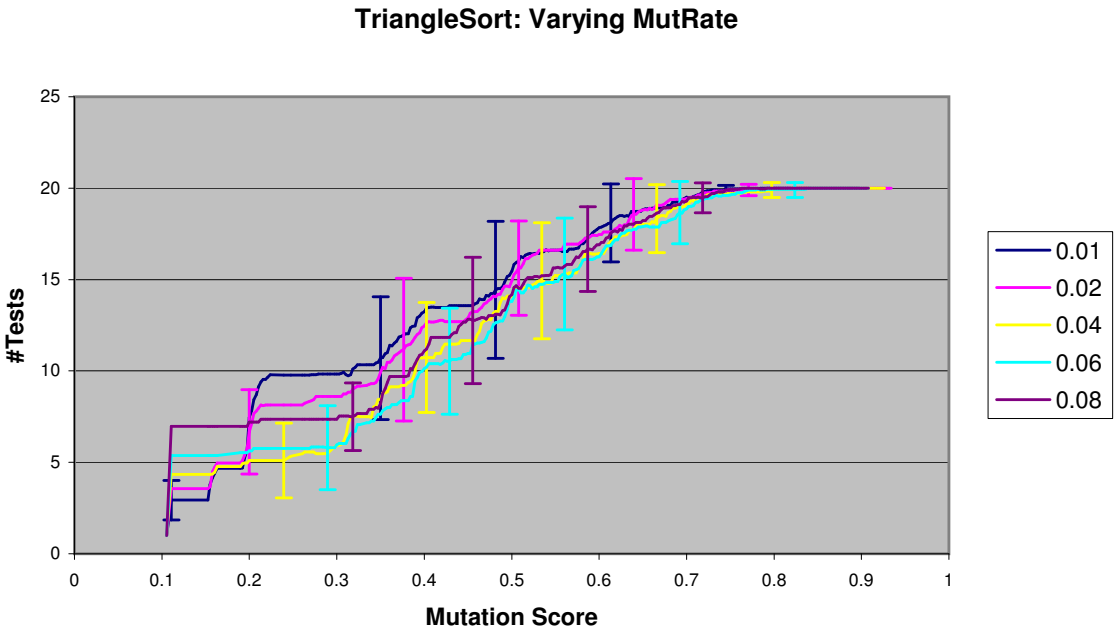

Figure 3.96: Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *TriangleSort* program.

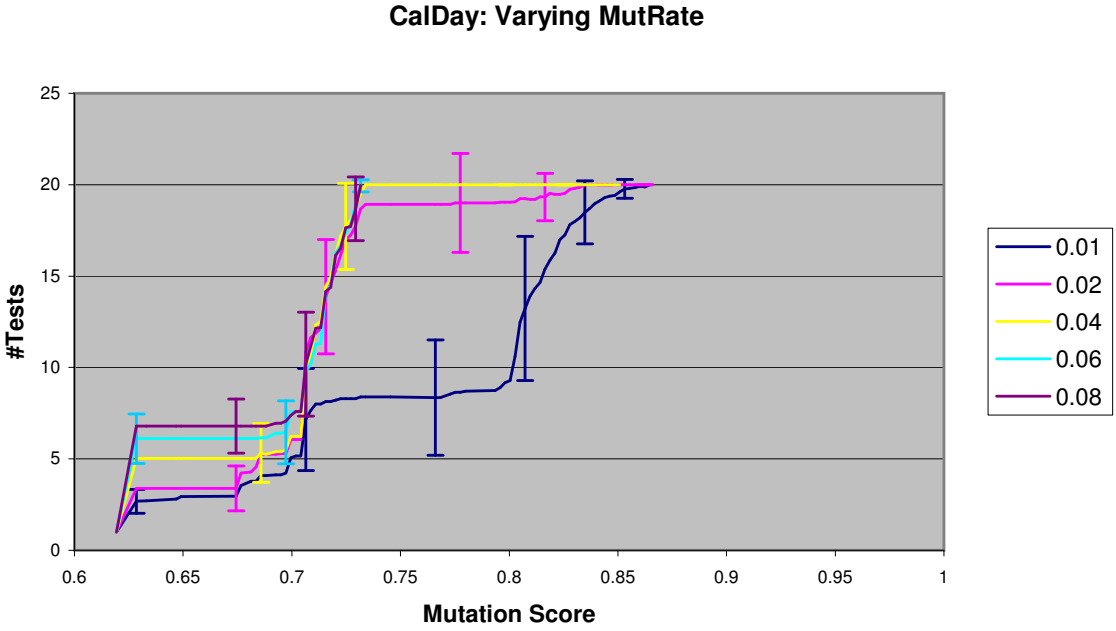

Figure 3.97: Effect of varying the mutation rate on the mean number of tests needed to achieve a specific mutation score for the *CalDay* program.

C.2 Immune Inspired Algorithm for Mutation Testing

C.2.1 nFittest: Effect on Number of Mutant Executions

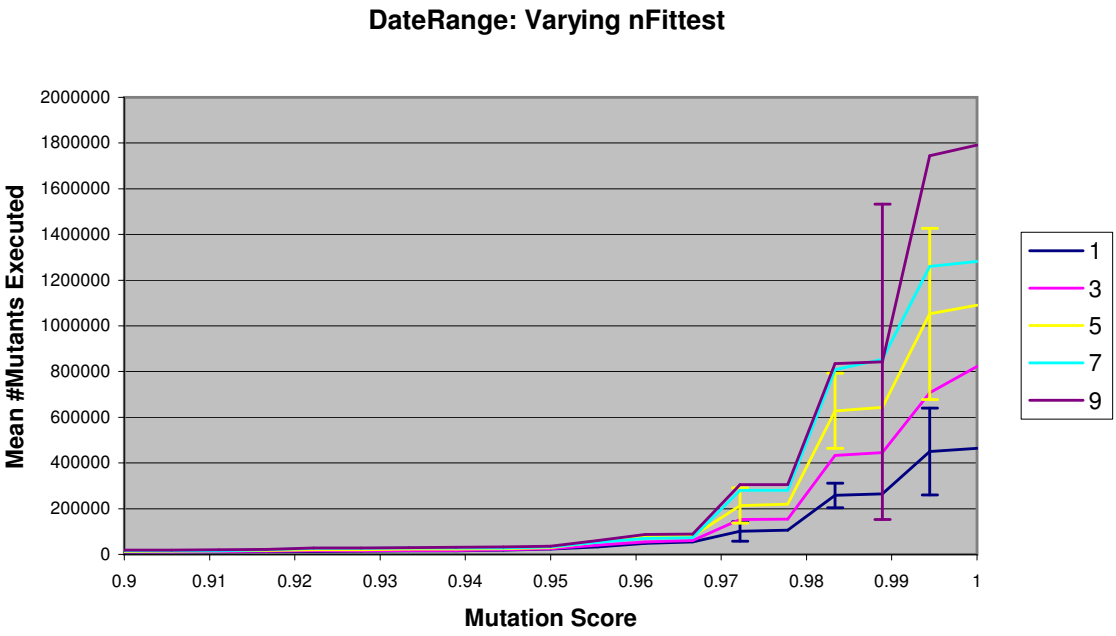

Figure 3.98: Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

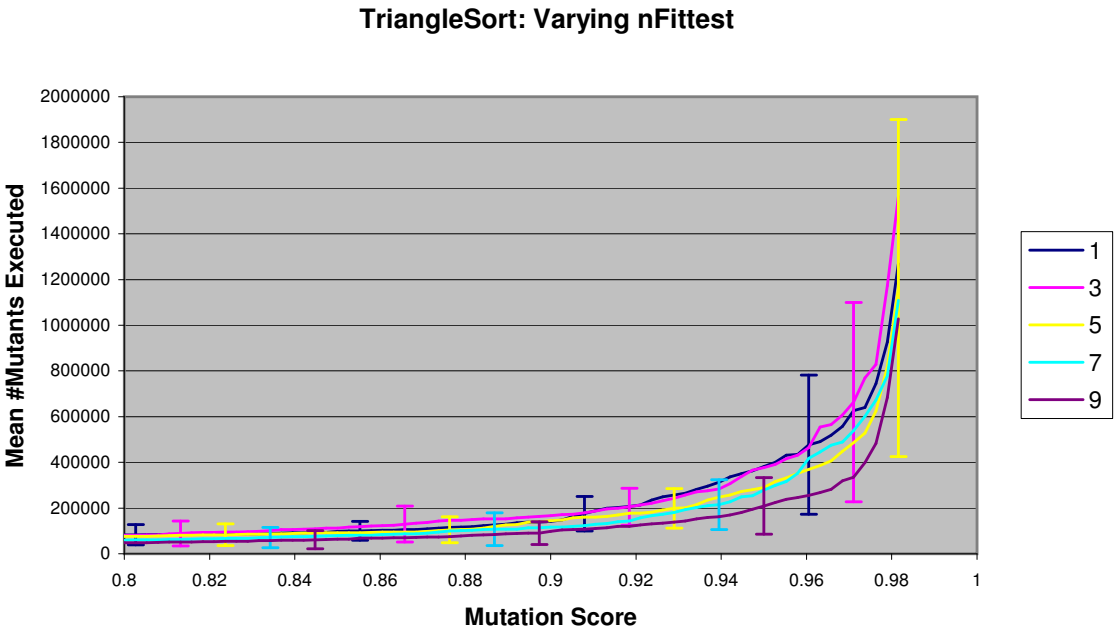

Figure 3.99: Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

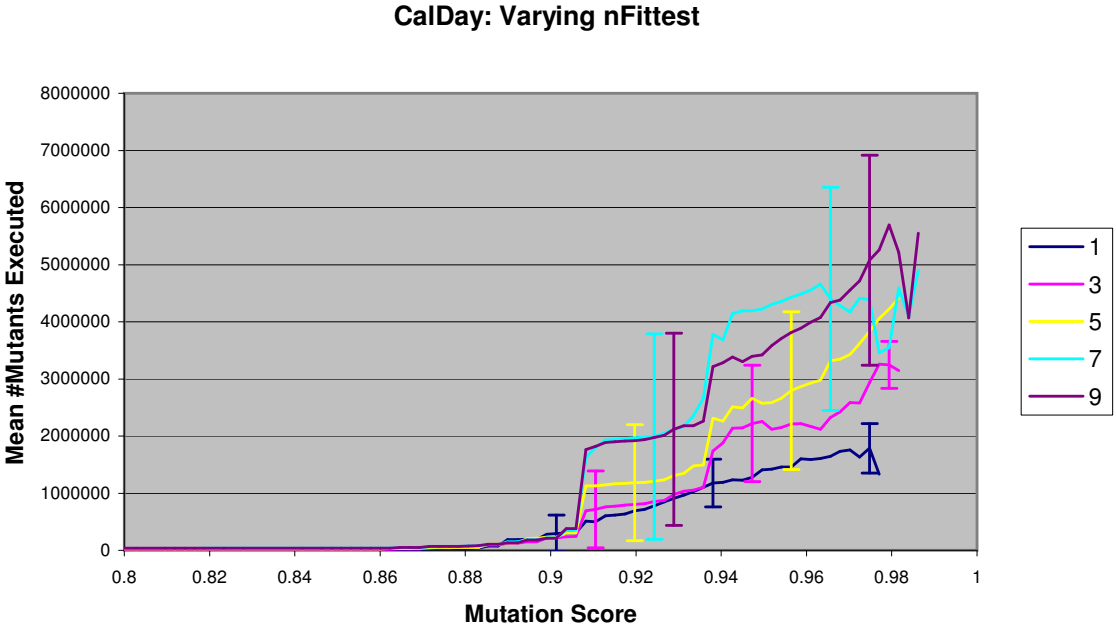

Figure 3.100: Effect of varying `nFittest` on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

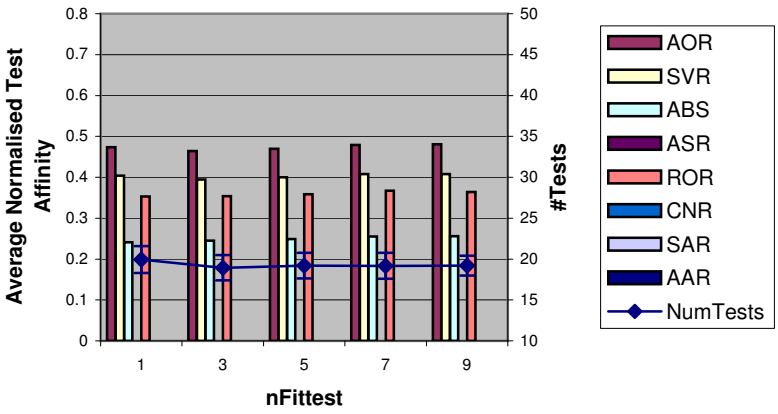

Figure 3.101: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program.

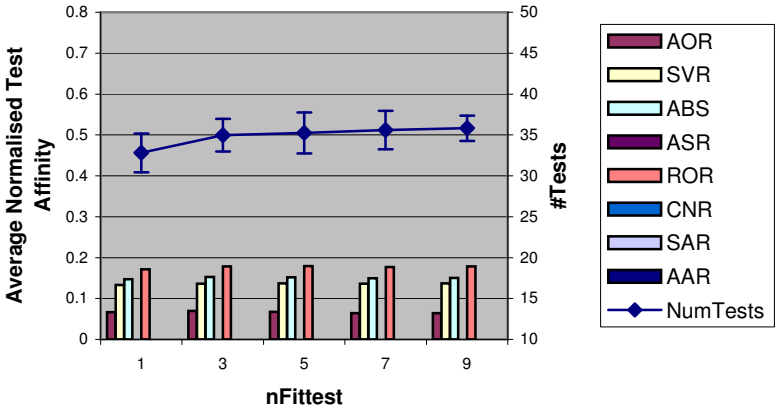

Figure 3.102: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program.

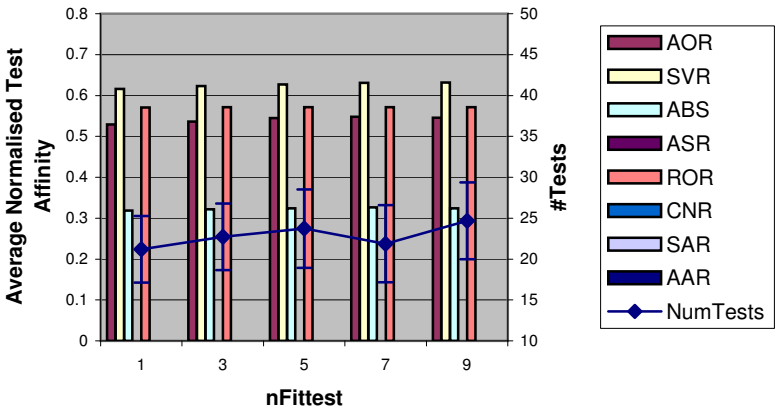

Figure 3.103: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program.

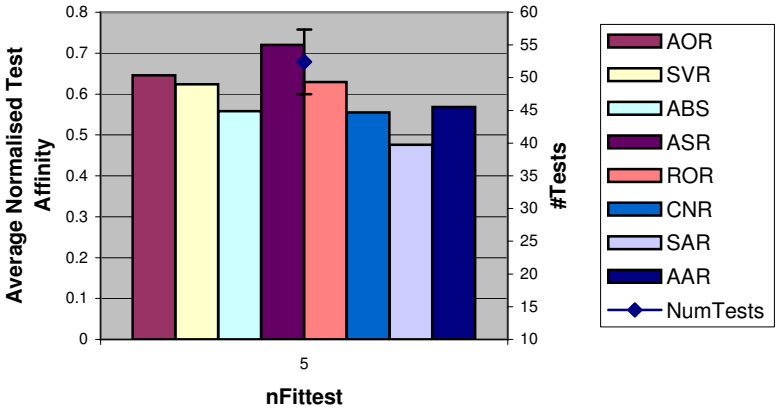

Figure 3.104: Effect of varying `nFittest` on the average memory set size and memory test affinity for each mutation operator, for the *Select* program.

C.2.2 nFittest: Effect on Mutation Score per Iteration

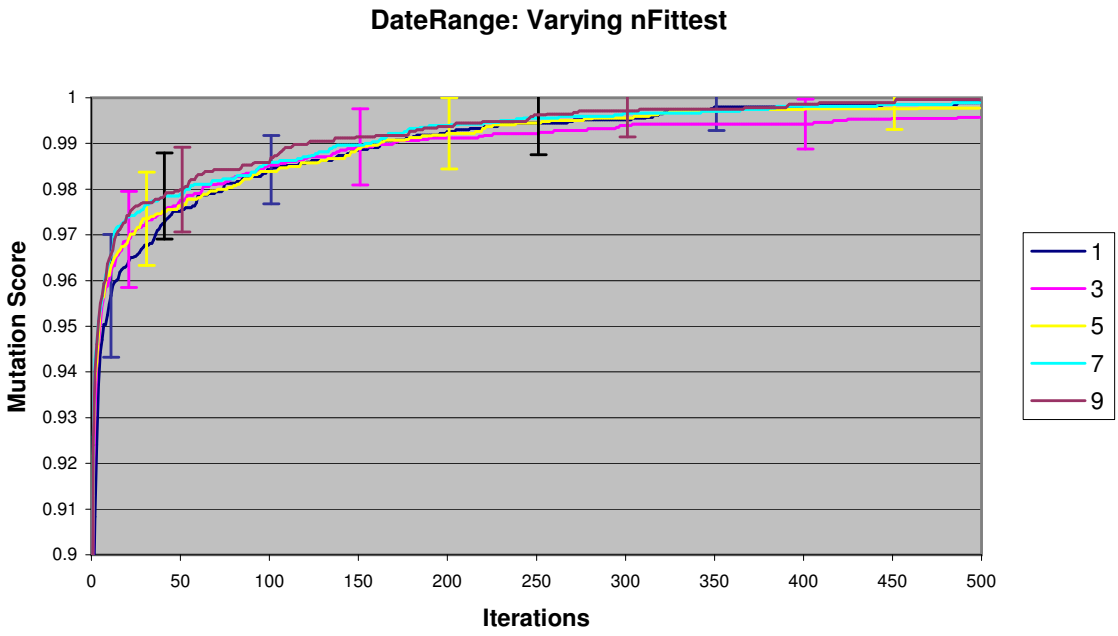

Figure 3.105: Effect of varying `nFittest` on the mean mutation score per iteration for the *DateRange* program.

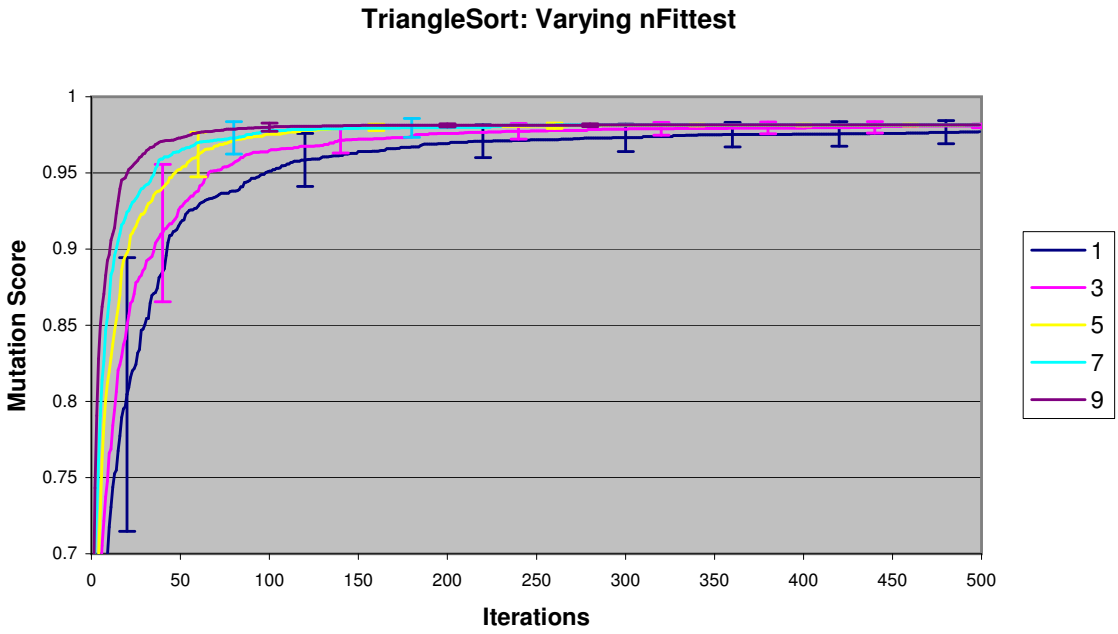

Figure 3.106: Effect of varying `nFittest` on the mean mutation score per iteration for the *TriangleSort* program.

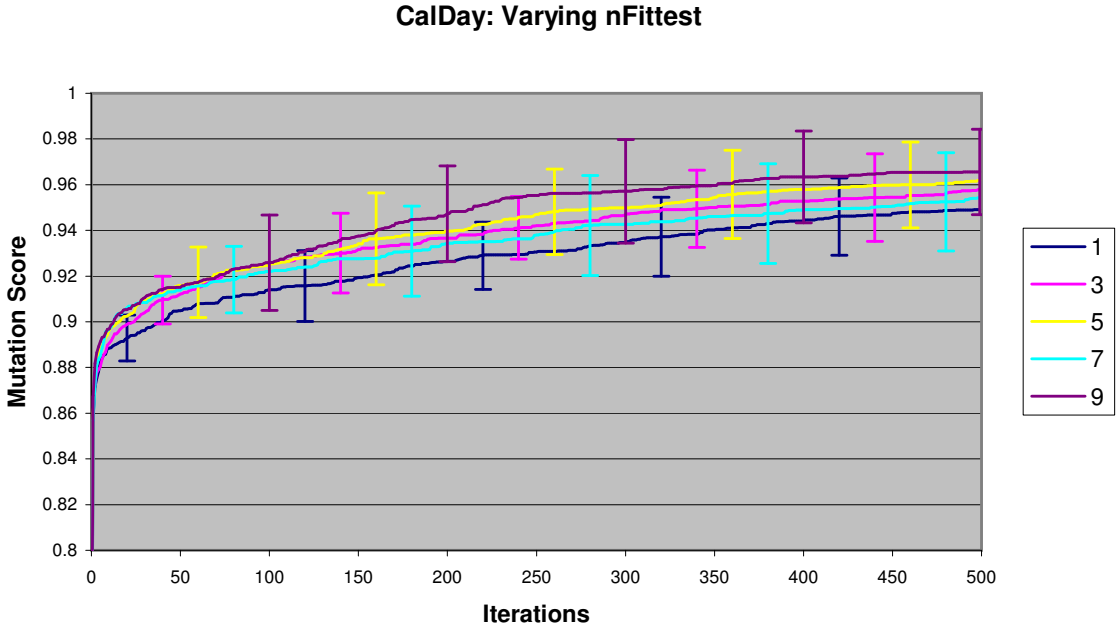

Figure 3.107: Effect of varying `nFittest` on the mean mutation score per iteration for the *CalDay* program.

C.2.3 nFittest: Effect on Number of HTK identified

`crossRate`	DR	TRI	CD
1	98.67% ± 1.95%	72.31% ± 3.12%	76.10% ± 1.20%
3	94.84% ± 1.48%	75.00% ± 2.69%	81.53% ± 1.35%
5	97.33% ± 1.47%	75.00% ± 2.14%	81.63% ± 1.17%
7	98.67% ± 1.51%	75.00% ± 2.65%	79.64% ± 1.19%
9	99.57% ± 1.28%	75.00% ± 2.37%	81.96% ± 1.26%
ANOVA			
f_{obt}	**4.69**	**8.20**	**4.83**
f_{crit}	2.43	2.43	2.43
1-3	**3.16**	**4.56**	**3.45**
1-5	1.09	**4.60**	**3.51**
1-7	0.00	**4.53**	2.21
1-9	0.75	**4.63**	**3.66**
3-5	2.06	0.00	0.06
3-7	**3.16**	0.00	1.18
3-9	**3.94**	0.00	0.26
5-7	1.09	0.00	1.24
5-9	1.85	0.00	0.20
7-9	0.75	0.00	1.42
c_{crit}	3.12	3.12	3.12

Table 3.44: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the five `nFittest` values: 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `nFittest` values result in significantly different mean percentages (in **bold**). *All results are to 2 decimal places.*

C.2.4 nFittest: Effect on Number of Tests

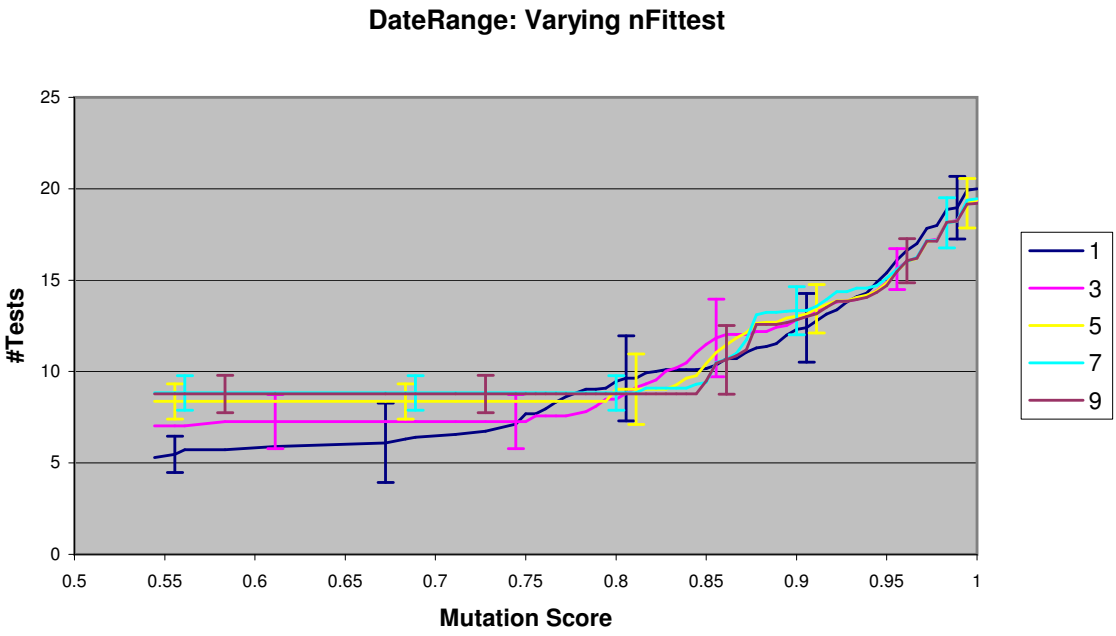

Figure 3.108: Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program.

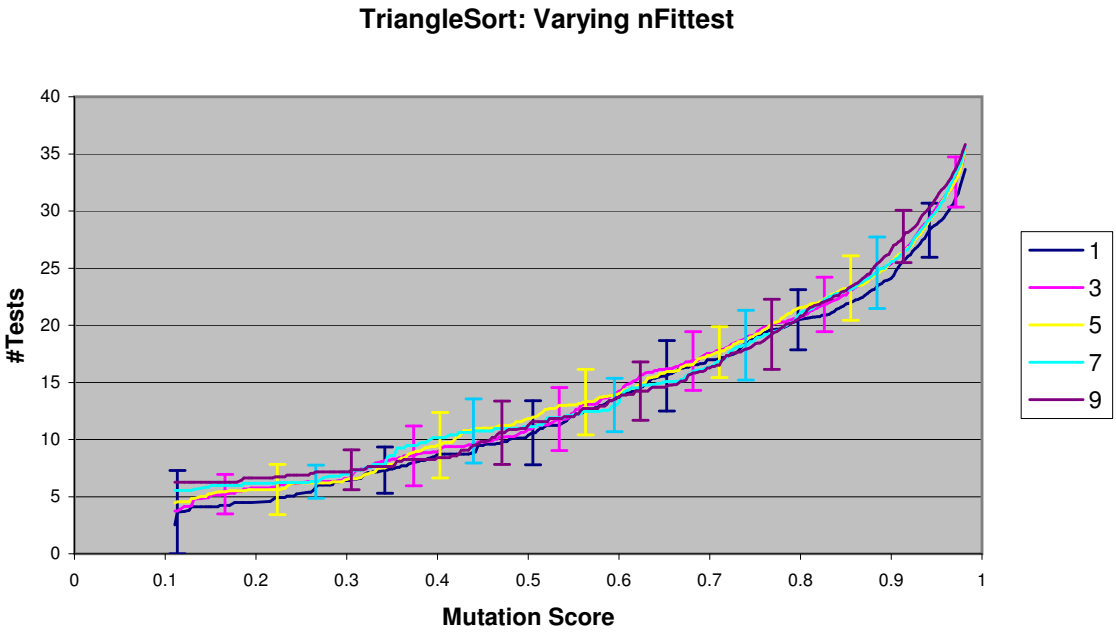

Figure 3.109: Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program.

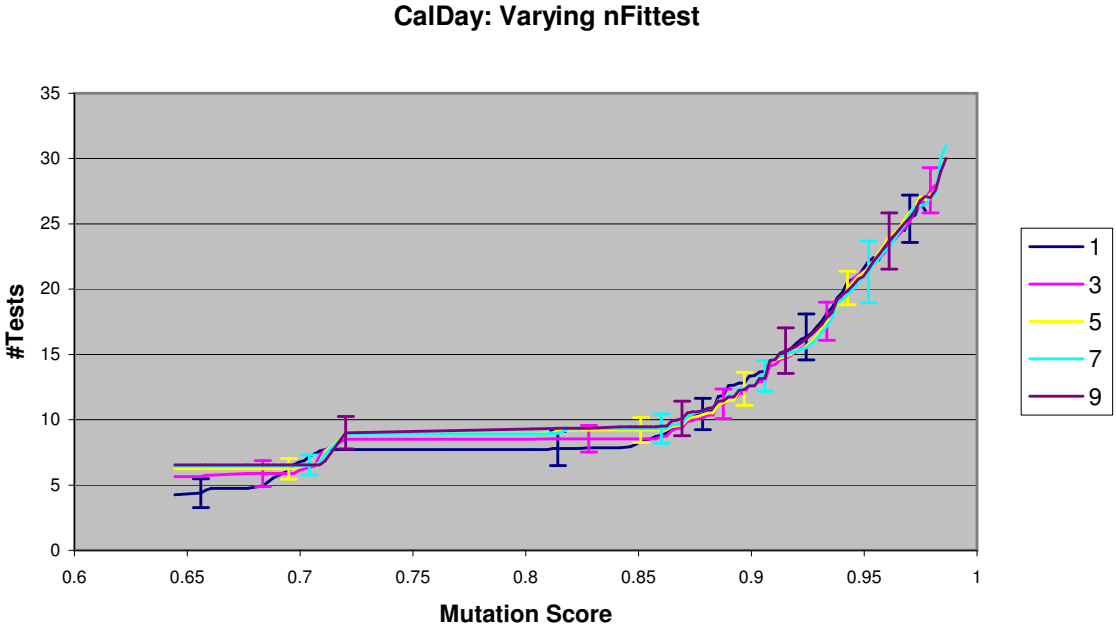

Figure 3.110: Effect of varying `nFittest` on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program.

C.2.5 nWorst: Effect on Number of Mutant Executions

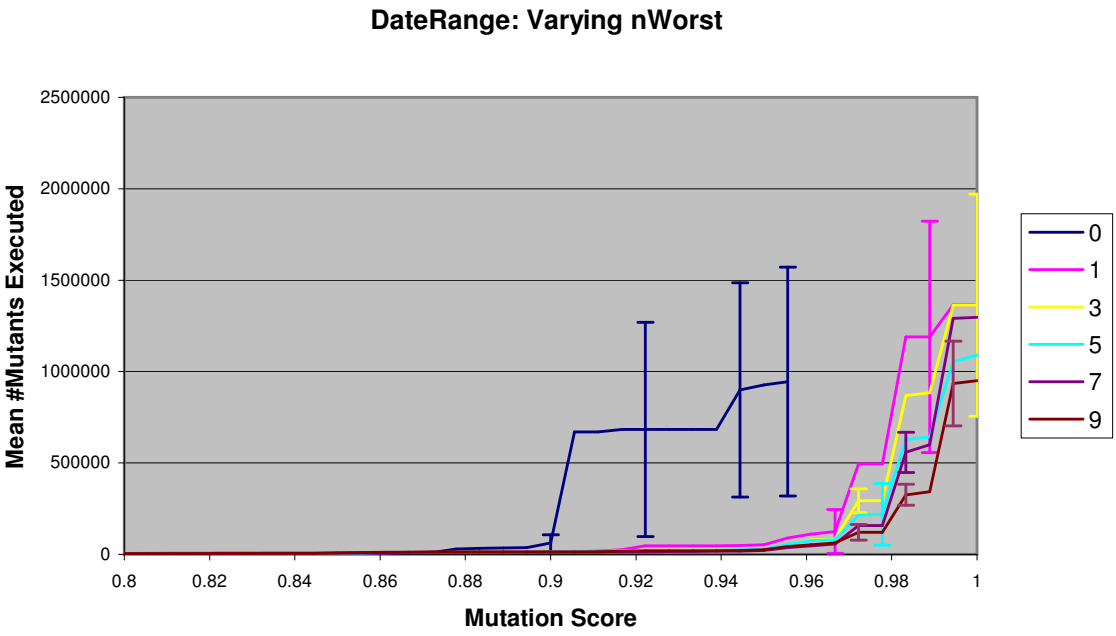

Figure 3.111: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

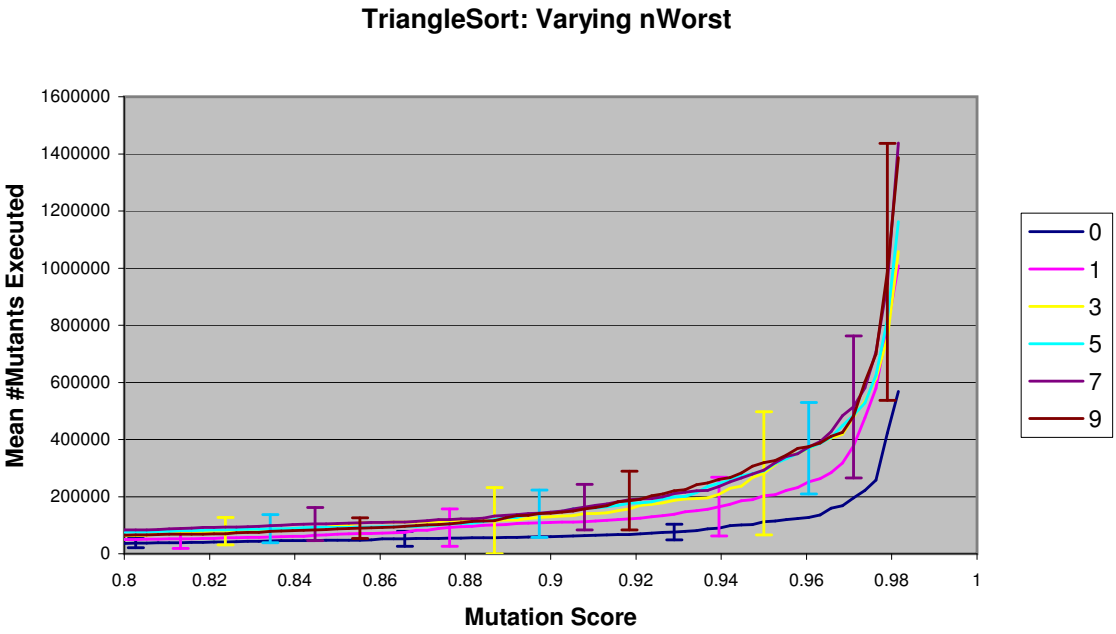

Figure 3.112: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

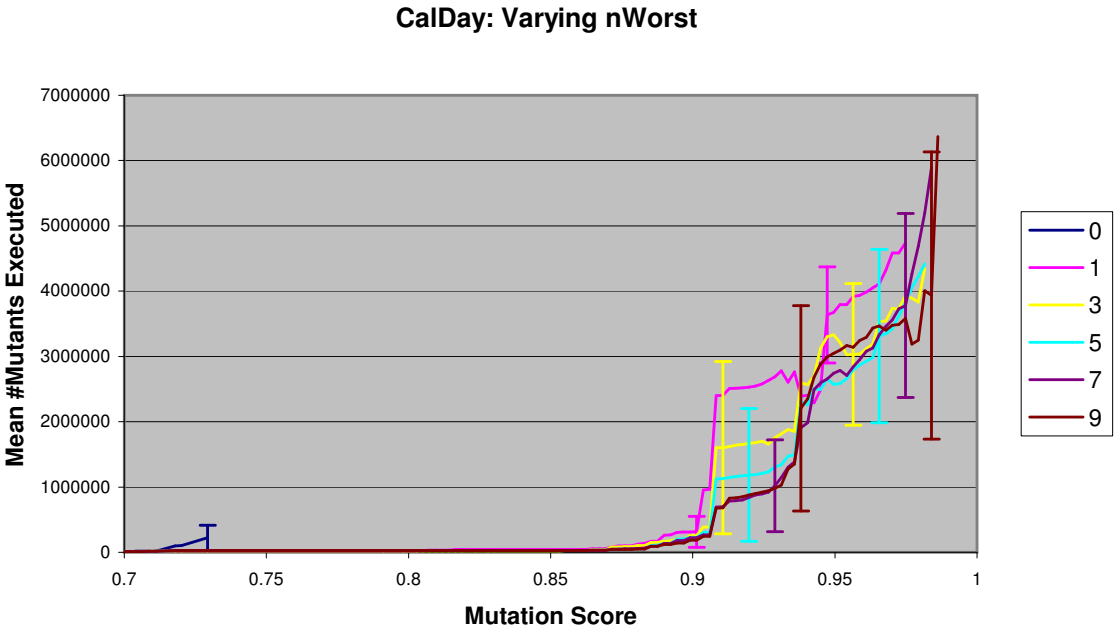

Figure 3.113: Effect of varying `nWorst` on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

C.2.6 nWorst: Effect on Mutation Score per Iteration

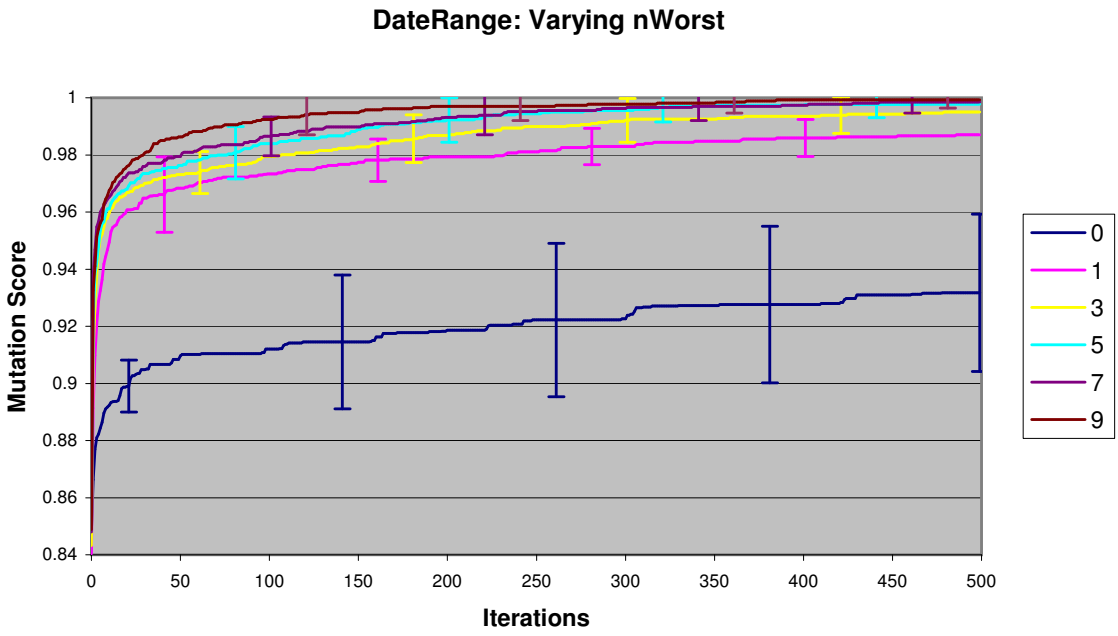

Figure 3.114: Effect of varying `nWorst` on the mean mutation score per iteration for the *DateRange* program.

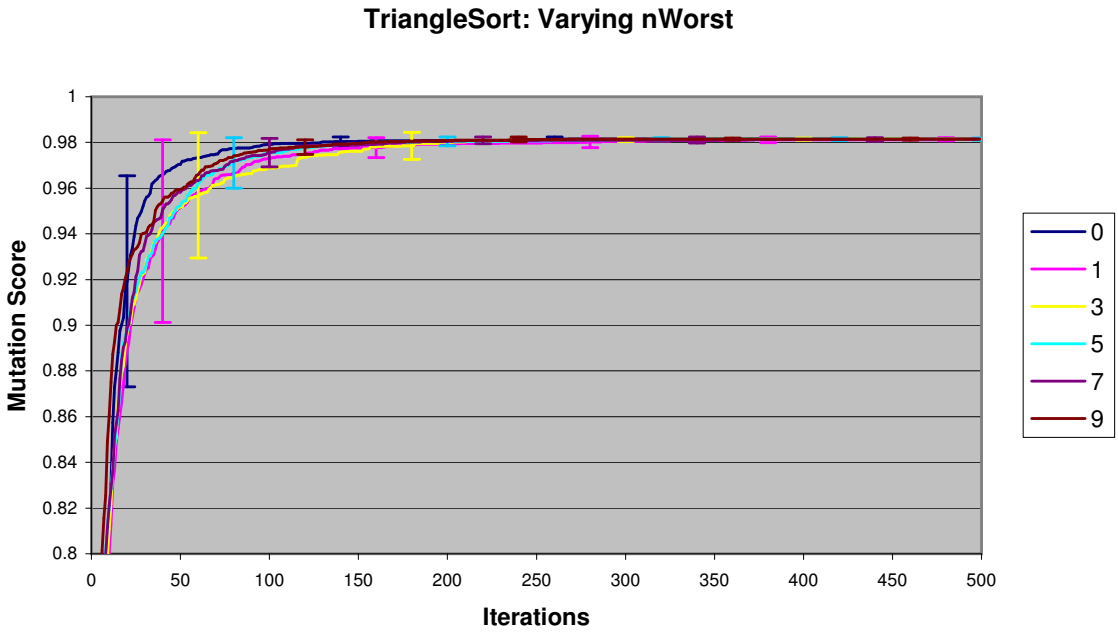

Figure 3.115: Effect of varying `nWorst` on the mean mutation score per iteration for the *TriangleSort* program.

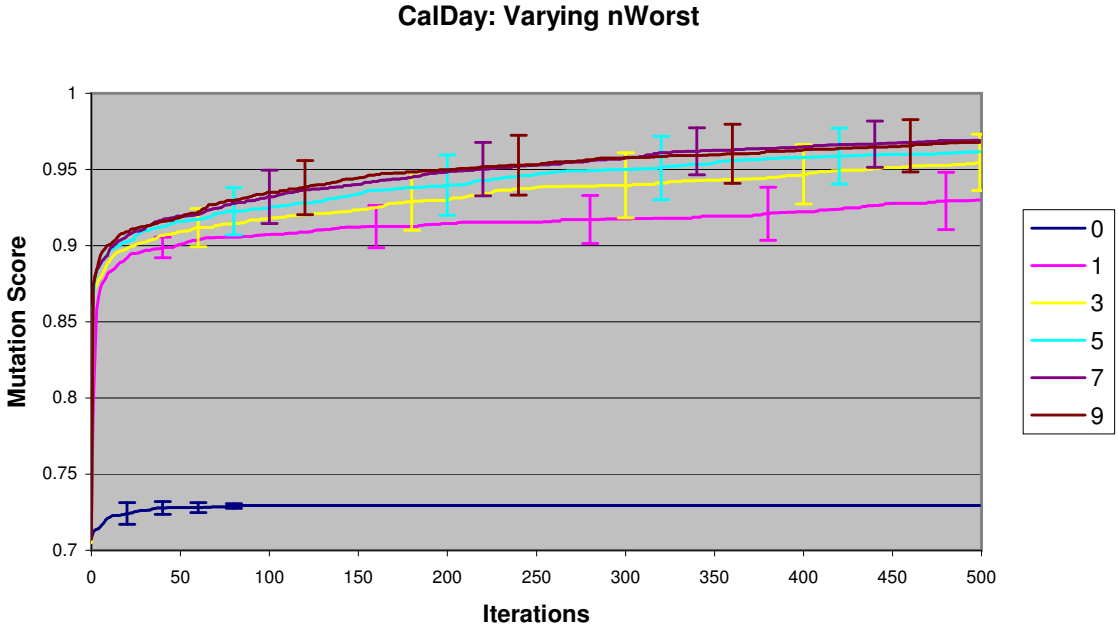

Figure 3.116: Effect of varying `nWorst` on the mean mutation score per iteration for the *CalDay* program.

C.2.7 nWorst: Effect on Number of HTK identified

`nWorst`	DR	TRI	CD
0	73.33% ± 0.00	75.00% ± 0.00	34.38% ± 0.00
1	85.33% ± 1.51	75.00% ± 0.00	58.92% ± 0.99
3	94.41% ± 1.05	75.00% ± 0.00	78.41% ± 1.15
5	97.33% ± 1.47	75.00% ± 0.00	81.63% ± 1.17
7	98.22% ± 1.46	75.00% ± 0.00	82.52% ± 1.11
9	99.11% ± 1.45	75.00% ± 0.00	82.16% ± 1.06
ANOVA			
f_{obt}	**109.76**	0	**133.26**
f_{crit}	2.27	2.26	2.26
0-1	**8.75**	-	**10.33**
0-3	**15.50**	-	**18.40**
0-5	**17.50**	-	**19.75**
0-7	**18.15**	-	**19.96**
0-9	**18.80**	-	**19.64**
1-3	**6.62**	-	**8.09**
1-5	**8.68**	-	**9.42**
1-7	**9.32**	-	**9.72**
1-9	**9.97**	-	**9.49**
3-5	2.13	-	1.33
3-7	2.78	-	1.68
3-9	**3.43**	-	1.52
5-7	0.64	-	0.36
5-9	1.29	-	0.21
7-9	0.64	-	0.15
c_{crit}	3.37	-	3.36

Table 3.45: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six `nWorst` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `nWorst` values result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.2.8 nWorst: Effect on Number of Tests

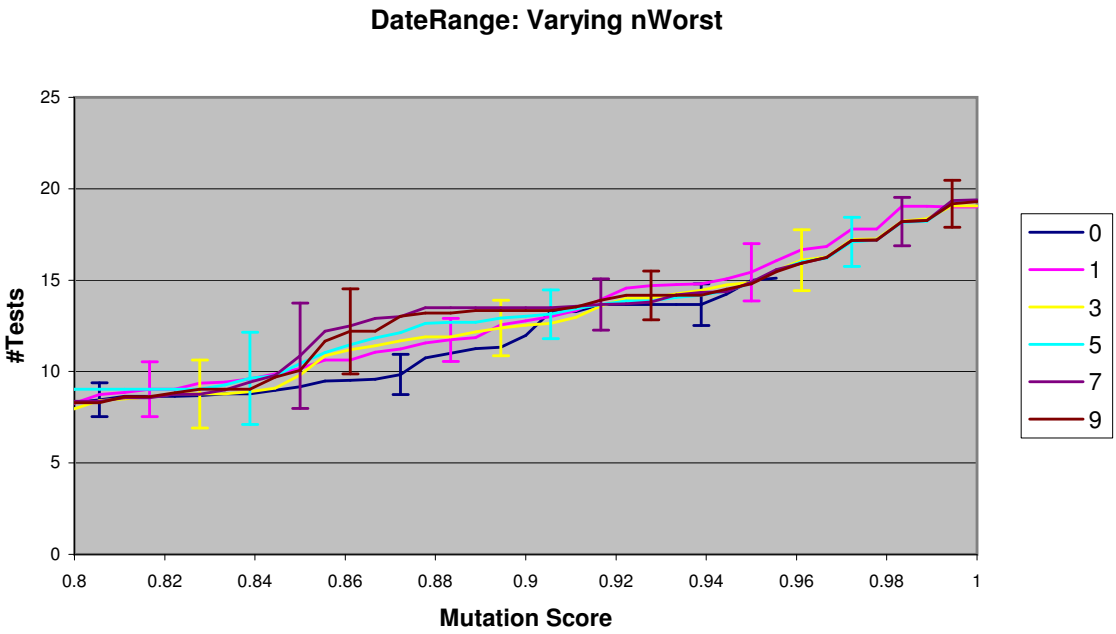

Figure 3.117: Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program.

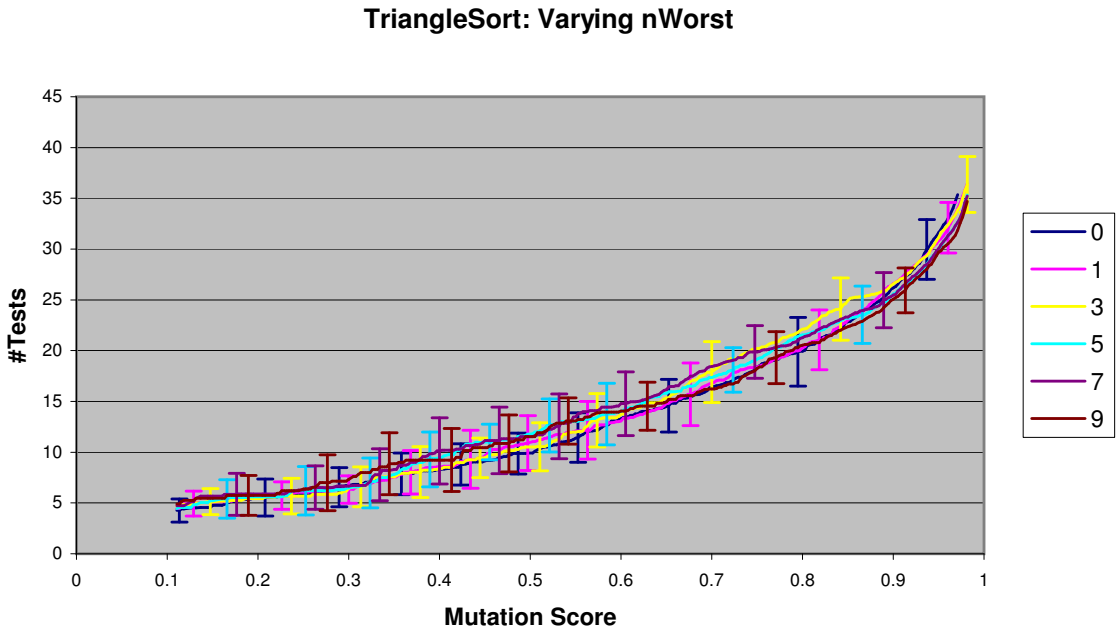

Figure 3.118: Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program.

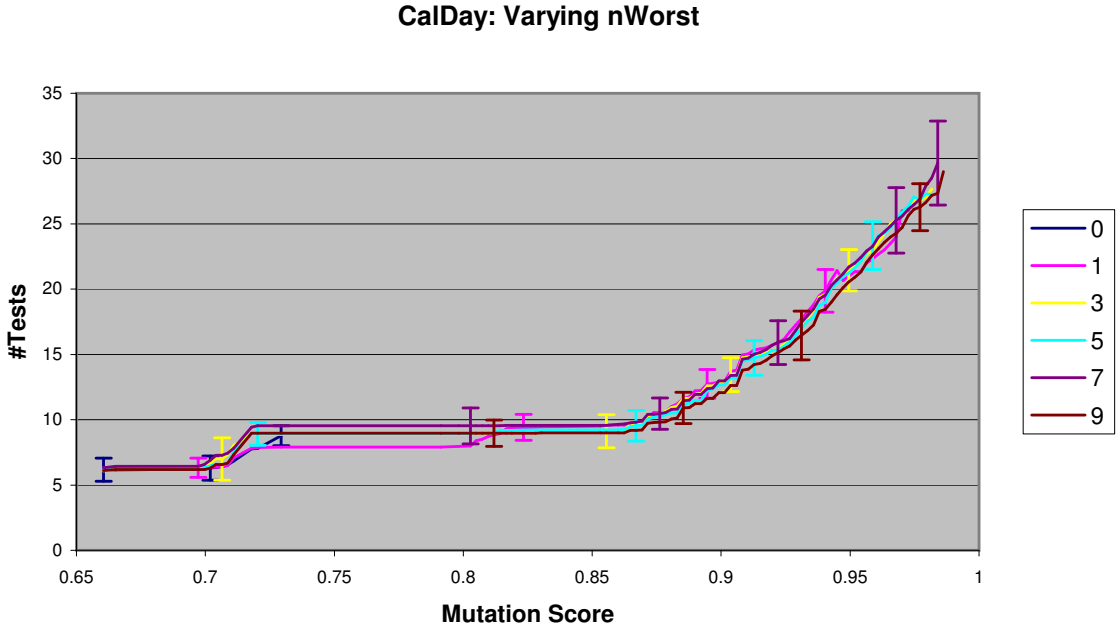

Figure 3.119: Effect of varying `nWorst` on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program.

C.2.9 cloneRate: Effect on Number of Mutant Executions

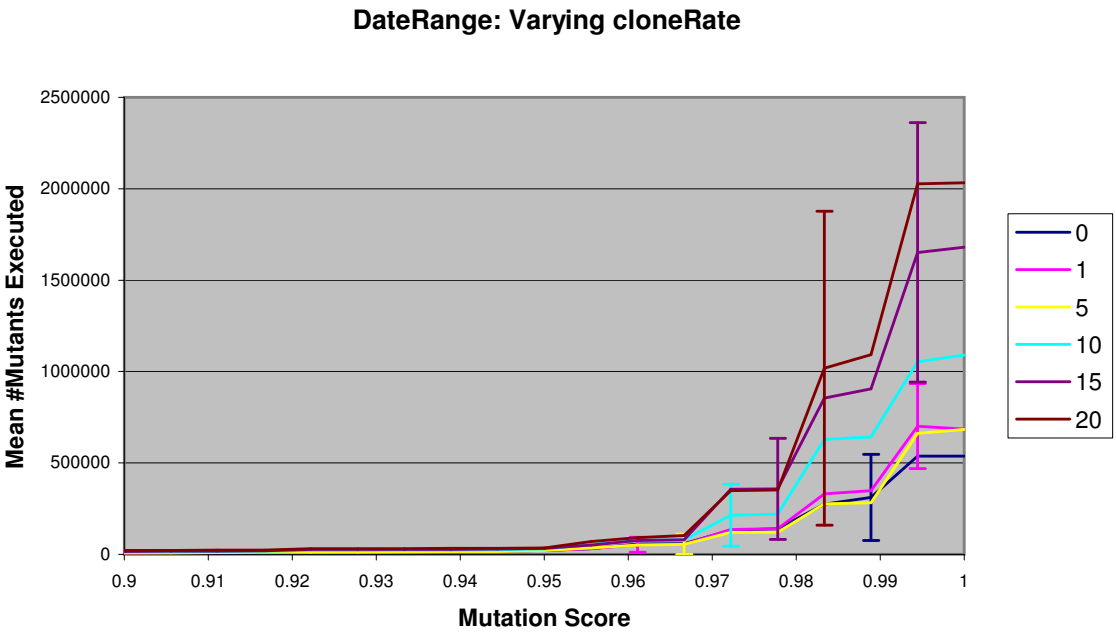

Figure 3.120: Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *DateRange* program.

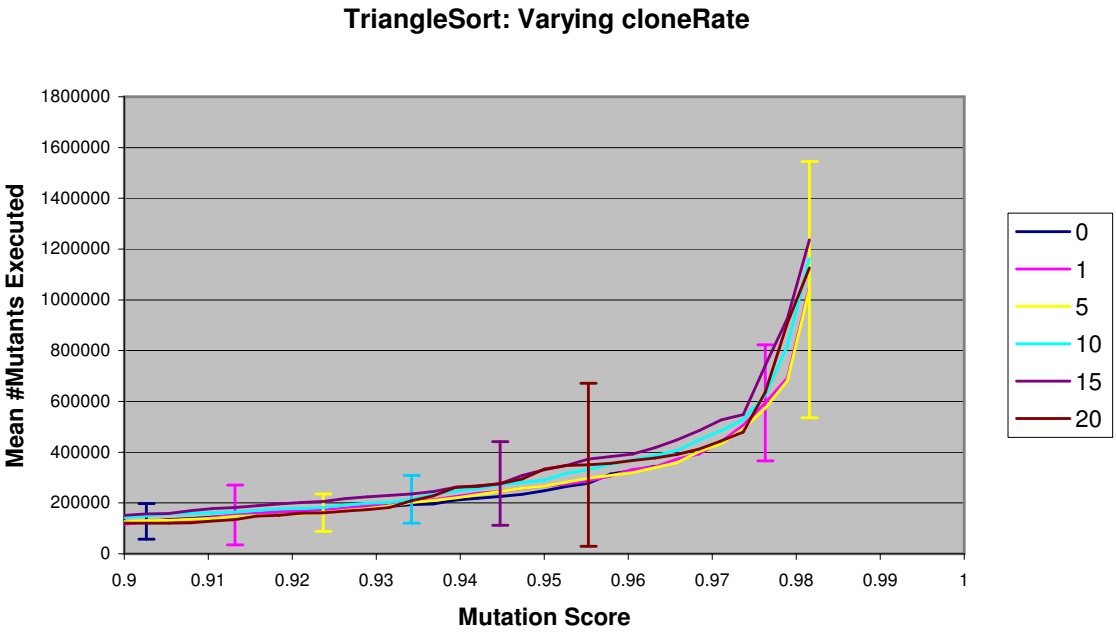

Figure 3.121: Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *TriangleSort* program.

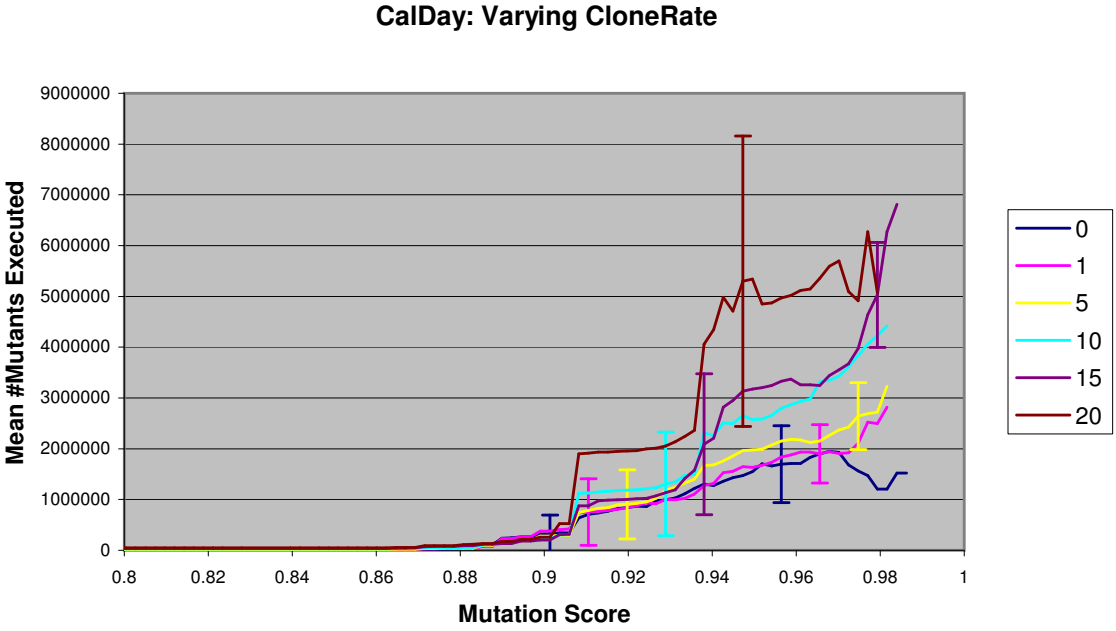

Figure 3.122: Effect of varying cloneRate on the mean number of executions to achieve specific mutation scores for the *CalDay* program.

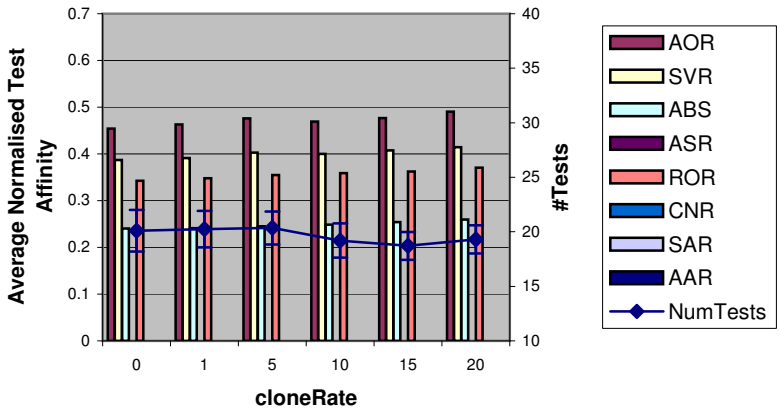

Figure 3.123: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *DateRange* program.

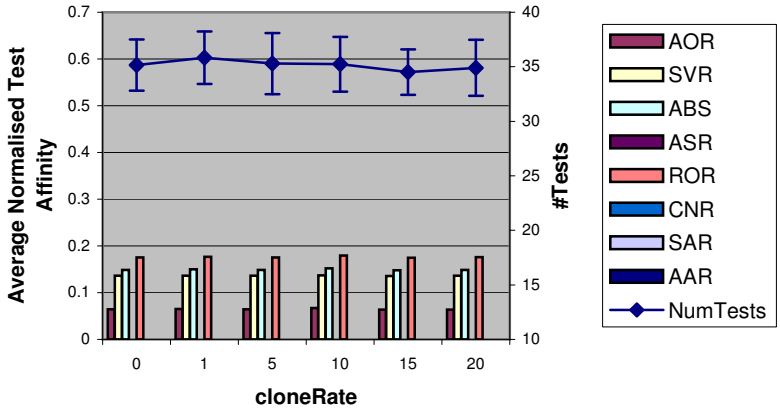

Figure 3.124: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *TriangleSort* program.

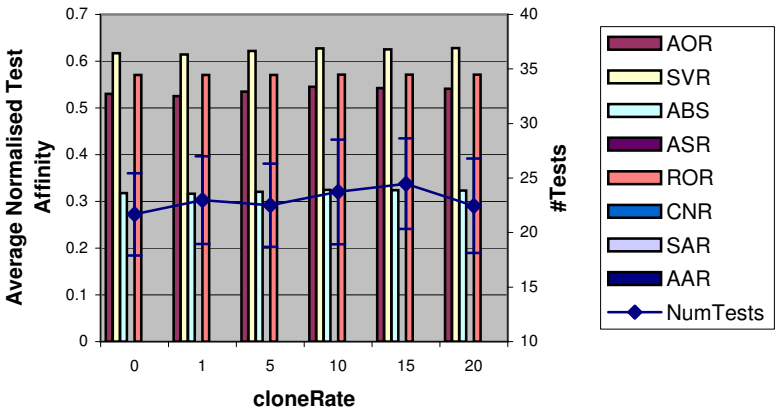

Figure 3.125: Effect of varying `cloneRate` on the average memory set size and memory test affinity for each mutation operator, for the *CalDay* program.

C.2.10 cloneRate: Effect on Mutation Score per Iteration

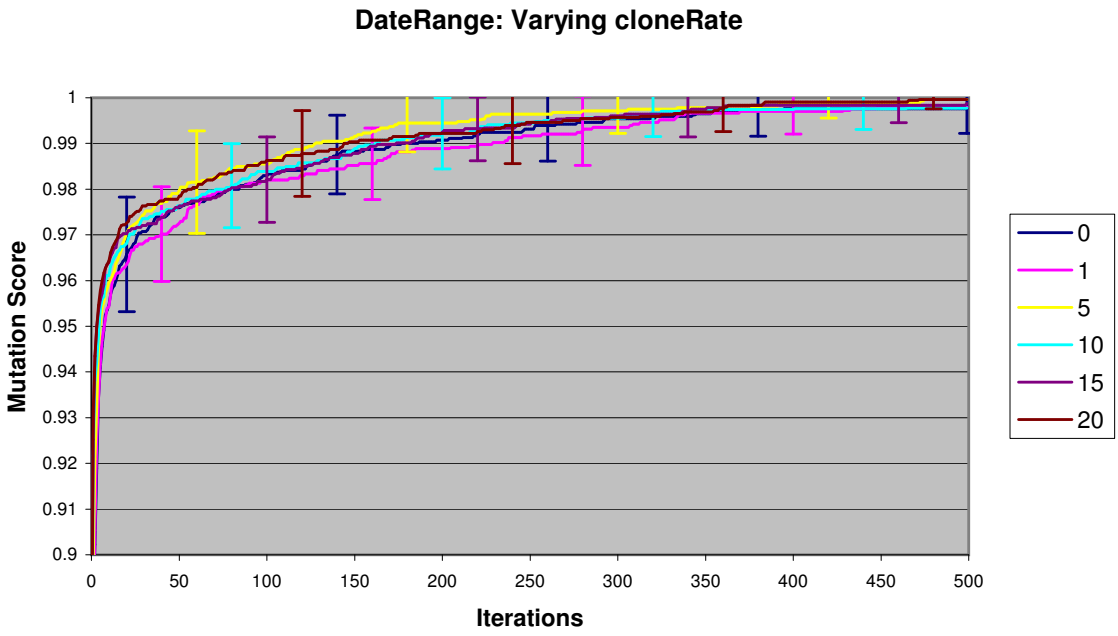

Figure 3.126: Effect of varying cloneRate on the mean mutation score per iteration for the *DateRange* program.

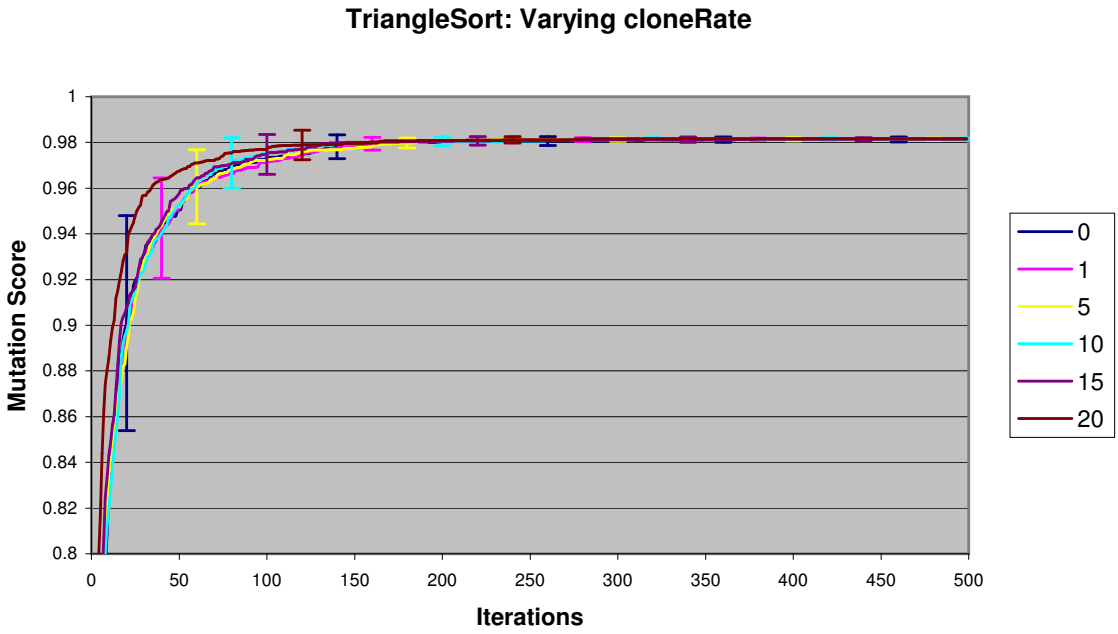

Figure 3.127: Effect of varying cloneRate on the mean mutation score per iteration for the *TriangleSort* program.

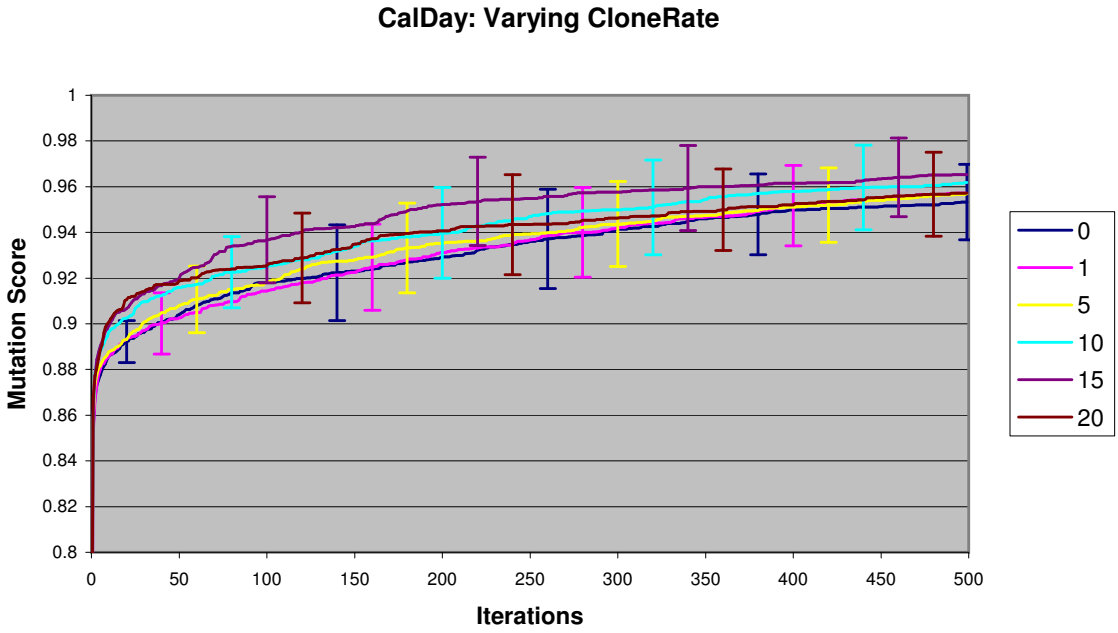

Figure 3.128: Effect of varying cloneRate on the mean mutation score per iteration for the *CalDay* program.

C.2.11 cloneRate: Effect on Number of HTK identified

`cloneRate`	DR	TRI	CD
0	97.11% ± 6.47%	74.72% ± 1.52%	76.86% ± 8.38%
1	97.56% ± 5.10%	75.00% ± 0.00%	77.96% ± 8.56%
5	99.57% ± 2.39%	75.00% ± 0.00%	79.38% ± 5.26%
10	97.33% ± 5.42%	75.00% ± 0.00%	81.63% ± 4.19%
15	98.92% ± 3.48%	75.00% ± 0.00%	83.05% ± 3.99%
20	99.56% ± 2.43%	75.00% ± 0.00%	79.74% ± 8.30%
ANOVA			
f_{obt}	1.95	1.03	**3.93**
f_{crit}	2.27	2.26	2.60
0-1	-	-	0.69
0-5	-	-	1.57
0-10	-	-	2.93
0-15	-	-	**3.80**
0-20	-	-	1.74
1-5	-	-	0.89
1-10	-	-	2.26
1-15	-	-	3.13
1-20	-	-	1.08
5-10	-	-	1.37
5-15	-	-	2.23
5-20	-	-	0.22
10-15	-	-	0.85
10-20	-	-	1.11
15-20	-	-	1.95
c_{crit}	3.37	3.37	3.36

Table 3.46: The mean average percentage of hard-to-kill mutants identified after 500 iterations, for each of the six `cloneRate` values: 0, 1, 3, 5, 7, 9. ANOVA and Scheffé calculations (at the 0.05 level) show which pairs of `cloneRate` values result in significantly different mean percentages (in **bold**). Values replaced with '-' are not important as the ANOVA results are not significant. *All results are to 2 decimal places.*

C.2.12 cloneRate: Effect on Number of Tests

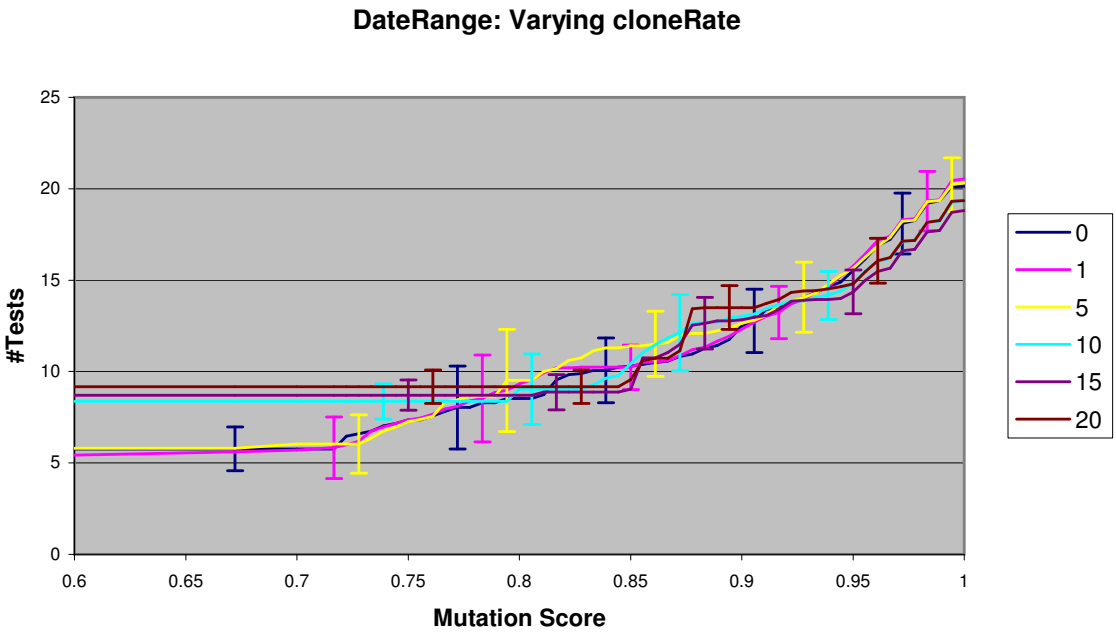

Figure 3.129: Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *DateRange* program.

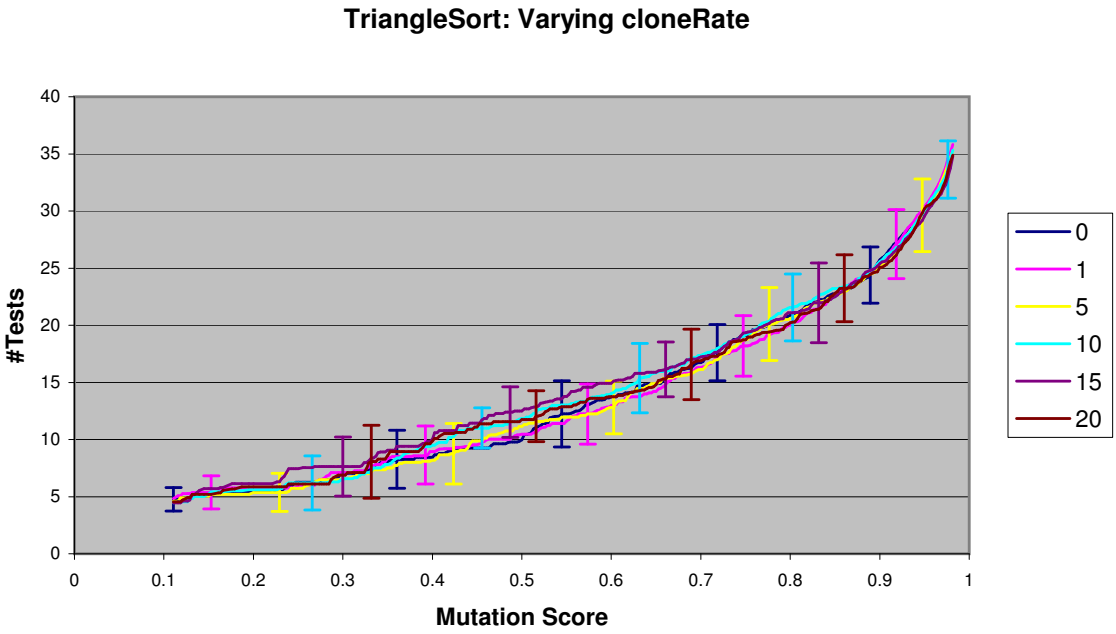

Figure 3.130: Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *TriangleSort* program.

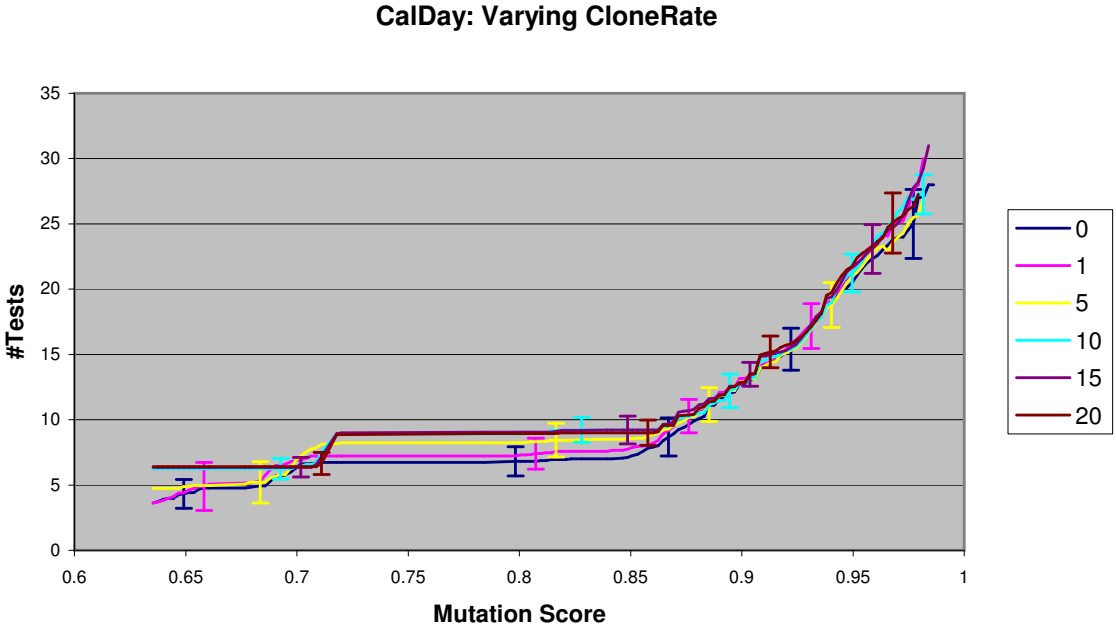

Figure 3.131: Effect of varying cloneRate on the mean number of distinct tests created to achieve specific mutation scores for the *CalDay* program.

Bibliography

[1] A. T. Acree. *On Mutation*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, 1980.

[2] A. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[3] ANTLR. ANTLR parser generator. Internet - http://www.antlr.org/. Accessed: September 2003.

[4] D. Baldwin and F. Sayward. Heuristics for determining equivalence of program mutations. Technical Report Research Report 276, Dept. of Computer Science, Yale University, 1979.

[5] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Automatic test cases optimization using a bacteriological adaptation model: Application to .net components. *17th IEEE International Conference on Automated Software Engineering (ASE'02)*, page 253, 2002.

[6] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Genes and bacteria for automatic test cases optimization in the .net environment. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE 02)*, pages 195–206, 2002.

[7] Benoit Baudry, Franck Fleurey, Jean-Marc Jezequel, and Yves Le Traon. Automatic test case optimization: A bacteriologic algorithm. *IEEE Software*, 22(2):76–82, 2005.

[8] Boris Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, New York, 2nd edition, 1990.

[9] P. J. Bentley, J. Greensmith, and S. Ujjin. Two ways to grow tissue for artificial immune systems. In *ICARIS*, pages 139–152, 2005.

[10] James M. Bieman, Sudipto Ghosh, and Roger T. Alexander. A technique for mutation of Java objects. *Proceedings of Automated Software Engineering*, pages 337–340, 2001.

[11] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools.* Addison-Wesley, 2000.

[12] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems.* Oxford University Press Inc., USA, 1999.

[13] Leonardo Bottaci. A genetic algorithm fitness function for mutation testing. Internet - http://www.dcs.kcl.ac.uk/projects/seminal/pastmeeting /(007)(12,13)-5-2001/bottaci.ps, May 2001. Accessed: July 2005.

[14] Leonardo Bottaci. Instrumenting programs with flag variables for test data search by genetic algorithm. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*, pages 1337–1342. Morgan Kaufmann, New York, USA, 2002.

[15] Jonathan Bowen. Formal specification and documentation using z: A case study approach. Internet - http://www.jpbowen.com/pub/zbook.pdf, 2003. Accessed: March 2007.

[16] Heather Brannon. The history of smallpox. Website - http://dermatology.about.com/cs/smallpox/a/smallpoxhx.htm.

[17] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18:31–45, 1982.

[18] T.A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven, CT, 1980.

[19] F. M. Burnet. *The Clonal Selection Theory of Acquired Immunity*. Cambridge University Press, 1959.

[20] B. Choi and A.P. Mathur. High-performance mutation testing. *The Journal of Systems and Software*, 20:135–152, 1993.

[21] T. Dandekar and P. Argos. Folding the main chain of small proteins with the genetic algorithm. *Journal of Molecular Biology*, 236(3):844–861, February 1994.

[22] Charles Darwin. On the origin of species by means of natural selection. Online Book - http://pages.britishlibrary.net/charles.darwin/texts/origin_6th/origin6th_fm.html, 1859. Accessed: October 2005.

[23] Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. *Proceedings of the ACM '81 conference*, pages 254 – 257, 1981.

[24] L. N. de Castro and F. J. Von Zuben. The clonal selection algorithm with engineering applications. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 36–37, 2000.

[25] L.N. de Castro and J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002.

[26] M. E. Delamaro and J. C. Maldonado. Proteum - a tool for the assessment of test adequacy for C programs: User's guide. Technical report, March 1996.

[27] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[28] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[29] E. Diaz, J. Tuya, and R. Blanco. An automated test data generation tool using tabu search. In *UK-Softest: UK Software Testing Workshop*, pages 21–30, 2003.

[30] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, 1996.

[31] S. Forrest, S. A. Hofmeyr, and A. Somayaji. Computer immunology. *Communications of the ACM*, 40(10):88–96, 1997.

[32] S. Forrest, A.S. Perelson, L. Allen, and R.Cherukuri. Self-nonself discrimination in a computer. In *IEEE Symposium on Research in Security and Privacy*, pages 202–212, Los Alamos, CA, 1994. IEEE Computer Society Press.

[33] Marie-Claude Gaudel. Testing can be formal, too. *Proceedings of the 6th International Joint CAAP/FASE Conference on Theory and Practice of Software Development (TAPSOFT 95)*, 915:82–96, 1995.

[34] M.R. Girgis and M.R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. *Proceedings of the Eighth International Conference on Software Engineering*, pages 313–319, August 1985.

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

[36] K. Gurney. *An Introduction to Neural Networks*. UCL Press, 1999.

[37] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, 1977.

[38] M. Harman, L. Hierons, A. Baresel, and H. Sthamer. Improving evolutionary testing by flag removal. In *Genetic and Evolutionary Computation Conference (GECCO 2002)*. Morgan Kaufmann Publishers, New York.

[39] R. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification, and Reliability*, 9:233–262, 1999.

[40] John Holland. *Adaptation in Natural and Artificial Systems.* The MIT Press, reprint edition, 1992.

[41] W.E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8:371–379, July 1982.

[42] Chris Hunter and Paul Strooper. Systematically deriving partial oracles for testing concurrent programs. *Australasian Computer Science Conference (ACSC '01)*, pages 83–91, 2001.

[43] Charles A. Janeway. How the immune system recognizes invaders. *Scientific American*, 269(3):72–79, September 1993.

[44] N.K. Jerne. Towards a network theory of the immune system. *Annals of Immunology*, 125C:373–389, 1974.

[45] Ian Johnston. Translation of philosophie zoologique by J. B. Lamarck. Internet - http://www.mala.bc.ca/ johnstoi/LAMARCK/tofc.htm, April 2000. Accessed: October 2005.

[46] B. F. Jones, D. E. Eyres, and H.-H.Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *The Computer Journal*, 41(2):98–107, 1998.

[47] Sunwoo Kim, John A. Clark, and John A. McDermid. Class mutation: Mutation testing for object-oriented programs. Internet - http://www-users.cs.york.ac.uk/ jac/papers/ClassMutation.pdf, 2000. Accessed: April 2003.

[48] K. N. King and A. Jefferson Offutt. A Fortran language system for mutation-based software testing. *Software - Practice and Experience*, 21(7):685–718, 1991.

[49] Bogdan Korel. Automated software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[50] John R. Koza. Genetic programming. In James G. Williams and Allen Kent, editors, *Encyclopedia of Computer Science and Technology*, volume 39, pages 29–43. Marcel-Dekker, Taylor & Francis Group Ltd, Oxford, 1998.

[51] Gerald Kranzler and Janet Moursund. *Statistics for the Terrified*. Prentice-Hall, Saddle River, New Jersey, US, second edition, 1999.

[52] Edward William Krauser. *Compiler-Integrated Software Testing*. PhD thesis, Purdue University, December 1991.

[53] E.W. Krauser, A.P. Mathur, and V. Rego. High performance testing on SIMD machines. *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 171–177, July 1998.

[54] J. B. Lamarck. *Philosophie zoologique, ou Exposition des considrations relatives l'histoire naturelle des animaux*. 1809.

[55] P. Larraaga, C.M.H. Kuijpers, R.H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13(2):129–170, April 1999.

[56] E. L. Lawler, J. K. Lenstra, and D. B. Shmoys. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. Wiley and Sons, New York, 1985.

[57] J.-C. Lin and P.-L. Yeh. Automatic test data generation for path testing using GAs. *Information Sciences: an International Journal*, 131:47–64, 2001.

[58] Yu-Seung Ma, Yong-Rae Kwon, and A. Jefferson Offutt. Inter-class mutation operators for Java. *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

[59] Yu Seung Ma, A. Jefferson Offutt, and Yong Rae Kwon. Mujava mutation system for java programs. Internet - http://www.ise.gmu.edu/ ofut/mujava/. Accessed: January 2004.

[60] Yu-Seung Ma, A. Jefferson Offutt, and Yong-Rae Kwon. Mujava: An automated class mutation system. *Journal of Software Testing, Verification and Reliability*, 15(2):97–133, June 2005.

[61] Patricia D. L. Machado. *Testing from Structured Algebraic Specifications: The Oracle Problem*. PhD thesis, University of Edinburgh, 2000.

[62] B. Marick. The weak mutation hypothesis. Technical Report UIUCDCS-R-90-1644, University of Illinois, November 1990.

[63] Philippa Marrack and John W. Kappler. How the immune system recognizes the body. *Scientific American*, 269(3):80–83, 86–89, September 1993.

[64] A. P. Mathur. Performance, effectiveness, and reliability issues in software testing. *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference*, pages 604–605, September 1991.

[65] Aditya P. Mathur and W. Eric Wong. Comparing the fault detection effectiveness of mutation and data flow testing: An empirical study. Technical Report SERC-TR-146-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana., December 1993.

[66] Aditya P. Mathur and W. Eric Wong. An empirical comparison of mutation and data flow based test adequacy criteria. Technical Report SERC-TR-135-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana., December 1993.

[67] P. May, K. Mander, and J. Timmis. Mutation testing: An artificial immune system approach. *UK-Softest: UK Software Testing Workshop*, pages 31–43, September 2003. University of York.

[68] P. May, K. Mander, and J. Timmis. Software vaccination: An artificial immune system approach. In J. Timmis, P. Bentley, and E. Hart, editors, *2nd International Conference on Artificial Immune Systems*, number 2787 in Series Lecture Notes in Computer Science, pages 81–92. Springer-Verlag, September 2003.

[69] Johannes Mayer and Ralph Guderlei. Test oracles using statistical methods. *Proceedings of the First International Workshop on Software Quality, Lecture Notes in Informatics P-58*, pages 179–189, 2004.

[70] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[71] P. McMinn and M. Holcombe. The state problem for evolutionary testing. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2003)*, pages 2488–2497. Springer-Verlag, Chicago, USA, 2003.

[72] P. McMinn and M. Holcombe. Hybridizing evolutionary testing with the chaining approach. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2004)*, pages 1363–1374, 2004.

[73] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2005)*, pages 1013–1020, 2005.

[74] C. Michael and G. McGraw. Automated software test data generation for complex programs. *13th IEEE International Conferance on Automated Software Engineering*, pages 136–146, 1998.

[75] C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[76] M. Mitchell. *An Introduction to Genetic Algorithms*. The MIT Press, 6th edition, 1999.

[77] Edward Jenner Museum. The Jenner Museum. Website - http://www.jennermuseum.com/. Accessed: October 2005.

[78] Gustav J. V. Nossal. Life, death and the immune system. *Scientific American*, 269(3):52–62, September 1993.

[79] A. Jefferson Offutt. Mothra mutation testing tool. Internet - http://www.ise.gmu.edu/faculty/ofut/rsrch/mut.html. Accessed: January 2004.

[80] A. Jefferson Offutt. *Automatic Test Data Generation*. PhD thesis, Georgia Institute of Technology, 1988. GIT-ICS 88/28.

[81] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering and Methodology*, 1(1):5–20, 1992.

[82] A. Jefferson Offutt. A practical system for mutation testing: Help for the common programmer. *Proceedings of the International Test Conference*, pages 824–830, 1994.

[83] A. Jefferson Offutt and W.M. Craft. Using compiler optimization techniques to detect equivalent mutants. *The Journal of Software Testing, Verification, and Reliability*, 4:131–154, September 1994.

[84] A. Jefferson Offutt, Zhenyi Jin, and Jie Pan. The dynamic domain reduction procedure for test data generation: Design and algorithms. Technical Report ISSE-TR-94-110, Department of Information and Software Systems Engineering, George Mason University, Fairfax VA, 1994.

[85] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[86] A. Jefferson Offutt and S.D. Lee. How strong is weak mutation? *Proceedings of the Fourth Symposium on Software Testing, Analysis and Verification*, pages 200–213, October 1991.

[87] A. Jefferson Offutt and S.D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20:337–344, May 1994.

[88] A. Jefferson Offutt, Yu-Seung Ma, and Yong-Rae Kwon. An experimental mutation system for Java. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–4, 2004.

[89] A. Jefferson Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, September 1997.

[90] A. Jefferson Offutt, R. Pargas, S.V. Fichter, and P. Khambekar. Mutation testing of software using a MIMD computer. *1992 International Conference on Parallel Processing*, II:257–266, August 1992.

[91] A. Jefferson Offutt and Roland H. Untch. *Mutation 2000: Uniting the Orthogonal*. Kluwer Academic Publishers, Massachusetts, USA, 2000.

[92] A. Jefferson Offutt, Christian Zapf, and Gregg Rothermel. An experimental evaluation of selective mutation. *Proceedings of the Fifteenth International Conference on Software Engineering*, pages 100–107, May 1993.

[93] World Health Organization. Influenza fact sheet. Internet, March 2003.

[94] R.P. Pargas, M.J. Harrold, and R.R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9(4):263–282, 1999.

[95] A. S. Perelson and G. F. Oster. Theoretical studies of clonal selection: Minimal antibody repertoire size and reliability of self-nonself discrimination. *Journal of Theoretical Biology*, 81(4):645–670, December 1979.

[96] Alan S. Perelson and Gerard Weisbuch. Immunology for physicists. *Review of Modern Physics*, 69(4):1219–1268, October 1997.

[97] Marc Roper. *Software Testing*. McGraw-Hill Book Company Europe, 1994.

[98] Alexandra Minna Stern and Howard Markel. The history of vaccines and immunization: Familiar patterns, new challenges. *Health Affairs*, 24(3):611–621, 2005.

[99] Sun. Java J2SE 5.0. Internet - http://java.sun.com/.

[100] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Automated Software Engineering*, pages 285–288, 1998.

[101] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Software - Practice and Experience*, 30:61–79, 2000.

[102] R. Unger and J. Moult. Genetic algorithms for protein folding simulations. *J Molecular Biology*, 231(1):75–81, May 1993.

[103] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. Mutation analysis using mutant schemata. *International Symposium on Software Testing and Analysis*, pages 139–148, 1993.

[104] Andrew Watkins. *Exploiting Immunological Metaphors in the Development of Serial, Parallel, and Distributed Learning Algorithms.* PhD thesis, University of Kent, Canterbury, UK, March 2005.

[105] Irving L. Weissman and Max D. Cooper. How the immune system develops. *Scientific American*, 269(3):64–71, September 1993.

[106] Wikipedia-Lamarck. Jean-Babtiste Lamarck. Internet - http://en.wikipedia.org/wiki/Jean-Baptiste_Lamarck. Accessed: October 2005.

[107] Wikipedia-Mendel. Mendelian inheritance. Internet - http://en.wikipedia.org/wiki/Mendelian_inheritance. Accessed: October 2005.

[108] Weichen E. Wong. *On Mutation and Data Flow.* PhD thesis, Purdue University, December 1993.

[109] M.R. Woodward and K. Halewood. From weak to strong, dead or alive? an analysis of some mutation testing issues. *Proceedings of the Second Workshop on Software Testing, Verification and Analysis*, pages 152–158, July 1988.