The Use of Automated Search in Deriving Software Testing Strategies

Simon Marcus Poulding

Submitted for the degree of Doctor of Philosophy
University of York
Department of Computer Science
July 2013

Abstract

Testing a software artefact using every one of its possible inputs would normally cost too much, and take too long, compared to the benefits of detecting faults in the software. Instead, a testing strategy is used to select a small subset of the inputs with which to test the software. The criterion used to select this subset affects the likelihood that faults in the software will be detected. For some testing strategies, the criterion may result in subsets that are very efficient at detecting faults, but implementing the strategy—deriving a 'concrete strategy' specific to the software artefact—is so difficult that it is not cost-effective to use that strategy in practice.

In this thesis, we propose the use of metaheuristic search to derive concrete testing strategies in a cost-effective manner.

We demonstrate a search-based algorithm that derives concrete strategies for 'statistical testing', a testing strategy that has a good fault-detecting ability in theory, but which is costly to implement in practice. The cost-effectiveness of the search-based approach is enhanced by the rigorous empirical determination of an efficient algorithm configuration and associated parameter settings, and by the exploitation of low-cost commodity GPU cards to reduce the time taken by the algorithm. The use of a flexible grammar-based representation for the test inputs ensures the applicability of the algorithm to a wide range of software.

Contents

1 Introduction **19**
1.1 Overview . . . 20
1.2 Software Testing Strategies . . . 21
1.2.1 The Software Testing Process . . . 21
1.2.2 Testing Strategies, Techniques, and Costs . . . 22
1.3 Coverage-Based Testing Strategies . . . 24
1.4 Structural Testing . . . 25
1.4.1 Adequacy Criteria . . . 25
1.4.2 Fault-Detecting Ability . . . 28
1.4.3 An Analytical Technique: Symbolic Execution . . . 29
1.4.4 A Dynamic Technique: Search-Based Software Testing . . . 30
1.5 Probabilistic Testing Strategies . . . 36
1.6 Random Testing . . . 38
1.6.1 Adequacy Criterion . . . 38
1.6.2 Fault-Detecting Ability . . . 38
1.7 Statistical Testing . . . 40
1.7.1 Adequacy Criterion . . . 40
1.7.2 Fault-Detecting Ability . . . 41
1.7.3 A Manual Analytical Technique . . . 41
1.7.4 An Automated Analytical Technique . . . 43
1.7.5 A Manual Dynamic Technique . . . 43
1.8 Research Hypothesis . . . 45
1.9 Thesis Structure . . . 46

2 Demonstration of a Viable Algorithm **47**
2.1 Introduction . . . 48
2.1.1 Motivation . . . 48
2.1.2 Contributions . . . 48
2.1.3 Chapter Outline . . . 48
2.2 Proposed Algorithm . . . 50
2.2.1 Representation . . . 50
2.2.2 Fitness Metric . . . 54
2.2.3 Search Method . . . 55
2.3 Implementation . . . 60
2.3.1 Language . . . 60
2.3.2 Search Method . . . 60
2.3.3 Argument Representation . . . 60

2.3.4 SUT Execution . . . 60
2.3.5 Input and Output . . . 61
2.3.6 Pseudo-Random Number Generation . . . 61
2.4 Software Under Test . . . 62
2.4.1 SUT Characteristics . . . 62
2.4.2 simpleFunc . . . 62
2.4.3 bestMove . . . 62
2.4.4 nsichneu . . . 63
2.4.5 Instrumentation . . . 63
2.5 Experiment I – Algorithm Viability . . . 65
2.5.1 Objectives . . . 65
2.5.2 Preparation . . . 65
2.5.3 Method . . . 65
2.5.4 Results . . . 66
2.5.5 Discussion and Conclusions . . . 67
2.5.6 Threats to Validity . . . 67
2.6 Experiment II – Fault-Detecting Ability . . . 69
2.6.1 Objectives . . . 69
2.6.2 Preparation . . . 69
2.6.3 Method . . . 70
2.6.4 Results . . . 70
2.6.5 Discussion and Conclusions . . . 72
2.6.6 Threats to Validity . . . 73
2.7 Experiment III – Change in Fault-Detecting Ability During the Search . . . 74
2.7.1 Objectives . . . 74
2.7.2 Method . . . 74
2.7.3 Results . . . 74
2.7.4 Discussion and Conclusions . . . 74
2.8 Profile Diversity . . . 77
2.9 Experiment IV – Effect of Diversity Measure . . . 80
2.9.1 Objectives . . . 80
2.9.2 Preparation . . . 80
2.9.3 Method . . . 80
2.9.4 Results . . . 81
2.9.5 Discussion and Conclusions . . . 81
2.9.6 Threats to Validity . . . 83
2.10 Conclusion . . . 84

3 Construction of an Efficient Algorithm **87**
3.1 Introduction . . . 88
3.1.1 Motivation . . . 88
3.1.2 Contributions . . . 88
3.1.3 Chapter Outline . . . 89
3.2 Empirical Approach . . . 90
3.2.1 Algorithm Configurations and Parameters . . . 90
3.2.2 Configuration Comparison . . . 91
3.2.3 Common Parameter Derivation . . . 91

3.2.4 Rationale . . . 91
3.2.5 Calculating Average Efficiency . . . 92
3.3 Response Surface Methodology . . . 94
3.3.1 Justification . . . 94
3.3.2 Standard RSM . . . 95
3.3.3 Implementation and Enhancements . . . 96
3.4 Software Under Test . . . 102
3.4.1 SUT Characteristics . . . 102
3.4.2 cArcsin . . . 102
3.4.3 fct3 . . . 103
3.5 Experiment V – Baseline Algorithm Efficiency . . . 104
3.5.1 Objectives . . . 104
3.5.2 Preparation . . . 104
3.5.3 Method . . . 106
3.5.4 Results . . . 106
3.5.5 Discussion and Conclusions . . . 106
3.5.6 Threats to Validity . . . 107
3.6 Experiment VI – Directed Mutation . . . 108
3.6.1 Objectives . . . 108
3.6.2 Motivation . . . 108
3.6.3 Preparation . . . 109
3.6.4 Method . . . 110
3.6.5 Results . . . 110
3.6.6 Discussion and Conclusions . . . 112
3.6.7 Threats to Validity . . . 112
3.7 Experiment VII – Optimal Algorithm Configuration . . . 114
3.7.1 Overview . . . 114
3.7.2 Experiment VIIa – Limit on Number of Parents . . . 116
3.7.3 Experiment VIIb – Limit on Number of Bins . . . 118
3.7.4 Experiment VIIc – Edge Initialisation . . . 120
3.7.5 Experiment VIId – Bin Initialisation . . . 121
3.7.6 Experiment VIIe – Omission of Mutation Operators . . . 122
3.7.7 Experiment VIIf – Non-Atomic Mutation Operators . . . 123
3.7.8 Experiment VIIg – Fitness Re-evaluation . . . 124
3.7.9 Experiment VIIh – Simulated Annealing . . . 126
3.7.10 Experiment VIIi - Hierarchical Fitness . . . 128
3.7.11 Consolidated Results . . . 129
3.7.12 General Discussion and Conclusions . . . 129
3.7.13 Threats to Validity . . . 131
3.8 Experiment VIII – Common Parameter Settings . . . 134
3.8.1 Objectives . . . 134
3.8.2 Approach . . . 134
3.8.3 Experiment VIIIa – Analysis of SUT-Specific Parameter Settings . . . 135
3.8.4 Experiment VIIIb – Optimisation of Common Parameter Settings . . . 138
3.8.5 Experiment VIIIc – Algorithm Efficiency at Common Parameter Settings 141
3.8.6 General Discussion and Conclusions . . . 142
3.8.7 Threats to Validity . . . 143

3.9 Experiment IX – Caching Instrumentation Data . . . 144
3.9.1 Objectives . . . 144
3.9.2 Preparation . . . 144
3.9.3 Method . . . 144
3.9.4 Results . . . 144
3.9.5 Discussion and Conclusions . . . 145
3.10 Experiment X – Utility of the Optimised Algorithm . . . 147
3.10.1 Objective . . . 147
3.10.2 Preparation . . . 147
3.10.3 Method . . . 147
3.10.4 Results . . . 148
3.10.5 Discussion and Conclusions . . . 148
3.11 Conclusion . . . 150

4 Strategy Representation Using Stochastic Grammars 153
4.1 Introduction . . . 154
4.1.1 Motivation . . . 154
4.1.2 Contributions . . . 154
4.1.3 Chapter Outline . . . 154
4.2 Grammar-Based Testing . . . 156
4.2.1 Formal Grammars . . . 156
4.2.2 Context-Free Grammars . . . 157
4.2.3 Stochastic Grammars . . . 158
4.2.4 Related Work . . . 158
4.3 A Stochastic Context-Free Grammar Representation . . . 161
4.3.1 Overview . . . 161
4.3.2 Conditional Production Weights . . . 161
4.3.3 Scalar Numeric Variables . . . 162
4.3.4 Relationship to Binned Bayesian Network Representation . . . 162
4.3.5 The Application of Automated Search . . . 163
4.4 Implementation . . . 166
4.4.1 GPGPU Constraints . . . 166
4.4.2 Assessing Valid Mutations . . . 166
4.4.3 Representation of Binned Variables . . . 167
4.4.4 Representation of Production Weights . . . 167
4.4.5 Limits on the Length of Generated Strings . . . 167
4.4.6 Pseudo-Random Number Generation . . . 168
4.5 Experiment XI – Compatibility . . . 169
4.5.1 Objectives . . . 169
4.5.2 Preparation . . . 169
4.5.3 Method . . . 171
4.5.4 Results . . . 172
4.5.5 Discussion and Conclusions . . . 172
4.5.6 Threats to Validity . . . 173
4.6 Experiment XII – Wider Applicability . . . 175
4.6.1 Objectives . . . 175
4.6.2 Preparation . . . 175

4.6.3 Method 179
4.6.4 Results 180
4.6.5 Discussion and Conclusions 180
4.6.6 Threats to Validity 181
4.7 Conclusion 183

5 Strategy Derivation Using Parallel Computation 185
5.1 Introduction 186
5.1.1 Motivation 186
5.1.2 Contributions 186
5.1.3 Chapter Outline 186
5.2 General Purpose Computation on GPUs 188
5.2.1 Background 188
5.2.2 CUDA 189
5.2.3 Physical Architecture 189
5.2.4 Programming Model 190
5.2.5 Performance Features 191
5.2.6 Application Development 191
5.3 Implementation 193
5.3.1 Architecture 193
5.3.2 Code 193
5.4 Experiment XIII – GPU Performance 195
5.4.1 Objectives 195
5.4.2 Preparation 195
5.4.3 Method 196
5.4.4 Results 197
5.4.5 Discussion and Conclusions 197
5.4.6 Threats to Validity 198
5.5 Experiment XIV – Multiple Parallel Searches 199
5.5.1 Objectives 199
5.5.2 Preparation 199
5.5.3 Method 200
5.5.4 Results 200
5.5.5 Discussion and Conclusions 201
5.6 Experiment XV – Spatial Migration 204
5.6.1 Objectives 204
5.6.2 Preparation 204
5.6.3 Method 205
5.6.4 Results 205
5.6.5 Discussion and Conclusions 207
5.7 Experiment XVI – One Level of Parallelisation 208
5.7.1 Objectives 208
5.7.2 Preparation 208
5.7.3 Method 208
5.7.4 Results 209
5.7.5 Discussion and Conclusions 209
5.8 Conclusion 210

6 Conclusions **213**
6.1 Evaluation of the Research Hypothesis 214
6.2 Opportunities for Further Research 215
6.2.1 Diversity 215
6.2.2 Novel Adequacy Criteria 216
6.2.3 Application to System-Level Testing 216
6.2.4 Use of Structural Search-Based Software Testing Approaches 216
6.2.5 Plans for Future Work 217

List of References **219**

List of Figures

1 Source code of the example SUT `daysInMonth`. 25
2 Control flow graph of the SUT `daysInMonth`. 26
3 The subsume relationship between adequacy criteria. 28
4 An illustration of approximation level for the SUT `daysInMonth`. 34
5 Test set construction via an intermediate concrete strategy and by direct generation. 36
6 The probabilities of edges in the control flow graph of `daysInMonth` being exercised during random testing. 39
7 The probabilities of edges in the control flow graph of `daysInMonth` being exercised during statistical testing. 43

8 The source code of the toy SUT `simpleFunc`. 51
9 Control flow graph of `simpleFunc`. 51
10 An example of Bayesian network for six input arguments. 52
11 An example of conditional distributions represented in terms of bins for two input arguments. 53
12 An illustration of the encoding of the board positions for `bestMove`. 63
13 Results of Experiment II. 71
14 Results of Experiment III. 75
15 An illustration of diverse and non-diverse input profiles. 78
16 Results of Experiment IV. 82

17 An illustration of response surface methodology. 95
18 An example of a cubic curve fitted to responses along the steepest path. 100
19 An illustration of the intended effect of directed mutation. 108
20 An illustration of the calculation of the confidence, γ. 112
21 A summary of the efficiencies of the tuned configurations measured in Experiments V, VI, and VII. 130
22 A summary of the efficiencies of the *untuned* configurations measured in Experiments V, VI, and VII. 132
23 Coded parameter settings across all SUTs plotted using parallel coordinates. The near-optimal settings are shown as solid lines and the remaining settings as fainter dotted lines for comparison. 135
24 Near-optimal coded parameter settings for each SUT plotted using parallel coordinates. 136
25 The optimal common parameter settings as coded values and the boundaries of the region of interest. 139

26 Binary trees represented by two strings from the language defined by the example grammar. 158
27 A grammar that defines valid inputs for testing circBuff. 176
28 The high-level control flow structure of the e-puck controller. 177
29 Plan view of an example mission configuration. 177
30 A grammar that defines valid inputs (mission configurations) for testing epuck. 178
31 The construction of the robot's environment using object clusters. 178
32 Mission configurations sampled from the best input profile derived by six of the 64 trials for epuck. 182

33 Single-precision performance of the highest-performance GPU cards of recent NVIDIA GeForce iterations. 188
34 The physical architecture of a CUDA GPU card. 189
35 The conceptual architecture of a CUDA GPU card. 190
36 The implementation of the search algorithm in the device-side (GPU) kernel. . 194
37 The timing start and end points for the CPU and GPU implementations. . . . 200
38 Boxplots comparing the distributions of run times for the CPU and GPU implementations for each SUT. 202
39 An example of the spatial migration method. 204
40 Boxplots comparing the distribution of run times for different frequencies of migration. 206
41 The revised algorithm architecture that performs only the execution of the SUT in the device-side (GPU) kernel. 208

List of Tables

1 Example test cases for `daysInMonth`. 26
2 Branch distance functions used by Tracey. 32

3 Characteristics of the SUTs `simpleFunc`, `bestMove`, and `nsichneu`. 62
4 Algorithm parameter settings used in Experiment I. 66
5 A summary of results of Experiment I. 67
6 Algorithm parameters used in Experiment IV. 81

7 Characteristics of the SUTs `bestMove`, `nsichneu`, `cArcsin`, and `fct3`. 102
8 Algorithm parameters for configuration A_{base}. 105
9 Results of Experiment V. 106
10 Algorithm parameters for configuration A_{dmut}. 110
11 Results of Experiment VI. 111
12 Results of Experiment VIIa. 116
13 Results of Experiment VIIb. 118
14 Results of Experiment VIIc. 120
15 Results of Experiment VIId. 121
16 Results of Experiment VIIe. 122
17 Additional algorithm parameters for configuration A_{natm}. 123
18 Results of Experiment VIIf. 123
19 The additional algorithm parameter for configuration $A_{\text{eval}}(*)$. 124
20 Results of experiment VIIg. 125
21 Additional algorithm parameters for configuration A_{sima}. 126
22 Results of Experiment VIIh. 126
23 Results of Experiment VIIi. 128
24 Consolidated results of Experiment VII. 129
25 The interquartile range of the near-optimal parameter settings. 135
26 The optimal common parameter settings for configuration $A_{\text{inib}}(1.0)$. 140
27 Results of Experiment VIIIc. 141
28 The efficiency of the baseline configuration, optimal SUT-specific parameters, and common parameters. 141
29 The median run times of the algorithm configurations A_{base} and A^{*}_{comm}. 142
30 A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^{*}_{comm}. 143
31 A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^{*}_{comm} *without* caching. 145

32 A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^{*}_{comm} *with* caching. 145
33 Characteristics of the SUTs `bestMove`,`nsichneu`, `cArcsin`, `fct3`, and `tcas`. . . . 148
34 The efficiency of the baseline configuration, optimal SUT-specific parameters, and common parameters applied to `tcas`. 148

35 Algorithm parameters used when input profiles are represented as an SCFG. . 165
36 The algorithm parameter settings used for the SCFG representation. 171
37 Results of Experiment XI. 172
38 The algorithm parameter settings used in Experiment XII. 179
39 Results of Experiment XII. 180

40 Results of Experiment XIII. 197
41 The algorithm parameter settings used in Experiment XIV. 199
42 Summary of results for Experiment XIV. 201
43 Comparison of algorithm timings with and without migration. 205
44 Results of Experiment XVI. 209

Acknowledgements

I consider myself very fortunate to have benefitted from the unparalleled knowledge and guidance of my supervisor, John Clark. My internal assessor, Jon Timmis, has provided thoughtful advice on planning and completing my PhD, and this thesis has been greatly enhanced by the thorough feedback from my two external assessors, Rob Hierons and Phil McMinn.

I would also like to acknowledge Rob Alexander for his invaluable comments on an early draft of this thesis, and both Hélène Waeselynck and Paul O'Dowd for providing software used as case studies. Feedback from Hélène, Robert Feldt, and many other researchers in the software testing and search-based software engineering communities has been invaluable in ensuring the utility of my research.

Meanwhile, the support of my colleagues, friends, and family—too numerous to mention by name—has been both generous and indispensable.

I am extremely grateful, and would like to thank them all.

Declaration

I declare that the contents of this thesis derive from my own original research between October 2007 and July 2013, the period during which I was registered for the degree of Doctor of Philosophy.

Contributions from this thesis have been published in the following papers:

- Simon Poulding and John A Clark. Efficient Software Verification: Statistical Testing Using Automated Search. *IEEE Transactions on Software Engineering* 36(6) 763–777, 2010.

  Based on research described in chapter 2 of this thesis.

- Simon Poulding, John A Clark, and Hélène Waeselynck. A Principled Evaluation of the Effect of Directed Mutation on Search-Based Statistical Testing, In *Proc. 4th International Workshop on Search-Based Software Testing (SBST 2011), held in conjunction with ICST 2011*, pages 184–193, 2011.
  (Winner of best paper award.)

  Based on research described in sections 3.3 to 3.6 of this thesis, and extended with contributions from Hélène Waeselynck.

- Simon Poulding, Robert Alexander, John A Clark, and Mark J Hadley. The Optimisation of Stochastic Grammars to Enable Cost-Effective Probabilistic Structural Testing. In *Proc. Genetic and Evolutionary Computation Conference (GECCO 2013)*, in press, 2013.
  (Winner of best paper award, SBSE track.)

  Based on research described in chapter 4 of this thesis, and extended with contributions from Robert Alexander and Mark J Hadley.

Chapter 1

Introduction

1.1 *Overview*

The economic impact of software faults is substantial: a report prepared for the US National Institute of Standards and Technology in 2002 estimated that the cost of such faults to the US economy was $59.5 billion annually (Research Triangle Institute, 2002). There are two potential solutions to this problem: either provide additional resources for testing, or use more efficient testing processes that detect more faults using the same resources. Since testing typically accounts for the majority of costs in software development—Myers et al. (2011) suggest that 50% of the elapsed time and more than 50% of the total budget are reliable rules of thumb—there may be little scope to dedicate additional resources to testing. There is therefore a strong motivation for improving the efficiency of software testing, and this desire for cost-effectiveness is the context of this thesis.

It is normally impractical to test a software artefact exhaustively using *all* of its possible inputs; instead the test engineer employs a testing strategy to select a small subset of the inputs with which to test the software. This inevitably means that some faults may not be detected, and for a given size of input subset, testing strategies differ greatly in the number of faults they are able to detect. It would therefore appear that cost-effective testing should use the most efficient of these strategies, but the cost of *implementing* the most efficient strategies—selecting a subset of inputs that must often satisfy complex constraints—may be greater than any savings achieved by *using* the efficient subset of inputs for testing the software. This is the problem that is addressed in this thesis: we show that the use of automated computational search algorithms can reduce the cost of implementing efficient software testing strategies and so enable more cost-effective software testing.

We expand on the problem outlined above in the remainder of this chapter. In section 1.2, we explain the need for a testing strategy and that the selection criterion used by the testing strategy affects how efficient the testing will be at detecting faults. We illustrate the relationship between testing strategy and efficiency using two commonly-used strategies: structural testing (section 1.4) and random testing (section 1.6). *Statistical testing*—an example of a potentially highly-efficient testing strategy that may, in practice, be prohibitively costly to implement—is introduced in section 1.7. In section 1.8, we define our research hypothesis: that concrete implementations of statistical testing can be derived cost-effectively using automated computational search. In section 1.9, we outline how this hypothesis is evaluated by the research described in the subsequent chapters of this thesis.

1.2 Software Testing Strategies

In this first section we briefly introduce software testing terminology that will be used throughout this thesis, explain the need for a testing strategy, and discuss the costs incurred when implementing a strategy.

1.2.1 The Software Testing Process

Testing Objective The objective of software testing is to provide empirical evidence about a hypothesised property of a software artefact.

The hypothesised property may be *functional*: concerning the actions of the software; or *non-functional*: concerning the mechanism by which the software performs these actions. Examples of functional properties are that the software behaves in a manner consistent with its specification, or that its behaviour is identical to the previous version of the software. Examples of non-functional testing are *dynamic timing analysis* for which the property may be a worst-case bound on the software's execution time (e.g. (Wilhelm et al., 2008)); and *reliability evaluation* for which the property is an estimate of the failure rate of the software in operation (e.g. (Littlewood, 2000)).

In this thesis the primary focus is on functional testing; nevertheless, the proposed approach could also be applied to the evaluation of non-functional properties.

Test Cases and Sets Executing the *software-under-test* (SUT) just once does not normally provide sufficient evidence about the functional behaviour of the SUT. Instead, the SUT is executed multiple times. Each execution of the SUT is a *test case*, and a *test set* is a coherent series of test cases which, as a whole, provide sufficient evidence as to the behaviour of the SUT.

Test Inputs The manner in which a test case exercises the behaviour of the SUT is determined by the *input* provided to the SUT. The set of possible inputs is the *input domain*.

For a simple function written in a procedural language, the input domain may be the range of values supplied as arguments to the function. For a web server, the input domain may be the set of possible HTTP requests.

The behaviour of the SUT may be affected by the SUT's state, the state of the system of which it is a part, or the state of the computing environment. For example, in object-oriented code, the behaviour of a method call may depend on the values of the object's data fields set by previous method calls (the SUT's own state). Similarly, a function call may raise an exception if there is insufficient system memory available (a state of the computing environment) when it attempts to allocate space for a data structure. In this situation, it is insufficient to define the test input solely in terms of the arguments passed explicitly to the SUT: the input must additionally specify relevant state.

Test Oracle For each test case, the SUT is executed using the specified input and the relevant behaviour of the SUT is observed. Examples of relevant behaviour are the value returned by a function, the HTTP response from a web server, or the change in the internal state of the system.

The correctness of the observed behaviour is determined using an *oracle*, such as (Sommerville, 2007):

- a human domain expert;

- a specification document interpreted by the test engineer;
- an earlier version of the SUT that has been verified;
- an independent implementation of the same specification (a technique called *back-to-back testing*).

If the observed behaviour is determined to be incorrect, a *fault* is assumed to exist in the SUT.

1.2.2 Testing Strategies, Techniques, and Costs

The Need for Strategy For unequivocal evidence that the SUT behaves correctly, one could construct a test set in which every possible input to the SUT has a corresponding test case. However, for anything other than the simplest of software, the input domain is too large for such exhaustive testing to be feasible. Instead a *testing strategy* is employed to select a finite subset of the input domain from which a test set is constructed.

In order to *guarantee* the absence of faults, Goodenough and Gerhart (1975) propose that an 'ideal' testing strategy—which they refer to as a *data selection criterion*—must have the following properties:

valid: for every possible fault, at least one test set constructed using the strategy detects the fault; and,

reliable: the SUT passes all test sets constructed using the strategy, or passes none.

The validity property ensures that all faults *could* be detected using the testing strategy; the reliability property ensures that all detectable faults are guaranteed to be detected by any *one* test set constructed using the strategy. Therefore, if the SUT passes all the test cases in any *one* test set constructed using an 'ideal' strategy, it is guaranteed to be free from all faults.

However, such an ideal testing strategy is impractical in practice. Weyuker and Ostrand (1980) argue that in the absence of any knowledge as to how the SUT is implemented, it is possible to choose *any* input from the input domain and envisage an implementation of the SUT containing a fault which is detected *only* by that input. Thus to ensure validity, every input must have a non-zero probability of being selected as a test case (which is often simple to ensure), but to ensure reliability *every* test set must include a test case for *each* input in the input domain. In other words, each and every test set is an exhaustive test of the input domain.

However, if information *is* available as to how the SUT is implemented—by examination of the source code, inferred from the specification, or based on the tester's expertise—the argument of Weyuker and Ostrand may no longer apply. Indeed, many testing strategies use information about the SUT's implementation to approximate (but not necessarily achieve) the properties of validity and reliability. Examples of such strategies are discussed later in this chapter.

Efficiency and Cost-Effectiveness The tasks of selecting inputs to construct test sets, executing the SUT for each test case, and using an oracle to check whether the observed behaviour is correct, are typically expensive in terms of both money and time. Similarly, failing to detect faults in the SUT may incur potentially substantial costs, and, for some types of software, may also lead to injury or death. There is therefore a trade-off between the cost of performing the

testing and consequences of not detecting faults. Nevertheless, the objective considered by Goodenough and Gerhart—to guarantee the absence of faults—is typically not an economically viable one. Instead the objective becomes the detection of as many faults as possible within the cost constraints on the testing process. To quote Myers et al. (2011):

> [S]ince exhaustive testing is out of the question, the objective should be to maximize the yield on the testing investment by maximizing the number of errors found by a finite number of test cases.

In order to compare testing strategies on this basis, we will use the term *cost-effectiveness* to refer to the average number of faults detected per unit cost of testing.

There may be more than one method of constructing test sets for a given testing strategy: for example, there are both analytical and dynamic methods of implementing structural testing strategies (we discuss these methods in section 1.4). We will use the term *testing technique* to refer to a particular method of implementing a testing strategy.

If a simplifying assumption is made that the *per test case* costs of executing the SUT and applying the oracle are the same for all *testing strategies*, then a cost-effective *testing technique* will be a combination of an effective testing strategy (one that detects many faults with a small test set) and a low-cost method of implementing that strategy (i.e. of generating test sets).

In the following sections we discuss three testing strategies—deterministic structural testing, random testing, and statistical testing—in terms of their ability to detect faults and the costs associated with the techniques that implement these strategies.

1.3 Coverage-Based Testing Strategies

Many testing strategies are based on the concept of *coverage* of a set of elements. The coverage elements are often derived from the SUT's implementation or specification. Test inputs are chosen so that, ideally, every coverage element is 'exercised' by one or more test cases in the test set. The assumption is that the guidance provided by coverage when selecting test cases will enable faults to be detected efficiently during testing.

Testing strategies of this type include:

Structural Testing: The coverage elements are structural aspects of the SUT's source code, such as statements or conditional branches. An element is exercised by a test case if the execution path passes through it.

Specification Testing: The coverage elements are aspects of the SUT's specification. If the specification is textual, the coverage elements may be atomic behaviours specified in the document. If the specification is expressed more formally as a model, the coverage elements may be based on the structure of the model. An element is exercised if the functionality corresponding to that part of the specification is demonstrated by a test case. (Zhu et al. (1997) use the term structural testing for coverage of *either* source code elements *or* specification elements and distinguish them as *program-based structural testing* and *specification-based structural testing* respectively.)

Mutation Testing: The coverage elements are *mutants*: versions of the SUT seeded with faults. A mutant is 'exercised' in the sense of a coverage element if it is *killed* by a test case: i.e. the fault in the mutant is detected because the observed behaviour of the mutant differs from that of the original SUT.

Coverage testing strategies are defined by a *test adequacy criterion* expressed in terms of a property of the generated test sets. In their comprehensive survey of coverage testing, Zhu et al. (1997) identify two forms of such criteria. The first form is a Boolean property; for example, a criterion that every statement in the SUT's source code is exercised by at least one test case in the test set. The criterion acts as a stopping rule during the construction of the test set: test cases are added until the property is satisfied.

The second form of adequacy criterion is a scalar property; for example, the proportion of coverage elements exercised by at least one test case. In this form the criterion acts as a metric that can be used to compare test sets against one another, or against a target value, and may be used as a *post hoc* measure of the potential efficacy of the generated test sets.

1.4 *Structural Testing*

In this section, we discuss the adequacy criteria and techniques used to implement one form of coverage-based testing strategy: *structural testing*.

1.4.1 *Adequacy Criteria*

Running Example Figure 1 is a listing of a function, `daysInMonth`, implemented in C. We will use this function as a running example of a SUT throughout this chapter.

```
1  int daysInMonth(int m, int y) {
2    int d;
3    if (m <= 7)
4      d = 30 + m % 2;
5    else
6      d = 31 - m % 2;
7    if (m == 2) {
8      if ((y % 4 == 0) & (y % 100 != 0))
9        d = 29;
10     else
11       d = 28;
12   }
13   return d;
14 }
```

Figure 1 – Source code of the example SUT `daysInMonth`.

The specification for the function is to return the number of days for the given month in the Gregorian calendar for any year between 1583 AD (the first full year of the Gregorian calendar) and (arbitrarily) 2199 AD inclusive. Its input arguments are: m, the month as an integer between 1 and 12; and y, the year. The input domain for this SUT is therefore:

$$D = \{(m, y) \in \mathbb{Z}^2 : (1 \leq m \leq 12) \wedge (1583 \leq y \leq 2199)\} \tag{1}$$

Control Flow Criteria Control flow criteria express adequacy in terms of the SUT's control flow graph: a directed graph illustrating possible execution paths through the code. Nodes in the graph represent blocks of contiguous statements such that if the first statement in the block is executed, then the remaining statements are also executed. Directed edges represent a transfer of control between nodes. Where a node has more than one outbound edge as a result of a conditional control statement, the value taken by the Boolean predicate in the conditional statement determines which edge is followed. Figure 2 shows the control flow graph for the SUT `daysInMonth`.

For *statement coverage*, the coverage elements are code statements. In terms of the control flow graph, statement coverage corresponds to coverage of the graph nodes.

For *branch coverage*, the set of structural elements are conditional branches in the code, or edges in the control flow graph. Table 1 lists some example test cases for `daysInMonth`, and the edges of the control flow graph (labelled in figure 2) they would exercise. A test set consisting of only the first three test cases (A, B and C) would cover all branches of `daysInMonth`: each of the 11 edges is executed by at least one of the three test cases.

For *path coverage*, the set of structural elements consists of all valid execution paths in the control flow graph. The SUT `daysInMonth` illustrates one of the practical considerations for

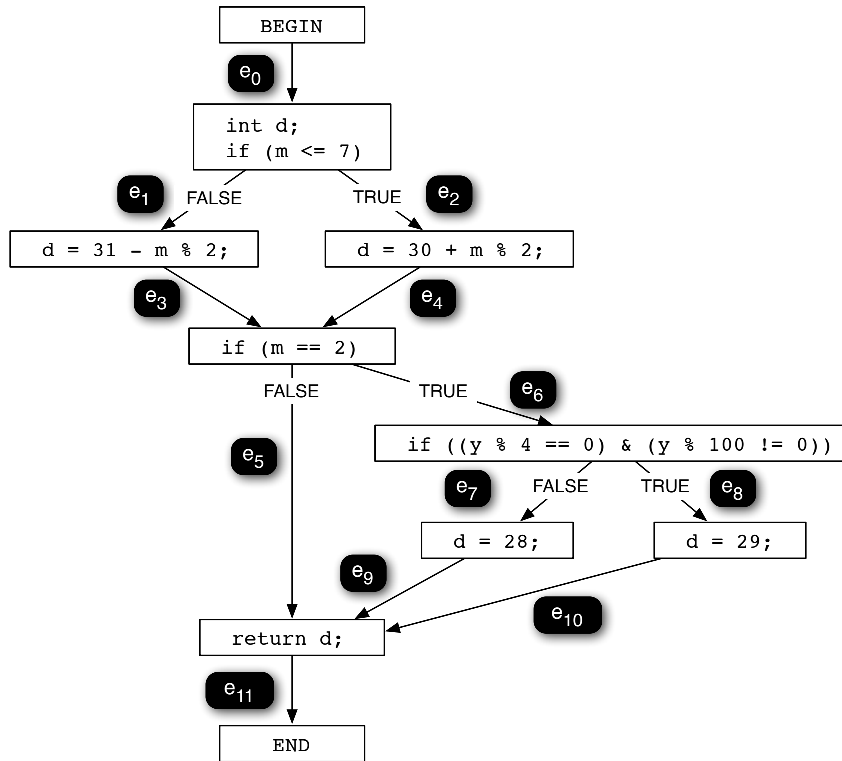


Figure 2 – Control flow graph of the SUT `daysInMonth`. Labels e_i identify the edges.

Test Case	Input m	Input y	Execution Path	Conditions m<=7	m==2	y%4==0	y%100!=0
A	10	1967	$e_0e_1e_3e_5e_{11}$	FALSE	FALSE		
B	2	2033	$e_0e_2e_4e_6e_7e_9e_{11}$	TRUE	TRUE	FALSE	TRUE
C	2	1780	$e_0e_2e_4e_6e_8e_{10}e_{11}$	TRUE	TRUE	TRUE	TRUE
D	3	1844	$e_0e_2e_4e_5e_{11}$	TRUE	FALSE		
E	2	2100	$e_0e_2e_4e_6e_7e_9e_{11}$	TRUE	TRUE	TRUE	FALSE

Table 1 – Example test cases for `daysInMonth`, with associated execution paths and condition values.

this criterion: many theoretical paths may be infeasible owing to dependencies between predicates and as a result of assignments to the variables used in the predicates. For `daysInMonth`, paths beginning $e_0e_1e_3e_6$ are infeasible owing to relationship between the predicates at lines 3 and 7: if `m<=7` is false (edge e_1), then `m==2` cannot be true (edge e_6). This leaves four feasible paths for `daysInMonth`, and a test set consisting of the first four test cases (A, B, C, and D) in table 1 would be sufficient to cover all four paths.

A further complication for path coverage is that unbounded loops in the control flow graph can lead to an infinite number of possible paths. Zhu et al. (1997) discuss a number of variations to the path coverage criterion that address this problem, such as the coverage of only simple paths (no edges are repeated), elementary paths (no nodes are repeated), or covering all feasible subpaths shorter than or equal to a predetermined length. The latter variation is described as ‘PATH$_n$’ testing by Gourlay (1983), where n is the number of nodes in the subpath. Gourlay suggests that values of $n \geq 3$ can force the execution of iterative statements without creating the infinite number of coverage elements that could occur with standard path coverage.

Logical Predicate Criteria Myers et al. (2011) describes statement, branch and path coverage as forms of 'logic-coverage testing', and describes potentially 'stronger' criteria that consider the composition of the predicates controlling control flow in addition to nodes and edges in the control flow graph itself. (The notion of 'strength' is discussed below.)

Myers et al. uses the term *decision* for the entire Boolean expression associated with a control statement, and the term *condition* for each atomic Boolean predicate that makes up the expression. For example, the decision at line 8 of `daysInMonth` consists of two conditions: `(y % 4 == 0)`, and `(y % 100 != 0)`.

The adequacy criterion for *condition coverage* is that each condition in the SUT evaluates to TRUE for at least one test case, and to FALSE for at least one test case. Table 1 shows that the condition coverage criterion could be satisfied by a test set consisting of all five example test cases.

Condition coverage does not necessarily imply branch coverage (*decision coverage* in the terminology used by Myers), and this can be shown by a simple counter-example. For a decision $(A \wedge B)$, condition coverage could be satisfied by two test cases: one for which condition A is TRUE and condition B is FALSE; and a second case for which A is FALSE and B is TRUE. In both cases the decision evaluates to FALSE, so branch coverage is not guaranteed. To rectify this problem, the criterion of *decision/condition coverage* may be applied which supplements the condition coverage criterion with the requirement that all decisions must evaluate to each possible outcome for at least one test case.

An even stronger criterion is *multiple-condition coverage* which requires that for each decision, all feasible combinations of values for its constituent conditions are exercised by the test set. If there are j conditions in a decision, the test set must contain a minimum of 2^j cases for all combinations of these conditions to be tested (assuming all combinations are feasible) in order to satisfy multiple-condition coverage. Hayhurst et al. (2001) argue that multiple-condition coverage is often impractical, citing evidence of real-world code containing more than 36 conditions in one decision.

Modified condition/decision coverage (MC/DC) is a practical alternative to multiple-condition coverage (Hayhurst et al., 2001). Rather than requiring all combinations of conditions, the criterion is that there must be sufficient test cases to demonstrate that each condition independently affects the outcome of the decision. In simple terms, this requires that for each condition, there is one test case where the condition evaluates to FALSE, and there is a second case where the condition evaluates to TRUE and all other conditions evaluate to the same value they took in the first case; in addition, the value of the entire decision must be different for both test cases. In general, this requires fewer test cases than multiple-condition coverage: for a decision affected by j inputs, a minimum of $j + 1$ test cases are required (Hayhurst et al., 2001). Unlike the definition of decision, decision/condition and multiple-condition coverage by Myer, MC/DC considers all Boolean expressions—including, for example, those in assignment statements—to be decisions that must be covered, not just those in control statements (Hayhurst et al., 2001).

Data Flow Criteria Data-flow criteria define the set of coverage elements by combining the control flow graph with variable usage.

Rapps and Weyuker (1982) define two categories of variable usage in the SUT: in a definition or *def*, a variable is assigned a value; in a *use* the value of the variable is referred to in an expression. The *use* category is further split into predicate use (or *p-use*) when the variable is used in a Boolean expression of a control statement, and computational use (or *c-use*) for all

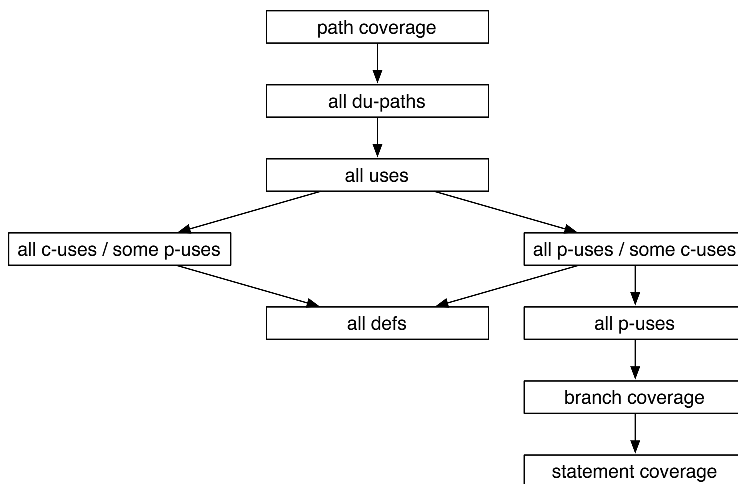


Figure 3 – The subsume relationship between adequacy criteria. The direction of arrows is from a strong criterion to a weaker criterion that it subsumes.

other cases. For example, in the code of `daysInMonth` in figure 1, defs of `m` and `y` occur at line 1; defs of the `d` occur at lines 4, 6, 9, and 11; p-uses of `m` occur in lines 3 and 7; p-uses of `y` occur at line 11; c-uses of `m` occur in lines 4 and 6.

Rapps and Weyuker define adequacy criteria in terms of *def-clear paths*: subpaths between a def and a use of the same variable where no other defs occur on the subpath. For example, the *all defs* criterion requires that for each def, at least one test case exercises a def-clear path between the def and one of the variable's uses. The *all uses* criterion requires that between each def and all reachable uses of the variable, there is at least one test case that exercises a def-clear path. The *all c-uses/some p-uses* criterion requires that for each def, one or more test cases exercise a def-clear path between the def and all-c-uses; if no c-uses are available, then one or more test cases must exercises a def-clear path between the def and one p-use. Further criteria such as *all p-uses/some c-uses* and *all p-uses* are defined analogously. The *all du-paths* criterion requires that for each def and each use of a variable, the test set exercises a def-clear path where there are no loops in the subpath between the def and use.

Relationship Between Adequacy Criteria It is possible to define an ordering on adequacy criteria. For example, we may consider an adequacy criterion A as being 'stronger' (in some sense) than a criterion B if the set of structural elements defined by A is a superset of the set of element defined by B. Zhu et al. (1997) formalise this ordering as the *subsume relation*: A subsumes B if for all SUTs, a test set that is adequate by criterion A is also adequate by criterion B.

Figure 3, adapted from (Rapps and Weyuker, 1982), shows the subsume relationship between some of the criteria described above.

1.4.2 Fault-Detecting Ability

Intuitively, it is reasonable to expect that stronger criteria result in test sets that are, on average, better at detecting faults. However, Frankl and Weyuker (1993) show theoretically that a stronger criterion does not *guarantee* a better ability to detect faults.

Moreover, there is no guarantee that a fault will be detected by a test set that satisfies even the strongest adequacy criteria. For example, there is a fault in the implementation of

`daysInMonth`: it does not correctly identify years that are multiples of 400 as being leap years. The test set formed by test cases A, B, C and D listed in table 1 satisfies the path coverage criterion, the strongest of the criteria shown in Figure 3, yet none of the four test cases would detect the fault. Similarly, a test set formed by all five test cases satisfies multiple-condition coverage, which for this particular SUT is a 'stronger' test set than for path coverage, but none of the test cases would detect the fault.

Identifying the subdomains of the input that exercise each structural element clarifies why this is the case. Each of the four feasible paths of `daysInMonth` are associated with a subdomain of the input domain: any test input chosen from the subdomain would exercise that particular path. (In this case, the subdomains partition the input domain, but adequacy criteria do not generally induce such a partition.)

The subset of the input domain that would detect the fault in the SUT is given by:

$$\begin{aligned} D_{\text{fault}} &= \{(m,y) \in \mathbb{Z}^2 : (m=2) \wedge (1583 \le y \le 2199) \wedge (y \bmod 400 = 0)\} \\ &= \{(2,1600),(2,2000)\} \end{aligned} \tag{2}$$

An input from D_{fault} would only be selected for a test case exercising the path $e_0e_2e_4e_6e_7e_9e_{11}$ in the faulty SUT. The subdomain of this path is:

$$D_{\text{path}} = \{(m,y) \in \mathbb{Z}^2 : (m=2) \wedge (1583 \le y \le 2199) \wedge (y \bmod 100 \notin \{4,8,12,\ldots,96\})\} \tag{3}$$

Since D_{fault} is not empty, it is possible to choose a test case that detects the fault (i.e. the adequacy criterion is *valid* in the terminology of Goodenough and Gerhart). However since D_{fault} is a proper subset of D_{path}, not all test sets will contain such a test case (i.e. the criterion is not *reliable*).

Since a testing strategy rather than exhaustive testing is being applied, there is no expectation that the reliability property is satisfied. However, for this particular fault, the property is not even approximated: the chance that a test set generated using the path coverage criterion would detect the fault is extremely small. Since $|D_{\text{path}}| = 545$ and $|D_{\text{fault}}| = 2$, if all inputs in D_{path} have the same chance of being chosen by the structural testing technique, only 0.367 % of the generated test sets would contain a test case that detects the fault.

1.4.3 An Analytical Technique: Symbolic Execution

Symbolic execution is an analytical technique for constructing test sets for structural testing, first proposed in the 1970s in the work of Clarke (1976), and Ramamoorthy et al. (1976), among others. For each structural element in the set defined by the adequacy criterion, a path is chosen in the SUT that exercises that element. The constraints on the inputs that ensure that the execution follows the chosen path are identified, and a solution (an input) that satisfies the accumulated constraints is found analytically. The path constraints are constructed by considering the inputs as algebraic variables and then deriving the outcome of each expression on the path in terms of these variables.

As an example, consider the path $e_0e_2e_4e_6e_8e_{10}e_{11}$ in `daysInMonth`. We denote the input arguments by variables M and Y, distinguishing these algebraic variables from program variables by using upper case. The first constraint that these variables must satisfy is the definition of the input domain in equation (1), giving a constraint $(1 \le M \le 12) \wedge (1583 \le Y \le 2199)$. In order to trace the edge e_2, the predicate `(m <= 7)` at line 3 must evaluate to TRUE: this provides the constraint $M \le 7$. Similarly, edges e_6 and e_8 provide the constraints

$M = 2$ and $(Y \bmod 4 = 0) \wedge (Y \bmod 100 \neq 0)$ respectively. The accumulated constraints are therefore:

$$(1 \leq M \leq 12) \wedge (1583 \leq Y \leq 2199) \wedge (M \leq 7) \\ \wedge (M = 2) \wedge (Y \bmod 4 = 0) \wedge (Y \bmod 100 \neq 0) \tag{4}$$

Any pair of values for M and Y that satisfies this constraint is a valid test input that exercises the chosen path.

However, there are number of practical difficulties in the use of symbolic execution. Firstly, finding a solution to the constraints may be difficult. When the variables are numeric and the constraints are linear, linear programming techniques may be used to find a solution; Clarke (1976) uses this approach. In general, however, the constraints are not linear and other approaches must be taken. Ramamoorthy et al. (1976), for example, use "systematic trial and error". As McMinn (2004) suggests, it is unlikely that such an approach is scalable to more complex programs.

A second issue arises when the SUT uses compound data types such as arrays. If an array subscript in an assignment or reference to the array is dependent on an input variable, and therefore expressible only in terms of an algebraic expression involving the inputs with a range of possible values, symbolic execution must consider each and every consequence of the subscript taking a value from this range. Techniques have been suggested to control the resultant complexity, such as the method described by Ramamoorthy et al. (1976) that delays the construction of algebraic expression for array elements until the generation of test data itself: at this point it may be possible to assign values to some of the 'simple' variables on which the subscripts depend, without resolving the values of the array elements themselves. Nevertheless, Korel (1990) describes the results as "unsatisfactory" and uses this as an argument in favour of dynamic approaches to test data generation.

1.4.4 A Dynamic Technique: Search-Based Software Testing

Dynamic testing techniques execute the SUT itself in order to provide information about the structural elements exercised by particular input variables. As Miller and Spooner (1976) note in one of the earliest papers on search-based software testing, "the program itself presumably gives a very efficient procedure for numerical evaluation of the [path constraints]" .

The techniques described in this section combine execution of the SUT with automated computational search in order to derive input data for structural testing, an approach often referred to as search-based software testing.

Search-Based Software Engineering Search-based software engineering (SBSE): the application of automated computational search to the solution of software engineering tasks. Search-based software testing (SBST) is a form of SBSE.

Clark et al. (2003) observe that many tasks in software engineering have the following characteristics:

- it is easy to *check* whether candidate solutions are acceptable, but difficult to *construct* such solutions;
- the requirement is to find an *acceptable* rather than *optimal* solution; and,
- there are often competing constraints that must be satisfied.

Such tasks, they argue, can be reformulated as a search problem by constructing a fitness metric that quantifies the quality of a candidate solution. The task may then be solved by automated metaheuristic search techniques—such as genetic algorithms (Holland, 1975; Goldberg, 1979), simulated annealing (Kirkpatrick et al., 1983), genetic programming (Koza, 1992) and artificial immune systems (De Castro and Timmis, 2002)—using the fitness metric to guide the search process.

There are two key advantages to reformulating the task as an optimisation problem to be solved by an automated search algorithm. Firstly, automated approaches require less effort on the part of the software engineer and so are typically cheaper than manual approaches to the same task. Secondly, as software engineering tasks become increasingly large and complex, SBSE approaches are able to scale with them through the use of increasingly affordable high-performance computing resources. As Harman (2007) points out, many search algorithms are "naturally parallelisable", enabling SBSE techniques to take advantage of the highly-parallel architectures employed by many high-performance computing resources.

Over the last ten years, this approach to software engineering has been successfully applied to tasks throughout the engineering lifecycle, including: requirements engineering (Zhang et al., 2007), project planning (Alba and Chicano, 2007), software task allocation (Emberson and Bate, 2007), code refactoring (Harman and Tratt, 2007), protocol synthesis (Hao et al., 2004), and the design of resource-constrained algorithms (White et al., 2008).

The extensive surveys by McMinn (2004) and Ali et al. (2010) demonstrate that SBST is one of the most active research fields in SBSE. SBST is often applied to the construction of test sets for structural testing since this task demonstrates the characteristics of potential SBSE applications proposed by Clark et al.: it is difficult to *generate* a test set satisfying the coverage adequacy criterion, but easy to *evaluate* whether a candidate test set satisfies the criterion.

In the following, we review SBST techniques for structural testing. All these techniques operate on one structural element at a time: the test set is constructed by applying the technique to each of the structural elements in turn. We follow McMinn (2004) in classifying SBST techniques according to the nature of fitness metric used to the guide the search.

Predicate-Based Fitness Metrics Korel (1990) describes an SBST technique for automatic test data generation which provides the foundation for much of the subsequent work in the field. The first step in the technique is to choose an execution path in the control-flow graph that exercises the given structural element; search is then used to find an input that takes this target path.

The fitness metric used by Korel considers the predicates that determine the control flow. Boolean conditions in the predicate are assumed to be a simple relational expression of the form:

$$X \,\mathrm{op}\, Y \tag{5}$$

where X and Y are expressions that evaluate to a real number, and op is the relational operator.

Each Boolean condition is transformed an equivalent metric. Here we use the more convenient, but otherwise similar, metrics defined by Tracey (2000) of the form:

$$G_{\mathrm{op}}(X, Y) \tag{6}$$

G_{op} is a real-valued function that takes the value of the expressions X and Y as its input. The functions corresponding to each relational operator (as defined by Tracey) are shown in

relational expression	value of G_{op} when expression is TRUE	FALSE
$X = Y$	0	$\|X - Y\| + \kappa$
$X \neq Y$	0	κ
$X < Y$	0	$X - Y + \kappa$
$X \leq Y$	0	$X - Y + \kappa$
$X > Y$	0	$Y - X + \kappa$
$X \geq Y$	0	$Y - X + \kappa$

Table 2 – Branch distance functions used by Tracey. κ is a small positive constant.

table 2.

The function G_{op} is called a *branch function* by Korel; more recent work on SBST typically refers to the value returned by the function as the *branch distance* (McMinn, 2004). It is a measure of how 'close' the predicate is to evaluating to TRUE for the current input: if the value of G_{op} is 0, then the expression is TRUE; larger (positive) values of G_{op} indicate that the expression is further away from being TRUE. (We assume here that the value of the predicate must be TRUE in order to execute the desired path. If the desired predicate value is FALSE, the branch function may be constructed by propagating a negation over the expression.)

As an example, consider the branch distance for evaluating predicate `(m <= 7)` at line 3 of the SUT `daysInMonth` in the listing of figure 1 as TRUE. An input of $(m, y) = (11, 1879)$ gives a branch distance of $11 - 7 + \kappa = 4 + \kappa$. An input of $(8, 1905)$ gives a branch distance of $1 + \kappa$, and is therefore 'closer' to evaluating the predicate as TRUE. An input $(6, 1915)$ gives a branch distance of 0 since the predicate is TRUE for this input.

To find a test input for a target path $e_0 e_1 \ldots e_m$ (where e_i are edges in the control flow graph), an input point, $d \in D$ is first selected at random. The actual executed path, $\varepsilon_1 \varepsilon_2 \ldots \varepsilon_n$, is determined by executing the instrumented SUT with the input d. If the executed path differs from the target path, the point at which the paths differ, i.e. the smallest j such that $e_j \neq \varepsilon_j$ is identified. Using Korel's terminology, the 'subgoal' is to ensure that the predicate associated with edges e_j and ε_j takes the appropriate value so that the desired edge e_j is taken. Without loss of generality, we assume that e_j is taken when the subgoal predicate is TRUE, or equivalently when the branch distance is 0.

To solve this subgoal, Korel assumes that the input is a vector of variables and applies the *alternating variable* search method. In an 'exploratory' phase, an input variable is increased and decreased by a small amount, keeping all other input variables fixed. The instrumented SUT is executed with the modified inputs, and the value of the branch distance at the subgoal predicate determined. If either change to the input variable results in an improved branch distance, i.e. a value closer to 0, and does not alter the subpath executed prior to the subgoal predicate, then the search continues in a 'pattern phase': increasingly large changes are made to the variable in the same direction until a change makes no improvement. If a further exploratory phase using the same variable results in no improvement, the value of the variable is fixed and the process is repeated with the next input variable. The process continues until the branch distance is zero, indicating the subgoal is solved.

Having solved the subgoal, the new executed path is compared to the target path. If they are not identical, a new subgoal—a new predicate at which the paths now diverge—is identified, and a further alternating variable search performed to solve this new subgoal. The process is repeated until a suitable input is found.

Control Flow-Based Fitness Metrics An alternative approach is to use a fitness metric that quantities the similarity between the executed and target paths. Pargas et al. (1999) uses such a metric based on the control-dependency between nodes in the control flow graph.

A node n_j is *control-dependent* on node n_i if all execution paths that include n_i must also include n_j later in the path. A control-dependency graph can be constructed between nodes where directed edges indicate control-dependency between the nodes. An acyclic path in the control-dependency graph to a given node identifies nodes that *must* be executed, and therefore predicates that *must* be satisfied, for the given node to be executed. Pargas et al. refer to this as the control-dependence predicate path.

The fitness of a candidate input is evaluated by executing the instrumented SUT and counting the number of predicates that the executed path has in common with the control-dependence predicate path: a higher fitness value indicates an input that is closer to executing the chosen node.

Combined Fitness Metrics The disadvantage of Pargas' fitness metric is that it provides little guidance to the search method. All inputs with the same number of predicates in common with the control-dependence predicate path have the same (integer) fitness value, and so there is no information as to which inputs come closer to having more predicates in common with the target path. Subsequent work, such as that by Wegener et al. (2001) and Tracey (2000), combines a graph-based metric equivalent to that of Pargas with the branch distance in order to provide better guidance to the search.

Wegener's equivalent of the Pargas' metric is the *approximation level*. It is calculated by counting the number of branching nodes on the executed path at which an edge is taken that potentially leads to the target node. The count only considers those branching nodes with at least one edge which, if taken, would miss the target node. The higher the approximation level, the better the candidate input.

As an example, we consider again the example SUT `daysInMonth`, and assume that the target node is the statement `d = 29;` which is shown using a double border in control flow graph of figure 2. A candidate input $(m, y) = (2, 1955)$ executes the path $e_0 e_2 e_4 e_7 e_9 e_{11}$ indicated by the shaded nodes and solid edges. The predicate labelled 'A' is not considered since it does not have an edge that, if taken, would always miss the target node. The predicates 'B'and 'C' both have an edge that misses the target node and so *are* considered. Only at 'B' is the correct edge taken by the executed path, so the input $(2, 1955)$ has an approximation level of 1. If the input had also taken the correct edge at node 'C', then the approximation level would be 2.

Wegener et al. (2001) combines the approximation level with branch distance to form the metric:

$$\text{approximation_level} + (1 - \text{branch_distance}) \tag{7}$$

The branch distance, normalised to values between 0 and 1, is assessed at the predicate of node at which the incorrect branch was taken. The higher the value of this combined metric, the fitter the candidate input.

This metric permits the fitness of very different paths to the target node to be compared. Wegener et al. speculate that this search technique will be more efficient than methods—such as that of Korel—where a specific, possibly *difficult*, path must be identified *a priori*. The search can explore all paths that potentially exercise the target structural element, and in doing so can find and use the path that is *easiest* for the search. The authors report empirical

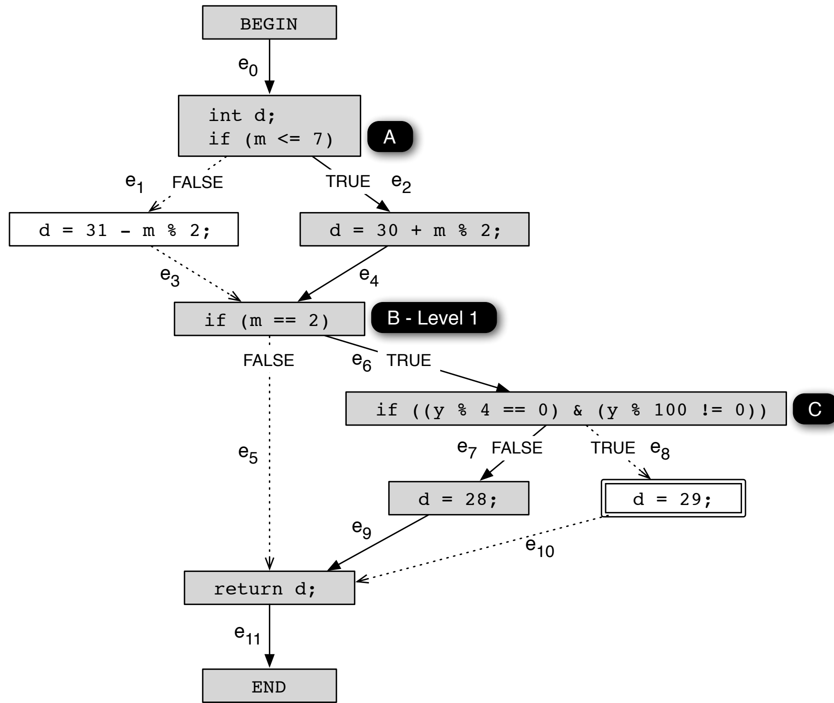


Figure 4 – An illustration of approximation level using the control flow graph of the SUT `daysInMonth`. The target node is shown with a double border, and the executed path for $(m, y) = (2, 1955)$ is indicated using shaded nodes and solid edges.

results for an evolutionary algorithm using this fitness function that demonstrate superior coverage of structural elements, compared to random testing, for a number of SUTs.

Program Stretching Ghani and Clark (2009) propose a technique that facilitates the derivation of test inputs using SBST techniques. The SUT is 'stretched' so that branch predicates that are difficult to satisfy in the original SUT become significantly easier in the stretched version.

For example, the predicate at line 7 of `daysInMonth`, originally `(m == 2)`, is rewritten as `((m+v1)>=2 & (2+v2)>=m)`. v1 and v2 are termed "auxiliary variables". When the auxiliary variables take positive values, the predicate is 'stretched'; when they have values of zero, the rewritten predicate is functionally identical to the original predicate.

The fitness metrics used by the SBST techniques described above are extended with the term:

$$\sum_i |v_i| \tag{8}$$

where the v_i are the auxiliary variables. The lower the value of this term, the better the fitness.

At that start of the search, the auxiliary variables take large positive values that stretch the predicate and enable inputs that exercise the desired execution path to be found with ease. The search proceeds by modifying *both* the input variables *and* the auxiliary variables, with the fitness metric term of equation (8) guiding the search to lower values of the auxiliary variables until they all reach zero. Thus, during the search the stretched predicates return gradually to their original form.

Ghani and Clark hypothesise that having initially found inputs for the stretched SUT, the search is able to refine the solution as the SUT returns to its unstretched form. They

report empirical evidence that for some SUTs, program stretching improves the performance of SBST techniques, measured as the number of fitness evaluations.

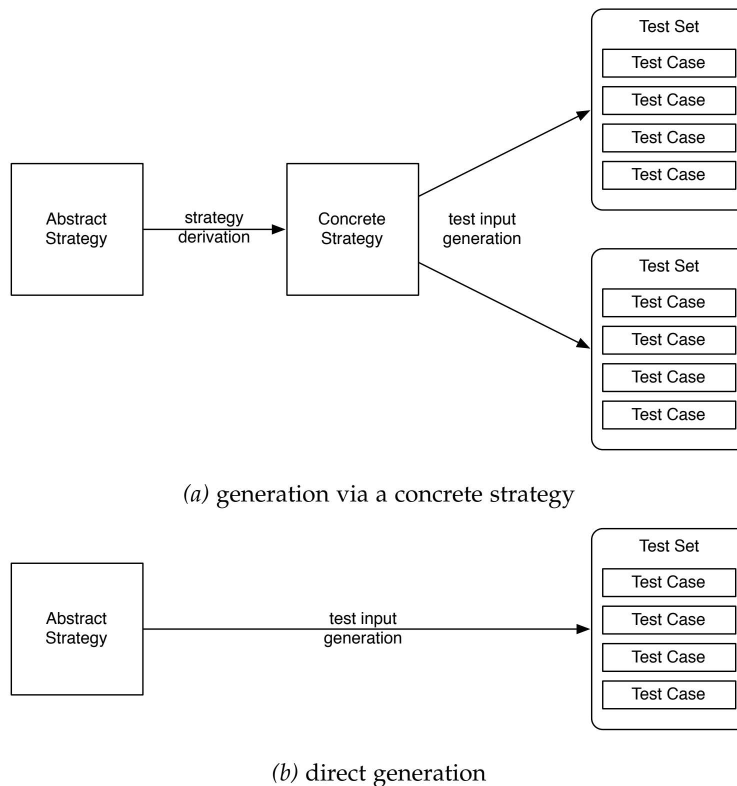
1.5 *Probabilistic Testing Strategies*

Probabilistic testing strategies specify an adequacy criterion in terms of an *input profile*—a probability distribution over the SUT's input domain—from which test inputs are sampled at random. This is in contrast to deterministic testing strategies, such as the structural testing strategy described in the previous section, for which the adequacy criterion is defined in terms of the test sets themselves.

Thus techniques that implement probabilistic strategies must generate test sets in two distinct steps. In the first step, an input profile is derived that satisfies the adequacy criterion, together with a mechanism of sampling from that profile. In the second step, a test set is generated by sampling inputs at random from this input profile.

The input profile is a *concrete strategy*: the realisation of the generic, abstract probabilistic strategy for the specific software-under-test. For many probabilistic testing strategies, it is the derivation of this concrete strategy that is the most costly step in the testing technique, while the second step—generating test sets using the concrete strategy—is relatively straightforward and cheap.

The two-step process enables a particular advantage of probabilistic testing strategies. Having derived the concrete strategy and generated one test set by random sampling from the input profile, further *different* test sets may be constructed at little additional cost by random sampling from the *same* input profile. (We refer here only to the cost of generating the test inputs; there may be additional costs involved in constructing an oracle for the additional test sets.)



(a) generation via a concrete strategy

(b) direct generation

Figure 5 – Test set construction via an intermediate concrete strategy and by direct generation.

This has two potential benefits. Firstly, the amount of testing can be easily adjusted to the available testing budget by altering the number of test sets (or, equivalently, the number of

test cases sampled in a test set). Secondly, the use of multiple *different* test sets during the testing process—e.g. during regression testing after each set of changes to the software—is likely to detect more faults than repeated application of the same test set: the use of multiple randomly-generated test sets comes closer to approximating the validity property of Goodenough and Gerhart than a single fixed test set.

The concept of an intermediate concrete strategy is illustrated in figure 5a. While deterministic testing techniques must also implement the testing strategy for the specific software-under-test, there is no explicit concrete strategy and the test set is generated directly in a single logical step (figure 5b). Therefore the generation of additional test sets for the same software artefact often requires the deterministic technique to be reapplied in its entirety, and—depending on the technique used—may not necessarily generate a *different* test set from the first.

1.6 Random Testing

In this section we consider the simplest form of probabilistic testing strategy: *random testing*.

1.6.1 Adequacy Criterion

Random testing requires the satisfaction of no adequacy criterion, and so the test engineer is free to use any suitable input profile that is defined over the SUT's input domain. Nevertheless, in the absence of any other guidance, the uniform distribution is often chosen as the input profile. We refer to testing using such an input profile as *uniform random testing*.

An alternative form of random testing uses the SUT-specific *operational profile*—the distribution of inputs that would occur when the SUT is used in an operational environment—as the input profile. The motivation is that this input profile focuses the testing effort on the detection of faults that are most likely to occur when the SUT is used. However, determining the operational profile for a SUT can be difficult: the operational profile of a new system may not be known at the time of generating test cases, and if the SUT is a component within a larger system, it may not be possible to derive the operational profile of the component from that of the system as a whole (Hamlet, 2006).

1.6.2 Fault-Detecting Ability

Since there is no adequacy criterion to be satisfied, the cost of deriving the concrete strategy for random testing is small: it is only necessary to construct a suitable random generator for the chosen input profile. This would appear to make random testing a cost-effective strategy, but in practice the fault-detecting ability of random testing is often poor. Although the actual distribution of faults cannot be known in advance of testing, other strategies utilise SUT-specific information, such as the structure of the code, to approximate the fault distribution and focus testing effort accordingly. Random testing makes use of no such information, and therefore is often inefficient: large test sets may be required to detect faults in the SUT.

As an illustration of this potential inefficiency, we consider the SUT `daysInMonth` under uniform random testing. The SUT is sufficiently simple that we are able to calculate the probability of each edge in the control flow graph of `daysInMonth` (figure 2) being exercised by one input sampled at random from the uniform input profile. The probabilities are plotted as a histogram in figure 6. It can be seen that inputs that exercise edges such as e_7 and e_8 are sampled only rarely, and so uniform random testing is unlikely to detect any faults in nodes reached by these edges.

The poor fault-detecting ability of uniform random testing is supported by empirical and theoretical analyses, two of which are described below.

Duran and Ntafos (1984) compared the fault-detecting ability of uniform random testing to partition testing. Partition testing divides the input domain into a non-overlapping subdomains and generates a specified number of test cases for each partition. Some forms of structural testing—such as the example of path testing used by the authors—have this property.

Duran and Ntafos considered a model where partition testing identifies k subdomains, $D_1, D_2, \ldots, D_k$ and that an input chosen from the subdomain D_i has a probability of θ_i of detecting a fault. If partition testing is performed where n_i samples are chosen from each

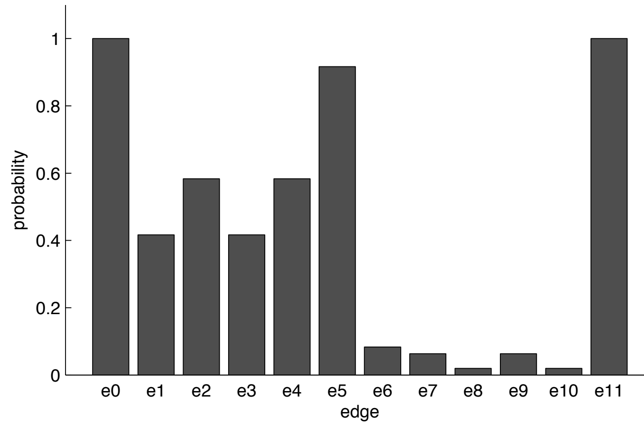


Figure 6 – Histogram showing the probability of edges in the control flow graph of `daysInMonth` being exercised by an input sampled from a uniform distribution.

subdomain, then the probability of detecting at least one fault is:

$$P_{\text{partition}} = 1 - \prod_{i=1}^{k} (1 - \theta_i)^{n_i} \tag{9}$$

If an input chosen at random from the entire input domain has a probability p_i of belonging to subdomain D_i, the probability of one random input detecting a fault is $\theta = \sum_{i=1}^{k} p_i \theta_i$. Thus the probability of a test set generated by uniform random testing of the same size, $n = \sum_{i=1}^{k} n_i$, is:

$$P_{\text{uniform}} = (1 - \theta)^n = (1 - \sum_{i=1}^{k} p_i \theta_i)^{(\sum_{i=1}^{k} n_i)} \tag{10}$$

Duran and Ntafos compared the probabilities P_{uniform} and $P_{\text{partition}}$ using simulation experiments for different (though somewhat arbitrary) values of θ_i, n_i, and p_i. The results showed that although partition testing was, on average, superior to uniform random testing, there were nevertheless a significant number of plausible cases in which uniform random testing was the superior strategy, both in terms of the probability of detecting one fault for a given number of test cases, and in terms of the total number of faults detected. Moreover, when uniform random testing was permitted twice as many test cases as partition testing, it was superior in the majority of cases. The authors argue that since generating test data for uniform random testing is cheap compared to partition testing, this last result suggests that uniform random testing will often be the more cost-effective.

Boland et al. (2003) used a similar model to that of Duran and Ntafos, but analysed the model rather than simulated model parameters. The outcome were a series of conditions, expressed in terms of the values of θ_i, n_i, and $|D_i|$, under which partition testing would be superior to uniform random testing. However, these conditions do not exclude the possibility that uniform random testing is the more cost-effective testing strategy.

Both Duran and Ntafos, and Boland et al., assume that the subdomains D_i partition the input domain, and—as Boland et al. point out—it is unclear whether these results generalise to forms of structural testing where the coverage elements do not partition the input domain.

1.7 Statistical Testing

Thévenod-Fosse and Waeselynck (1991, 1993, 1997) describe a probabilistic strategy for coverage testing which they refer to as *statistical testing*. Here we consider statistical testing applied to structural coverage, but it may also be used for specification coverage (e.g. (Thévenod-Fosse and Waeselynck, 1993)).

The premise of statistical testing is that the potentially poor fault-detecting ability of deterministic structural testing (discussed in section 1.4.2) may be compensated for by generating test sets that, instead of exercising each structural element once, exercise each element *multiple* times with different inputs. The strategy combines the advantages of structural testing with those of probabilistic testing strategies:

- information about the SUT's structure is used as a guide to the possible distribution of faults;
- it is straightforward to generate—by repeated sampling from the input profile—test sets of sufficient size to exercise each structural element multiple times.

Unlike random testing, the input profile (the concrete strategy) for statistical testing must satisfy an adequacy criterion, and thus the cost of deriving the input profile can be prohibitive. In this section we describe the adequacy criterion used by statistical testing, and consider the costs of the techniques that implement it.

1.7.1 Adequacy Criterion

The property required of the input profile by statistical testing is that, with probability Q, all the structural elements are exercised by a test set of size N sampled from the profile. This property is the probabilistic equivalent of the adequacy criterion used by deterministic structural testing.

Since the test set is generated by random sampling, it is impossible to *guarantee* complete coverage; instead Q is a measure of the confidence that any one randomly-sampled test set exercises all the coverage elements. Q is usually set close to 1, and at such a high value, each structural element is typically exercised multiple times in a single test set.

For the research in this thesis, the criterion is more conveniently expressed in terms of the set $\{p_i\}$, where p_i is the probability that a *single* input sampled at random from the input profile exercises the structural element c_i one or more times. (The formulation of p_i used by Thévenod-Fosse and Waeselynck does not distinguish between inputs that exercise a given coverage element only once, and those that exercise the coverage element multiple times in a single execution of the SUT. We apply the same definition in this thesis.)

We introduce the term *minimum coverage probability* for the minimum value of the p_i, and denote it by the symbol p_{min}, such that:

$$p_{\text{min}} = \min_{c_i \in \mathcal{C}} \{p_i\} \tag{11}$$

where $\mathcal{C}$ is the set of structural elements.

The adequacy criterion for the input profile is now expressed as p_{min} must be greater than or equal to a predetermined threshold, $\tau_{p_{\text{min}}}$. The values of $\tau_{p_{\text{min}}}$, Q and N are related by the equation (Thévenod-Fosse and Waeselynck, 1991):

$$(1 - \tau_{p_{\text{min}}})^N = 1 - Q \tag{12}$$

1.7.2 Fault-Detecting Ability

Thévenod-Fosse and Waeselynck (1991) compared the fault-detecting ability of statistical testing to both deterministic structural testing and uniform random testing on four real-world SUTs. A range of coverage criteria were used for deterministic structural testing: for three of the SUTs, the strongest criterion used was path coverage; owing to its complexity, the strongest practical criterion for the fourth SUT was 'all uses'. Statistical testing used the same set of structural elements as the strongest deterministic criterion, and a test size and input profile were chosen so that $Q = 0.9999$. The test sets generated by uniform random testing had the same size as those generated by statistical testing.

The fault detecting-ability of each testing method was estimated using mutation analysis: an empirical application of the mutation testing strategy described in section 1.3. The test sets generated by each testing strategy were applied to a set of mutants containing seeded faults. The mutation score, the proportion of mutants identified as containing a fault by one or more test cases, was used as the metric to compare the testing techniques.

The mutation scores of deterministic structural testing did not exceed those of statistical testing for any SUT using any of the deterministic test adequacy criteria. Moreover, the authors found that different deterministic structural test sets derived from the same adequacy criterion demonstrated great variance in the mutation score, supporting a hypothesis that the fault-detecting ability of structural testing is highly dependent on the specific data chosen. Statistical testing demonstrated a more robust fault detecting-ability: analysis of the results showed that any mutant killed by statistical testing had a probability of at least 0.9 of being killed by any test set generated from the input profile.

Similarly, the mutation scores of uniform random testing never exceeded those of statistical testing, and for the more complex SUTs, random testing was significantly worse than statistical testing. Since both random and statistical testing used test sets of the same size, this supports that hypothesis that statistical testing is more efficient at detecting faults than random testing. In other words, the cost invested in deriving a suitable input profile for statistical testing is rewarded by a greater ability to detect faults.

The authors note that the mutants that were not discovered by any of the testing methods required the use of "extremal/special" input values, and therefore recommend that statistical testing is supplemented by the deterministic testing of such values.

1.7.3 A Manual Analytical Technique

For relatively simple SUTs, input profiles suitable for statistical testing may be derived by manual analysis.

To illustrate the technique, we apply a manual analysis similar to that taken by Thévenod-Fosse et al. (1995) to our example SUT, `daysInMonth`, using branches as the structural elements to be covered.

We first denote the probabilities of the predicates in conditional statements evaluating to TRUE as follows:

$$\begin{aligned}
\pi_0 &= \mathbb{P}\{m = 2\} \\
\pi_1 &= \mathbb{P}\{m \leq 7 \ \wedge m \neq 2\} \\
\pi_2 &= \mathbb{P}\{y \bmod 4 = 0 \wedge y \bmod 100 \neq 0\} \qquad (13)
\end{aligned}$$

Let p_i be the probability of exercising edge e_i. Then, by reference to the SUT's control flow graph (figure 2), we obtain the following equalities:

$$\begin{aligned}
p_1 &= 1 - p_2 = 1 - (\pi_0 + \pi_1) \\
p_2 &= \pi_0 + \pi_1 \\
p_5 &= 1 - p_6 = 1 - \pi_0 \\
p_6 &= \pi_0 \\
p_7 &= p_6 - p_8 = \pi_0 \times (1 - \pi_2) \\
p_8 &= \pi_0 \times \pi_2
\end{aligned} \tag{14}$$

(And, trivially, $p_0 = p_{11} = 1$, $p_3 = p_1$, $p_4 = p_2$, $p_9 = p_7$, $p_{10} = p_8$.)

By inspection of control flow graph, the maximum value that can be taken by p_{min} is $\frac{1}{3}$ since the edges e_5, e_7 and e_8 are mutually exclusive. Using the equalities of equation (14), we derive the following values of the π_j for such a value of p_{min}:

$$\begin{aligned}
\pi_0 &= \tfrac{2}{3} \\
\pi_1 &= 0 \\
\pi_2 &= \tfrac{1}{2}
\end{aligned} \tag{15}$$

Taking into account the cardinality of the subsets that satisfy the predicates in equation (13), and assuming that the inputs already satisfy the input domain constraints, a probability distribution (but not the only one) that induces these values for the π_j is:

$$\begin{aligned}
\mathbb{P}(m = M) &= \begin{cases} \frac{2}{3} & \text{if } M = 2 \\ \frac{1}{15} & \text{if } M = 8, 9, 10, 11, 12 \\ 0 & \text{otherwise} \end{cases} \\
\mathbb{P}(y = Y) &= \begin{cases} \frac{1}{296} & \text{if } Y \bmod 100 \in \{4, 8, 12, \ldots, 96\} \\ \frac{1}{14512} & \text{otherwise} \end{cases}
\end{aligned} \tag{16}$$

and therefore, since the distributions of m and y are independent,

$$\mathbb{P}\{(m, y) = (M, Y)\} = \mathbb{P}\{m = M\}\, \mathbb{P}\{y = Y\} \tag{17}$$

(In general, a joint probability distribution suitable for statistical testing is not separable into independent marginal distributions for each input variable as it is here.)

The resultant histogram of the edge probabilities using this input profile is shown in figure 7. The minimum coverage probability, p_{min}, is the height of the shortest bar in the histogram.

Figure 6, the equivalent histogram for uniform random testing, showed that the edges e_7 and e_8 were rarely exercised by the uniform profile. Figure 7 demonstrates that these edges would be exercised much more frequently by the input profile derived for statistical testing, and so faults in code reached by these edges are more likely to be detected.

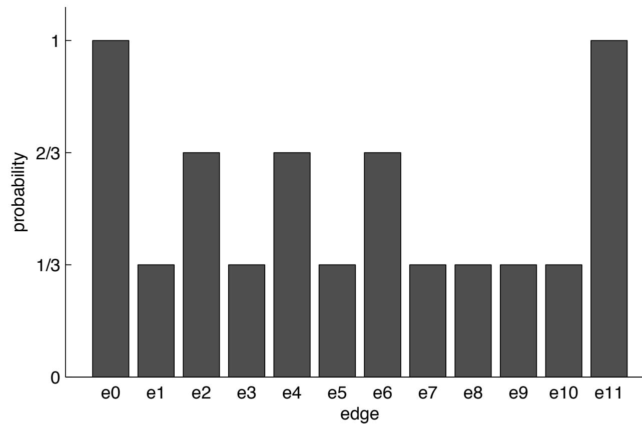


Figure 7 – Histogram showing the probabilities of edges in the control flow graph of `daysInMonth` being exercised by an input sampled from an optimal statistical testing profile.

1.7.4 An Automated Analytical Technique

Gouraud and Denise describe an automated analytical technique which can be applied when the structural elements are paths (Gouraud et al., 2001; Denise et al., 2004; Gouraud, 1995). Outgoing edges in the SUT's control flow graph are assigned weights in proportion to the number of paths to the terminal node from the node reached by the edge. If paths in the control flow graph are chosen by selecting edges at branching nodes according to these assigned weights (ignoring semantic feasibility), each path has the same chance of being selected. This distribution over the paths is therefore optimal in terms of the statistical testing adequacy criterion: all probabilities p_i have the same value.

There are, however, two practical limitation of this technique. Firstly, having chosen a path at random, constraint solving is used to generate an input that exercises the path. As for symbolic execution (section 1.4.3), constraint solving scales poorly to larger and more complex SUTs. Secondly, since semantic feasibility is ignored when sampling a path at random, many of the sampled paths are infeasible. Significant effort may be wasted because the infeasibility of a path is only detected during the process of solving the path constraints. To rectify this problem, Baskiotis et al. (2007) propose a machine learning approach to dynamically refine the distribution over the paths based on the feasibility of sampled paths.

1.7.5 A Manual Dynamic Technique

Thévenod-Fosse and Waeselynck describe a manual technique for deriving input profiles when the SUT is too complex for the use of analytical techniques (Thévenod-Fosse and Waeselynck, 1993; Thévenod-Fosse et al., 1995).

The technique assesses the coverage of candidate input profiles by executing an instrumented version of the SUT using a large number of inputs sampled from the profile. Starting with the uniform distribution, the empirical coverage data guides the manual adjustment of the profile in order to create a potentially better candidate profile. The process is repeated until the minimum coverage probability of the candidate profile is considered "sufficiently high".

This technique was found to be feasible only for adequacy criteria that, in terms of the 'subsume' ordering of Zhu et al., were relatively weak. For example, in the empirical work of Thévenod-Fosse et al. (1995), the criterion was statement coverage—which is subsumed by

all other adequacy criteria shown in figure 3—and the fault-detecting ability of the generated test sets was therefore relatively poor. In (Thévenod-Fosse and Waeselynck, 1993), the authors suggest that there is an upper limit on the size SUT for which this empirical technique is tractable using even the weakest of coverage criteria.

1.8 Research Hypothesis

The empirical investigations of Thévenod-Fosse and Waeselynck suggest that the strategy of statistical testing can be highly effective at detecting faults. But deriving a SUT-specific concrete strategy—a suitable probability distribution over the input domain—for statistical testing is challenging: existing analytical and dynamic techniques are only practical for relatively simple SUTs and for relatively weak adequacy criteria. For this reason, statistical testing is unlikely to be a cost-effective strategy for many SUTs using these techniques.

We note, however, that the task of deriving concrete strategies for statistical testing has the characteristics identified by Clark et al. (2003) as indicative of problems that could be solved using search-based software engineering (section 1.4.4):

- It is difficult to *construct* a concrete strategy—an input profile satisfying the adequacy criterion—but easy to *check* that a given input profile satisfies the criterion by executing the instrumented SUT with inputs sampled from the profile.
- It is not necessary to derive the *optimal* strategy: an input profile with the highest possible minimum coverage probability. An *adequate* strategy—a profile with a high (not necessarily optimal) minimum coverage probability—may be sufficient for efficient testing.

We therefore propose that concrete strategies for statistical testing could be derived using a search-based technique. Moreover, we believe that use of automated search would enable strategies to be derived at lower cost, and for a wider range of SUTs, than existing techniques.

Thus our research hypothesis is:

> *Concrete strategies for statistical testing can be derived cost-effectively for a wide range of software using automated computational search.*

Such a cost-effective search-based technique would have two major impacts. From a practical perspective, the superior fault-detecting ability of statistical testing could be applied, at acceptable cost, to many more SUTs than it is currently. From an academic perspective, it would raise the abstraction level at which search-based software testing is applied from the level of individual test inputs to the level of concrete strategies from which entire test sets are generated.

1.9 Thesis Structure

The remainder of this thesis addresses the research hypothesis as follows.

Viability In chapter 2, we demonstrate a computational search algorithm for deriving concrete strategies for statistical testing. The algorithm represents candidate input profiles as a Bayesian network, and uses hill climbing to search for suitable strategies. The algorithm is shown to be viable for SUTs of realistic size and complexity, and that test sets generated from the derived strategies possess the superior fault-detecting ability observed by Thévenod-Fosse and Waeselynck.

Efficiency The algorithm evaluated in chapter 2 requires substantial computing resources. This limits the cost-effectiveness of the testing strategy and the scalability of the algorithm to larger SUTs. In chapter 3, novel enhancements are proposed to the algorithm with the objective of improving its efficiency and thereby reducing the computing resources required. The outcomes are an improved algorithm that uses, on average, one-eighth of the computing resources of the original, and practical guidance as to the setting of algorithm parameters.

Applicability The Bayesian network representation used in chapters 2 and 3 is most easily applied to SUTs whose inputs are a fixed number of numeric arguments. In chapter 4, we propose a new representation based on stochastic context-free grammars that is able to represent input profiles for a much wide range of SUTs. The wider applicability of the grammar-based representation is demonstrated using two SUTs that take highly-structured inputs of varying length. Moreover, we show that the search algorithm using the grammar-based representation retains much of the efficiency exhibited by the Bayesian network representation.

Parallelisation We propose that parallelising the search algorithm may further improve its performance. In chapter 5, low-cost commodity GPU cards are used as a highly-parallel computing environment for this purpose. We demonstrate the parallel execution of both the search algorithm and instrumented SUTs on a GPU, and show that for SUTs with long execution times, concrete strategies may derived much more quickly using on a GPU than on a CPU.

Conclusions In chapter 6, we evaluate the research hypothesis and propose opportunities for further research.

Chapter 2

Demonstration of a Viable Algorithm

2.1 Introduction

2.1.1 Motivation

In this chapter we show that concrete testing strategies for statistical testing may be derived automatically using computational search. The concrete strategies are input profiles—probability distributions over the input domain of the software-under-test (SUT)—that induce a suitably high minimum coverage probability over the structural elements of the SUT.

We propose a search algorithm for this purpose, and demonstrate empirically that the proposed algorithm is *capable* of deriving suitable input profiles and is *viable* in terms of the computing resources required and the time taken.

2.1.2 Contributions

The contributions in this chapter are:

- a computational search algorithm for deriving input profiles that are suitable for statistical testing;
- empirical evidence of the viability of the proposed algorithm for SUTs of realistic size and complexity;
- empirical evidence that input profiles derived by the algorithm demonstrate the superior fault-detecting ability observed by Thévenod-Fosse and Waeselynck when using manually-derived profiles;
- empirical evidence that by making the 'diversity' of the input profile an additional search objective, the fault-detecting ability of the derived input profiles can be improved.

2.1.3 Chapter Outline

The search algorithm is proposed in section 2.2, and its implementation discussed in section 2.3.

The empirical work in this chapter considers three SUTs with contrasting features; their provenance and relevant characteristics are discussed in section 2.4.

Experiment I (section 2.5) demonstrates that the proposed algorithm is viable: it is able to derive input profiles with a suitable minimum coverage probability for all three SUTs, and has an acceptable performance when using computing resources typical of a desktop PC.

Experiment II (section 2.6) confirms that for two of the SUTs, input profiles derived automatically by the algorithm demonstrate the superior fault-detecting ability previously shown for manually-derived profiles. However for one SUT, the automatically derived profiles detect fewer faults than uniform random profiles at large test sizes.

Experiment III (section 2.7) explores this unexpected result in more detail by evaluating how the fault-detecting ability of candidate profiles changes over the course of the search.

In section 2.8, we hypothesise that the poor fault-detecting ability observed during Experiments II and III is caused by a loss of 'diversity' in the derived profiles, and propose a modification to the fitness metric with the objective of retaining this diversity.

Experiment IV (section 2.9) demonstrates that the modified fitness metric *does* improve the fault-detecting ability of derived profiles, but that the uniform random profile nevertheless remains superior for one SUT.

In section 2.10, we reflect on the outcomes of the research described in this chapter and identify issues that motivate the research in subsequent chapters of this thesis.

2.2 Proposed Algorithm

The three main components of a computational search algorithm are a representation for candidate solutions, a fitness metric for comparing the suitability of candidate solutions, and a search method that derives new candidate solutions using information gained from the solutions considered to date. In this section, we describe the design of these components for our proposed algorithm.

The objective of the research described in this chapter is to demonstrate that it is *viable* to use computational search to derive input profiles; it is not to demonstrate that the proposed algorithm is the *most* efficient form of computational search for this purpose. For this reason, the proposed designs for the representation, fitness metric, and search method used in this chapter are the most 'straightforward' that meet the objective of viability.

2.2.1 Representation

Assumptions The input domain of a SUT is potentially very complicated. At its simplest, the input domain may be defined by the explicit arguments to the software such as the values passed to a function undergoing unit testing, but even in this simple case the arguments may have complex data types such as strings, pointers, or objects. If the SUT has internal state that affects its function—for example, the SUT is a class with member data fields—the internal state is a component of the input domain. Similarly, if the SUT interacts with a wider system that has state—for example, reads from files or calls other stateful system modules—the system state may need to be considered as part of the input domain.

A simplifying assumption is made in this chapter for the purpose of demonstrating viability of the the proposed algorithm: that the input to the SUT is a fixed number of numeric values: $a_0, a_1, \ldots, a_{k-1}$, and that the domain of individual argument a_i may be defined using a finite interval. An obvious example of a SUT satisfying this assumption is a stateless function with a fixed number of integer or floating point arguments, and for this reason, the individual values a_i are described as input *arguments* throughout this chapter. (In chapter 4, the representation is extended in order to relax this assumption.)

The Need to Represent a Joint Probability Distribution The input arguments $a_0, a_1, \ldots, a_{k-1}$ cannot be assumed to be independent.

As an example, consider the 'toy' SUT, `simpleFunc`, shown in figure 8, and its control flow graph figure 9. Let us assume that the adequacy criterion for statistical testing is defined in terms of branch coverage; in this context an 'optimal' input profile is one that has the highest possible minimum coverage probability across the set of branches.

The SUT has six branches leading from conditional statements: the edges labelled e_0 to e_5 in the control flow graph. Since the branches e_0 and e_1 are executed at least as frequently as the branches e_2 to e_5, we need only consider the latter four branches when deriving an input profile. The four branches e_2 to e_5 are mutually exclusive and so an optimal (in the sense defined above) input profile will be one for which that probability of exercising each branch is the same.

The pair of branches e_2 and e_3 are exercised when $1 \leq a \leq 5$, and the other pair e_4 and e_5 only when $6 \leq a \leq 50$. Thus a probability distribution over a that enables an optimal input

```
1  /* 1 <= a <= 50, 1 <= b <= 20 */
2  int simpleFunc(int a, int b) {
3    int r;
4    if (a<=5) {
5      if (b>=18) {
6        r = abs(b-19);
7      else
8        r = b;
9    } else {
10     if (b<=3)
11       r = abs(b-2);
12     else
13       r = 10+b;
14     }
15   return r;
16 }
```

Figure 8 – The source code of the toy SUT `simpleFunc`.

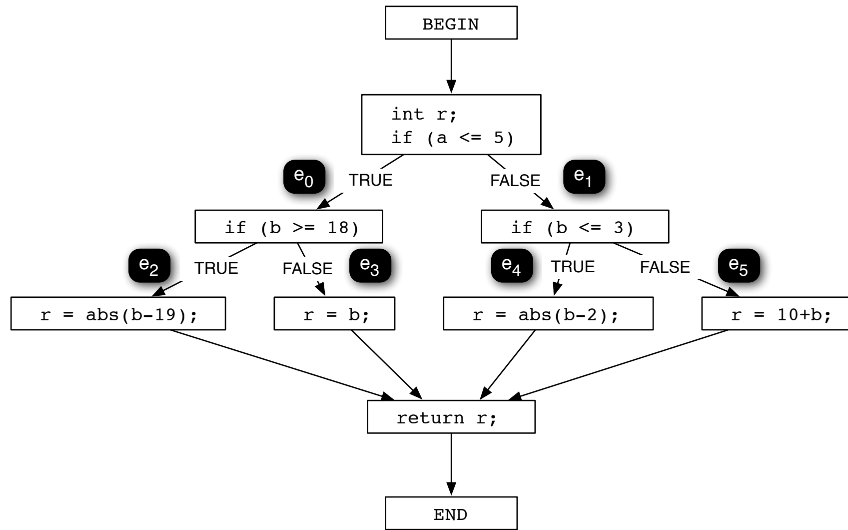


Figure 9 – Control flow graph of `simpleFunc`. Labels e_i identify edges leading from conditional statements.

profile is:

$$\mathbb{P}(a) = \begin{cases} 1/10 & \text{if } 1 \leq a \leq 5 \\ 1/90 & \text{if } 6 \leq a \leq 50 \end{cases} \tag{18}$$

(This is not the only distribution that is optimal, but in some sense is the most 'natural': it partitions the domain of a into two subsets and each integer value within a subset has the same probability.)

When $1 \leq a \leq 5$, the distribution over b that exercises e_2 with the same probability as e_3 is:

$$\mathbb{P}(b) = \begin{cases} 1/34 & \text{if } 1 \leq b \leq 17 \\ 1/6 & \text{if } 18 \leq b \leq 20 \end{cases} \tag{19}$$

However when $6 \leq a \leq 50$, the above distribution for b is far from optimal: the branch e_5 would be exercised much more frequently than e_4. An optimal probability distribution over

b for this second pair of branches is instead:

$$\mathbb{P}(b) = \begin{cases} 1/6 & \text{if } 1 \leq b \leq 3 \\ 1/34 & \text{if } 4 \leq b \leq 20 \end{cases} \tag{20}$$

Since two different probability distributions for b are required—the choice between the two conditional on the value taken by a—`simpleFunc` demonstrates that it is not generally possible to represent an optimal input profile as independent marginal probability distributions over each of the input arguments. Instead, a joint probability distribution is required, such as the following for `simpleFunc`, formed by combining equations (18), (19), and (20):

$$\mathbb{P}(a,b) = \begin{cases} 1/340 & \text{if } (1 \leq a \leq 5) \wedge (1 \leq b \leq 17) \\ 1/60 & \text{if } (1 \leq a \leq 5) \wedge (18 \leq b \leq 20) \\ 1/540 & \text{if } (6 \leq a \leq 50) \wedge (1 \leq b \leq 3) \\ 1/3060 & \text{if } (6 \leq a \leq 50) \wedge (4 \leq b \leq 20) \end{cases} \tag{21}$$

Dependency Representation We propose to use a Bayesian network to represent the dependency between input arguments. A Bayesian network is a directed acyclic graph. Here the nodes in the graph are the input arguments, and a directed edge from a parent to child node indicates that the probability distribution of the child input argument is conditionally dependent on the value taken by the parent input argument.

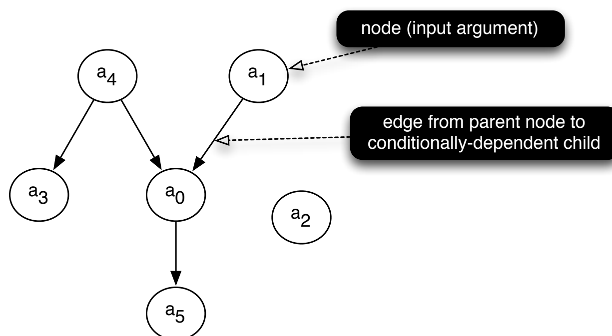


Figure 10 – An example of Bayesian network for six input arguments.

Figure 10 is an example of a Bayesian network for six input arguments: argument a_0, for example, is conditionally dependent on arguments a_1 and a_4, while arguments a_4, a_1, and a_2 are conditionally independent of the other arguments.

The Bayesian network decomposes the joint probability distribution into a product of the conditional probability distributions at each argument. Let the k input arguments be $a_0, a_1, \ldots, a_{k-1}$; and the set $\mathcal{P}_{a_i}$ be the parents of a_i in the network; then the joint probability distribution is:

$$\mathbb{P}(a_0, a_1, \ldots, a_{k-1}) = \prod_{i=0}^{k-1} \mathbb{P}(a_i | \mathcal{P}_{a_i}) \tag{22}$$

We use this representation for two reasons. Firstly, a Bayesian network is capable of representing any joint probability distribution over the input arguments (Korb and Nicholson, 2011), and thus this representation does not restrict the type of input profiles that could be synthesised by the algorithm. Secondly, the representation is concise in the sense that it only

represents conditionally between arguments where this is necessary: it is more concise than a representation that assumes, for example, that there is always a dependency relationship between all input arguments. We intend that this conciseness will facilitate the search process since the size of the representation is no larger than is required, and it avoids unnecessarily large data structures in the implementation of the algorithm.

Conditional Probability Representation To represent the conditional probability distribution of each input argument—the terms $\mathbb{P}(a_i|\mathcal{P}_{a_i})$ in equation (22)—the argument's domain is partitioned into a set of *bins*. Each bin is itself an interval; the number and length of the bins are variable, and may differ between arguments.

Conditional probabilities are represented in terms of bins, rather than the values taken by the input argument. If the input argument is conditionally *independent*, then each bin has an associated probability, and the set of bin probabilities forms a probability distribution. If the input argument is conditionally *dependent* on one or more parents, then there are many such probability distributions: one for each combination of bins in the parents.

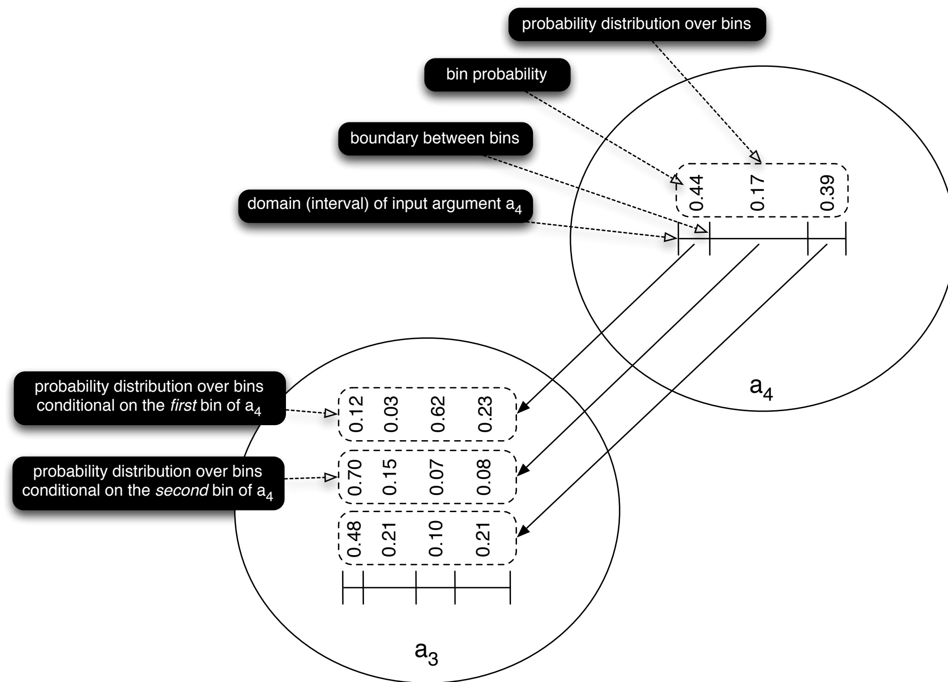


Figure 11 – An example of conditional distributions represented in terms of bins for two input arguments. Argument a_3 is conditionally dependent on a_4.

The representation of conditional distributions in terms of bins is illustrated by the example in figure 11 which shows input arguments a_4 and a_3 from the Bayesian network of figure 10. The domain for argument a_4—which, by the assumption above, is an interval—is shown as a horizontal line partitioned into three bins. Since a_4 is conditionally independent, there is only one distribution at the node, i.e. one set of bin probabilities. Argument a_3 has four bins, but since it is conditionally dependent on a_4, has three different sets of bin probabilities: one corresponding to each bin of a_4. The length of the bins is the same for all three distributions over a_3, but the bin probabilities differ.

When sampling from the binned Bayesian network, conditionally independent input arguments are considered first. For each of these independent arguments, a bin is selected

at random according to the probability distribution. Next, arguments that are conditionally dependent on parents for which bins have already been selected are considered. Bins are selected at random for these dependent arguments according to the conditional probability distributions corresponding to the bins selected for the parents. This process is repeated until a bin has been selected for every argument. Finally, a specific argument value is chosen at random from each of the selected bins using a uniform distribution across the bin's interval. (The use of a uniform distribution is arbitrary, but following the design principle discussed above, it is a 'straightforward' choice.)

The representation of the probability distribution in terms of bins, rather than argument values, again encourages conciseness in the representation. Consider the case where the valid domains of both a_3 and a_4 are $2^{16} = 65\,536$ integer values: to represent the conditional distribution of a_3 directly in terms of values rather than bins would require $2^{16}.2^{16} = 2^{32} = 4\,294\,967\,296$ probability values. This would require 16 GB of memory should the probabilities be stored as 4-byte single-precision floats. The use of bins also overcomes the issue of representing distributions over real-numbered domains for which the cardinality is infinite in theory, and very large in practice when using floating-point data types.

We hypothesise that the use of bins may enable a particularly concise representation in the case of SUTs where neighbouring points in the input domain typically exercise the same coverage elements; in other words, when the subsets of the input domain exercising each coverage element are contiguous regions. In this situation, the search may be able to align the bins closely to the element subsets. However the representation can also accommodate input domains in which the subsets do not form contiguous regions by using multiple bins for each element subset. One of the SUTs used in the empirical work is chosen to investigate this latter case.

We refer to the combination of Bayesian network to represent dependency, and bins to the represent marginal distributions, as a *binned Bayesian network* (BBN).

2.2.2 Fitness Metric

Statistical testing specifies an adequacy criterion for the input profile: that its minimum coverage probability $p_{\min}$ must be at least a predetermined threshold, $\tau_{p_{\min}}$. This adequacy criterion is discussed in section 1.7.1 where $p_{\min}$ is defined as:

$$p_{\min} = \min_{c_i \in \mathcal{C}} \{p_i\} \tag{23}$$

where $\mathcal{C}$ is the set of coverage elements, and p_i is the probability that the coverage element c_i is exercised (one or more times) by a single input sampled at random from the profile.

The *fitness metric* is a value calculated for each candidate solution generated during a search process and is used to guide the search towards the objective, in this case a suitably high value of $p_{\min}$. If the search objective is quantifiable—as it is here—the objective may itself be used as the fitness metric, but this need not be the case: a different function may be better at guiding the search to the same objective. However, using the design principle above, the most straightforward choice for the proposed algorithm *is* to use the objective, $p_{\min}$, as the fitness metric.

For the simplest SUTs and weakest coverage criteria, it *may* be possible to analytically calculate the minimum coverage probability for a candidate input profile using static analysis using a process similar to that illustrated in section 1.7.3. We instead propose a more general

approach that *estimates* the minimum coverage probability by executing the SUT. A set, size K, of inputs is sampled from the candidate profile and used to execute an instrumented version of the SUT. The instrumentation records which of the structural elements, c_i, are exercised when the SUT is run with each of the sampled inputs. The variable e_i is the count of the number of sampled inputs for which c_i is exercised one or more times. The estimate of $p_{\min}$, and the fitness metric used by the proposed algorithm is:

$$\widehat{p}_{\min} = \frac{1}{K} \min_{c_i \in C} \{e_i\} \tag{24}$$

Since this estimate is calculated from a finite set of inputs sampled from the candidate profile, the metric is noisy in the sense that is only an approximation for the minimum coverage probability; calculating the metric for the same candidate profile a second time, using a new sample of inputs, will typically result in a slightly different value. An inaccurate estimation of the minimum coverage probability could not only mislead the search, but also cause the actual minimum coverage probability of profiles synthesised by the algorithm to be much lower than desired if the fitness of an inadequate candidate profile included, by chance, sufficient noise to take it above the target value and thus satisfy the algorithm's termination condition. The noise could be reduced by using a larger sample of inputs (a higher value of K), but since each execution of the SUT is likely to be expensive in terms of both time and computing resources, there is trade-off between the accuracy of the fitness and the resources required to evaluate the fitness.

2.2.3 Search Method

Hill Climbing Hill climbing is a local search method—it considers only the local neighbourhood of the candidate solution at each iteration—that moves to a solution in this neighbourhood only if the solution is better than the current candidate solution. Although very straightforward, its major disadvantage is that by considering only the local neighbourhood, it is unable to escape local optima in the search landscape; more sophisticated search methods such as simulated annealing and genetic algorithms are designed to avoid this problem. It might be expected that noise in the fitness metric of the proposed algorithm could mislead hill climbing by creating artificial local optima in the landscape should a candidate profile have an estimated minimum coverage probability that is much larger than its true value.

However, preliminary experimentation demonstrated that this was not necessarily the case: hill-climbing was able to synthesise input profiles despite noise in the fitness metric. (We describe later in this section a technique of 're-evaluating' the fitness of candidate profiles that is designed to minimise the impact of noise on the search method, and this might account for the unexpected efficacy of hill climbing.) Hill climbing has fewer parameters than more sophisticated search methods, and does not require the a crossover operator—as would a genetic algorithm—which would be difficult to define and implement for the Bayesian network representation. Thus, invoking the design principle of a 'straightforward' algorithm, hill-climbing is chosen as the search method.

In *steepest-ascent hill climbing*, all neighbours of the current candidate solution (in this case, an input profile) are evaluated at each iteration of the climb and the neighbour giving the best improvement in fitness replaces the current solution. Such an algorithm would find the (locally) optimum solution in the least number of iterations compared to other hill climbing methods, but since the neighbourhood of an input profile can be very large (see the discussion

of the neighbourhood defined by mutation below), each iteration of the climb would require very many fitness evaluations. Each fitness evaluation itself requires multiple executions of the instrumented SUT, and therefore an algorithm using steepest-ascent hill climbing is likely to perform poorly in terms of both time and computing resources.

Instead, the form of hill climbing chosen for the algorithm selects a small sample, size λ, of neighbours of the current solution at random and evaluates their fitnesses. If the best of these neighbours has a fitness that is greater than *or equal* to the current solution, it replaces the current solution. (The justification for moving to the best neighbour when the fitness is *equal* to the current solution is to enable the search to escape neutral plateaux in the landscape, i.e. regions where all solutions have the same fitness, by continually moving to new solutions.) This form of hill climbing is similar to *random mutation hill climbing* described by Forrest and Mitchell (1993), but instead of evaluating just one random mutation at each iteration, a small number of random mutations are considered.

Initialisation The hill climb always starts from the same candidate input profile: one with no edges in the Bayesian network, and one bin for each input argument. Since there are no edges, each argument is conditionally independent. Since there is only one bin for each argument, the probability distribution for that argument is uniform. Thus this initial profile is a joint uniform probability distribution over the input domain.

This design choice is again the most straightforward: an alternative would be to start the climb from a randomly-selected input profile, but this would require a suitable distribution over the set of possible input profiles to be defined for this purpose and it is not obvious how to define such a distribution nor what form it should take. In addition, the initial uniform profile is 'diverse' in the sense that for the subset consisting of inputs that exercise a given coverage element, all inputs have the same probability of being sampled from the profile. As discussed later in the chapter, such diversity is important for detecting faults; we intend that by starting the hill climb from an already diverse input profile, diversity is encouraged in the profile derived by the search.

Mutation At each iteration of the hill climb, a small number of immediate neighbours of the current candidate input profile are evaluated. This neighbourhood is defined as the profiles created by a single application of one of the following mutation operators to the current profile.

M_{prb} increases or decreases one bin probability by multiplying or dividing it by a factor ρ_{prb}. For a conditionally dependent input argument, each conditional probability associated with a bin is considered a separate value that is mutated independently of the others: i.e. only one such value is mutated by a single application of the operator. After mutation, the probabilities in the distribution are normalised so that they sum to one. The factor ρ_{prb} is a parameter to the algorithm.

M_{len} increases or decreases one bin length by multiplying or dividing it by a factor ρ_{len}. After mutation, the bin lengths are normalised so that the total length of the bins are equal to the length of the valid interval of the argument. The factor ρ_{len} is a parameter to the algorithm.

M_{spl} splits a bin into two new bins, each half the length of the original. The new bins have probabilities that are half that of the original bin, and the conditional probability

distributions at dependent child input arguments for the new bins are the same as the original bin.

M_{joi} joins two adjacent bins to form a single bin with a length equal to the sum of the two original bins' lengths. The new bin has probabilities that are the sum of the two original bins' probabilities, and the conditional probability distributions at dependent child input arguments are formed by averaging the corresponding probabilities in the distributions associated with the two original bins.

M_{add} adds an edge between two nodes (i.e. input arguments) in the Bayesian network. The new edge is not permitted if it were to create a cycle in the network graph. Each of the new conditional distributions at the child node is the same as the previous unconditional distribution at the node.

M_{rem} removes an edge between two nodes in the Bayesian network. The new probability distribution at the child node is formed by averaging the previous conditional distributions at the node.

For each mutation operator, there are normally many different *atomic mutations* that could be made to create a neighbour profile. For example, if there are n_{prb} probability values in the the current profile, n_{prb} different atomic mutations could be made by the M_{prb} operator. We choose to make only one of them—to a probability value chosen at random—during a single application of the operator. (This, and many of the other operator design decisions, are revisited in chapter 3.)

This set of mutation operators affect all aspects of the BBN representation: the structure of Bayesian network (M_{add} and M_{rem}), the number and length of bins (M_{spl}, M_{joi}, and M_{len}), and the conditional probability distributions (M_{prb}).

The mutation operators were designed with two principles in mind. Firstly, each operator has a corresponding one that reverses it: an increase in bin probability has a corresponding decrease by the same factor; a split of a bin has a corresponding join; an addition of an edge has a corresponding removal. (An exact reversal is not always possible. For example, a join of two bins results in a summing of the lengths, while the corresponding split would halve the lengths: the new bins would not necessarily correspond with the original bins prior to the join.) The intention is that the search method is theoretically able to 'back out' of any change and thus avoids becoming trapped in a region of the search space. (Normally such back-tracking is not possible in a hill climbing algorithm unless the region is a neutral plateau, i.e. locally all solutions have the same fitness. However, back-tracking to solutions of lower fitness *can* occur in this algorithm as a consequence of the noise in the fitness metric. This behaviour is discussed further below.)

Secondly, since hill-climbing is a local search method, the mutation operators were chosen so that they, where possible, make relatively small changes to the representation and thus enable exploration of local neighbourhood of the current candidate input profile:

- The operators M_{add} and M_{rem} make the smallest possible change to the Bayesian network structure—the addition or removal of only one edge—but these are nonetheless unavoidably large changes to the *genotypic* representation: the probability distribution at the child node has a new structure and the overall size of the representation is modified. However, the *phenotypic* impact of these operators is kept small by ensuring that the mutations are as neutral as possible in terms of fitness. For example, on adding

an edge to the network, the conditional distributions at the child node are all the same as the original unconditional distribution and so the joint distribution over the input domain is unchanged. The intention is that the benefit (or otherwise) of such changes to the structure of the Bayesian network can be explored gradually by *subsequent* mutations to the bin probabilities.

- The operators M_{spl} and M_{joi} make local changes to a single bin, or a pair of neighbouring bins, respectively, of a single node in the network. Although this is a relatively small genotypic change, these operators are also designed to be neutral in terms of fitness. For example, on splitting a bin, the lengths and probabilities of the two new bins are both one half of those of the original bin, resulting in no overall change to the distribution.
- The operators M_{prb} and M_{len} make a change to the probability or length of one bin in a single node in the network. However, the renormalisation of the probability distribution at the node after applying M_{prb}, or of the lengths of the bins after M_{len}, will modify all bins of the node. For M_{len} in particular, this may have a relatively large impact on fitness as the renormalisation changes the boundaries of all bins for the input argument represented by the node. (Alternative mutation operators might avoid this problem: for example, an operator that moves a bin boundary directly—rather than via changes to bin lengths—so that only the two bins sharing the changed boundary are affected.)

Each mutation operator, M_{x} has an associated mutation weight w_{x}. When sampling a random neighbour of the current candidate input profile, the probability of making any one atomic mutation (in the sense discussed above) is proportional to this weight. Thus the probability of the operator M_{x} being applied when creating a neighbour is proportional to $n_{\text{x}} w_{\text{x}}$ where n_{x} is the number of different atomic mutations the operator M_{x} could make to the current profile. These mutation weights are parameters to the algorithm.

By choosing $w_{\text{rem}} > w_{\text{add}}$ and $w_{\text{joi}} > w_{\text{spl}}$, parsimony is encouraged by preferring mutations that decrease the total number of probability values in the representation (removing edges and joining bins) to those that increase it (adding edges and splitting bins). Our intention is that encouraging parsimony not only avoids large memory structures in the implementation, but may also increases the efficiency of the search in terms of the number of mutations required to derive a suitable input profile.

There is little advantage in including the same neighbour more than once in the sample of the neighbourhood evaluated at each iteration of the hill climb. It is easy to identify when the same atomic mutation has been applied more than once in the sample to create identical neighbours, and this enables the neighbourhood to be sampled *without replacement*.

Fitness Re-evaluation The fitness metric is noisy as a result of sampling a finite number of inputs from the candidate profile. As a result, it is possible that the hill climb will sometimes move to a neighbouring profile for which the fitness metric has been overestimated, i.e. the estimate of the minimum coverage probability is much higher than the true value. This behaviour could mislead the search, and in the worst case could create an artificial local optimum from which the hill climb finds it difficult to escape.

A solution to this problem would be to estimate the minimum coverage probability using a very large number of inputs sampled from the candidate profile in order to reduce the degree of noise in this estimate. However, a large sample could increase the computing resources required by the search.

We propose instead that the current candidate input profile is re-evaluated at each iteration. If the fitness metric were *not* noisy, there would be no advantage in evaluating the fitness of the same input profile more than once. Here, however, multiple evaluations of the same input profile enable a more accurate estimate of its minimum coverage probability. Therefore, after a neighbour profile has been evaluated once and selected as the current profile, it continues to be evaluated at each iteration until it is replaced by a fitter neighbour. The fitness is calculated from the instrumentation data accumulated across all of its evaluations: during earlier search iterations as well as the current iteration. In this way, an initially overestimated fitness is likely to converge to a value closer to the true minimum coverage probability at subsequent iterations.

We hypothesise that the combination of a noisy fitness metric and this method of re-evaluation enables the hill climbing search method to move to solutions of lower fitness. Consider the situation when the current solution, X, is a local optimum: none of its neighbours has a better minimum coverage probability. In standard hill-climbing, moves are only possible to solutions with higher fitness and so the search will not escape from the local optimum X. However, in our proposed algorithm the *estimated* fitness of one the neighbours, Y, could, by chance, be evaluated as better than X as a result of noise in the fitness metric. Solution Y becomes the current solution, but over the next few search iterations the process of re-evaluation enables a more accurate prediction of its minimum coverage probability, and this value is lower than X. In this way the hill climb has escaped a local optimum at X. Although we do not perform an experiment that specifically explores this hypothesis, individual runs of the algorithm do indeed show the trajectory of the search occasionally moving to solutions of lower fitness.

Termination The algorithm terminates when one or more of the following conditions is met.

- The fitness of the current solution reaches or exceeds a target value for the minimum coverage probability, $\tau_{p_{\text{min}}}$. Termination as a result of this condition is a success for the algorithm: a suitable input profile has been derived. We note, however, that since the fitness is a noisy *estimate* of the minimum coverage probability, it is possible that the *true* minimum coverage probability of the derived profile may be less than target.

- The number of hill-climbing iterations taken exceeds a threshold, τ_{iter}. This enables the algorithm to be used to find the best possible solution using a fixed amount of computing resource and, particularly for the purposes of experimentation, avoids excessively long-running instances of the algorithm.

Both $\tau_{p_{\text{min}}}$ and τ_{iter} are parameters to the algorithm.

As discussed above, the hill climb may move temporarily towards solutions of lower fitness. Should the algorithm terminate owing to the second condition, the current input profile at the point of termination is not necessarily the best profile found during the search. Therefore a record is maintained of the fittest current candidate input profile at any iteration of the hill climb, and this 'elite' profile, rather than the current profile at termination, is reported by the algorithm.

2.3 *Implementation*

This section discusses relevant details of the implementation of the algorithm used for the empirical work in this chapter. The focus is on aspects of the implementation that might affect the performance of the algorithm; that are relevant to reproducing the empirical work; and that support the validity of the empirical work.

2.3.1 *Language*

The algorithm is implemented in C++. This decision is based on potential performance: an implementation in C or C++ is likely to be faster than one using, for example, Java. C++ is chosen over C since the container classes and algorithms in the C++ Standard Template Library facilitate the implementation of the variable-length binned Bayesian network representation.

2.3.2 *Search Method*

The implementation of the search method is bespoke. While it would have been possible to use a toolkit or library for this purpose, the core hill climbing method is very simple to implement and regardless of whether a toolkit were used or not, the majority of effort would have been expended on implementing the custom representation, the fitness evaluation, and the mutation operators.

2.3.3 *Argument Representation*

The implementation supports both integer and floating point input arguments in the binned Bayesian network. Internally both types of arguments are represented as floating point values in the semi-open interval $[0, 1)$, and are converted to the actual intervals of the input arguments by a suitable linear function. For example, if the actual interval of the input argument is the closed interval $[l, u]$ and the data type of the argument is integer, then the linear function is:

$$a = l + \lfloor (u - l + 1)a^* \rfloor \tag{25}$$

where a and a^* are the actual and internal argument values, respectively. This generic internal representation enables the implementation of initialisation and mutation operators that are independent of the actual data type of the argument.

2.3.4 *SUT Execution*

An instrumented version of SUT must be executed many times by the algorithm in order to assess the fitness of candidate input profiles. The instrumented SUT can be a standalone executable or linked with the algorithm executable itself. A standalone executable has fewer restrictions on the language of the SUT: it need only accept its arguments on the standard input stream and output instrumentation data to the standard error stream which the algorithm retrieves via a pipe. However, a new subprocess is created each time the standalone SUT executable is run, and this can be a significant overhead. For the empirical work in this thesis, the SUTs are themselves written in C or C++ and therefore can be linked directly with the algorithm implementation to create a single executable in order to avoid the overhead of subprocess creation.

2.3.5 Input and Output

The input to the algorithm executable is a file containing the algorithm parameters, and details about the SUT such as the valid interval for each input argument and number of coverage elements. A log file is output by the executable that records the best input profile found by the algorithm, its fitness, and statistics about the algorithm run. Both parameter and log files are XML format; this enables the creation of multiple parameter files from an experimental design, and the analysis of empirical results, using XML transformation and parsing tools.

2.3.6 Pseudo-Random Number Generation

The Mersenne Twister algorithm implementation in the GNU Scientific Library (Galassi et al., 2009) was used to generate pseudo-random numbers. The seed to the generator is specified in the parameter file.

2.4 *Software Under Test*

This section describes the three SUTs used in the experiments of this chapter: the toy function `simpleFunc` described above, and two real-world SUTs, `bestMove` and `nsichneu`. The provenance and features of the SUTs are described below. The instrumented source code of all three SUTs is available at: `http://www-users.cs.york.ac.uk/smp/supplemental/`.

2.4.1 *SUT Characteristics*

Relevant characteristics of the SUTs are summarised in table 3. These characteristics are different measures of the 'scale' of the problem, either in terms of the size and complexity of the SUT (lines of code, number of loops, number of branches), or the size and complexity of the input domain (number of arguments, argument data types, domain cardinality). It is reasonable to expect that the scale of the SUT and input domain affect the performance of the algorithm, and so the SUTs are chosen so that there is diversity in the values of the associated characteristics.

SUT	`simpleFunc`	`bestMove`	`nsichneu`
lines of code	15	89	1 967
no. loops	0	7	1
no. conditional branches	6	42	490
no. input arguments	2	2	11
argument data types	`int`	`int`	`int`
domain cardinality	1.00×10^3	2.62×10^5	3.40×10^6

Table 3 – Characteristics of the SUTs `simpleFunc`, `bestMove`, and `nsichneu`.

For all three SUTs, the input profile will be derived for the purpose of branch coverage, and thus the number of conditional branches listed in table 3 is also the number of coverage elements.

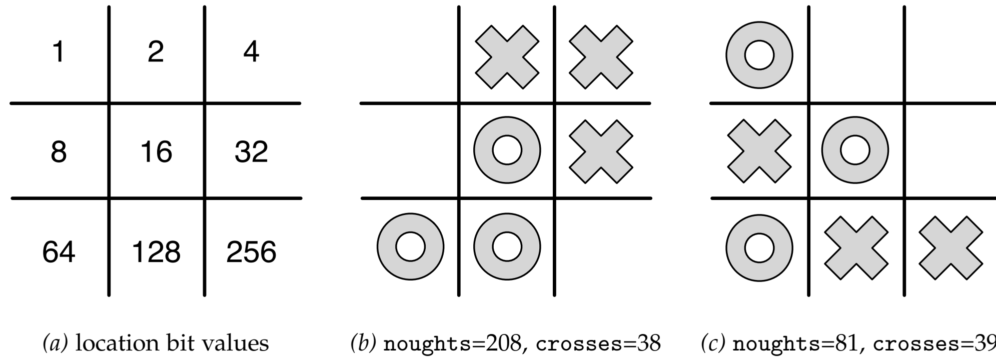
2.4.2 *simpleFunc*

Although `simpleFunc` is a small 'toy' function of our own design, it was demonstrated in section 2.2.1 that a suitable input profile for this SUT may be derived by manual analysis, and thus it provides a useful comparison between the automated computational search and manual analysis. (Both `bestMove` and `nsichneu` are significantly more complex than `simpleFunc` and deriving input profile by manual analysis for these SUTs would be very difficult at best.)

2.4.3 *bestMove*

The function `bestMove` is adapted from a demonstration applet from the Java 1.4 tutorial (Sun Corporation, 2006) that plays noughts-and-crosses (also known as tic-tac-toe). The code was converted from Java to C and additional validation of the input arguments was added. Given a game position, the function calculates the best move for the computer player to make, or identifies that the game has already been won or has reached stalemate. The game position is passed as two integer arguments; for clarity we refer to them here as `noughts` and `crosses`. The bits set in the `noughts` argument identify which locations are occupied by the computer player; the bits set in `crosses`, the locations occupied by the human player.

The arguments are illustrated in figure 12. Figure 12a shows the bit value assigned to each location: for example, a symbol in the centre square will add 16 to the value of the argument by setting the 5th bit, counting from the least significant bit. Figures 12b and 12c are examples of board positions and the corresponding argument values.



(a) location bit values *(b)* `noughts`=208, `crosses`=38 *(c)* `noughts`=81, `crosses`=392

Figure 12 – An illustration of the encoding of the board positions for `bestMove`. (a) is the mapping of locations to bits in the input arguments. (b) and (c) are example board positions with the corresponding input argument values shown below the board.

We suspect that this encoding of board positions presents a particular challenge to proposed algorithm since functionally similar inputs are widely separated in the input domain. For example, figures 12b and 12c both illustrate board positions at which the computer player (noughts) can win the game at the next move. These inputs to the function would therefore be expected to exercise similar branches in the SUT, but the input arguments are distant from one another when interpreted as integers. As discussed in section 2.2.1 above, the use of binning in the representation of candidate input profiles might be expected to be less effective in this situation, and so `bestMove` provides a useful case study.

2.4.4 *nsichneu*

The function `nsichneu` is automatically generated C code that implements an extended Petri net. It is taken from a set of benchmark SUTs used for research on worst-case execution time (Gustafsson et al., 2010). The function was modified to restore the number of iterations of the main loop to its original value; the number of iterations had been reduced in the benchmark version of the SUT, but the original value retained as a comment. This modification was made in order to increase the complexity and decrease the number of unreachable branches; nevertheless, some branches appear to be unreachable even in this modified version. Such branches are omitted from the set of coverage elements considered when calculating the fitness metric.

We suspect that the large number of conditional statements in the function, and the complex data-flow between them, presents a challenge to the proposed algorithm. In addition, `nsichneu` takes significantly longer to execute than the other two SUTs.

2.4.5 *Instrumentation*

The SUTs were instrumented manually using a simple textual substitution: the Boolean predicates in conditional statements were modified so in addition to evaluating the predicate, its value was recorded. This was a convenient approach for these example SUTs, but should

the algorithm be used in practice by test engineers, an automated approach could be applied using a tool such as ANTLR (Parr, 2007) to parse and rewrite the SUT's abstract syntax tree.

2.5 Experiment I – Algorithm Viability

2.5.1 Objectives

In this experiment, we demonstrate the viability of the proposed algorithm in terms of the time taken to derive a suitable input profile for the three examples SUTs.

The research questions addressed are:

RQ-1 Is the proposed algorithm capable of deriving near-optimal input profiles (i.e. profiles for which the minimum coverage probability is close to the maximum possible)?

RQ-2 How much time is required, on average, for the algorithm to derive such profiles?

2.5.2 Preparation

The algorithm parameter settings used in this experiment are listed in table 4. Since the objective of the experiment is to demonstrate viability rather than to determine the *best* performance of the algorithm, only a small amount of effort was spent in tuning the parameters.

The trade-off between an accurate estimation of the minimum coverage probability for use as the fitness metric and the resources consumed in executing the instrumented SUT more times to achieve this accuracy was noted in section 2.2.2. The settings of the sample size parameter, K, were chosen in the context of this trade-off, but erring on the size of accuracy: the settings are sufficiently high that even for candidate profiles with minimum coverage probabilities a magnitude or more lower than the target minimum coverage probability for these SUTs, every coverage element is exercised many times and thus a relatively accurate estimate of the minimum coverage probability can be made. The choice of a lower setting of K for `simpleFunc` is based on the smaller number of coverage elements and the higher target minimum coverage probability achievable for this SUT than for `bestMove` and `nsichneu`.

As discussed in section 2.2.3 above, the weights of mutations that decrease the size of the representation (M_{joi} and M_{rem}) are set higher than those that increase it (M_{spl} and M_{add}) in order to encourage parsimony in the representation, with the objective of improving algorithm performance and avoiding excessive memory usage.

For `simpleFunc` and `bestMove`, the optimal minimum coverage probability can be determined by manual examination of the code as 0.25 and 0.1667, respectively. For these SUTs, the target fitness, $\tau_{p_{\min}}$, was set to be close to, but not at, these optimum values: obtaining *exactly* the optimal fitness may require an infrequent chance occurrence or may be impossible given that the fitness is calculated using a finite sample. For example, if there are four mutually exclusive coverage elements, obtaining an optimal fitness of 0.25 would require each coverage element to be exercised by exactly one quarter of the sample, which may occur only rarely even if the *actual* minimum coverage probability of the profile is this optimal value, and would be impossible if the number of samples is not a multiple of four.

For `nsichneu`, the code is too complex to derive an optimal probability manually and so the target was set to close to the maximum value observed during preliminary experimentation.

2.5.3 Method

For each of the three SUTs, the algorithm was executed 32 times. We will refer to each execution of the algorithm as a *trial*.

Parameter	Effect	SUT	Value
λ	neighbourhood sample size		4
ρ_{prb}	bin probability mutation factor		10.0
ρ_{len}	bin length mutation factor		10.0
w_{prb}	M_{prb} mutation weight		1.0
w_{len}	M_{len} mutation weight		1.0
w_{spl}	M_{spl} mutation weight		0.8
w_{joi}	M_{joi} mutation weight		1.0
w_{add}	M_{add} mutation weight		0.8
w_{rem}	M_{rem} mutation weight		1.0
K	evaluation sample size	`simpleFunc`	200
		`bestMove`	1 000
		`nsichneu`	1 000
$\tau_{p_{\min}}$	target min. coverage probability	`simpleFunc`	0.24
		`bestMove`	0.14
		`nsichneu`	0.035
τ_{iter}	maximum no. iterations	`simpleFunc`	4×10^3
		`bestMove`	6×10^4
		`nsichneu`	4×10^3

Table 4 – Algorithm parameter settings used in Experiment I. The SUT name is listed in the column 'SUT' only for those settings that differ between the SUTs.

Each trial was provided with a different seed to the pseudo-random number generator. The seeds were obtained from the website `random.org` that generates random numbers from atmospheric noise at radio frequencies.

The choice of 32 trials is a compromise between the accuracy of these results and the resources—computing power and time—that were available for experimentation. During the analysis of the results below, confidence intervals are provided as an indication of the accuracy achieved.

The trials were run on a set of Linux servers. Each trial used one core of a CPU running at 3 GHz.

The responses measured were whether or not the algorithm trial was successful in deriving a profile with the target fitness, and the algorithm's run time. We choose to use the processor time (obtained using the standard C function `clock()`) as the run time metric since this will be less sensitive to the effect of other processes running on the same server than the alternative measure of elapsed time.

2.5.4 *Results*

The results are summarised in table 5. The reported values are the mean of the responses observed in the 32 trials for each SUT. The 95% confidence intervals are estimated by a bootstrapping technique.

The response π_+ is the mean proportion of trials that succeeded in deriving a suitable input profile: one for which the fitness—the estimated minimum coverage probability—met or exceeded the target probability. T_+ is the mean processing time taken by trials that are successful, and T_- the mean time taken when the trail is unsuccessful and is terminated when the maximum number of iterations is reached.

If we assume that after an unsuccessful trial, the test engineer would in practice run further trials of the algorithm (with different seeds) until one is successful, then we may estimate the total time taken until a near-optimal profile is obtained. To do this, the algorithm trials are modelled as a Bernoulli process: each trial is an independent random variable with

Mean Response	simpleFunc	bestMove	nsichneu
Proportion of trials successful, π_+	1.0	$0.72\,^{+0.13}_{-0.19}$	$0.31\,^{+0.16}_{-0.16}$
Time taken by successful trial, T_+ (min)	$0.04\,^{+0.02}_{-0.01}$	$80\,^{+12}_{-14}$	$144\,^{+25}_{-30}$
Time taken by unsuccessful trial, T_- (min)		$124\,^{+3}_{-4}$	$204\,^{+5}_{-4}$
Estimated total time until success, C_+ (min)	$0.04\,^{+0.02}_{-0.01}$	$129\,^{+76}_{-40}$	$592\,^{+702}_{-252}$

Table 5 – A summary of results of Experiment I. Upper and lower bounds of the 95% confidence intervals are reported as differences from the mean response in smaller type after the mean.

two outcomes, success and failure, where the probability of success is π_+. For such processes, the number of failures until the first success follows an exponential distribution with mean $(1 - \pi_+)/\pi_+$. Thus the average time taken for the mean number of failures followed by one successful trial is:

$$C_+ = \frac{1 - \pi_+}{\pi_+} T_- + T_+ \tag{26}$$

This estimate is reported in the final row of table 5.

The estimate is conservative: in practice the test engineer is likely to be able to run multiple trials of the algorithm in parallel on a multicore computer (we explore the benefits of such parallelisation in section 5). However, it does provide an upper bound on the time taken to obtain a near-optimal input profile.

2.5.5 *Discussion and Conclusions*

The results show that the algorithm is capable of deriving near-optimal profiles for all three SUTs (RQ-1). In the worst case—that of `nsichneu`—it would take approximately 10 hours on average to derive a profile (RQ-2).

We did not specify *a priori* an upper limit on the computing resources and time taken by the algorithm for it to be considered viable. However, we note that if a test engineer were to run a set of trials overnight on a single core of a computer equivalent in power to a desktop PC, there would be a good chance of deriving a profile by morning for any of the three SUTs. We argue that this is indicative that the proposed algorithm is a viable method of deriving input profiles for statistical testing.

2.5.6 *Threats to Validity*

We identify two important threats to the validity of our conclusions:

Generality It is not possible to extrapolate the results to other software with confidence given the small number of SUTs considered in the experiment. However, the SUTs were chosen to be diverse in terms of the characteristics listed in table 3. Moreover, `bestMove` and `nsichneu` were chosen specifically because they are non-trivial in size and have characteristics that we believed would challenge the algorithm. If this is indeed the case, we may reasonably expect the existence of a fairly large number of real-world SUTs—at the very least, those SUTs 'smaller' or 'less complex' than `bestMove` and `nsichneu`—for which the algorithm is a viable method of synthesising profiles. It is not clear, however, exactly which characteristics should be used to identify 'smaller' and 'less complex' SUTs.

Implementation We assume that the code used in this and subsequent experiments is an accurate implementation of the proposed algorithm described in section 2.2. For practical purposes any differences in implementation are not necessary a problem: we have an implemented *an* algorithm that is fit for purpose, even if it not exactly the proposed one. However, in the context of research, it would be difficult to interpret experimental results for the purpose of enhancing the algorithm if there are meaningful differences between the implemented and proposed algorithms.

We have reduced the chance of any such differences by testing the implementation during and after the development process. Many aspects of the algorithm's operation may be logged to the output file in addition to the data required for the experiments so as to support this testing.

2.6 Experiment II – Fault-Detecting Ability

2.6.1 Objectives

Thévenod-Fosse and Waeselynck (1991, 1993) demonstrated empirically that statistical testing is more capable at detecting faults than both deterministic structural testing and uniform random testing. The sizes of the test sets used for structural testing were the smallest that exercised each coverage element at least once. The test sets generated for statistical testing were larger and exercised each coverage element multiple times since this is the premise of statistical testing: executing each coverage element many times overcomes some of the limitations of deterministic structural testing (see section 1.7.1). The size of the test sets for random testing were the same as those for statistical testing.

Thévenod-Fosse and Waeselynck had the objective of demonstrating the advantages of statistical testing as a verification technique. In this experiment we perform a similar demonstration, but with the objective of showing that input profiles derived automatically by the search algorithm continue to demonstrate the superior fault-detecting ability that Thévenod-Fosse and Waeselynck found for profiles derived manually. (For conciseness, we refer in the following to the fault-detecting ability of the profiles themselves, even though this is a property of test sets *generated* from the profile.)

Experiment II therefore addresses the following research questions:

RQ-1 Do test sets sampled from profiles derived by the search algorithm detect more faults than test sets of the same size randomly sampled from a uniform profile?

RQ-2 Do test sets sampled from profiles derived by the search algorithm detect more faults than minimal test sets generated using deterministic structural testing?

2.6.2 Preparation

The fault-detecting ability of a test set was assessed using mutation analysis. The Proteum/IM tool (Delamaro and Maldonado, 2001) was used to create mutant versions of the SUT by applying the large number of mutation operators built in to the tool. For `simpleFunc` and `bestMove`, all possible single-point mutations were created, resulting in 178 and 1923 mutant SUTs respectively. Since `nsichneu` is larger and therefore has many more potential mutants, 10% of the possible single-point mutations were selected at random, resulting in 6699 mutant SUTs.

A mutant is regarded as being 'killed' by the test set if for at least one test case, the mutant produced a different output, or was terminated by a different operating system signal, from the unmutated SUT. (To accommodate infinite loops that could be created by mutation, mutant executables were terminated by an operating system signal if they ran for more than one second of CPU time; this is much longer than the execution time of any of the unmutated SUTs.)

The *mutation score* is the proportion of the mutants that a test set kills; a higher score indicates that the test detects more faults. The mutation score was used to compare the fault-detecting ability of the test sets constructed using the different testing strategies.

2.6.3 Method

Automated Statistical Testing For each SUT, 10 input profiles derived during the successful trials of Experiment I were identified. From each of the 10 profiles per SUT, 10 tests sets were generated by random sampling test inputs from the profile. The sampling was *without replacement*: if the sampled test input was identical to one already present in the test set, it was rejected and another sampled. We believe that this is consistent with how tests would be generated in practice. Preliminary experiments using *with replacement* sampling gave results that were qualitatively no different to those reported here.

As each new input was sampled from the profile and added to the test set, the set of mutants were executed using the newly sampled input, and the mutant score for the entire test set re-calculated by combining the results of the new input with the results previously obtained for the rest of the inputs. This piecewise assessment of the mutation score enables the fault-detecting ability of different test set sizes to be compared. The maximum size of test sets was 50 for `simpleFunc` and `bestMove`, and 150 for `nsichneu`.

Uniform Random Testing To enable a comparison against uniform random testing, 100 test sets were sampled from the uniform profile for each SUT. As above, the test input sampling was without replacement, and the mutation score of the test sets were assessed after adding each new test case.

Deterministic Structural Testing Instead of generating test sets using a structural testing technique, an upper bound for the fault-detecting ability of structural testing was estimated from the mutation scores of the statistical testing profiles derived by search. For each statistical testing test set, the size at which the set first satisfies the adequacy criterion for deterministic structural testing—exercising each coverage element at least once—was calculated. The mutation score at this test size is then an estimate for the mutation score of test sets generated by deterministic structural testing techniques.

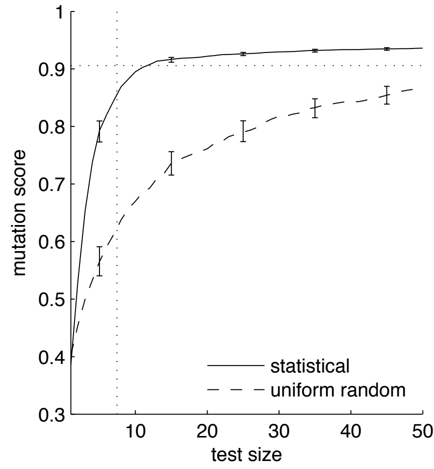
When the test set first satisfies the adequacy criterion, it is does not necessarily represent a minimal structural testing test set. Some test cases may be redundant in the sense that they could be removed and the test set would still satisfy the adequacy criterion. These redundant test cases can only improve the mutation score. Therefore, this approach will overestimate the fault-detecting ability of deterministic structural testing, but nonetheless provide an indicative upper bound.

2.6.4 Results

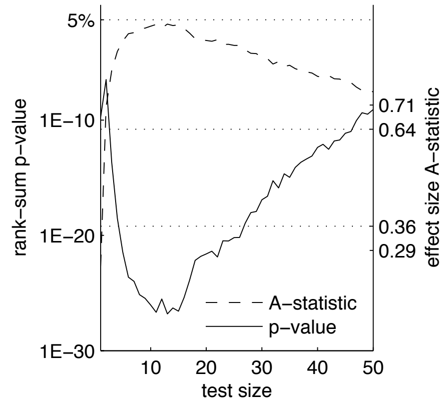
The results are summarised in figure 13.

Graphs (a), (c), and (e) compare the mean mutation scores of algorithm-derived profiles for statistical testing (solid lines) to those of the uniform profile (dashed lines). Error bars indicate 95% confidence intervals estimated by bootstrapping using the same technique as Experiment I. Although mutation scores were assessed for every integer test size, for clarity error bars are only shown at regular intervals.

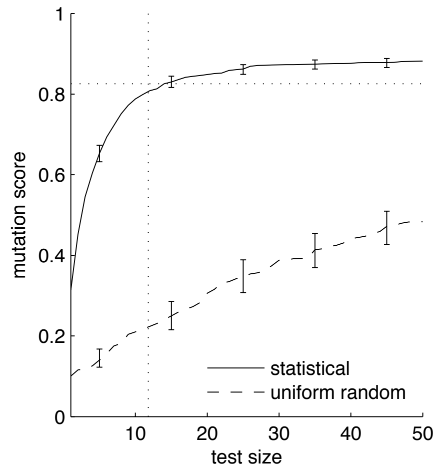
The horizontal dotted line is the mean estimated mutation score that would be achieved by deterministic structural testing techniques. The vertical dotted line is the mean size at which test sets generated from search-synthesised profiles satisfy the deterministic structural testing adequacy criterion. As discussed above, both the mean mutation scores and sizes for deterministic structural testing are upper bounds.



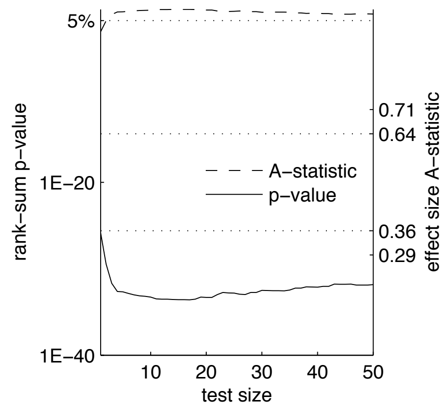
(a) `simpleFunc` mutation scores



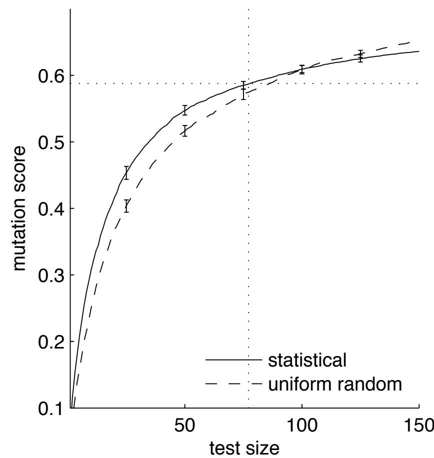
(b) `simpleFunc` significance



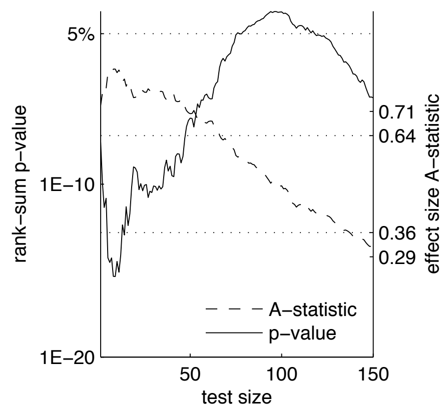
(c) `bestMove` mutation scores



(d) `bestMove` significance



(e) `nsichneu` mutation scores



(f) `nsichneu` significance

Figure 13 – Results of Experiment II. Graphs (a), (c), and (e) compare the mutation scores of algorithm-derived profiles for statistical testing (solid line) to uniform random testing (dashed line). The horizontal dotted line is the estimate of the mutation score of test sets generated using deterministic structural testing. Graphs (b), (d), and (f) show the corresponding statistical significance (solid line and left-hand axis) and effect size (dashed line and right-hand axis) of the differences in mutation score.

It was not necessary to remove equivalent mutants—mutants that are functionally indistinguishable from the unmutated SUT—in order to compare the fault-detecting ability of the three testing strategies: the mutation score is reduced by the same amount for each strategy in the presence of these equivalent mutations. However, since equivalent mutants were not removed, the mutation scores are not expected to reach the value of 1.0 even at the largest test sizes.

Graphs (b), (d), and (f) shows the statistical significance (solid line and left-hand axis) and effect size (dashed line and right-hand axis) of the differences in mutation score between the algorithm-synthesised and uniform profiles. The statistical significance is calculated using the non-parametric Mann-Whitney-Wilcoxon (rank-sum) test (Wilcoxon, 1945). The null hypothesis of this test is that for the two profiles, the distribution of the mutation scores are the same. The test returns a p-value; the smaller the value, the less likely it is that the observed difference is a result of chance.

The effect size is calculated using the non-parametric Vargha-Delaney A-test (Vargha and Delaney, 2000). The test returns an A-statistic which has a simple real-world interpretation: it is the chance that if one mutation score is picked from each distribution at random, the mutation score for the algorithm-synthesised profile would be higher than that for uniform profile. An A-statistic of 0.5 indicates no effect. Vargha and Delaney suggest that, in general, values greater than 0.64 (or less than 0.36) may be interpreted as a 'medium' effect size, and greater than 0.71 (or less than 0.29) as a 'large' effect size. The lower boundaries of the medium effect size region are indicated by dotted lines on the graphs.

The calculation of effect size is to support the claim of statistical significance. Even if the difference in the distribution of mutation scores between the profiles is so small as to have no practical impact, it could nevertheless be shown to be statistically significant given a sufficiently big sample. Should the sample size of 100 test sets used here be big enough for this to be the case, a medium or large effect size statistic—which is independent of sample size—demonstrates that observed difference would have a meaningful effect when the testing techniques are used in practice. At the medium effect size boundary of 0.64, it is almost twice as likely that one test set generated by an algorithm-derived profile would find more faults than one test set generated from the uniform profile.

2.6.5 Discussion and Conclusions

Graphs (a) and (c) of figure 13 show, for `simpleFunc` and `bestMove`, the profiles derived using the algorithm have a superior fault-detecting ability to those used by uniform random testing (RQ-1). This is most noticeable for `bestMove` where, for a range of moderate test sizes, the algorithm-derived profiles detect over twice as many faults. The differences observed are statistically significant—the p-values are much less than 5%—and for most test sizes the Vargha-Delaney A-statistics suggest a large effect size.

The results for `nsichneu` in graph (e) differ from the first two SUTs. For test sizes less than approximately 70, the profiles synthesised using the algorithm *do* detect more faults than the uniform profile. The difference is relatively small compared to the other two SUTs, but statistically significant. However, at test sizes above approximately 100, an unexpected reversal occurs: uniform random testing is superior and this difference is statistically significant at the highest test sizes.

Graphs (a), (c), and (e) also show that algorithm-derived profiles can be used to generate test sets that detect more faults than minimally-sized tests used by deterministic structural

testing: the mutation scores for the algorithm-derived profiles exceed the (conservative) estimate for deterministic structural testing (the dotted horizontal lines) given a sufficiently large test set size (RQ-2).

This is the potential efficiency of statistical testing: it allows the easy generation of test sets that exercise each coverage element multiple times, and such test sets detect more faults that a minimal test set that exercises each coverage element only once. The number of additional faults that can be detected in this way is relatively small for `simpleFunc` and `bestMove`: the lines for statistical testing plateau at values just above the upper bound for deterministic structural testing. Since the mutation score for `nsichneu` has yet to plateau at the highest test size measured, the difference for this SUT might be larger.

In general, the results provide evidence that the superior fault-detecting ability of statistical testing, demonstrated by Thévenod-Fosse and Waeselynck for manually derived input profiles, is also true of profiles derived using our computational search algorithm.

Nevertheless we highlight the unexpected result for `nsichneu`, where the algorithm-derived profile detects fewer faults than uniform random testing at the largest test sizes. This result provides the motivation for the next experiment.

2.6.6 *Threats to Validity*

In addition to the threats identified for Experiment I:

Mutation Analysis In this experiment, we implicitly assume that the mutation score obtained by mutation analysis is a good indication of real-world fault-finding ability. In particular, an assumption is made that the mutation operators applied to create mutants, and the use of single point mutations, are representative of the type of faults that could occur in practice. We do not address this assumption here, but note that mutation analysis is widely used for this purpose in other software testing research, and that Thévenod-Fosse and Waeselynck used mutation analysis in their empirical demonstration of the efficacy of statistical testing.

2.7 Experiment III – Change in Fault-Detecting Ability During the Search

2.7.1 Objectives

An intriguing result of Experiment II was that for `nsichneu`, the fault-detecting ability of input profiles derived by the search algorithm was, at some test sizes, less than that of the uniform random testing. This is particularly surprising since the candidate profile from which search begins is a uniform profile, and so the effect of the algorithm in this case was to *reduce* the fault-detecting ability.

This counter-intuitive effect motivates this experiment to address the following research questions:

RQ-1 How does the fault-detecting ability of the current candidate input profile (i.e. the candidate profile at the start of each iteration) change over the course of the search?

RQ-2 How does the fault-detecting ability of input profiles change with minimum coverage probability?

2.7.2 Method

For each of the three SUTs, the current candidate input profile was recorded at regular intervals (defined in terms of the number of iterations) during the course of a single successful algorithm trial, i.e. one that reached the target fitness. From each of these profiles, 20 test sets were generated and the mutation score of the test set assessed using the process described in Experiment II. In addition, an accurate estimation of the minimum coverage probability of the profile was evaluated by using a sample size much larger than that used by the algorithm itself.

2.7.3 Results

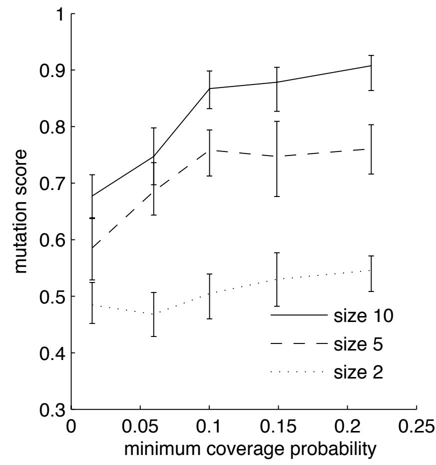
The results are summarised in figure 14.

The mutation scores are plotted for three different test sizes, the mean mutation score (over the 20 test sets generated from each input profile) on the vertical axis and the accurate estimation of the minimum coverage probability on the horizontal axis. Error bars at each point indicate the 95% confidence interval for the mutation score, and lines connect points with the same test set size. Since the minimum coverage probability generally—but, as discussed in section 2.2.3, not necessarily always—increases during the search, the search trajectory is from left to right in these graphs. The left-hand most set of points of each graph are for the initial input profile provided to the search, i.e. a uniform profile.

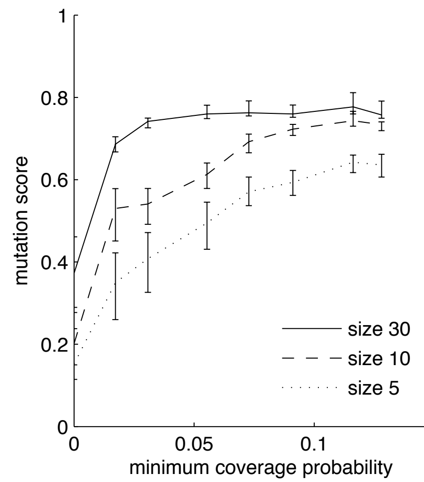
2.7.4 Discussion and Conclusions

At the the beginning of the search, the fault-detecting ability of the current profile improves as the search proceeds. Later in the search, the fault-detecting ability can plateau (e.g. `simpleFunc` test sizes 5 and 10; `bestMove` test size 30) or get worse as the search proceeds (e.g. third and fourth profiles of `nsichneu` at test sizes 80 and 150).

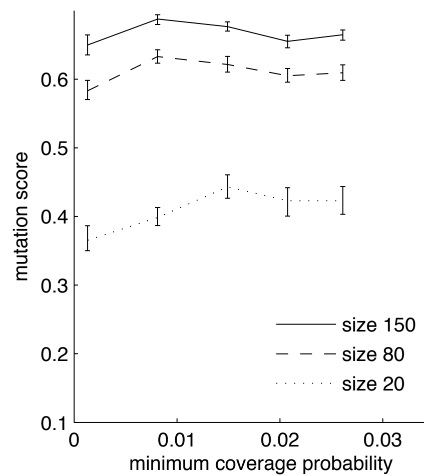
The results therefore provide evidence that the fault-detecting ability of the current profile does not always increase as the search progresses (RQ-1). They also show that higher minimum coverage probabilities are not necessary indicative of better fault-detecting ability (RQ-2).



(a) `simpleFunc`



(b) `bestMove`



(c) `nsichneu`

Figure 14 – Results of Experiment III: the mean mutation scores (vertical axis) and minimum coverage probability (horizontal axis) of input profiles at regular interval during an algorithm run. The mutation scores are assessed at three test set sizes, and lines connect points for test sets of the same size.

A positive interpretation of the latter result is that it may not always be necessary to target the highest possible minimum coverage probability. For example, the search for an input profile for `bestMove` could have used a much lower target, e.g. $\tau_{p_{\min}} = 0.10$ rather than 0.14, and the derived profile would still had nearly the same fault-detecting ability.

A negative interpretation of the same result is that minimum coverage probability is not a sufficient fitness metric for the search algorithm if the objective is to derive an efficient profile, i.e. one that detects as many faults as possible per test case.

2.8 *Profile Diversity*

In Experiment II, algorithm-derived profiles were found to detect fewer faults than the uniform profile for larger test set sizes for `nsichneu`. In Experiment III, the fault-finding ability of the current candidate input profile sometimes decreased as the search progressed to solutions with higher minimum coverage probabilities. We hypothesise that there is a common cause for both of these observations: input profiles derived by the search algorithm can lack *diversity*.

We informally define diversity in terms of the subsets of the input domain that exercise each coverage element. In a diverse profile, each input in a subset has a similar probability of being sampled. In large test sets sampled from a diverse profile, each coverage element will be exercised more than once, and it is likely to be exercised by *dissimilar* inputs from the subset each time. However in a non-diverse profile, only a limited number of the inputs in each subset have a high probability of being sampled. In a test set sampled from such a profile, a coverage element may be exercised more than once, but often it is exercised by *similar* inputs. Exercising a coverage element multiple times with dissimilar inputs will typically detect more faults than if a coverage element is repeatedly exercised with similar inputs. It is therefore reasonable to assume that diversity contributes to the fault-finding ability of an input profile.

A number of testing strategies explicitly consider diversity when selecting test inputs. All are based on the assumption that the points in the input domain that reveal a given fault are likely to form contiguous regions. If so, it would be inefficient to execute the SUT using multiple points in the same fault-revealing region, and so the strategy should instead ensure that future test inputs are 'far away' from existing test inputs in the test set in order to avoid being part of the same fault-revealing regions. The strategy of *antirandom testing* (Malaiya, 1995) starts with an arbitarily chosen test input, and then deterministically constructs additional inputs such that the total distance—either Euclidean or Hamming—of each test input from previous members of the test set is maximised. Bueno et al. (2011) propose a similar stategy called *diversity orientated test data generation* (DOTG) that applies metaheuristic search to optimise test sets for diversity using a fitness metric based on the Euclidean distance between the inputs in a candidate test set. An empirical comparison using mutation analysis demonstrates that test sets constructed using DOTG have a higher fault-detecting ability than uniform random testing. *Adaptive random testing* (Chen et al., 2010) differs in that it permits any distance metric to be used, and generates test inputs randomly: for each new test case, a small number of candidate inputs are generated at random and the one that maximises the minimum distance from all of the existing test inputs is selected.

Our hypothesis is that since the fitness metric of the profile-synthesing algorithm does not include a measure of the diversity, diversity can be lost in the process of increasing the minimum coverage probability. As an illustration, consider the optimal profile manually derived for `simpleFunc` in section 2.2.1:

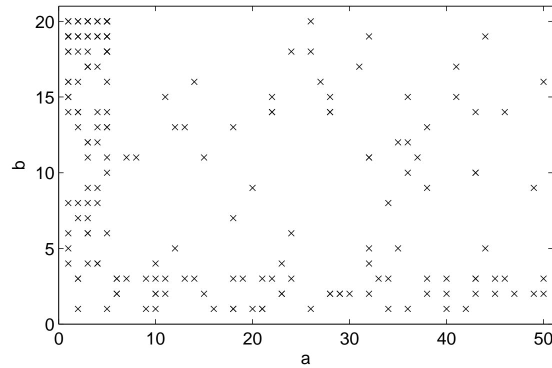
$$\mathbb{P}(a,b) = \begin{cases} 1/340 & \text{if } (1 \leq a \leq 5) \wedge (1 \leq b \leq 17) \\ 1/60 & \text{if } (1 \leq a \leq 5) \wedge (18 \leq b \leq 20) \\ 1/540 & \text{if } (6 \leq a \leq 50) \wedge (1 \leq b \leq 3) \\ 1/3060 & \text{if } (6 \leq a \leq 50) \wedge (4 \leq b \leq 20) \end{cases} \tag{27}$$

The following profile is also optimal in the sense that it has the same, highest possible mini-

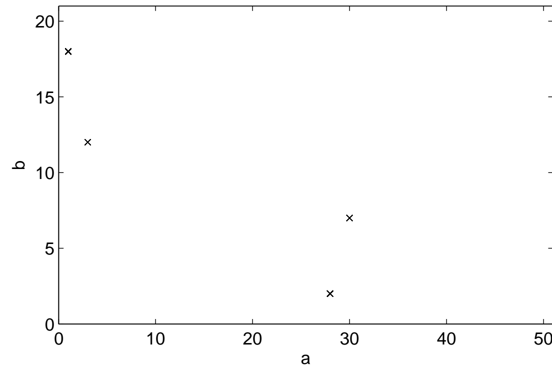
mum coverage probability:

$$\mathbb{P}(a,b) = \begin{cases} 1/4 & \text{if } (a,b) = (3,12) \\ 1/4 & \text{if } (a,b) = (1,18) \\ 1/4 & \text{if } (a,b) = (28,2) \\ 1/4 & \text{if } (a,b) = (30,7) \\ 0 & \text{otherwise} \end{cases} \tag{28}$$

These two optimal profiles are illustrated in figure 15 by the plots of 200 inputs sampled at random from each profile.



(a)



(b)

Figure 15 – An illustration of the profiles of (a) equation (27); and (b) equation (28). Each plot shows 200 points sampled at random from the profile.

We consider the four subsets of the input domain corresponding to the four mutually-exclusive branches of `simpleFunc` (e_2 to e_5 in figure 9). In the first profile of equation (27) all the inputs in each subset have the same probability of being sampled. However, in the second profile of equation (28), only one input from each subset is ever sampled. If a fault existed in the SUT and was only detectable when, say, $1 \leq a \leq 5$ and $b = 19$, then it could be found using inputs sampled from the first profile, but never by inputs sampled from the second profile. In the terminology of Goodenough and Gerhart, the concrete strategy represented by the second profile lacks 'validity'.

A uniform profile is inherently diverse: every input in the subset corresponding to each

coverage element has the same probability of being sampled. Therefore, if the effect of the algorithm is to reduce the diversity, the fault-detecting ability of the profile derived by the algorithm could be less than that of the uniform profile.

This loss of diversity could explain the results of Experiment II for `nsichneu`. If the algorithm starts from a diverse uniform profile but loses diversity as it progresses, it could also explain the reduction in fault-detecting ability that occurs along some sections of the algorithm trajectories in Experiment III: while an increase in maximum coverage probability contributes to the fault-detecting ability of the profile, it does not always offset the reduction resulting from the loss of diversity.

We suspect that diversity is implicitly retained when input profile for statistical testing are derived manually. However, in the absence of any explicit guidance, there is no reason for the computational search algorithm to do the same. We therefore propose that a suitable diversity measure be incorporated as an additional component of the fitness metric for this purpose, and test the efficacy of such a measure in the next experiment.

2.9 Experiment IV – Effect of Diversity Measure

2.9.1 Objectives

To test our hypothesis that profiles derived by the algorithm can lack diversity, we repeat Experiment II, but using the fitness metric that incorporates a diversity measure. The research question is:

RQ-1 Does the incorporation of a diversity measure in the fitness metric improve the fault-detecting ability of the derived profiles?

2.9.2 Preparation

For this experiment, we propose the following diversity measure calculated from the data returned by the instrumentation of the SUT during fitness evaluation.

During a fitness evaluation, each coverage element, c, will be exercised by a subset of the K inputs sampled from the profile. Let n_c be the size of this subset. The most frequently occurring element in the subset is identified, and let h_c be the number of times this element occurs. For example, if an element is exercised by the inputs $(1,19), (3,20), (1,18), (1,20), (3,20)$, then $n_c = 5$, and $h_c = 2$ since $(3,20)$ is most frequently occurring input and it occurs twice.

The ratio h_c/n_c is a simple measure of the diversity of the inputs sampled for the coverage element: the more distinct inputs that exercise the coverage element (without repetition), the lower the value of this ratio.

The diversity of the input profile as a whole can be calculated as:

$$\widehat{g}_{\text{div}} = 1 - \frac{1}{|\mathcal{C}|} \sum_{c \in \mathcal{C}} \frac{h_c}{n_c} \tag{29}$$

where $\mathcal{C}$ is the set of coverage elements. The measure takes a value in the range $0 \leq \widehat{g}_{\text{div}} < 1$, and is defined so that the more diverse the input profile, the further the value is from 0. This is consistent with the direction of the estimated minimum coverage probability, $\widehat{p}_{\text{min}}$, for which fitter profiles take values further from 0.

For each SUT, the diversity measure is given a target value, $\tau_{g_{div}}$, that is a new parameter to the algorithm.

The minimum coverage probability and diversity are combined as a single fitness metric:

$$w \max(\tau_{p_{\text{min}}} - \widehat{p}_{\text{min}}, 0) + (1 - w) \max(\tau_{g_{\text{div}}} - \widehat{g}_{\text{div}}, 0) \tag{30}$$

where the coefficient w weights the contributions of the minimum coverage probability and diversity, and is a further new parameter to the algorithm. Fitter input profiles—those demonstrating high diversity and high minimum coverage probability—have lower values of this new fitness metric, and so the algorithm now minimises the fitness. The fitness metric reaches zero only when both the minimum coverage probability and diversity reach the specified target values, and so a fitness of zero is now the termination criterion for the algorithm.

2.9.3 Method

For each SUT, two sets of 10 input profiles were derived using the algorithm.

The first set used the new fitness metric of equation (30) using the target minimum coverage probability, target diversity parameters and weight shown in table 6. Suitable values

for the target diversity and weight were found by preliminary experimentation. All other algorithm parameters are the same as those used in Experiment I and listed in table 4.

Parameter	Effect	`simpleFunc`	`bestMove`	`nsichneu`
$\tau_{p_{\min}}$	target min. coverage probability	0.24	0.10	0.025
$\tau_{g_{\text{div}}}$	target diversity	0.95	0.80	0.995
w	weight of min. coverage prob.	0.8	0.8	0.95

Table 6 – Algorithm parameters used in Experiment IV. (All other parameters as table 4.)

In order to demonstrate that the lack of diversity has an effect even when the minimum coverage probability is not close to the optimal value, the target minimum coverage probabilities for `bestMove` and `nsichneu` were less than in Experiment I. There was also a practical consideration for making this change: the incorporation of a diversity metric was found to increase the run time of the algorithm and so a reduction in the target minimum coverage probability was used to counter this increase.

The second set of 10 profiles was derived using the same parameters, but with the weight, w, set to one. These profiles are therefore derived without a diversity metric, and provide a comparison to the first set. Despite a constant target $\tau_{p_{\min}}$, profiles derived by the algorithm in practice exhibit a range of *actual* minimum coverage probabilities. In order to minimise the effect of different minimum coverage probabilities on the experiment, more than 10 profiles were synthesised for each SUT. From these a subset of 10 was chosen in a principled manner so that the distribution of minimum coverage probabilities across the subset was as similar as possible to the same distribution across the profiles of the first set (those derived using the fitness metric incorporating the diversity measure).

From every input profile, 10 test sets were generated by random sampling from the profile. The fault-detecting ability of the test sets were assessed using the mutation analysis technique described in Experiment II.

2.9.4 Results

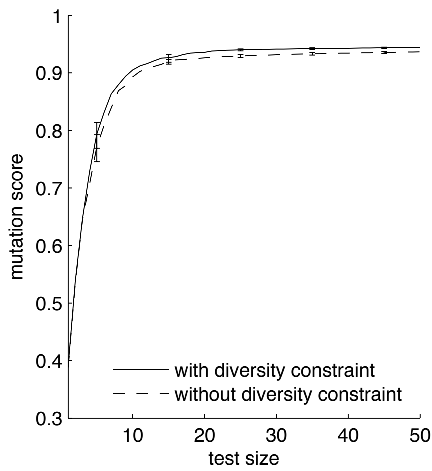
The results are summarised in figure 16.

Graphs (a), (c), and (e) compare the mean mutation scores of profiles derived using a fitness metric (solid lines) with a diversity measure to those derived without (dashed lines). For `nsichneu`, the fault-detecting ability of the uniform profile assessed in Experiment II is repeated here for comparison (dotted line).

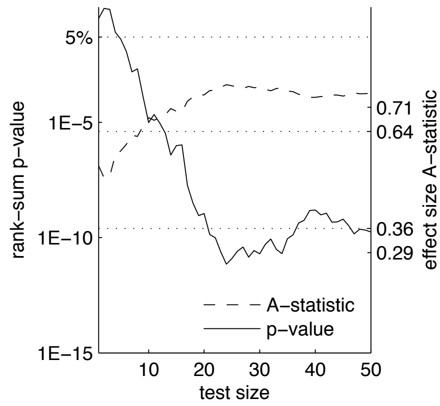
Graphs (b), (d), and (f) shows the statistical significance (solid line and left-hand axis) and effect size (dashed line and right-hand axis) of the differences in mutation score calculated using the non-parametric Mann-Whitney-Wilcoxon (rank-sum) test and Vargha-Delaney A-test as described in Experiment II.

2.9.5 Discussion and Conclusions

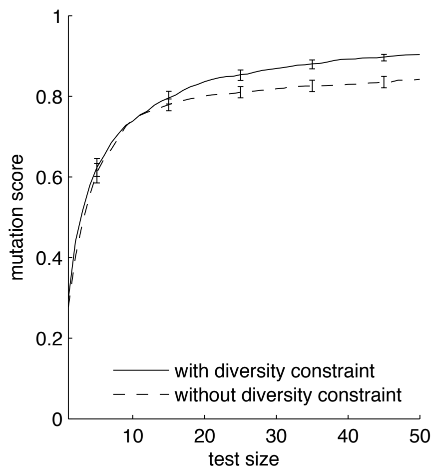
For `simpleFunc` and `bestMove`, the use of diversity component improves the fault-detecting ability of the derived profiles. The effect is relatively small, but strongest and most significant at larger test sizes, especially for `bestMove`. This would be expected: for smaller test sets, each coverage element is exercised by only a relatively few inputs and so multiple similar inputs are less likely to be sampled.



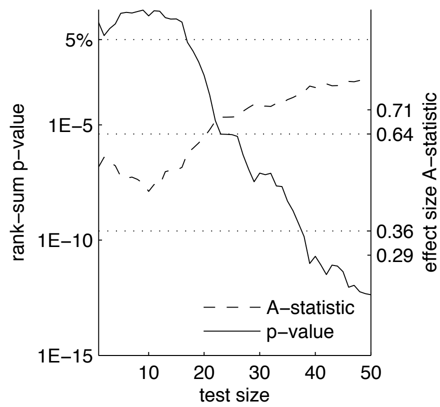
(a) `simpleFunc` mutation scores



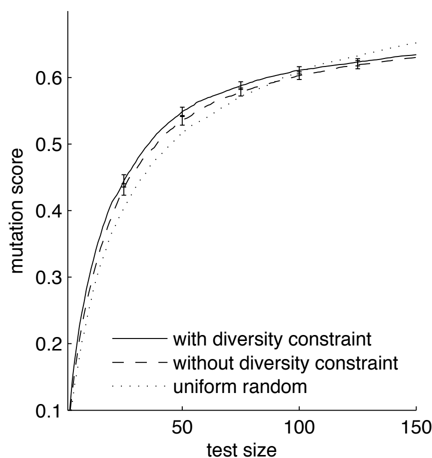
(b) `simpleFunc` significance



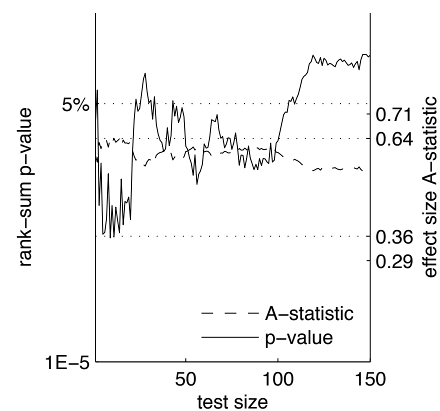
(c) `bestMove` mutation scores



(d) `bestMove` significance



(e) `nsichneu` mutation scores



(f) `nsichneu` significance

Figure 16 – Results of Experiment IV. Graphs (a), (c), and (e) compare the mutation scores of profiles derived using fitness metric with (solid line) and without (dashed line) a diversity measure. Graphs (b), (d), and (f) show the corresponding statistical significance (solid line and left-hand axis) and effect size (dashed line and right-hand axis) of the differences in mutation score.

For `nsichneu`, there is a smaller effect, though it is significant for many test sizes. In contrast to `simpleFunc` and `bestMove`, the effect diminishes at the largest test sizes, and the uniform random profile (dotted line of graph (c)) detect more faults than either set of synthesised profiles.

The results provide evidence that the use of a diversity measure *can* improve the fault-detecting ability of derived profiles, but not necessarily for all SUTs and at all test sizes (RQ-1).

We suspect that the simple diversity measure used in the fitness function does not capture the diversity property discussed in section 2.8 sufficiently well. In particular, the measure assesses only the number of distinct inputs exercising each coverage element; it does not *directly* measure how *similar* those inputs are to one another. The degree of similarity of two sampled inputs is dependent on the nature of the SUT itself, but for some SUTs an enhanced diversity metric might consider the Euclidean distance between inputs as an approximate measure of similarity.

An alternative explanation for the continued poor fault-detecting ability of statistical testing compared to uniform random testing for `nsichneu` is that this SUT has the characteristics identified by Boland et al. under which partition testing (of which statistical testing could be considered a form) is less efficient than uniform random testing (see section 1.6.2). However, we would expect uniform random testing to outperform statistical testing at all test sizes if this were the case, but at smaller test sizes statistical testing is superior.

2.9.6 Threats to Validity

In addition to the threats identified in Experiments I and II:

Parameter Selection The choice of algorithm parameters relating to diversity—the target diversity and weight—may have affected the outcome. For example, a higher diversity target for `nsichneu` might be achievable and could have resulted in better mutation scores. For the diversity measure used in this experiment, it would be difficult to analytically derive a suitable target value *a priori* as the analysis would require knowledge of the subsets of the input domain exercising each coverage element. If such knowledge were available, there would be no need to use an algorithm to derive input profiles as they could be constructed manually. The use of preliminary experimentation to explore the range of diversity values seen in practice is a practical method of setting the diversity target, but does not necessarily lead to good parameter choices.

Generality The current diversity measure may extend poorly to SUTs with real-valued inputs since floating point values sampled from the profile will almost always be distinct, and so the value h_c will be at most 1. The use of Euclidean distance (or another measure of similarity between inputs) could be used in place of equality when deriving a measure for real-valued inputs.

2.10 Conclusion

In this chapter, we proposed a computational search algorithm—a representation, fitness metric, and search method—that derives input profiles suitable for efficient statistical testing.

The algorithm was demonstrated on three example SUTs, two of which were real-world functions. The algorithm was assessed empirically in terms of both a functional property: that it was able to derive suitable profiles; and non-functional properties: the time and computing resources required. The algorithm was shown to be capable of deriving profiles for all three SUTs in a practical time using computing resources typical of a desktop PC.

The profiles derived by the algorithm had a fault-detecting ability that was, for the most part, superior to uniform random testing and to deterministic structural testing. This is consistent with the efficacy that Thévenod-Fosse and Waeselynck demonstrated for manually-derived profiles.

We conclude, therefore, that it is viable to derive input profiles (SUT-specific concrete strategies) for statistical testing using automated computational search.

Nevertheless, for one SUT, we observed that algorithm-derived profiles detected fewer faults than uniform random testing at the largest test sizes. We also observed for other SUTs that the fault-detecting ability of candidate input profiles sometimes decreased over some parts of the algorithm trajectory even though the minimum coverage probability increased.

We hypothesised that the cause for both these observations was that the algorithm fitness metric did not contain any measure of diversity, and so the resulting profiles did not generate test sets that exercised coverage elements using as many different inputs as possible. We obtained empirical evidence to support this hypothesis: when a simple diversity measure was added to the fitness metric, the fault-detecting ability of synthesised profiles was improved. For one SUT, however, uniform random testing continued to detect more faults than the profiles derived by the algorithm.

Unaddressed Issues We note the following unaddressed issues:

Performance and Scalability: While the algorithm was shown to be viable, the time taken to derive a near-optimal profile for the largest and complex SUT (`nsichneu`) is, at 10 hours, arguably at the limits of what may considered practical. Therefore the algorithm, as is it currently implemented, is unlikely to be practical for more complex SUTs, or for coverage criteria stronger than branch coverage: both these factors are likely to increase the time taken by algorithm. Since the objective of the empirical work in this chapter was only to demonstrate viability, only a little time was spent in 'tuning' the algorithm. To what degree can changes to the algorithm configuration and tuning the parameter settings improve performance and therefore permit the algorithm to scale to more complex SUTs?

Diversity: We have demonstrated the need for the algorithm to derive diverse profiles, and proposed a change to the fitness metric that was shown to be only partially effective. What changes to the algorithm configuration can further improve the diversity of the derived profiles, and thus the fault-detecting ability of generated test sets?

Applicability: In order to formulate the binned Bayesian representation, a number of assumptions were made as to the nature of the input domain (section 2.2.1). In particular, the input is assumed to be a fixed number of numeric arguments, and the valid domain of each argument is assumed to form a finite interval. It would be easy to extend the

algorithm to input arguments which have non-numeric ordered data types (e.g. single alphabetic characters) as long as the domain of the argument could be mapped to the real-valued interval $[0, 1)$ used internally by the current algorithm implementation (section 2.3) in such a way that ordering, and therefore the interpretation of the bins partioning the argument's domain in the Bayesian network representation, is preserved by the mapping. However, can these assumptions be relaxed further in order to apply the algorithm to a wider range of SUTs, such as those taking variable-length inputs with highly-structured composite data types?

Chapter 3

Construction of an Efficient Algorithm

3.1 Introduction

3.1.1 Motivation

In the preceding chapter, an algorithm for deriving concrete strategies for statistical testing was proposed and evaluated. Although the algorithm was viable, the performance of the algorithm was relatively poor. The algorithm performance is important both in terms of the computing resources required—and therefore the cost-effectiveness of the technique—and in terms of satisfying time constraints that are placed on the testing process in practice. The poor performance currently demonstrated by the algorithm would limit its ability to scale to larger and complex SUTs.

The algorithm used in the empirical work of the preceding chapter was purposely designed on the principle of a 'straightforward' *algorithm configuration*, i.e. the choices of profile representation, fitness metric and search method, and little effort was spent in tuning the *algorithm parameters*. In this chapter, we undertake a more rigorous evaluation of algorithm configurations and parameter settings, with the objective of improving algorithm performance. In addition to proposing and testing new algorithm configurations, some of the design choices made in the preceding chapter are validated.

Performance, in the sense used in the preceding chapter of the time taken to derive a suitable input profile, is a function of not only the algorithm itself, but also its implementation and the nature of the computing environment in which it executes. These additional factors make it more difficult to compare different algorithm configurations and parameter settings in a rigorous and reliable manner. Therefore in this chapter, we use a different performance metric: the number of times the instrumented SUT is executed during an algorithm trial. We refer to this metric as the algorithm *efficiency*. Efficiency is a suitable measure of performance since executing the instrumented SUT in order to evaluate the fitness of a candidate input profile typically accounts for the majority of the algorithm's run time. Since it is a function of only the algorithm configuration and parameters—and not the algorithm's implementation nor the computing environment—efficiency is a more robust metric than the run time.

3.1.2 Contributions

The primary contributions in this chapter are:

- A novel enhancement to the search method, that we call *directed mutation*, which improves efficiency by biasing the mutation operators towards those parts of the representation that appear to have the greatest impact on fitness. On average, directed mutation improves algorithm efficiency by a factor of approximately five.

- The empirical identification of an optimised algorithm configuration that demonstrates the best efficiency, on average, across a range of SUTs. Enhancements to the representation, fitness metric, and search method are proposed and assessed. These enhancements to the algorithm configuration improve its efficiency, on average, by a further factor of approximately two.

- The empirical derivation of common parameter settings for the algorithm configuration that demonstrate the best efficiency, on average, across a range of SUTs.

Secondary contributions related to the empirical method, rather than directly to the research hypothesis, are:

- An implementation of response surface methodology, the optimisation method used to tune the parameters of new algorithm configurations to ensure the reliable comparison of algorithm efficiency. The implementation includes a number of innovations that enable automation, facilitate objectivity, and improve efficacy.
- The use of bootstrapping analysis to quantify how likely any observed difference in algorithm efficiency is real and not the result of the large stochastic variance in the efficiency measurements.

3.1.3 Chapter Outline

Section 3.2 describes the high-level approach to the empirical investigations in this chapter. A key aspect of this approach is the separate treatment of categorical algorithm design decisions (referred to as the algorithm *configuration*) and scalar algorithm *parameters*. To permit the reliable comparison of algorithm configurations, the parameter settings for each configuration are independently tuned using response surface methodology; this approach is described in section 3.3.

The empirical work of this chapter considers two further real-world SUTs in addition to those used in chapter 2; these SUTs are described in section 3.4.

Experiment V (section 3.5) assesses the baseline efficiency of the algorithm prior to applying enhancements.

Experiment VI (section 3.6) evaluates the improvement in algorithm efficiency when directed mutation is used. This experiment also introduces the bootstrapping analysis technique used to estimate the statistical significance of the observed differences in algorithm efficiency in this and later experiments.

In Experiment VII (section 3.7), a further nine enhancements to the algorithm configuration are proposed and assessed. These enhancements are alternative choices for profile representation, fitness metric, and search method. The algorithm configuration is optimised by incorporating the best of these enhancements.

In Experiments V, VI and VII, the algorithm configurations are tuned independently for each SUT. In Experiment VIII (section 3.8), a common set of parameter settings is derived, and the use of common settings is shown to result in a relatively small loss of efficiency compared to SUT-specific settings.

In Experiment IX, (section 3.9), the caching of instrumentation data is proposed as a means of further improving the algorithm performance. It is shown to be viable for some SUTs.

The utility of the empirical results of this chapter are illustrated in Experiment X (section 3.10). The optimised algorithm configuration and common parameter settings are applied to a previously unseen SUT. The improvement in algorithm efficiency for the new SUT is similar to that observed for the example SUTs used in the earlier experiments.

In section 3.11, we reflect on both the outcomes of the research described in this chapter and the empirical method used to optimise the algorithm efficiency.

3.2 Empirical Approach

This section describes the high-level approach to the experimentation performed in this chapter.

3.2.1 Algorithm Configurations and Parameters

We use the term *algorithm configuration* to refer to a particular combination of representation, fitness metric, and search method. For example, the search method used in the preceding chapter was hill climbing, but could also have been simulated annealing or a genetic algorithm. Using either of these alternative choices of search method would result in a new algorithm configuration.

Each algorithm configuration has parameters that control its operation, the number of type of which may differ between configurations. For example, the hill climbing search method used in the preceding chapter had 9 parameters: 6 mutation weights, 2 mutation factors, and the neighbourhood sample size.

The distinction between configuration and parameter is somewhat arbitrary, especially when the parameter takes a small number of categorical values. For example, the algorithm of the preceding chapter used re-evaluation as a method of handling noise in the fitness. If we were to turn off this feature so that re-evaluation is not performed, we may consider this to be a new algorithm configuration. Alternatively, we may think of a single algorithm configuration with a Boolean parameter that controls whether re-evaluation is performed or not.

The empirical work in this chapter utilises *design of experiments* (DOE) techniques in order to tune parameter values. These techniques measure the effect of changing a parameter value on the algorithm response, in this case the efficiency. DOE techniques assume that a meaningful distance metric can be applied to parameter values; typically this assumption limits the data type of the algorithm parameters to be integers or reals. We therefore treat, for the purpose of the empirical work in this chapter, any categorical parameters without a meaningful distance metric as a set of algorithm configurations. Algorithm parameters that act as limits, for example an upper bound on the number of bins in the binned Bayesian network representation, are also considered to be part of the configuration rather than parameters for a similar reason: if the limit is rarely encountered during an algorithm run, then changes to the parameter will have little effect on the response and so DOE techniques will have difficulty in finding an optimal setting for the parameter.

The overall task is to find a choice of configuration, and optimal settings for its parameters, that enable the most efficient algorithm considered on average across a range of SUTs. We split this task into two parts:

1. Propose a set of algorithm configurations, and evaluate the efficiency of each in order to identify the optimal—the most efficient—configuration. (Experiments V, VI and VII.)

2. For the optimal configuration, derive *common* parameter settings that give the best average efficiency over the range of SUTs. Common parameters may either be constants, or settings that may be derived from easily measurable characteristics of the SUT. (Experiment VIII.)

3.2.2 *Configuration Comparison*

We may compare two algorithm configurations, A_1 and A_2, by running each algorithm multiple times for each SUT, measuring the number of times the instrumented SUT was executed during each trial, and using this data to calculate an average efficiency for each configuration.

The question arises as to how to set the algorithm parameters for each of the configuration. One option would be to pick a single set of parameter settings—for example, those used in the preceding chapter—and run both A_1 and A_2 with these same parameters. However, the algorithm efficiency might be very sensitive to the parameter settings and so using the same set of arbitrary parameter settings could lead to unreliable results. For example, the parameter settings may be, by chance, particularly good for A_1 and poor for A_2 with the outcome that A_1 is considered the more efficient algorithm; but other choices for the parameter setting could give the opposite result.

Our solution is to independently tune the parameters for A_1 and A_2, and when comparing the efficiency, run each configuration using its own tuned parameters. This approach is consistent with how the algorithm would be ideally used in practice: with the parameters chosen so that the algorithm is as efficient as it can be. For the purposes of comparing algorithm configurations here, it is not practical to derive *optimal* parameter settings, i.e. those giving the best possible efficiency: given the large number of parameters, guaranteeing the derivation of the optimal settings would require too many computing resources. Instead, a smaller, fixed and practical amount of computation is used to tune the parameters with the aim of deriving representative *near-optimal* settings.

Response surface methodology (RSM), a DOE technique, is used to tune the parameters of the algorithm configuration for two reasons. Firstly, it enables an equivalent amount of computation to be used in tuning each configuration, and thus a fair comparison. Secondly, the process can be automated: by requiring little input from the experimenter the process is more objective. The RSM process—including our innovations that facilitate its automation and improve its objectivity—is described in section 3.3.

3.2.3 *Common Parameter Derivation*

Having identified the most efficient algorithm configuration, a common set of parameters is derived. Since the number of parameters is relatively large, our approach uses another DOE technique: a linear model of the effect of parameter settings on algorithm efficiency. The coefficients of the model are estimated by performing algorithm trials at a sample of parameter settings. The most efficient parameter settings are predicted from the model. This process is described in detail in section 3.8.

3.2.4 *Rationale*

Any approach to optimising the algorithm configuration and parameter settings will require extensive computing resources since there are a very large number of algorithm configurations that could be evaluated, even relatively simple configurations of the algorithm can have nine or more parameters to tune, the stochastic nature of the algorithm requires that it be run multiple times to accurately assess its performance, and there is an infinite number of SUTs to which the algorithm could be applied. The approach outlined above is a pragmatic approach that reduces the resources required by means of the following choices:

- The space of algorithm configurations and the space of parameter settings are treated separately and sequentially: first the configuration is optimised, and then a common set of parameter settings are tuned. If the configurations and parameter settings were optimised simulateously, the space would be unfeasibly large: the dimensionality of the combined space would be the product of the dimensionalities of the separate configuration and parameter spaces.
- The optimisation of the algorithm configuration considers a small sequence of configuration enhancements (chosen from a larger set of potential enhancements) in a defined order. This requires fewer resources than considering all possible combinations of independent configuration enhancements, and is analogous to performing one-factor-at-a-time optimisation rather than applying design-of-experiments optimisation methods such as the use of a factorial design. (In addition to reducing computing resources, the order in which the configuration enhancements are considered can take into account some dependencies between the enhancements: for example, an upper limit on the number of bins is evaluated prior to evaluating whether the number of bins in the initial profile should be set to this limit.)
- Although the algorithm configuration is optimised prior to deriving common parameter settings, the parameter settings of each new algorithm configuration must nonetheless be tuned to permit a fair comparison between configurations. The use of RSM permits the application of limited, but equivalent, resources to this process of tuning each configuration. This 'best-effort' tuning enables a principled comparison to be made without the extensive resources that would be required to find the optimal parameter settings for each configuration.
- Once the best algorithm configuration has been determined, the common parameter settings are estimated using an efficient experimental design that extracts more information from a limited number of design points than would a naive design.
- The algorithm configurations and parameter settings are measured against a small set of four SUTs; the SUTs chosen have relatively diverse characteristics with the intention that the set is representative of the range of software to which the algorithm would be applied in practice (section 3.4).

Although these choices reduce the resources required, there are trade-offs with effectiveness, objectivity, and generality. The first two choices in particular may fail to consider the most optimal combination of configurations and parameter settings, the choice of configuration enhancements and the order in which they are assessed is subjective, and the use of a small set of SUTs may limit the generality of the optimised configuration and parameter settings.

3.2.5 Calculating Average Efficiency

The SUTs used in the empirical work have very different characteristics and so the observed algorithm efficiencies differ greatly, often by an order of magnitude or more. If the average efficiency were calculated using the *arithmetical* mean, it is possible that the average would be dominated by the SUT that requires the largest number of executions.

Instead, the *geometric* mean is used for the average efficiency since it enables an assessment of how the efficiency changes across *all* SUTs without one SUT dominating the value.

As an illustration, let N_i be the efficiency (number of SUT executions) of the algorithm configuration A_1 where the index i indicates the SUT. Assume that A_2 *improves* the efficiency of the algorithm for SUT i by the factor r_i so that the efficiency of A_2 is $\frac{1}{r_i}N_i$.

The average efficiency of configuration A_1 calculated using the geometric mean over the k SUTs is therefore:

$$\overline{N}_{A_1} = \left(\prod_{i=1}^{k} N_i \right)^{1/k} \tag{31}$$

Similarly for configuration A_2:

$$\begin{aligned} \overline{N}_{A_2} &= \left(\prod_{i=1}^{k} \frac{1}{r_i} N_i \right)^{1/k} \\ &= \left(\prod_{i=1}^{k} \frac{1}{r_i} \right)^{1/k} \left(\prod_{i=1}^{k} N_i \right)^{1/k} \\ &= \left(\prod_{i=1}^{k} \frac{1}{r_i} \right)^{1/k} \overline{N}_{A_1} \end{aligned} \tag{32}$$

Therefore,

$$\frac{\overline{N}_{A_1}}{\overline{N}_{A_2}} = \left(\prod_{i=1}^{k} r_i \right)^{1/k} = r \tag{33}$$

Thus, the ratio of geometric mean efficiencies for the two configurations is the geometric mean of the SUT-specific improvement factors r_i. We use r to denote this estimate of the average efficiency improvement as result of the configuration changes in A_2. The calculation of r does not depend on the actual values of N_i and thus avoids the issue of one SUT dominating the estimate.

Initial experimentation showed that the distributions of the random variables N_i (estimated by running the algorithm multiple times against SUT i) are skewed, but that the distributions of $\log N_i$ are more symmetrical. This symmetry enables the mean and median of $\log N_i$ to be used interchangeably as measures of central tendency in the empirical work: many DOE techniques implicitly use the mean, but the median is used elsewhere when robustness to outliers is required. Moreover, the variance of the distribution of $\log N_i$ was similar across different SUTs, and this property of homoscedasticity is an assumption of some DOE techniques. Therefore $\log N_i$ rather N_i is used as the response variable throughout the empirical work. The results are nevertheless reported in terms of N_i for clarity. (The distribution of $\log N_i$ was also found to satisfy tests of normality in many cases, but this property is not relied upon for the analysis.)

Equation (31) may be reformulated as:

$$\log \overline{N}_{A_1} = \frac{1}{k} \sum_{i=1}^{k} \log N_i \tag{34}$$

In other words, the logarithm of the geometric mean efficiency is the *arithmetic* mean of the logarithms of the efficiencies for each SUT. Since DOE techniques often take the arithmetic mean as the average measure, our use of geometric mean and of $\log N_i$ as the response measure are therefore consistent.

3.3 *Response Surface Methodology*

This section explains the use of response surface methodology (RSM) to tune algorithm parameters, and describes the specific implementation of RSM used for the empirical work of this chapter.

3.3.1 *Justification*

As discussed above in section 3.2.2, RSM is used to tune the parameters of algorithm configurations prior to assessing their efficiency.

Since the tuning of parameters is an optimisation problem, one approach would be to apply evolutionary search algorithms to the parameters, resulting in a *meta-evolutionary algorithm*. For example, Grefenstette (1986) uses one genetic algorithm to tune the parameters of another. (To avoid confusion in the following, we refer to such a parameter-tuning algorithm as the 'meta-algorithm' to distinguish it from the search algorithm used to derive input profiles.)

The empirical work of the preceding chapter demonstrated that each trial of the profile-deriving search algorithm can take many minutes, and since the algorithm is stochastic, multiple trials of the algorithm would be required to evaluate its average efficiency for the 'meta' fitness metric. Thus the meta-algorithm would need to be extremely efficient itself in terms of the number of trials of the profile-deriving algorithm it performed. Moreover, the meta-algorithm would need to be reliable—consistently returning near-optimal parameter settings for any configuration of the profile-deriving algorithm—if a fair comparison of algorithm configurations is to be achieved. Such efficiency and reliability of a stochastic meta-evolutionary algorithm may be very difficult to achieve, so we consider alternative, more deterministic, tuning approaches.

One such deterministic approach is the creation of a linear model of algorithm efficiency in terms of algorithm parameters. This technique defines a region of the parameter space and samples the algorithm efficiency at points with the region in order to estimate the model coefficients. The sample points are chosen according to experimental designs that enable the most accurate estimates of the model coefficient with the smallest number of samples. The optimal parameters are predicted from the estimated model.

Previous work has demonstrated that this is a viable technique for tuning metaheuristic algorithm parameters: Ridge and Kudenko (2007) used a linear model to tune the parameters of ant colony systems, and White and Poulding (2009) used this technique to tune the parameters of genetic programs prior to comparison. However, it depends on choosing the region over which the model is derived so that it contains the globally optimal parameters: it is unreliable to extrapolate the model outside the region. Correctly choosing such a region would be difficult for the novel algorithm configurations considered in this chapter without substantial preliminary experimentation.

RSM is an extension of the linear model technique which does not assume that the initial region chosen contains the optimal parameters. Instead a hill climb is performed to other regions of the parameter space, using results from the initial region to guide the climb. This flexibility makes RSM a more suitable tuning technique for the empirical work of this chapter. RSM is used to efficiently optimise industrial processes (Myers and Montgomery, 2005), but has only occasionally been applied to software: one of the few examples is the optimisation of wireless protocol parameters (Vadde et al., 2006).

3.3.2 *Standard RSM*

We outline here the standard form of response surface methodology as described by Myers and Montgomery (2005), and Montgomery (2005), amongst others.

Coded Parameters In RSM, *coded* parameters are typically used instead of the actual parameters. The coded parameters initially take values in the range in the interval $[-1, 1]$ where the values -1 and 1 represent reasonable bounds on the values the actual parameters may take. If the bounds for the actual parameter x_i are L_i and U_i, the transformation from actual to coded parameter values, z_i, is straightforward:

$$z_i = 2\frac{x_i - L_i}{U_i - L_i} - 1 \tag{35}$$

The use of coded parameters is for convenience and to avoid inaccuracies than can arise in the analysis techniques when parameters differ greatly in magnitude. Although deciding what constitutes 'reasonable' ranges for the parameters can be subjective, it does *not* place constraints on the parameters: the coded parameters can take values outside the interval $[-1, 1]$.

Response Surface For an algorithm configuration with n parameters, the *response surface* is the theoretical n-dimensional surface formed by points $(z_1, z_2, \ldots, z_n, y)$, where y is the response—here, the algorithm efficiency—at the parameters $z_1, z_2, \ldots, z_n$. The objective of RSM is to reach the optimum on this surface using a process similar to hill climbing.

The standard methodology is illustrated using figure 17 in which the response surface is indicated by a contour map.

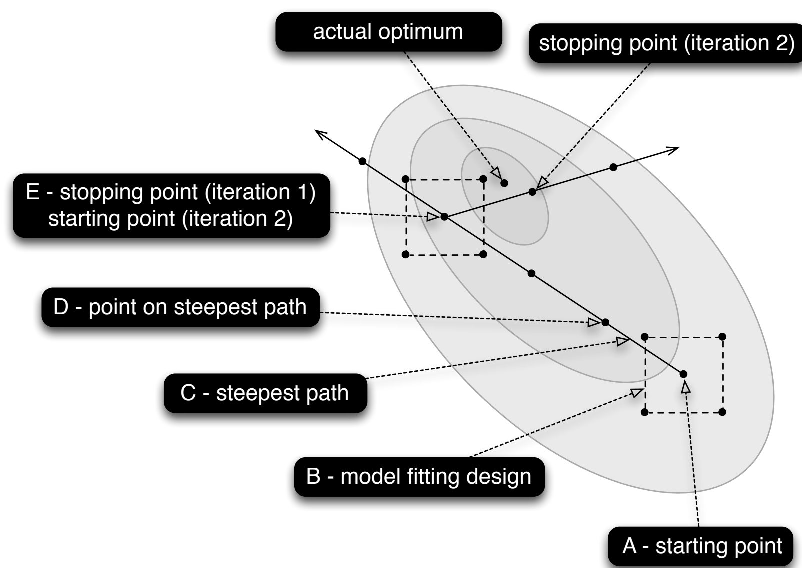


Figure 17 – An illustration of response surface methodology. The height of the response surface is indicated by a contour map, with higher areas in darker grey.

Starting Point A starting point is chosen in the coded parameter space, such as the origin at which parameter values are the centre of the 'reasonable' range of values. This point is

labelled 'A' in the figure. The methodology proceeds by iterative application of the following two phases.

Phase 1 - Slope Estimation In the first phase, algorithm trails are run at a set of parameter design points in a small region around this starting point in order to estimate the gradient of the surface at this point. The design points are typically chosen according to fractional factorial or central composite designs since these designs enable the gradient to be accurately estimated with the fewest design points. The design points are connected by dotted lines in figure 17) and labelled 'B'.

The observed responses of the algorithm trials at the design points is used to fit a first-order linear model:

$$Y = \beta_0 + \sum_{i=1}^{n} \beta_i z_i + \epsilon \tag{36}$$

In this model, Y is a random variable representing the response, z_i are the coded algorithm parameters, β_0 and the β_i are the unknown model parameters that are estimated by the analysis of the observed responses, and ϵ is a 'noise' or 'error' term that accounts for the variance in the algorithm response. For the algorithms considered in this chapter, this variance is entirely due to the stochastic nature of the algorithm.

The region sampled around the starting point is assumed to be sufficiently small that a linear model gives a reasonable approximation of the surface at this point, and so that the following vector gives an estimate of the direction of the steepest slope *up* the hill on the response surface:

$$(\beta_1, \beta_2, \ldots, \beta_n) \tag{37}$$

The path in this estimated direction of the steepest slope is labelled 'C' in the figure.

Phase 2 - Hill Climb In the second phase, algorithm trails are run at series of equally spaced points ('D' in the figure) along a path from the starting point in the direction of the steepest slope vector. The path is followed until no further improvement in algorithm response is observed. Usually a simple stopping rule is applied, such as two successive points along the path showing no improvement.

Iterations and Termination The vector defined by equation (37) is an *estimate* of the direction towards the optimum, and the further away from the starting point, the less accurate it will be. Therefore, the stopping point on the the path is unlikely to be precisely at the optimum on the surface, but may be, for example, on a 'ridge' to one side of the optimum. For this reason, the stopping point becomes the starting point (labelled 'E') for a further iteration of the two phases in order to get closer to the optimum on the surface.

Iterations of these two phases—slope estimation and hill climb—are performed until analysis suggests that the first-order linear model is a poor fit to the observed responses owing to curvature in the surface. This is indicative of reaching the 'summit' of the hill where the optimum lies. At this stage, a second-order linear model is often fitted in order to accurately estimate the optimum point on the curved surface near summit.

3.3.3 *Implementation and Enhancements*

This section describes the implementation of RSM used for the empirical work of this chapter. It also describes some innovative enhancements to the standard technique that facilitate the

use of RSM for this particular application: tuning the parameters of a stochastic algorithm. The implementation details and enhancements are based on experience gained during preliminary experimentation, and were motivated by the need to accommodate the stochastic variance in the efficiency of the profile-deriving algorithm, to ensure the objectivity of the methodology, and to enable automation of the process.

Terminology In the following, an *algorithm trial* refers to a single execution of the algorithm, using a defined specific set of parameter values, and returning a single response, i.e. a measure of the algorithm efficiency. A *tuning run* refers to a single application of RSM, consisting of many algorithm trials, that returns a single set of tuned algorithm parameters. The term *experiment* will be reserved for a set of tuning runs that compare the performance of different algorithm configurations.

Coded Parameter Space For some algorithm parameters, the algorithm behaviour may be reasonably be expected to be related to the *ratio* of the change rather than the absolute difference in parameter settings. For example, a change in K, the evaluation sample size, from 10 to 20, is likely to have more effect than a change from 110 to 120. In addition these, and other parameters, may differ by orders of magnitude between different SUTs.

For these reasons, it is therefore convenient to relate the coded value of such parameters to the *logarithm* of the actual parameter values. Equation (35) is adapted for these parameters as:

$$z_i = 2\frac{\log x_i - \log B_i}{\log U_i - \log L_i} - 1 \tag{38}$$

(where L_i and U_i are 'reasonable' bounds on the actual parameter values, as discussed above). The use of logarithm maps an absolute change in the coded parameter space to multiplicative change in the actual parameter space. This is a more convenient mapping for parameters where algorithm performance is related to the relative change, and allows parameter values to range more easily across orders of magnitude during the tuning.

Trial Termination Criteria An individual algorithm trial terminates if a profile is derived whose fitness reaches or exceeds the target minimum coverage probability, $\tau_{p_{\min}}$.

Two additional termination criteria are used to avoid long-running jobs. This is a practical consideration: occasionally, some algorithm configurations with particular parameter choices can produce large data structures that are very slow to process. The additional termination criteria are specified as limits on:

- the number of times an instrumented SUT is executed during a trial (parameter τ_{exec}); and,
- the processor time used by the trial (parameter τ_{time}).

If either limit is reached, the algorithm terminates cleanly, and the accumulated number of executions and fitness at point of termination are recorded.

Termination based on the number of executions is compatible with repeatable experimentation: it is dependent only on the algorithm configuration and the SUT. However, termination based on processor time is less repeatable: the computing environment in which the trial executes is an additional factor. For this reason, τ_{time} is set so that is exceeded only in exceptional cases. (The actual settings used for both of these limits is specified during the description of each experiment in this chapter.)

Starting Point RSM is a form of hill climbing that locates a local optimum reached by a climb from its starting point. There is no reason to assume that this local optimum is the global optimum: the response surface may be rugged with many local optima. To increase the probability of locating the global optimum, or at least a 'good' local optima, multiple tuning runs are performed, each time starting at a different, random starting point.

The starting points are chosen by randomly sampling each coded parameter from the interval $[-0.5, 0.5]$ using a uniform distribution. By selecting the point from a large space around the centre of the 'reasonable' region for parameter values, there is a chance that the hill climb could start in the basin of attraction of the global optimum (or a 'good' local optimum).

Response To accommodate censored observations, i.e. trials that terminate as a result of the two limits described above before a suitable input profile is derived, statistical techniques such as the Tobit model (Tobin, 1958) or Cox's Proportional Hazards Model (Cox, 1972) could be applied during the analysis. We have previously successfully applied such techniques to the tuning of search algorithm performance (Poulding et al., 2007). However both these techniques make assumptions as to the distribution of the responses. In order to avoid these assumptions, we instead incorporate information from censored observations through the use of the following modified response metric:

$$y = \log \left(N \frac{\tau_{p_{\min}}}{\min(\widehat{p}_{\min}, \tau_{p_{\min}})} \right) \tag{39}$$

where N is the number of times the instrumented SUT was executed during the trial (i.e. the original response measure), $\tau_{p_{\min}}$ the target minimum coverage probability, and $\widehat{p}_{\min}$ the fitness of elite input profile, i.e. the candidate profile with highest estimated minimum coverage probability found during the algorithm trial.

If the algorithm completes and an input profile was derived that has target minimum coverage probability, then the response, y, is the original measure of efficiency, $\log N$. If the trial terminated for any other reason, then the number of SUT executions at the point of termination is increased by a proportion dependent on how close the algorithm was to achieving the target fitness. This is *not* intended to be an estimate of the number of SUT executions that would be made by the algorithm trial if it had run to completion, but simply as a means of guiding the tuning process by incorporating information from censored observations. (In the very rare cases that $\widehat{p}_{\min}$ is zero, an arbitrary response of $y = 22$—equivalent to $e^{22} \approx 3.58 \times 10^9$ SUT executions—is recorded. This value is much higher than normally possible, but not so high as to distort the analysis.)

Preliminary experiments demonstrated that the limit on the number of executions was set sufficiently high that although some algorithm parameter choices resulted in a relatively large proportion of censored observations at the initial parameter settings, very few, if any, trials were censored at the final parameter settings. For this reason, we may be confident that tuned parameters derived using RSM are indeed very efficient, rather than being an artefact of this modified response.

Design for Linear Model Fitting A two-level fractional factorial design is used for the design points at which algorithm trials are run in order to estimate the model parameter in the first phase of each RSM iteration. If the starting point is $(z_1^*, z_2^*, \ldots, z_n^*)$, then the design is a subset of the points formed by combinations of coordinates $z_i^* \pm 0.1$. The value of 0.1 was chosen

with the intention that a first order-linear model would be a reasonable approximation of the surface in this small region, but large enough that there is sufficient difference in algorithm performance (compared to the intrinsic stochastic noise) across the region to establish a steepest path on the surface.

The number of parameters varies depending on the algorithm configuration (the combination of search method, representation and fitness metric), and this would normally affect the size of the design. In order to apply equivalent computation to the tuning of each configuration, we generate the smallest fractional factorial design with a minimum of 128 points and a *resolution* of 4 (or better), regardless of the number of parameters. The resolution refers to the degree with which the coefficients of higher order terms cannot be determined through regression analysis using such a fractional factorial design. A resolution of 4 is the least which ensures that the coefficients in the first-order linear model may be accurately determined, and is much smaller (and therefore requires less computation) than a full two-level factorial design for the same number of parameters.

Steepest Path Descent The steepest path is explored at points separated by the vector:

$$\frac{-0.2}{\sqrt{\sum_{i=1}^{n} \beta_i}}(\beta_1, \beta_2, \ldots, \beta_n) \tag{40}$$

Here the steepest path vector of equation (37) is normalised and multiplied by a scalar value. Since better algorithm parameters have lower values of the response metric defined by (39), the steepest path should *descend* and so this scalar is negative. The magnitude of 0.2 was found to be effective during preliminary experimentation.

To assess the average algorithm efficiency at points on the steepest path, 32 trials of the algorithm are performed, each trial using the same parameter settings but a different seed to the pseudo-random number generator. The efficiency is calculated as the median of the response (equation (39)) of each trial. The median is used as it is more robust to outliers than the mean.

We introduce an innovation in how the coded parameters are converted to the actual parameters at points on the steepest descent path. Some of the actual parameters take only integer (or similarly discrete) values. The process of transforming coded values to actual values and then rounding to the nearest integer can cause inconsistencies in the differences between parameter values at consecutive points. For example, the actual values, before rounding, for a parameter at five consecutive points along the path may be: $3.1, 4.3, 5.5, 6.7, 7.9$. When rounded to the nearest integer, the parameter values are: $3, 4, 6, 7, 8$, resulting in an inconsistent difference of 2 between the second and third points compared to 1 between the other points. These inconsistencies might result in sudden changes in the response metric, and since our chosen descent stopping criteria (described below) assumes a smooth curve can be fitted to the response along the steepest path, the sudden change could lead to a poorly fitted curve and therefore unreliable results.

This potential source of error is minimised by applying 'probabilistic' rounding when deriving the actual parameter values for points on the steepest path. If the fractional part of the unrounded parameter value is g, then the value is rounded up to the next integer if $\gamma < g$, where γ is a random value uniformly sampled from the interval $[0, 1)$, and rounded down otherwise. This rounding rule is applied independently in each of 32 trials, so that the mean value of this parameter considered across the trails is close to the value before rounding, but each individual trial uses an integer value. Assuming an approximately linear relationship

between response and this parameter value, the median response will be an estimate of that at the unrounded parameter value, and even without a linear relationship, it will almost certainly be a closer approximation than if the direction of rounding were the same for all 32 trials.

Descent Stopping Criterion In most applications of the RSM, the steepest path is followed until a simple stopping criterion is met, such as two consecutive points on the path giving a worse response than the previous point. During preliminary experimentation it was found that a simple criterion such as this was unreliable as a result of the stochastic noise in the algorithm response: occasionally the median responses at a point was slightly worse than at the previous point, but the overall trajectory of the response was still towards better response.

For this reason, a second innovation was made to the methodology: the stopping criterion was based on a smooth curve fitted to the median responses at points along the path. The process of curve fitting incorporates information from all points along the path, and is therefore robust to the occasional outlier value for the median response.

The fitted curve is a cubic form:

$$\tilde{y} = as^3 + bs^2 + cs + d \tag{41}$$

where $\tilde{y}$ is the median response and s the number of steps along the path from the starting point. A cubic curve accommodates both a minimum and maximum along the path, and was found to produce a good fit to observed results during preliminary experimentation: an example is shown in figure 18. The coefficients a, b, c, and d are estimated using least-squares regression.

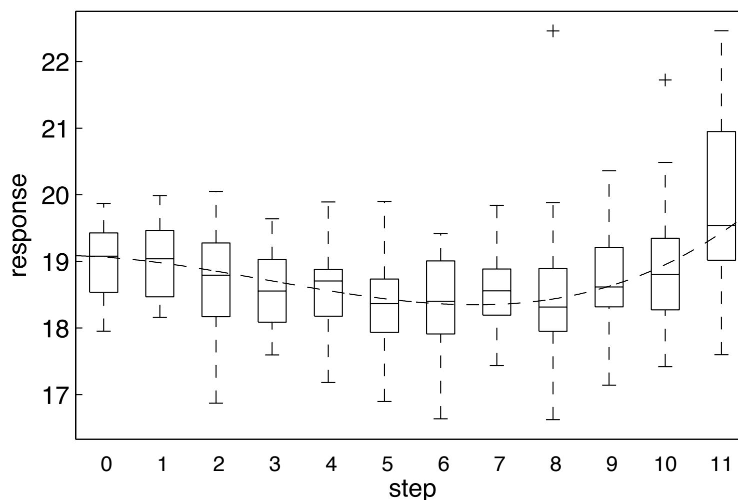


Figure 18 – An example of a cubic curve (dashed line) fitted to responses along the steepest path. At each step along the path, a boxplot illustrates the distribution of observed algorithm efficiencies over 32 trials: the box is drawn between the first and third quartiles, and the horizontal line across the box is the median response. (The response is the metric of equation 39, equivalent to the logarithm of the number of SUT executions.)

Initially the path is followed for a total of 8 steps—the starting point plus 7 steps along the path—and the cubic fitted to the observed median responses. If the fitted cubic has a minimum, and this predicted minimum is at least two steps from the last point sampled (i.e. it is a true minimum and not simply the lowest point sampled so far) then the descent of

the path stops. Otherwise subsequent batches of 4 steps are taken up to a maximum of 21 steps: the starting point plus 20 points along the path. The limit on the number of steps is used since the vector is unlikely to still be a good estimate of the steepest path so far from the starting point. (20 steps represents a distance travelled in the coded parameter space of 10 units, but is otherwise an arbitrary limit.)

Path descent may also stop at fewer than 20 steps if any of the parameters reaches a hard bound on its value, for example, a parameter that must be positive would become negative at the next step. In this case, a minimum is estimated that leaves sufficient 'space' for the fractional factorial design in the first phase of the next RSM iteration.

As well as, we believe, accommodating the noise in algorithm responses more effectively than a simpler stopping criterion, the curve fitting criterion has an additional advantage: the minimum predicted by the fitted curve can lie between the sampled points, potentially resulting in a better estimation of the minimum.

Iterations When RSM is applied to, for example, industrial processes, there is significant benefit in performing many iterations in order to find the best possible settings for the process. However the goal here is find only *near-optimal* parameter settings for the purpose of reliably comparing algorithm configurations; finding the *best* possible settings is not necessary and would require too many computing resources. For this reason, the RSM process applies just two iterations of hill climbing; preliminary experimentation suggested that further iterations typically produced little improvement in the response. Always performing two iterations also ensures that equivalent effort is expended in tuning each algorithm configuration.

A second-order linear model is *not* fitted after that last iteration because fitting such a model to a noisy response and a relatively large number of parameters is unlikely to be reliable, and is unnecessary since only *near-optimal* settings are required. Instead, the minimum point estimated during the steepest path descent of the second iteration is used as the tuned algorithm parameters.

Automation The configuration and analysis of each RSM phase is performed using a combination of shell and MATLAB scripts. The algorithm trials and the scripts implementing RSM run on a computing cluster. Each server in the cluster has Linux as the operating system, and the cluster is managed using Oracle Grid Engine software. A single tuning run, consisting of all the phases of RSM, executes without input by the experimenter: at the end of each phase a grid job is automatically submitted to perform the next phase.

3.4 *Software Under Test*

This section describes the SUTs used in the experiments of this chapter. A threat to validity identified for the experiments of chapter 2 was the relatively small number of SUTs. Therefore two new real-world SUTs, `cArcsin` and `fct3`, are added to the existing SUTs `bestMove` and `nsichneu`. The very simple toy SUT `simpleFunc` is no longer used as an example in the experiments.

Ideally, the set of example SUTs would be larger, but further SUTs would add to the computing resources required to the perform the experiments. The set of four diverse SUTs is a trade-off between the generality of the empirical results and the available computing resources.

The characteristics and provenance of the new SUTs are described below. The instrumented source code of `cArcsin` is available at: `http://www-users.cs.york.ac.uk/smp/supplemental/`; only the object code of `fct3` was provided to us and thus we are unable to share the source code for this SUT.

3.4.1 *SUT Characteristics*

Relevant characteristics of the new SUTs are summarised in table 7. The characteristics of the SUTs `bestMove` and `nsichneu`, previously described in chapter 2 are shown for comparison.

SUT	`bestMove`	`nsichneu`	`cArcsin`	`fct3`
lines of code	89	1 967	70	135
no. loops	7	1	0	0
no. conditional branches	42	490	18	19
no. input arguments	2	11	3	8
argument data types	`int`	`int`	`double,bool`	`int`
domain cardinality	2.62×10^5	3.40×10^6	∞	6.71×10^7

Table 7 – Characteristics of the SUTs `bestMove`, `nsichneu`, `cArcsin`, and `fct3`.

As for the existing SUTs, input profiles are derived for the purpose of branch coverage of the new SUTs.

3.4.2 *cArcsin*

This function returns the arcsin of a complex number and is adapted from the GNU Scientific Library (Galassi et al., 2009). The arguments to the function are the real and imaginary parts of the complex number, and are passed as double-precision floating point numbers. For testing purposes, we restrict each of these two arguments to the semi-open interval $[-10, 10)$. The function behaves very differently when the imaginary argument is zero, and since the chance of this occurring during random sampling from the floating point interval $[-10, 10)$ is essentially zero, a third Boolean-valued argument is introduced into the representation of the input profile (the SUT itself is not changed). When the value of this argument is true, the imaginary argument passed to the SUT is set to zero.

This function differs from the other SUTs in that it has real-valued arguments, and therefore an input domain of, effectively, infinite cardinality.

3.4.3 *fct3*

This function is a component in a nuclear reactor safety shutdown system. It was used by Thévenod-Fosse et al. (1991) in an evaluation of the fault-detecting ability of statistical testing compared to traditional random and structural testing approaches.

This function differs from the other integer-valued SUTs in that the cardinality of the input domain is an order of magnitude larger: it has approximately 6.71×10^7 different inputs. As for `nsichneu`, the relatively large number of input arguments may present a challenge for the algorithm in that many dependencies between arguments may be need to represented.

3.5 Experiment V – Baseline Algorithm Efficiency

3.5.1 Objectives

This experiment assesses the efficiency of an algorithm configuration equivalent to that used in Experiments I–IV of the preceding chapter. The efficiency of this configuration provides a baseline against which other configurations may be compared.

The primary research question is:

RQ-1 What is the average efficiency, i.e. the number of SUT executions required to derive profile, of the baseline configuration?

We also validate the use of RSM as the parameter tuning method:

RQ-2 To what extent does the use RSM improve the algorithm efficiency by tuning the parameters?

3.5.2 Preparation

The baseline configuration evaluated in this experiment is similar to that the algorithm used in chapter 2, with the following modifications.

1. The fitness metric is that used in Experiments I–III, i.e. without a diversity measure. There are two reasons to omit the diversity measure. Firstly, its use in Experiment IV was shown to be only partially successful in improving fault-detecting ability. Secondly, the maintenance of the data structure used to calculate the diversity measure was found to greatly increase the time taken by the algorithm.

2. A limit of 10 is placed on the number of re-evaluations (section 2.2.3) of a single candidate input profile. Although re-evaluation improves the accuracy of the estimate of minimum coverage probability, it can require the maintenance of a very large data structures. (This issue of memory usage is in addition to the performance impact of the diversity measure discussed above.) By placing a limit on the number of re-evaluations, large data structures are avoided, but since the incremental gain in accuracy decreases for each subsequent re-evaluation, the loss of accuracy is likely to be small.

3. A limit is placed on the number of bins for each argument: 4 for integer-valued SUT arguments where the domain size is 4 or less, and 32 for real-valued arguments and integer-valued arguments with a domain size greater than 4. (In error, the third argument of `cArcsin` was given a limit of 32 bins, even though as a Boolean argument, a limit of 4 is more appropriate.) In the experiments of chapter 2 the choice of mutation weights prevented an excessive number of bins, and therefore avoided the data structures representing conditional probabilities becoming too large, but in the empirical work of this chapter the mutation weights change freely and so parsimony can no longer be encouraged by this method.

4. For the SUTs `bestMove` and `nsichneu`, the target minimum coverage probability is 80% of the target used in Experiment I. The results of Experiment III suggested that input profiles with a minimum coverage probability lower than optimal detected almost as many faults as optimal profiles. Since the derivation of a profile with a lower target minimum coverage probability requires fewer computing resources, such a lower target could be set by test engineer when the algorithm is used in practice.

The first three of these modifications improve the algorithm run time by avoiding large data structures that are manipulated slowly, and therefore enable more algorithm trials to be run given the same computing resources. We expect these modification to have only a minor effect on the algorithm efficiency measured in terms of the number of SUT executions. However, the change to the target minimum coverage probabilities *will* change the efficiency as well as the run time. Since all algorithm configurations considered in this chapter use the same targets, a reliable comparison of efficiencies can still be made.

Modifications (2) and (3) introduce limits on the number of re-evaluations and the number of bins. As discussed in section 3.2.1, it is convenient to treat these limits as configuration options rather than algorithm parameters. The values chosen for these limits are validated in Experiment VII.

We denote this algorithm configuration A_{base}. The parameter settings for A_{base} are listed in table 8.

Parameter	Effect	SUT	Value
$x_1 = \log(K)$	evaluation sample size		6.0 – 8.0
$x_2 = \lambda$	neighbourhood sample size		2 – 22
$x_3 = \log(\rho_{\text{prb}})$	bin probability mutation factor		0.01 – 3.00
$x_4 = \log(\rho_{\text{len}})$	bin length mutation factor		0.01 – 3.00
w_{prb}	M_{prb} mutation weight		1 000
$x_5 = \log(w_{\text{rem}}/w_{\text{prb}})$	M_{rem} mutation weight		−0.5 – 0.5
$x_6 = \log(w_{\text{add}}/w_{\text{rem}})$	M_{add} mutation weight		−0.75 – 0.25
$x_7 = \log(w_{\text{joi}}/w_{\text{prb}})$	M_{joi} mutation weight		−0.5 – 0.5
$x_8 = \log(w_{\text{spl}}/w_{\text{joi}})$	M_{spl} mutation weight		−0.75 – 0.25
$x_9 = \log(w_{\text{len}}/w_{\text{prb}})$	M_{len} mutation weight		−0.5 – 0.5
τ_{exec}	max. no. SUT executions		1×10^8
τ_{time}	max. processor time (s)		3 600
$\tau_{p_{\text{min}}}$	target min. coverage probability	`bestMove`	0.112
		`nsichneu`	0.028
		`cArcsin`	0.12
		`fct3`	0.05

Table 8 – Algorithm parameters for configuration A_{base}.

The parameters listed as x_i are those that are tuned using RSM. The column 'Value' is the interval of actual parameter value corresponding to the interval $[-1, 1]$ of the coded parameter. If the parameter is the logarithm of the natural interpretation of the parameter, this interval is expressed in terms of the logarithm value.

The mutations weights are interpreted by the algorithm relative to the total of the weights and so the RSM linear model would be over-parameterised if 6 parameters were used to represent the 6 weights. One solution is to use a *mixture model*, such as the canonical forms pioneered by Scheffé, in place of the standard linear model, and a simplex-lattice experimental design instead of a fractional factorial design (Scheffé, 1958). (We have previously demonstrated a successful application of such an approach (Poulding et al., 2007).)

However, it would difficult to incorporate the non-weight parameters into such a mixture model and so an alternative parameterisation of the weights is used: one weight, w_{prb}, is kept constant and 5 parameters represent the remaining weights.

The 5 weight parameters are ratios of the mutation weight to w_{prb}, apart from w_{add} and w_{spl} which are expressed as ratios of the corresponding 'reverse' operators, w_{rem} and w_{joi}. This formulation of w_{add} and w_{spl} enables parsimony to be encouraged (at least in the initial settings) by biasing the coded interval for these parameters to values less than 1.

3.5.3 *Method*

The RSM tuning procedure was applied three times for each SUT. Each tuning run started from different random parameter settings (from the the interval $[-0.5, 0.5]$ of each coded parameter, as described in section 3.3.3), and used a different set of pseudo-random number generator seeds for each algorithm trial.

Once the RSM procedure was complete, 64 algorithm trials were performed for each SUT in order to assess the efficiency of the tuned parameter settings.

If the response surface does not have a single optimum but is instead rugged with many local optima, the use of multiple RSM tuning runs provides more information about the surface than would a single tuning run. Although none of the runs will necessarily locate the global optimum, the sample of three local optima provides information about the distribution of heights of the local optima, and this information will be useful when comparing different algorithm configurations.

3.5.4 *Results*

The results are summarised in table 9. For each tuning run, the median efficiency is reported 'Before Tuning', i.e. at the random starting point, and 'After Tuning', i.e. at the near-optimal parameter settings predicted by the RSM. The average efficiency calculated as described in section 3.2.5 from the *best* efficiency observed for each SUT. The efficiencies are expressed in multiples of 10^6 executions of the instrumented SUT.

SUT	Efficiency (10^6 executions)	
	Before Tuning	After Tuning
`bestMove`	151	136
	139	120
	86.0	108
`nsichneu`	20.0	8.86
	23.0	5.77
	17.1	22.7
`cArcsin`	32.8	11.1
	28.4	5.31
	21.6	9.18
`fct3`	34.0	34.8
	40.1	15.9
	84.7	28.4
Average	32.3	15.1

Table 9 – Results of Experiment V showing the median efficiency before and after each of the three tuning runs per SUT, and the average efficiency of A_{base} calculated using the best efficiency observed for each SUT.

3.5.5 *Discussion and Conclusions*

The average efficiency of the baseline configuration is 15.1×10^6 executions (RQ-1).

If the efficiency response surface has a single (and therefore global) optimum, the results for each of three tuning runs for a particular SUT should be similar even though the starting points differ. However, there is significant variation: in the worst case, `nsichneu`, the lowest and highest efficiency differ by a factor of almost 4. We conclude that the response surface is

either rugged with multiple local optima, or that the RSM procedure is ineffective at finding the optimum.

For `bestMove` and `nsichneu`, the number of executions (108×10^6 and 5.77×10^6, respectively) is larger than SUT's domain cardinality (0.262×10^6 and 3.40×10^6). Thus systematically executing the SUT with each possible input would be a more efficient method of deriving a suitable input profile than the baseline algorithm configuration. We address this issue later in the chapter.

A comparison of the average efficiency before and after tuning shows that RSM improves the efficiency by a factor of more than 2 (RQ-2). Surprisingly, the use of RSM has resulted in a relatively large *decrease* in efficiency during the third runs for `bestMove` and `nsichneu` (and a small decrease during the first run of `fct3`). We speculate this is caused by the large variance in the algorithm response leading to estimates made during the RSM process that prove to be inaccurate. Nevertheless, for many runs the improvement in efficiency is substantial, and the use of multiple tuning runs for each SUT reduces the impact of such anomalous tuning runs.

3.5.6 *Threats to Validity*

Generality Although more SUTs are used than in Experiments I–IV, they do not necessarily form a representative sample of SUTs to which the algorithm may be applied. Therefore the efficiency figure must be considered as being specific to the set of four SUTs chosen.

Efficiency Variance As discussed above, there is significant variation in the efficiency estimated by each tuning run. Performing additional tuning runs would reduce the uncertainty resulting from this variability. However, since each tuning run can involve more than a thousand algorithm trials, more than three repetitions was not practical given the available computing resources. We note, however, that the average efficiency is calculated from the best efficiency observed for each of the four SUTs, and therefore a relatively large set of 12 tuning runs are used in determining the efficiency.

Starting Point Since the results are indicative of a response surface with many local optima, and RSM is a hill-climbing method, the starting point determines which optimum is located by the tuning run. Although the starting points are sampled at random from a large region of the parameter space—the region equivalent to the interval $[-0.5, 0.5]$ of each coded parameter—the choice of this region could bias the result. As the region is based on parameters found to produce acceptable results for the configuration used in the empirical work of chapter 2, the bias is likely to be towards similar configurations—such as the baseline configuration considered here—and away from novel configurations.

Target Minimum Coverage Probability The efficiency observed is specific to the choice of targets for the minimum coverage probability. If the relationship between efficiency and target minimum coverage probabilities is different for each algorithm configuration, the result of the comparison may be dependent on the choice of target values: one configuration may be the more efficient at one choice of targets, and the other configuration at a different choice of targets. However, the target minimum coverage probabilities used in this experiment are justified above as being realistic: they generate test sets with good fault-finding abilities but do not require as much time to derive as near-optimal profiles.

3.6 Experiment VI – Directed Mutation

3.6.1 Objectives

In this experiment, we propose an enhancement to the algorithm configuration, which we call *directed mutation*, and assess the improvement in the efficiency as a result of the change. We denote the new algorithm configuration A_{dmut}.

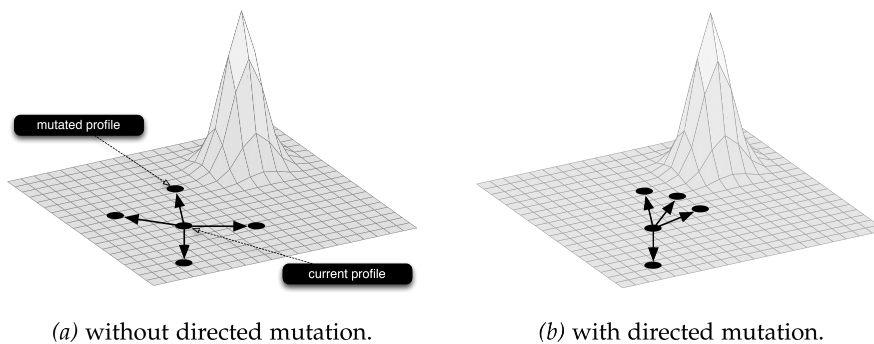
The research question addressed is:

RQ-1 Is average efficiency of the configuration A_{dmut} significantly better than that of the baseline configuration, A_{base}?

3.6.2 Motivation

During the empirical work in chapter 2, it was observed that for some SUTs, one or more of the coverage elements are only very rarely exercised by the uniform distribution. Since the initial input profile is the uniform distribution, the fitness metric of this initial profile will often be zero and the fitness landscape is essentially flat in the region from which the hill climb starts. Such a landscape provides very little guidance to the search, and we speculate that the algorithm can spend many iterations performing a random walk on this flat region before locating a region that has a gradient which can guide the search towards a solution.

The proposed solution is to more effectively utilise the chance executions of the most rarely exercised coverage elements through the use of *directed mutation*. During the fitness evaluation, the bins in the Bayesian network representation that gave rise to inputs which exercise the least-exercised coverage elements (i.e. those elements having the minimum coverage probability) are recorded. In the next hill climbing iteration, the mutation operators are biased, or '*directed*', so that mutations affecting these bins have a higher probability of occurrence. In this way, the sample of neighbours considered at each iteration of the hill climb is more likely to contain candidate input profiles that more frequently exercise the least-exercised elements.



(a) without directed mutation. *(b)* with directed mutation.

Figure 19 – An illustration of the intended effect of directed mutation.

The intended effect of directed mutation during one iteration of the search algorithm is illustrated in figure 19. It shows a fitness landscape of which a substantial proportion is flat with a fitness value at or near zero. The current candidate input profile is indicated with a small circle and is connected by arrows to a sample of four neighbours derived by mutation. Since the fitness landscape is essentially flat, the fitness metric provides little guidance as to which neighbour should become the current distribution at the next iteration: any of the neighbours is equally likely to be chosen.

In (a), the mutation operators sample neighbours without any direction. Although the fitness metric is normally zero in this region, the least-exercised coverage elements are occasionally exercised by chance during a fitness evaluation. In (b), the data from these chance occurrences is used to bias the choice of mutations made and thus the neighbours sampled at each iteration. The intention is that this bias guides the search towards regions of the landscape where there is sufficient gradient in the fitness metric to enable the search to find a near-optimal profile.

Directed mutation is thus providing a second mechanism, in addition to the fitness metric, that feeds back the data provided by dynamic execution of the SUT to the search algorithm.

A similar concept of directed (or targeted) mutation operators has been applied, for example, to the construction of timetables using memetic algorithms (Ross et al., 1994; Paechter et al., 1996). In these studies, the mutation operator is directed towards those parts of the representation that have the potential to resolve violations of timetabling constraints in the current individual. The infected genes evolutionary algorithm (igEA) of Tavares et al. (1999) is a generalisation of this approach: information from phenotypic evaluation is used to 'protect' desirable genes in the representation from variation by mutation and crossover.

Directed mutation is distinct from *adaptive mutation*. In most forms of adaptive mutation, such as those discussed by Thierens (2002), the *overall* mutation rate is modified—over time, based on the recent changes in fitness, or by self-adaption as part of the representation—but the mutation operator itself is unchanged. In the form of *directed mutation* proposed here, the overall mutation rate is unchanged; instead the mutation operator is dynamically modified in order to bias mutations towards particular parts of the representation.

3.6.3 Preparation

Implementation Directed mutation is implemented through the use of *mutation groups*. Each of the mutation operators is assigned to one or more groups, denoted G, and each group has an associated weight, W. The random selection of an atomic mutation now becomes a two stage process. In the first stage, one of the mutation groups is chosen at random according to the group weights; this occurs without regard to the number of atomic mutations possible using operators in the group. The second stage is to randomly select, as before, an atomic mutation using the process described in section 2.2.3, but considering only those mutations enabled by mutation operators belonging to the chosen mutation group.

Three mutation groups are used:

G_{edge} consists of the operators that modify edges in the Bayesian network: M_{add} and M_{rem};

G_{bins} consists of the remaining operators that modify bins directly: M_{len}, M_{spl}, M_{joi}, and M_{prb}, and applies to *all* bins;

G_{drct} consists of the same bin-modifying operators as G_{bins}, but applies them *only* to bins that contributed inputs that exercised the least-exercised coverage element.

It is the last group, G_{drct}, that implements directed mutation through restricting the mutation operators to a subset of the bins. Bins associated with the least-exercised coverage elements may be mutated by operators in *both* the G_{bins} and G_{drct} groups, while other bins are mutated only by operators in the G_{bins} group. Thus, bins associated with the least-exercised coverage elements are mutated more frequently, and the degree to which this occurs is controlled by the relative weights of the G_{bins} and G_{drct} groups.

The initial selection by mutation group weight, regardless of the number of atomic mutations possible using operators in the group, ensures that direct mutation occurs at a rate that is independent of the number of bins.

Parameters The parameters that differ in the new configuration A_{dmut} are listed in table 10. All other parameters are as for A_{base}, and listed in table 8.

The mutation group weight parameters are handled in a similar manner to those for mutation weight: one group weight is fixed and the others are expressed as ratios of the fixed group weight. The use of mutation *group* weights therefore adds to the number of parameters to be tuned. However, since mutation *operator* weights are now only relative to other operators in the same group, some of these operator weights can be fixed (in the same way as w_{prb} was in Experiment V) and this reduces the number of parameters that are tuned. Overall, A_{dmut} has only one more parameter to be tuned than A_{base}.

Parameter	Effect	Value
W_{bins}	G_{bins} mutation group weight	1 000
$x_5 = \log(W_{\text{edge}}/W_{\text{bins}})$	G_{edge} mutation group weight	−3.0 – 0.0
$x_6 = \log(W_{\text{drct}}/W_{\text{bins}})$	G_{drct} mutation group weight	−2.0 – 2.0
w_{prb}	M_{prb} mutation weight	1 000
w_{rem}	M_{rem} mutation weight	1 000
$x_7 = \log(w_{\text{add}}/w_{\text{rem}})$	M_{add} mutation weight	−0.75 – 0.25
$x_8 = \log(w_{\text{joi}}/w_{\text{prb}})$	M_{joi} mutation weight	−0.5 – 0.5
$x_9 = \log(w_{\text{spl}}/w_{\text{joi}})$	M_{spl} mutation weight	−0.75 – 0.25
$x_{10} = \log(w_{\text{len}}/w_{\text{prb}})$	M_{len} mutation weight	−0.5 – 0.5

Table 10 – Algorithm parameters for configuration A_{dmut}.

3.6.4 *Method*

The empirical procedure was the equivalent to that of Experiment V: three tuning runs were performed for each SUT using A_{dmut}, and the efficiency of the tuned parameter settings predicted by each tuning run assessed by a further 64 trials of the configuration.

It is possible that the use of mutation groups, rather than directed mutation itself, may be the cause of any improvement observed in the efficiency of the algorithm. For example, it is possible that the random selection of mutation groups independent of the number of possible atomic mutations may be beneficial when the size of the representation varies throughout the course of the search. To exclude this explanation, we also applied the same procedure to a configuration A_{mgrp}, which omits G_{drct} mutation group (and therefore parameter x_6) but is otherwise identical to A_{dmut}.

3.6.5 *Results*

The results are summarised in table 11. As in Experiment V, the average efficiency of the configurations A_{dmut} and A_{mgrp} are calculated from the best of three tuning runs for each SUT, and averaged using the geometric mean. The efficiency of A_{base}, evaluated in Experiment V, is shown for comparison.

The ratio r (introduced in section 3.2.5) is the average multiplicative change in efficiency on going from the original to new configuration, calculated from the average of efficiencies of each algorithm configuration using equation (33). Values greater than 1 indicate that the

new configuration makes fewer executions of the SUT and so is more efficient than the old configuration.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
A_{base}	15.1			
A_{mgrp}	14.4	A_{base} to A_{mgrp}	1.05	73 %
A_{dmut}	2.76	A_{base} to A_{dmut}	5.48	100 %

Table 11 – Results of Experiment VI.

Confidence Estimation The quantity γ is an estimate of the confidence that any observed difference is a real effect and not a result of chance, and so a measure of statistical significance. Higher values of γ indicate that the observed difference is real. In Experiment V, the large variance in the estimated efficiency was noted, and thus it is important to assess the statistical significance, i.e. whether the observed difference is a result of this variability or as a result of the configuration changes.

The confidence is estimated using a bootstrapping analysis technique which accounts for variances arising as a result of *both* the random starting point of each tuning run, *and* the variability inherent in the stochastic profile-deriving algorithm itself. (Feelders (2003) provides an introduction to bootstrapping techniques.)

The bootstrapping analysis repeats the following three steps 1000 times:

1. For each combination of algorithm and SUT, three tuning runs are resampled *with replacement* from the original three runs. Since the sampling is with replacement, the same tuning run may occur more than once in the new sample.

2. For each resampled tuning run, 64 algorithm trails are resampled *with replacement* from the 64 trials performed at the tuned parameter settings. If a tuning run is resampled more than once in the first step, its trials are resampled independently for each occurrence.

3. The average efficiency of each algorithm is re-calculated from the resampled tuning runs and trials using the same procedure for the original results: the median efficiency is calculated across the resampled trials of each resampled tuning run, and the average algorithm efficiency calculated as the geometric mean of the best median efficiencies within each SUT. The ratio, r, is calculated from the re-calculated average efficiencies.

Across the 1000 recalculations, r becomes a random variable, with a distribution resulting from the variance in both the tuning run starting points and the efficiency of the profile-deriving algorithm itself. The 1000 re-sampled values of r are an estimate of this distribution. Let $n_{<1}$ and $n_{>1}$ be the number of the 1000 re-sampled values of r that are less than and greater than 1 respectively, and calculate:

$$\gamma = 2\frac{\max(n_{<1}, n_{>1})}{1000} - 1 \tag{42}$$

This value of γ defines the largest region from the $(\frac{1-\gamma}{2})^{\text{th}}$ to the $(1 - \frac{1-\gamma}{2})^{\text{th}}$ quantiles of r (i.e. located symmetrically in the middle of the distribution) that does *not* contain the value $r = 1$, and γ is the proportion of the distribution contained in this region. Since a value

of $r = 1$ would indicate no difference in algorithm efficiency, γ is thus an estimate of the likelihood that there is a real difference in algorithm efficiency.

Figure 20 illustrates the calculation of γ from the distribution of *r* estimated by resampling.

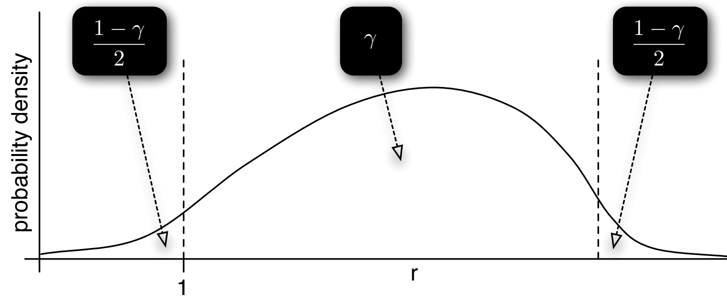


Figure 20 – An illustration of the calculation of γ. The vertical dashed lines indicate the $(\frac{1-\gamma}{2})^{\text{th}}$ and $(1 - \frac{1-\gamma}{2})^{\text{th}}$ quantiles of the distribution. Labels indicate the proportion of the distribution contained in each area under the curve bounded by the quantiles.

3.6.6 *Discussion and Conclusions*

The results provide strong evidence that the efficiency of A_{dmut} is much better than that of A_{base}: the efficiency is improved by a factor of more than five—i.e. one-fifth of number of SUT evaluations are required to derive a profile—and there is a very high confidence that this observed difference is not a result of chance (RQ-1).

Moreover, the improvement is largely the result of directed mutation rather than introduction of mutation groups: the efficiency of the configuration A_{mgrp} that uses mutation groups, but not directed mutation, is only slightly better than A_{base}.

3.6.7 *Threats to Validity*

In addition to the threats identified for Experiment V:

Comparison at Local rather than Global Optima Our empirical approach is to use RSM to find near-optimal parameter settings for each configuration prior to the comparison. However, the results of Experiment V suggest that the tuning runs find local optima, but not necessarily the global optimum. As long as a 'good' local optimum is indicative of the algorithm efficiency, then the comparison is still valid since an equivalent effort has been spent in tuning each algorithm.

The parameter settings used in practice by a test engineer will be those found by RSM or a similar tuning technique. If it is resource-intensive to locate the globally optimal settings—as appears to be the case—then we argue 'good' local optimum may well be representative of the practical efficiency achievable by the algorithm.

If this argument is accepted, then there remains the question of whether sampling only three local optima enables a reliable estimate of a 'good' local optimum. This threat could be reduced by additional tuning runs, but this is impractical given the available computing resources, as discussed in Experiment V. Instead, we use of confidence estimate, γ, to avoid conclusions being drawn from unreliable comparisons as a result of this small sample size.

Difference in the Number of Parameters 9 parameters were tuned for the algorithm configurations A_{base} and A_{mgrp}, but 10 for A_{dmut}. It is reasonable to expect that tuning more parameters is more difficult, and so it is possible that the effort spent tuning each configuration is not equivalent and the comparison unreliable.

The extra effort would occur during the model fitting phase of RSM (see section 3.3.3). However, the size of the fractional factorial design used for this purpose is not fixed. Instead, a minimum design resolution is specified, and the size of design changes to ensure this resolution is achieved for the given number of parameters. In this sense, the fixed resolution—but not necessarily fixed size—of the design ensures that equivalent effort is spent tuning configurations with different numbers of parameters.

3.7 *Experiment VII – Optimal Algorithm Configuration*

3.7.1 *Overview*

Experiment VI demonstrated that the use of directed mutation significantly improves algorithm efficiency. In this experiment, the same empirical method is used to assess the impact on efficiency of the following incremental enhancements to the algorithm's configuration:

- a limit on the number of parents of each node;
- a limit on the number of bins per node;
- the random initialisation of edges;
- the initialisation of bins;
- the omission of some mutation operators;
- the use of non-atomic mutation operators;
- a limit on the number of fitness re-evaluations;
- the use of simulated annealing as the search method;
- a hierarchical fitness comparison.

These proposed enhancements encompass changes to the representation, the fitness metric, *and* the search method. In the process of evaluating these proposed enhancements, many of the design decisions made in the previous chapter (section 2.2) are validated.

Although a potential change of search method from hill-climbing to simulated annealing will be considered, we do not evaluate population-based evolutionary algorithms such as a genetic algorithms. There are two reasons for this decision. Firstly, such search methods introduce multiple additional parameters: the larger number of parameters would increase the difficulty of deriving common parameters that are effective across a range of SUTs. Secondly, an evolutionary algorithm would require a recombination operator such as crossover, and such an operator would be difficult to define and implement for the Bayesian network representation.

The primary objective is to derive an algorithm that is highly efficient in terms of the number of times the instrumented SUT is executed, and so the research question is:

RQ-1 Which configuration of the algorithm is, on average, the most efficient?

Each one of the proposed enhancements listed above is considered, in turn, by a separate sub-experiment which has a method similar to that used in Experiment VI for directed mutation. If an enhancement results in an improvement in efficiency, then the enhancement is included in the configuration evaluated in subsequent sub-experiments.

However, should the difference in efficiency between configurations be negligible, two non-functional properties are considered: the number of parameters and the potential memory usage. An algorithm configuration with fewer parameters will normally require less guidance as to the parameter settings in practice. An algorithm implementation that uses less memory is more likely to scale to larger SUTs. Therefore configurations that have less parameters or constrain the memory usage are preferred, and such enhancements are included in the algorithm configuration even if their impact on efficiency is negligible.

The nine sub-experiments are described in the following subsections. After describing the sub-experiments, we consolidate the results, draw general conclusions from the results, and discuss potential threats to validity.

3.7.2 *Experiment VIIa – Limit on Number of Parents*

Motivation We hypothesise that it is difficult for the search method to derive a suitable input profile when each node (input argument) in the binned Bayesian network representation has a large number of probability values since each atomic mutation changes only *one* of these probability values. If a node has b bins and p parents (each having b bins also), then the number of probability values used to represent the conditional probability distribution at the node is b^{p+1}: the number of probability nodes scales exponentially with the number of parents. Therefore, we propose limiting the number of parents that a node may have in order to improve algorithm efficiency, with the desirable secondary effect of utilising less memory.

Preparation A limit of the number of parents is implemented as a configuration option μ_{prnt} which specifies the maximum number of parents a node may be dependent on. The mutator M_{add} does not add parents to child nodes if the limit would be exceeded.

As for all sub-experiment of Experiment VII, the enhancement is applied to the most efficient algorithm configuration observed so far, in this case, A_{dmut}. In other words, the configurations considered here include *both* directed mutation *and* a limit on the number of parents.

Method As discussed in section 3.2.1, it would be unreliable to treat μ_{prnt} as a *parameter* to be tuned using RSM since it is a limit that may only be occasionally encountered by the algorithm. Instead four different algorithm *configurations* are compared having μ_{prnt} set to 0, 1, 2, and 3. We denote these configurations $A_{\text{prnt}}(x)$ where x is the value of μ_{prnt}. Since the mutation operators M_{add} and M_{rem} that add/remove edges have no effect in configuration $A_{\text{prnt}}(0)$, these operators, their associated groups, and parameters were omitted for this configuration.

The SUTs `bestMove` and `cArcsin` do not have sufficient input arguments to permit more than two parents at any one node, and so a limit of $\mu_{\text{prnt}} \geq 2$ would have no effect. Configurations $A_{\text{prnt}}(2)$ and $A_{\text{prnt}}(3)$ were therefore tuned against only `nsichneu` and `fct3`, and the results of tuning A_{dmut} from Experiment VI re-used for `bestMove` and `cArcsin` when calculating the average efficiency. Similarly, configuration $A_{\text{prnt}}(1)$ was tuned against `nsichneu`, `cArcsin`, and `fct3`, and the results of tuning A_{dmut} re-used for `bestMove`.

The empirical procedure is otherwise the same as Experiment VI.

Results The results are summarised in table 12. The proposed configurations are compared against that of the most efficient configuration observed so far, A_{dmut}. The ratio of efficiencies, r, and the confidence, γ are calculated as described in Experiment VI.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
A_{dmut}	2.76			
$A_{\text{prnt}}(0)$	3.55	A_{dmut} to $A_{\text{prnt}}(0)$	0.78	91 %
$A_{\text{prnt}}(1)$	2.12	A_{dmut} to $A_{\text{prnt}}(1)$	1.30	53 %
$A_{\text{prnt}}(2)$	3.01	A_{dmut} to $A_{\text{prnt}}(2)$	0.92	32 %
$A_{\text{prnt}}(3)$	2.47	A_{dmut} to $A_{\text{prnt}}(3)$	1.12	47 %

Table 12 – Results of Experiment VIIa.

Discussion and Conclusions The configuration $A_{\text{prnt}}(1)$ is the most efficient of new configurations. It is more efficient than A_{dmut}, but the improvement is relatively small and there is little confidence that there is a real difference in efficiency between $A_{\text{prnt}}(1)$ and A_{dmut}. However, $A_{\text{prnt}}(1)$ is likely to utilise much less memory as a result of the limit on the number of parents, and is to be preferred to A_{dmut} for this practical reason. The enhancement, using $\mu_{\text{prnt}} = 1$, is therefore incorporated into the algorithm.

In the configuration $A_{\text{prnt}}(0)$ each node is independent since no parents are permitted. This configuration is not very much worse than the most efficiency configuration, $A_{\text{prnt}}(1)$, and, counter to our expectations, demonstrates that it is possible to derive suitable profiles using only marginal distributions, i.e. no conditionality is required at the nodes. (However, the target probabilities are less than 80% of the optimal minimum coverage probability, and it is possible that input profiles with optimal minimum coverage probabilities may require conditional distributions.) It is possible that omission of the M_{add} and M_{rem} mutation operators contributed to the efficiency of the configuration $A_{\text{prnt}}(0)$.

The relationship between average efficiency and μ_{prnt} is not linear: the observed efficiencies at values of 1 and 3 for μ_{prnt} are better than those at 2 and 4. This may be an artefact of the variability of the average efficiency calculated by this empirical procedure.

3.7.3 *Experiment VIIb – Limit on Number of Bins*

Motivation The motivation for limiting the number of bins is similar to that for limiting the number of parents. As discussed in Experiment VIIa, the number of probabilities is exponential in terms of the number of parents, but is only polynomial in terms of the number of bins. However, the number of atomic mutations possible through the operators M_{joi}, M_{spl}, and M_{len} (which join, split and change the length of bins, respectively) is proportional to the number of bins (regardless of the number of parents) and we hypothesise that a large number of atomic mutations for these operators reduces the algorithm efficiency.

As discussed in section 3.5.2, the configurations considered so far already possess such a limit on the number of bins in order to prevent excessively large representation: a limit of 32 for arguments with domain cardinality of 5 or more, and a limit of 4 otherwise. This experiment acts therefore as a validation of these limits.

Preparation We hypothesise that an appropriate limit on the number of bins is related to the number of coverage elements, $|\mathcal{C}|$, as follows. We argue that, in the worst case, a suitable input profile over the $|\mathcal{C}|$ coverage elements requires separate probabilities to be assigned to $|\mathcal{C}|$ subsets of the input domain. If we assume all nodes have one parent—consistent with the limit $\mu_{\text{prnt}} = 1$ from the previous sub-experiment—and denote the number of bins per node as b, then the number of conditional probability values at each node is b^2. Therefore separate probabilities for the $|\mathcal{C}|$ subsets of the input domain could be represented at each dependent node when the number of bins per node, b, is $\sqrt{|\mathcal{C}|}$.

We do not present this argument as rigorous, but instead use it as a starting point for the derivation of suitable values for limit on the number of bins. We assume that a suitable limit on the number of bins at each node, μ_{bins}, is proportional to $\sqrt{|\mathcal{C}|}$:

$$\mu_{\text{bins}} = \eta \left\lceil \sqrt{|\mathcal{C}|} \right\rceil \tag{43}$$

where $\lceil \ \rceil$ is the ceiling function; and η, a positive constant, is an algorithm parameter. (If the value of μ_{bins} is greater than the cardinality of the input argument represented by the node, the cardinality is used as limit.)

This limit is implemented by preventing the mutation operator M_{spl} splitting a bin if the resulting number of bins at the node would exceed μ_{bins}.

Method As for Experiment VIIa, η implements a limit that cannot be reliably tuned using RSM. Instead the experiment compared three configurations in which the values of η are 1, 2, and 3 respectively. These configurations are denoted $A_{\text{bins}}(x)$ where x is the value of η.

Results The results are summarised in table 13.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{prnt}}(1)$	2.12			
$A_{\text{bins}}(1)$	5.12	$A_{\text{prnt}}(1)$ to $A_{\text{bins}}(1)$	0.41	100 %
$A_{\text{bins}}(2)$	1.73	$A_{\text{prnt}}(1)$ to $A_{\text{bins}}(2)$	1.23	46 %
$A_{\text{bins}}(3)$	2.72	$A_{\text{prnt}}(1)$ to $A_{\text{bins}}(3)$	0.78	49 %

Table 13 – Results of Experiment VIIb.

Discussion and Conclusions The algorithm configuration $A_{\text{bins}}(2)$ is the most efficient and is slightly more efficient than $A_{\text{prnt}}(1)$, the configuration resulting from the previous sub-experiment. Although there is little confidence that there is a real difference in efficiency between $A_{\text{prnt}}(1)$ and $A_{\text{bins}}(2)$, the configuration $A_{\text{bins}}(2)$ is preferred since it introduces another method by which memory usage may be controlled. The enhancement is therefore incorporated into the algorithm, and the limit on the number of bins per node, μ_{bins}, is calculated from properties of the SUT using equation (43) with $\eta = 2$.

3.7.4 *Experiment VIIc – Edge Initialisation*

Motivation We hypothesise that the random initialisation of edges in the Bayesian network from which the search starts will improve algorithm efficiency. Although the edges add dependencies between input arguments, since all the conditional distributions at each node in the initial candidate profile are marginal uniform distributions, the initial profile remains a joint uniform distribution over the input domain.

Preparation Since the network must be acyclic, whether an edge can be added to a node in the initial profile depends on the edges added so far. Therefore, to avoid bias, the nodes are considered in a random order. For each node, the set of potential parents—those that would not create a cycle—is considered, and one is selected at random. (If the limit on the number of parents was higher than one, this process would be repeated until the limit were reached.) We denote the algorithm configuration incorporating this enhancement A_{inie}.

Results The results are summarised in table 14.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{bins}}(2)$	1.73			
A_{inie}	3.04	$A_{\text{bins}}(2)$ to A_{inie}	0.57	89 %

Table 14 – Results of Experiment VIIc.

Discussion and Conclusions The results demonstrate, with high confidence, that randomly initialising edges does not improve algorithm efficiency, and so this enhancement is not incorporated into the algorithm.

3.7.5 *Experiment VIId – Bin Initialisation*

Motivation Currently each node in the initial profile has one bin. We hypothesise that initialising nodes to have more than one bin will improve algorithm efficiency. This initialisation, by itself, does not change the initial conditional distribution at each node, and so the initial profile remains a joint uniform distribution.

Preparation The number of bins initially created in each node is expressed in terms of the maximum number of bins:

$$\max(\lfloor \zeta \mu_{\text{bins}} \rfloor, 1) \tag{44}$$

where $\lfloor \; \rfloor$ is the floor function, μ_{bins} the maximum number of bins that the node may have, calculated using equation (43), and ζ a ratio between 0 and 1.

Method Two algorithm configurations were considered with values of ζ of 0.5 and 1.0, respectively. The configurations are denoted $A_{\text{inib}}(x)$ where x is the value of ζ.

Results The results are summarised in table 15.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{bins}}(2)$	1.73			
$A_{\text{inib}}(0.5)$	1.39	$A_{\text{bins}}(2)$ to $A_{\text{inib}}(0.5)$	1.24	53 %
$A_{\text{inib}}(1.0)$	1.34	$A_{\text{bins}}(2)$ to $A_{\text{inib}}(1.0)$	1.29	85 %

Table 15 – Results of Experiment VIId.

Discussion and Conclusions Both algorithm configurations improve the efficiency, but the configuration $A_{\text{inib}}(1.0)$ demonstrates the better efficiency and there is a much higher confidence that there is a real difference in efficiency between this configuration and $A_{\text{bins}}(2)$. The enhancement, using $\zeta = 1.0$, is therefore incorporated into the algorithm: each node is initialised to its maximum number of bins.

3.7.6 *Experiment VIIe – Omission of Mutation Operators*

Motivation In one tuning run during preliminary experimentation, particularly efficient parameter settings were located for which ρ_{len}, the factor by which bin lengths are increased or decreased by the mutation operator M_{len}, was very close to 1. At this value, the mutation operator has little or no effect, and so we hypothesise that this, and perhaps other, mutation operators may not be required for an efficient algorithm.

Method We consider two algorithm configurations. $A_{\text{-len}}$ omits the mutation operator M_{len} and its associated parameters. $A_{\text{-bin}}$ omits the mutation operators M_{spl} and M_{joi} that split and join bins respectively, and their associated parameters. The latter is a realistic configuration only because each node initially has its maximum number of bins as a result of the enhancement of the previous sub-experiment: if each node had only one bin, the operator M_{spl} would be the only mechanism to create additional bins.

We do not explicitly consider configurations that omit the operators M_{add} and M_{rem}, but note that the configuration $A_{\text{prnt}}(0)$ of Experiment VIIa is the equivalent of this for the case when the representation is initialised with no edges.

This experiment validates the design decision (section 2.2.3) to include these mutation operators.

Results The results are summarised in table 16.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{inib}}(1.0)$	1.34			
$A_{\text{-len}}$	3.69	$A_{\text{inib}}(1.0)$ to $A_{\text{-len}}$	0.36	100 %
$A_{\text{-bin}}$	4.20	$A_{\text{inib}}(1.0)$ to $A_{\text{-bin}}$	0.32	100 %

Table 16 – Results of Experiment VIIe.

Discussion and Conclusions Both configurations are much less efficient than $A_{\text{inib}}(1.0)$, and there is high confidence that these observed differences are real. Therefore neither of the enhancements are incorporated into the algorithm. Moreover, the design decision to include these mutation operators is justified.

3.7.7 Experiment VIIf – Non-Atomic Mutation Operators

Motivation In the algorithm configurations considered so far, the neighbours of the current input profile are created by making *one* atomic mutation. For the mutation operators M_{prb} and M_{len}, an atomic mutation is a change to *one* probability value or *one* bin length, respectively, chosen from the entire representation. We hypothesise that changing multiple probabilities, or multiple bin lengths, during a single mutation will improve efficiency. Only the mutation operators M_{prb} and M_{len} are considered in this hypothesis as they represent 'small scale' mutations; the other mutation operators make 'large scale' mutations that change the structure of the representation and making more than one such mutation is likely to extend the neighbourhood of the current candidate profile too far.

Preparation The non-atomic version of the mutation operator M_{prb} has an additional parameter, the mutation probability p_{prb}. Every probability value in the entire representation is considered in turn, and at each a random number sampled from the uniform distribution over the interval $[0, 1)$. If the random number is less than p_{prb}, then the probability value is mutated using the same mechanism as for the atomic version of the operator. The non-atomic version of the operator M_{len} processes the bins in an equivalent way, using the parameter p_{len}.

The non-atomic versions of these operators are enabled *only* in the G_{bins} mutator group. Preliminary experimentation showed that enabling the non-atomic operators in both the G_{bins} and G_{drct} groups created an extremely inefficient algorithm configuration.

In section 2.2.3, it was explained that sampling of neighbours at each hill climbing iteration occurs *without replacement* as the use of atomic mutations enables identical neighbours to be easily identified. This is not as easy with the non-atomic mutation operators and so sampling occurs *with replacement*. However, the likelihood of sampling two identical neighbours is very small.

Method The algorithm is denoted A_{natm}. The non-atomic mutation operators introduce two additional algorithm parameters, listed in table 17, that are tuned using RSM.

Parameter	Effect	Value
$x'_{11} = \log(p_{\text{prb}})$	non-atomic M_{prb} mutation probability	-4.0 – -2.0
$x'_{12} = \log(p_{\text{len}})$	non-atomic M_{len} mutation probability	-4.0 – -2.0

Table 17 – Additional algorithm parameters for configuration A_{natm}.

Results The results are summarised in table 18.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{inib}}(1.0)$	1.34			
A_{natm}	1.58	$A_{\text{inib}}(1.0)$ to A_{natm}	0.85	53 %

Table 18 – Results of Experiment VIIf.

Discussion and Conclusions The use of non-atomic mutation operators does not improve the algorithm efficiency, and so this enhancement is not incorporated into the algorithm.

3.7.8 Experiment VIIg – Fitness Re-evaluation

Motivation In section 2.2.3, the re-evaluation of the fitness of the current profile at each iteration of the hill climb, and the re-calculation of the estimated minimum coverage probability from the accumulated instrumentation data resulting from these re-evaluations, were proposed as a mechanism to efficiently accommodate the noise in the fitness metric. In section 3.5.2, a limit of 10 was placed on the number of re-evaluations of any one profile should they persist over many iterations of the hill climb in order to avoid large data structures in memory.

The assumptions behind these design decisions—that fitness re-evaluation enables an efficient algorithm and that enforcing a limit of 10 re-evaluations does not significantly reduce efficiency—are validated in this experiment.

Preparation The limit on the number of evaluations of any one profile is specified by an algorithm parameter, μ_{eval}. The value includes the evaluation that occurs when the profile is first created as a neighbour of the then current candidate input profile, and so a setting of $\mu_{\text{eval}} = 1$ prevents subsequent re-evaluation.

Method As it is a limit, the parameter μ_{eval} was treated as a configuration option and not a parameter to tuned. Three configurations were considered with μ_{eval} settings of 1, 5, and 20, respectively. These configurations are denoted $A_{\text{eval}}(x)$ where x is the value of μ_{eval}.

The most efficient configuration so far, $A_{\text{inib}}(1.0)$, already uses $\mu_{\text{eval}} = 10$, and so it was considered to be $A_{\text{eval}}(10)$.

The initial results for the four configurations, $A_{\text{eval}}(x)$ ($x = 1, 5, 10, 20$), suggested that the efficiency is very sensitive to the value of μ_{eval}, and that these four configurations may be insufficient to establish an optimal value. For this reason, an additional configuration was considered in which μ_{eval} *is* an additional tunable parameter, listed in table 19. The range of this parameter was chosen from the results from $A_{\text{eval}}(x)$ so that the changes to the parameter *are* likely to affect the efficiency (the possibility that changes to parameters acting as limits may not change the response was the reason to treat such parameters as configuration options and not as tunable parameters). The configuration with the additional parameter is denoted $A_{\text{eval}}(*)$.

Parameter	Effect	Value
$x''_{11} = \mu_{\text{eval}}$	max no. evaluations per profile	6 – 16

Table 19 – The additional algorithm parameter for configuration $A_{\text{eval}}(*)$.

Results The results are summarised in table 20.

Discussion and Conclusions In $A_{\text{eval}}(1)$, no re-evaluation occurs and this configuration is an order of magnitude less efficient than $A_{\text{inib}}(1.0)$. It is very clear that the use of re-evaluation greatly improves the efficiency of the algorithm, and this result validates the design decision to use re-evaluation.

The configurations $A_{\text{eval}}(5)$ and $A_{\text{eval}}(20)$ are less efficient than $A_{\text{inib}}(1.0)$ for which $\mu_{\text{eval}} = 10$, suggesting an optimal value between 5 and 20.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{inib}}(1.0) \equiv A_{\text{eval}}(10)$	1.34			
$A_{\text{eval}}(1)$	18.8	$A_{\text{inib}}(1.0)$ to $A_{\text{eval}}(1)$	0.07	100 %
$A_{\text{eval}}(5)$	2.22	$A_{\text{inib}}(1.0)$ to $A_{\text{eval}}(5)$	0.60	99 %
$A_{\text{eval}}(20)$	2.15	$A_{\text{inib}}(1.0)$ to $A_{\text{eval}}(20)$	0.62	92 %
$A_{\text{eval}}(*)$	1.56	$A_{\text{inib}}(1.0)$ to $A_{\text{eval}}(*)$	0.86	66 %

Table 20 – Results of experiment VIIg.

This is the motivation for tuning the configuration $A_{\text{eval}}(*)$. $A_{\text{eval}}(*)$ would be expected to demonstrate an efficiency better than or equal to that of $A_{\text{inib}}(1.0)$ since $A_{\text{eval}}(*)$, at the very least, could take $\mu_{\text{eval}} = 10$. However, the observed efficiency of $A_{\text{eval}}(*)$ is slightly worse than that of $A_{\text{inib}}(1.0)$ and there is little confidence that there is any real difference between these configurations. We interpret this result as evidence that $\mu_{\text{eval}} = 10$ is close to the optimal value, and that the slightly worse observed efficiency of $A_{\text{eval}}(*)$ is a result of the difficulty of tuning parameters given the large variance in the response. This interpretation is supported by the tuned values of μ_{eval} resulting from the 12 tuning runs performed for $A_{\text{eval}}(*)$: the median value of the tuned μ_{eval} parameters was 9.

We conclude that re-evaluation using the parameter setting $\mu_{\text{eval}} = 10$ is an efficient algorithm configuration, and thus no additional change is required to the configuration $A_{\text{inib}}(1.0)$.

3.7.9 *Experiment VIIh – Simulated Annealing*

Motivation In section 2.2.3, we hypothesised that the combination of noise in the fitness metric and the process of re-evaluation would enable the hill-climbing to escape local optima. If this is the case, then the use of simulated annealing, a search method that explicitly includes a mechanism to escape local optima, may result in a more efficient algorithm.

Preparation Since both hill climbing and simulated annealing are local search methods, the only change in the algorithm implementation is the rule for deciding which input profile—either the current profile or the fittest of the sample of neighbours—becomes the current profile in the next iteration. The Lundy-Mees (Lundy and Mees, 1986) formulation of the decision rule is used: if the fittest neighbour is fitter than the current profile it always replaces it, but if the fittest neighbour is less fit (with a difference in fitness, $\delta < 0$) then the neighbour replaces the current profile with probability:

$$\mathbb{P}_{\text{replace}} = e^{\delta/u} \tag{45}$$

where $u > 0$ is a variable that decreases over the course of the search and is equivalent to the temperature used in other formulations of simulated annealing.

The Lundy-Mees formulation is used here because of the following simple cooling schedule that requires relatively few parameters. Let u_i be the value of u at iteration i; the value at the next iteration is given by:

$$u_{i+1} = \frac{u_i}{1 + \beta u_i} \tag{46}$$

where $\beta > 0$ is a constant that controls the cooling rate.

Method The values of β and u_0 (the initial value of u) are additional algorithm parameters to be tuned, and are listed in table 21. The ranges for these parameters are based on guidance provided by Lundy and Mees, and from the results of preliminary experimentation. The configuration is denoted A_{sima}.

Parameter	Effect	Value
$x'''_{11} = u_0$	initial 'temperature'	0.02 – 0.18
$x'''_{12} = \log(\beta)$	cooling rate	−2.0 – 0.0

Table 21 – Additional algorithm parameters for configuration A_{sima}.

Results The results are summarised in table 22

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{inib}}(1.0)$	1.34			
A_{sima}	2.69	$A_{\text{inib}}(1.0)$ to A_{sima}	0.50	100 %

Table 22 – Results of Experiment VIIh.

Discussion and Conclusions The use of Lundy-Mees simulated annealing results in reduced algorithm efficiency, and there is high confidence that there is a real difference in efficiency compared to hill climbing. The search method is therefore not changed to simulated annealing in the algorithm configuration.

3.7.10 *Experiment VIIi - Hierarchical Fitness*

Motivation The fitness metric used so far uses information about only the coverage elements with the lowest coverage probability. We hypothesise that algorithm efficiency will be improved if the coverage probabilities of the remaining elements are used to break ties when the minimum coverage probabilities are the same for two candidate input profiles.

Preparation Let coverage probabilities of candidate profiles P and Q be $p_1, p_2, \ldots, p_n$ and $q_1, q_2, \ldots, q_n$, respectively (where n is the number of coverage elements). The probabilities are sorted into ascending order, giving $p'_1, p'_2, \ldots, p'_n$ and $q'_1, q'_2, \ldots, q'_n$. Then P is fitter than Q if there exists m ($1 \leq m \leq n$) such that $p'_i = q'_i$ for all $i < m$, and $p'_m > q'_m$. In other words, if p'_1 and q'_1 are the same, then p'_2 and q'_2 are compared, and so on, until a pair of non-equal coverage probabilities are found.

We call this method of comparing input profiles *hierarchical fitness*, and denote the algorithm configuration that incorporates it A_{hier}. A single metric that expresses fitness is not derived, but this is compatible with hill climbing since the search method requires only a mechanism to *compare* candidate profiles. (Hierarchical fitness would not be compatible with simulated annealing which requires a quantitative fitness.) Nevertheless, the minimum coverage probability must still be calculated in order to check whether the target has been reached. This calculation is straightforward: it is the value p'_1.

Results The results are summarised in table 23

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)	Confidence (γ)
$A_{\text{inib}}(1.0)$	1.34			
A_{hier}	1.71	$A_{\text{inib}}(1.0)$ to A_{hier}	0.78	79 %

Table 23 – Results of Experiment VIIi.

Discussion and Conclusions Configuration A_{hier} demonstrates less efficiency than $A_{\text{inib}}(1.0)$, and there is reasonable confidence that a real difference in efficiency exists. Therefore this enhancement is not incorporated into the algorithm.

3.7.11 Consolidated Results

Table 24 illustrates the overall improvement in efficiency as a result of the sub-experiments VIIa–VIIi. The configuration $A_{\text{inib}}(1.0)$ is the most efficient derived during the experiment, and in the table its efficiency is compared to that of the base configuration A_{base}, and of the configuration A_{dmut} that incorporates directed mutation and is the starting point for the sequence of enhancements in this experiment. For each of three configurations, the table shows the best efficiency observed during the three tuning runs for each SUT and the average efficiency of the configuration.

SUT	Efficiency (10^6 executions)		
	A_{base}	A_{dmut}	$A_{\text{inib}}(1.0)$
`bestMove`	108	9.46	14.3
`nsichneu`	5.77	1.80	0.802
`cArcsin`	5.31	0.687	0.249
`fct3`	15.9	4.95	1.11
Average	15.1	2.76	1.34

Table 24 – Consolidated results of Experiment VII. A_{base} is the baseline configuration, A_{dmut} the starting point for the experiment, and $A_{\text{inib}}(1.0)$ the most efficient algorithm configuration derived.

The results of this experiment, and of the earlier Experiments V and VI, are summarised by figure 21. The filled points indicate the proposed enhancements that were incorporated into the algorithm because they demonstrate an improvement in efficiency.

3.7.12 General Discussion and Conclusions

Based on the consolidated results, the most efficient ('optimal') algorithm configuration is a combination of the algorithm described in section 2.2, the minor pragmatic changes discussed in Experiment V, directed mutation as described in Experiment VI, and the following enhancements (RQ-1):

- a limit of one parent per node (in Experiment VIIa);
- a limit on the number of bins per node, calculated from the characteristics of the SUT using equation (43) (Experiment VIIb);
- the initial creation of the maximum number of bins at each node (Experiment VIId).

The enhancements considered in this experiment improve the algorithm efficiency by a factor of 2.06 compared to the configuration A_{dmut}. When combined with the efficiency improvement enabled by directed mutation, the best configuration, $A_{\text{inib}}(1.0)$, is 11.3 times more efficient than the base configuration, A_{base}, on average across the four SUTs.

In addition, a number of design decisions have been validated:

- the mutation operators M_{add} and M_{rem} that add and remove edges in the Bayesian network representation (by Experiment VIIa, configuration $A_{\text{prnt}}(0)$);
- the mutation operator M_{len} that changes bin lengths (Experiment VIIe, configuration $A_{\text{-len}}$);
- the mutation operators M_{joi} and M_{spl} that join and split bins (Experiment VIIe, configuration $A_{\text{-bin}}$);

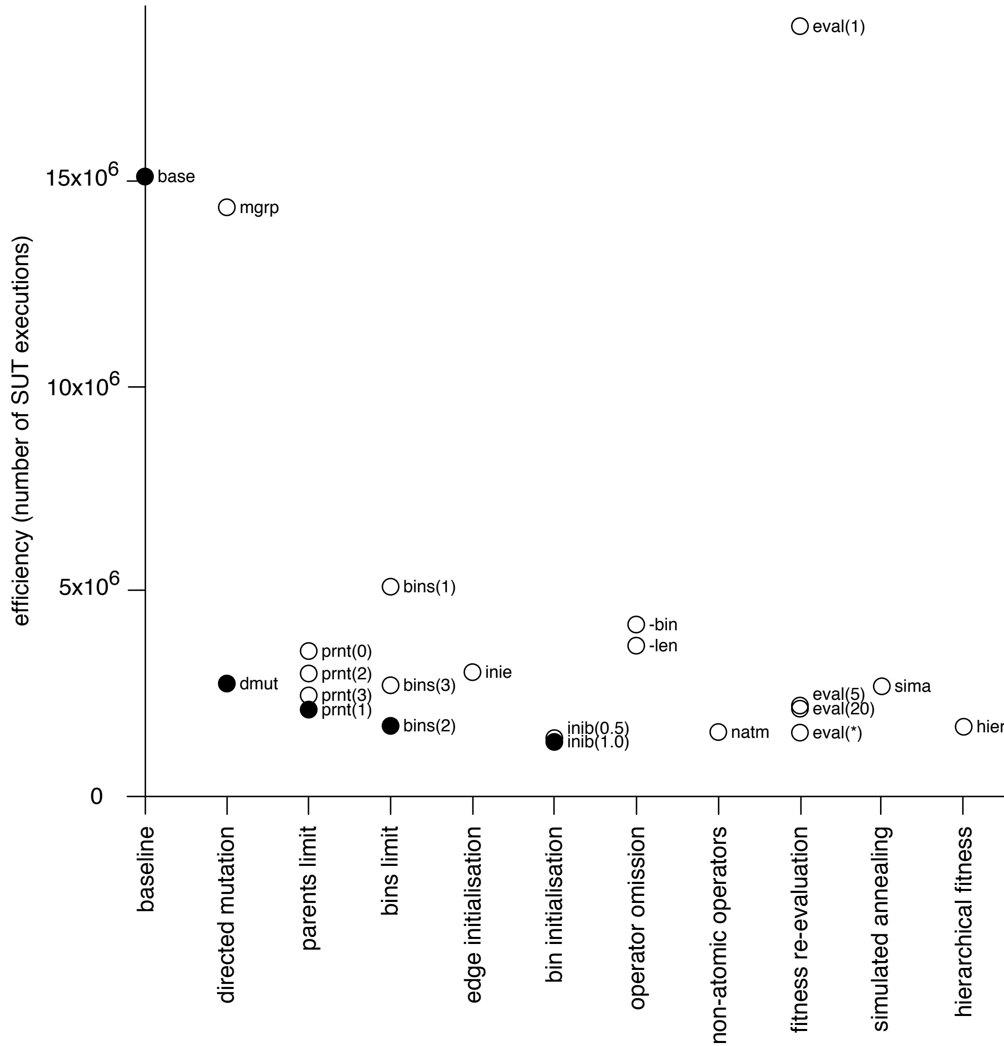


Figure 21 – A summary of the efficiencies of the tuned configurations measured in Experiments V, VI, and VII. The filled points indicate proposed enhancements that were incorporated into the algorithm.

- a search neighbourhood defined by a single atomic mutation (Experiment VIIf);
- fitness re-evaluation (Experiment VIIg);
- the use of a form of random mutation hill climbing (rather than simulated annealing as the search method) (Experiment VIIh);
- the use of the estimated minimum coverage probability as the fitness metric (rather than a hierarchical fitness comparison) (Experiment VIIi).

A Reflection on the Empirical Approach Although it is not an explicit research question for this experiment, we reflect on the decision to tune each configuration before assessing its efficiency, an approach used in Experiments V and VI as well as for this experiment.

Figure 22 is the equivalent of figure 21, but plots the efficiency for each algorithm *prior* to tuning, i.e. calculated using the most efficient of the randomly selected parameter settings from which the three tuning runs began for each SUT.

The scale of the efficiency axis differs from that of figure 21, but the pattern is otherwise remarkably similar. (A notable difference is $A_{\text{eval}}(1)$, the configuration in which no fitness re-evaluation occurs: although both its untuned and tuned efficiencies are the worst in each graph, the efficiency of its untuned configuration is particularly poor.) The filled points indicate the enhancements that were incorporated into the algorithm based on the assessment of their *tuned* efficiencies, and it can be seen that these configuration generally demonstrate the best efficiency prior to tuning as well as after. The exceptions are: untuned $A_{\text{bins}}(3)$ is very slightly more efficient than untuned $A_{\text{bins}}(2)$, and untuned A_{hier} is a little more efficient than untuned $A_{\text{inib}}(1.0)$.

These findings suggests that an alternative approach may have been to employ the resources that were used to tune the configurations to instead assess the efficiency of a large number of randomly-selected parameter settings. However, the correlation between untuned and tuned efficiencies cannot have been confidently predicted prior to the experiment. In addition, the tuned parameter settings for each SUT are important data for the experiment described in the next section that derives common parameter settings.

3.7.13 *Threats to Validity*

The threats to validity discussed in Experiments V (section 3.5.6) and VI (section 3.6.7) also apply to this experiment. In addition:

Order of Enhancements The outcome is likely to have been influenced by the order in which the enhancements were assessed if, as seems likely, the efficiency of each new enhancement is dependent on which earlier enhancements had already been incorporated into the algorithm. As a purely illustrative example, if simulated annealing (Experiment VIIh) had been assessed prior to apply a limit on the number of parents (Experiment VIIa), it might have been found to be more efficient than hill climbing, in which case simulated annealing would have been incorporated into the algorithm. The order was partially determined by dependencies between enhancements, such as the need to establish the maximum number of bins (Experiment VIIb) prior to assessing bin initialisation (Experiment VIId), but was otherwise subjective.

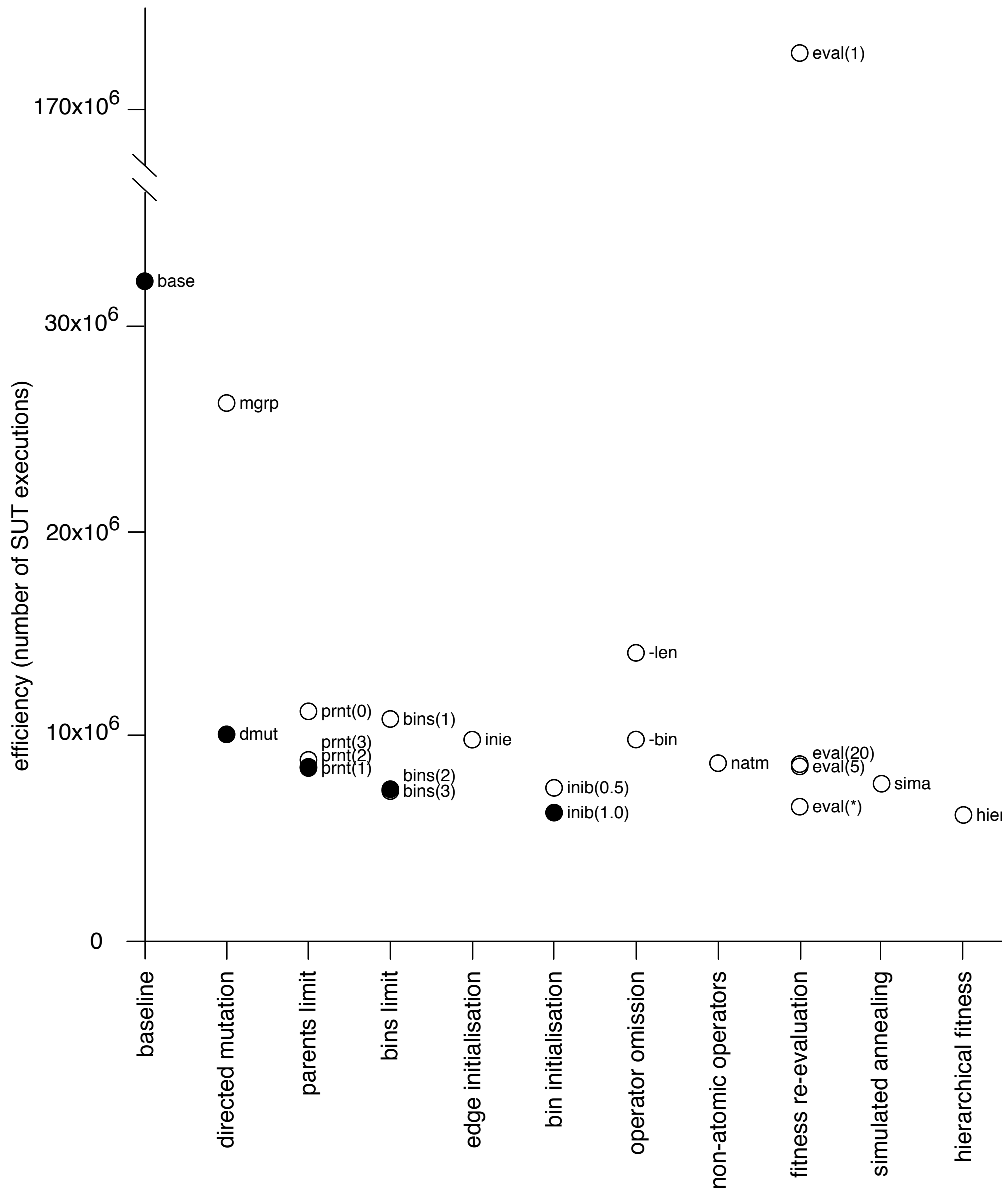


Figure 22 – A summary of the efficiencies of the *untuned* configurations measured in Experiments V, VI, and VII. The filled points indicate proposed enhancements that were incorporated into the algorithm based on their *tuned* efficiencies.

The issue is similar in nature to that which arises during *one-factor-at-a-time* optimisation: each factor is optimised independently by fixing the values of the other factors, but the outcome is affected by the order in which the factors are optimised. The solution for factor optimisation is the use of experimental designs, such as fractional factorial designs, that better accommodate potential interactions between factors. An equivalent solution here would have been to perform the entire experiment many times, each repetition using a different permutation of the enhancement ordering. However this would not have been practical given the substantial resources—both computing and time—were required for just *one* repetition of this experiment.

Bias Towards Early Enhancements The large variance in the observed efficiencies may introduce a bias towards enhancements assessed earlier in the ordering. If one of the earlier enhancements demonstrates, by chance, an *observed* efficiency that is much better than its *actual* value, it could be difficult for a later enhancement to demonstrate a better observed efficiency even if its actual efficiency were the better. In section 2.2.3, it was proposed that a similar problem may occur in the profile-deriving algorithm itself as a consequence of the stochastic noise in the fitness metric. The solution proposed for the algorithm was to re-evaluate the fitness at each iteration and to accumulate the data to provide more accurate fitness estimates. An equivalent form of re-evaluation could be used here: the efficiency of best configuration so far could have been re-evaluated by additional tuning runs at each sub-experiment, but this would again increase the required computing resources.

SUT-Specific Parameter Tuning The efficiency of each configuration is assessed by tuning the parameters *separately* for each SUT. An alternative would have been to tune a single set of parameters *common* to all SUTs, and this may have resulted in the construction of a different optimal configuration. However we have yet to establish that the use of common parameter settings is feasible without a significant reduction in algorithm efficiency compared to SUT-specific settings. (This feasibility is assessed in the next experiment.) If the use of common parameter settings were found not to be feasible in practice, constructing an optimal algorithm configuration by tuning a common set of parameters would have been inappropriate.

3.8 Experiment VIII – Common Parameter Settings

3.8.1 Objectives

The outcome of Experiments V, VI, and VII was a common—to all four SUTs—and highly efficient algorithm configuration, $A_{\text{inib}}(1.0)$.

However, the process used to identify this configuration tuned the parameters separately for each of the four SUTs. If the algorithm is to be used in practice, it will be necessary to provide guidance to the test engineers as to the parameter settings that enable good algorithm performance; it would not be cost-efficient for users to expend resources specifically tuning the parameters for each SUT to which the algorithm is applied.

In this experiment, we provide such guidance by attempting to establish optimal 'common' parameter settings that enable the best efficiency (and therefore best performance). We interpret 'common' to mean that the setting for each parameter is either a constant value or its value may be calculated from an easily measurable characteristic of the SUT. (The value μ_{bins} that limits the number of bins at each node in the representation is an example of the latter type: it is calculated from the number of coverage elements as described in section 3.7.3.)

The first research question is therefore:

RQ-1 What are the optimal common parameter settings for the algorithm configuration $A_{\text{inib}}(1.0)$?

The common parameters will be less efficient than parameters tuned specifically to each SUT. The second research question is:

RQ-2 To what degree is algorithm efficiency reduced as a consequence of using common parameter settings?

3.8.2 Approach

As outlined in section 3.2.3, the common parameter settings are derived using a linear model to predict the optimal settings. The process has three stages:

1. Analyse the tuned SUT-specific parameter settings for configuration $A_{\text{inib}}(1.0)$ in order to determine (a) which parameters could be expressed in terms of SUT characteristics, and (b) a range for each parameter that is likely to include the optimal setting.

2. Run algorithm trials in the parameter space defined by these ranges, and use the observed responses to fit a second-order linear model. Predict the optimal common parameter settings from the model.

3. Measure the efficiency of the algorithm at the predicted optimal settings.

This approach is discussed in detail in the following three sub-experiments; each sub-experiment performs one of the above stages.

3.8.3 *Experiment VIIIa – Analysis of SUT-Specific Parameter Settings*

Method In order to assess the SUT-specific parameter settings, 16 tuning runs are performed for each SUT using the algorithm configuration, $A_{\text{inib}}(1.0)$. Each run begins from parameter settings chosen randomly in the range [-0.5,0.5] of the coded parameter.

Results For each of the four SUTs, the 4 runs demonstrating the best efficiency at the tuned parameter settings are identified from the results of the 16 tuning runs. We refer to the parameter settings of the identified runs as the 'near-optimal' parameter settings.

Table 25 shows the interquartile range of the values taken by each parameter in the near-optimal settings considered across all SUTs. (In the next experiment, we will assume that the most efficient common parameter settings exist in the region of the parameter space defined by these ranges.)

Parameter	Effect	Interquartile Range
$x_1 = \log(K)$	evaluation sample size	4.77 – 6.13
$x_2 = \lambda$	neighbourhood sample size	4.50 – 8.50
$x_3 = \log(\rho_{\text{prb}})$	bin probability mutation factor	1.80 – 3.99
$x_4 = \log(\rho_{\text{len}})$	bin length mutation factor	1.01 – 2.33
$x_5 = \log(W_{\text{edge}}/W_{\text{bins}})$	G_{edge} mutation group weight	-2.65 – -1.27
$x_6 = \log(W_{\text{drct}}/W_{\text{bins}})$	G_{drct} mutation group weight	1.23 – 2.36
$x_7 = \log(w_{\text{add}}/w_{\text{rem}})$	M_{add} mutation weight	-0.56 – -0.14
$x_8 = \log(w_{\text{joi}}/w_{\text{prb}})$	M_{joi} mutation weight	-0.38 – 0.14
$x_9 = \log(w_{\text{spl}}/w_{\text{joi}})$	M_{spl} mutation weight	-0.24 – 0.27
$x_{10} = \log(w_{\text{len}}/w_{\text{prb}})$	M_{len} mutation weight	-0.29 – 0.29

Table 25 – The interquartile range of the near-optimal parameter settings.

In order to analyse any patterns in the parameter settings resulting from the tuning runs, they are plotted using *parallel coordinates* in figures 23 and 24. Parallel coordinates are a method of visualising high-dimensional data pioneered by Inselberg and Dimsdale (1990) in which normally orthogonal axes are drawn as parallel lines. In parallel coordinates, the vector $(v_1, v_2, \ldots, v_n)$, normally plotted as a *point* in a Cartesian coordinate system, is instead plotted as a *line* connecting the points (i, v_i) for $i = 1, 2, \ldots, n$, in a two-dimensional diagram. For clarity, the coded rather than natural parameter values are plotted since the latter differ by orders of magnitude; the parameter z_i is the coded version of parameter x_i listed in table 25.

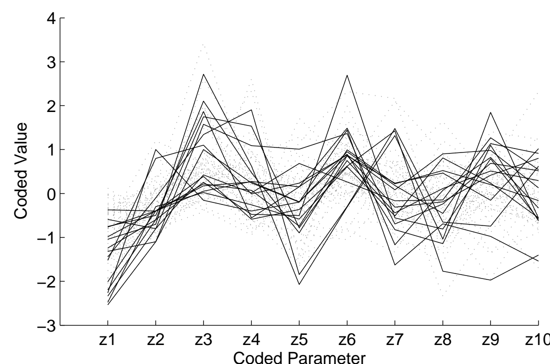
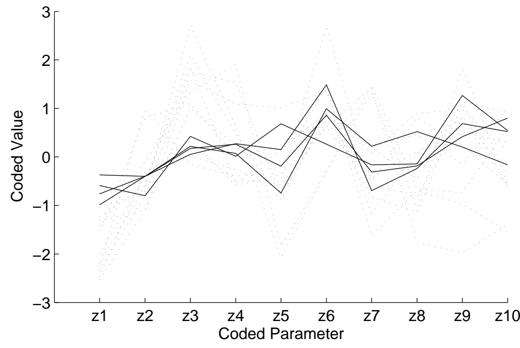
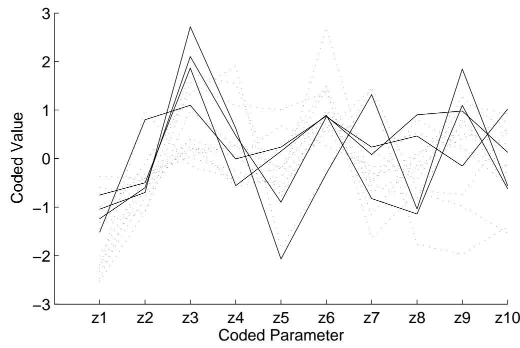


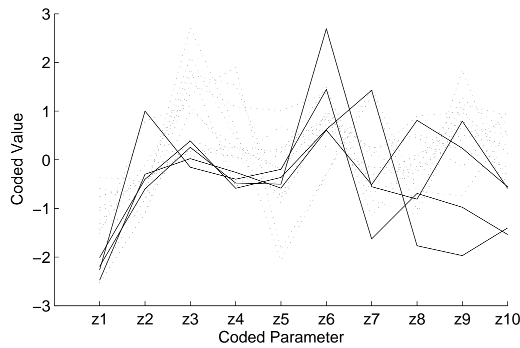
Figure 23 – Coded parameter settings across all SUTs plotted using parallel coordinates. The near-optimal settings are shown as solid lines and the remaining settings as fainter dotted lines for comparison.



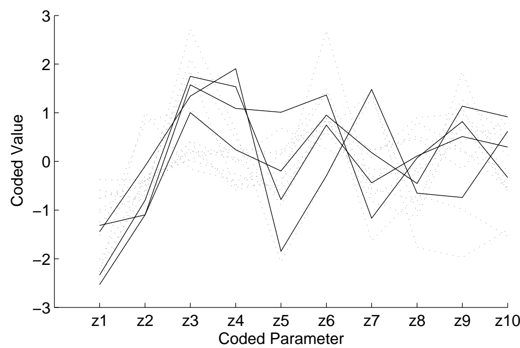
(a) `bestMove`



(b) `nsichneu`



(c) `cArcsin`



(d) `fct3`

Figure 24 – Near-optimal coded parameter settings for each SUT plotted using parallel coordinates. The four settings for the SUT are shown as solid lines and settings for other SUTs as fainter dotted lines for comparison.

Discussion and Conclusions Figure 23, in which the near-optimal settings of all four SUTs are plotted as solid lines, suggests little consistency in the settings *between* SUTs. To clarify this, the near-optimal settings are separated by SUT in the parallel coordinate plots of figure 24; in each, the solid lines show the near-optimal setting for the SUT, and fainter dotted lines are the near-optimal settings of the other SUTs. While figure 24 demonstrates some consistency of near-optimal settings *within* each SUT, there is little consistency with the other SUTs.

There is no obvious evidence in these results to justify the derivation of any parameter setting (other than the configuration option μ_{bins} already identified) from SUT characteristics. As an example, consider parameter x_1, the evaluation sample size. Figure 24 shows that the best settings of x_1 are reasonably consistent within each SUT, but differ between SUTs. We might hypothesise that the evaluation sample size is related to the number of coverage elements using the argument that the more coverage elements, the more SUT executions are required for an accurate estimate of coverage probability. However, the best settings of x_1 are generally lower for `nsichneu` than `bestMove`, even though `nsichneu` has more than ten times as many coverage elements (table 7).

3.8.4 *Experiment VIIIb – Optimisation of Common Parameter Settings*

In this experiment, the optimal settings are predicted from a model of algorithm efficiency.

Preparation The model is a second-order quadratic linear model of the form:

$$\log N = \beta_0 + \sum_{i=1}^{10} \beta_i z_i + \sum_{i=1}^{10} \sum_{j=i+1}^{10} \beta_{ij} z_i z_j + \sum_{i=1}^{10} \beta_{ii} z_i^2 \tag{47}$$

where N is the efficiency (and $\log N$, the chosen response measure as discussed in section 3.2.5), z_i are coded algorithm parameters, and all β are model coefficients to be estimated.

Experiments V, VI, and VII provide strong evidence of a rugged response surface with multiple optima, but equation (47) is a model of a smooth response surface which, since it is quadratic, has at most a single optimum. This simplification of the surface by the model is intentional. We speculate that it captures large-scale features of the response surface by 'smoothing out' smaller-scale features such as local optima and so may produce a more reliable estimate of the optimum than a technique such as RSM.

Given the relatively large number of parameters and the apparent complexity of the response surface, a linear model is likely to be a good fit over only a relatively small region of the parameter space. We therefore use the interquartile parameter ranges derived in the preceding experiment (listed in table 25) to define the boundaries of a suitably small region in which we estimate optimum parameter settings to lie. A minor adjustment is made: for integer-valued parameters, the lower boundary of the range is rounded down to the nearest integer, and the upper boundary rounded up. We refer to this region as the 'region of interest'.

(The requirement for a small region of interest is the reason why a linear model could not be used as an alternative to RSM for tuning parameters in Experiments V, VI and VII: it was not possible to confidently and objectively identify a small region of interest prior to experimentation. In those experiments, RSM is advantageous because it 'moves' the region of interest around the parameter space until an optimum is found.)

Method In order to estimate the model coefficients, β, algorithm trials are performed at selected points in the region of interest in order to measure the efficiency. A *central composite design* is used for this purpose as it provides accurate estimates of the coefficients using relatively few design points.

A central composite design is a combination of a fractional factorial design in which factors have coded values ± 1; repetitions of the centre point at which all factors have coded value 0; and 'star' points at which one factor has the coded values $\pm\alpha$ (where α is normally greater than 1 and depends on the number of factors) and all other factors are 0.

For ten parameters, the value of α would be normally be 3.36 and so, apart from the star points, the design points would be clustered in the centre of the region of interest. Such a design would provide little information about parameter interactions nearer the edges of the region. We therefore choose instead a *face-centered* central composite design in which the value of α is 1, and equate the boundaries of the region of interest with coded values ± 1 in the design.

A central composite design for the 10 algorithm parameters has 178 design points. At each design point, 16 algorithm trials are run (each with a different seed) for each of the four

SUTs in order to accommodate the noise in algorithm response; a total of 11,392 algorithm trials.

For consistency with calculation of efficiency used in Experiments V, VI, and VII, at each design point the median of the 16 observed responses is calculated for each SUT and the 'average' efficiency calculated from the medians as discussed in section 3.2.5. The model is fitted to the average efficiency observed at each design point using standard linear regression to estimate the model coefficients.

The optimal parameter settings are derived by optimising the fitted model using the MATLAB optimization toolbox. Since it would be unreliable to use the model to estimate the efficiency outside the region of interest, the optimisation process is constrained to this region.

Results In figure 25, parallel coordinates are used to plot the optimal common parameter settings predicted by the model as coded values (solid line), and the boundaries of the region of interest (dotted lines).

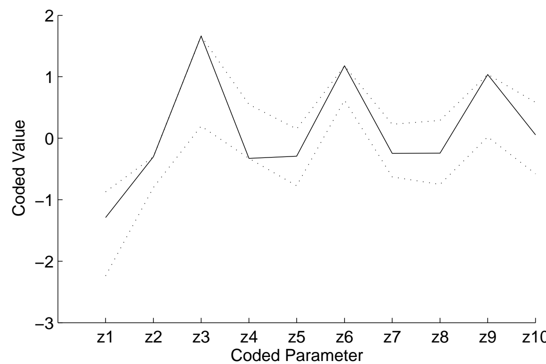


Figure 25 – The optimal common parameter settings as coded values (solid line), and the boundaries of the region of interest (dotted lines) for comparison.

Table 26 lists the optimal common parameter settings in their natural forms rather than the transformed forms, x_i, used for the experiments in this chapter; the column 'Source' indicates from which x_i the natural parameter values are derived.

For completeness, the table includes four parameters that were temporarily considered configuration options for the purpose of the empirical work: the maximum number of parents and bins per node, the proportion of the maximum number of bins that are initially created in the representation, and the maximum number of fitness evaluations for each candidate profile (which limits the number of re-evaluations). The maximum number of bins is calculated from the number of coverage elements in the SUT according to equation (43).

Discussion and Conclusions Figure 25 shows that the predicted optimum settings are at a 'corner' of the region of interest: for five of the parameters the optimal value is at one of the bounds defined by the interquartile range. This suggests that the 'true' optimum of the fitted model is at a point *outside* the chosen region of interest, and that the predicted optimum using constrained optimisation is simply the best parameter settings *inside* the region.

Possible explanations for such a result are: (a) the noise in the algorithm response reduces the fit between the model and response surface leading to some inaccuracy in the prediction; and, (b) there is no reason why the optimum should necessarily be in the chosen region of interest. It would be possible to 'move' the region of interest in the direction of the model's

Parameter	Effect	Source	Optimal Setting
K	evaluation sample size	x_1	302
λ	neighbourhood sample size	x_2	9
ρ_{prb}	bin probability mutation factor	x_3	54.002
ρ_{len}	bin length mutation factor	x_4	2.755
W_{bins}	G_{bins} mutation group weight		1 000
W_{edge}	G_{edge} mutation group weight	x_5	144
W_{drct}	G_{drct} mutation group weight	x_6	10 584
w_{rem}	M_{rem} mutation weight		1 000
w_{add}	M_{add} mutation weight	x_7	689
w_{prb}	M_{prb} mutation weight		1 000
w_{joi}	M_{joi} mutation weight	x_8	886
w_{spl}	M_{spl} mutation weight	x_8,x_9	1 159
w_{len}	M_{len} mutation weight	x_{10}	1 028
μ_{prnt}	Maximum no. parents per node		1
μ_{bins}	Maximum no. bins per node		($\eta = 2.0$)
ζ	Initial no. bins (as proportion of μ_{bins})		1.0
μ_{eval}	Maximum no. evaluations per profile		10

Table 26 – The optimal common parameter settings for configuration $A_{\text{inib}}(1.0)$.

optimum and then repeat the experiment using the new region of interest; such a process is essentially an iteration of RSM. Each repetition requires 11,392 algorithm trials and so would be require substantial computing resources, especially if more than one such 'move' is required. Instead, we regard the optimal parameter settings listed in table 26 as the 'best effort' prediction given practical resources.

We briefly comment on some individual parameter values in the optimal settings:

- The value of ρ_{prb}, the multiplicative factor by which a probability value is increased or decreased during mutation, is surprisingly high. It was expected that this mutation operator would make small refinements to the probability distributions, but—especially when the number of bins is small—one atomic mutation of this operator using such a large factor will be a substantial change to the distribution.

- The value of ρ_{len}, which serves a similar purpose to ρ_{prb} for bin lengths, is relatively small in comparison to ρ_{prb}.

- The weight, W_{drct}, for the mutation group implementing directed mutation is more than 10 times higher than the other groups, confirming that such mutations are important to the efficiency of the algorithm.

- The ratio of w_{add} to w_{rem} is less than 1 and so parsimony in the edges of the Bayesian network is encouraged

- The ratio of w_{spl} to w_{joi} is greater than 1 and so parsimony in the number of bins is *not* encouraged. However, in this algorithm configuration, $A_{\text{inib}}(1.0)$, a limit is placed on the number of bins, and the number of bins is initialised to this limit: therefore such parsimony may longer be required.

3.8.5 *Experiment VIIIc – Algorithm Efficiency at Common Parameter Settings*

For clarity, we use the symbol A^*_{SUT} in the following to refer to the optimal configuration, $A_{\text{inib}}(1.0)$, when using parameters tuned specifically to each SUT; and A^*_{comm} when using common parameters.

Method In order to assess the relative efficiency of the common parameter settings, 64 trials of A^*_{comm} (each with a different seed) are run for each SUT.

Results The results are presented by SUT in table 27, with the efficiencies observed for A^*_{SUT} in Experiment VIId included for comparison.

SUT	Efficiency (10^6 executions)		Ratio
	A^*_{SUT}	A^*_{comm}	(r)
`bestMove`	14.3	11.3	1.27
`nsichneu`	0.802	0.440	1.82
`cArcsin`	0.249	0.940	0.26
`fct3`	1.11	2.59	0.43

Table 27 – Results of Experiment VIIIc: the efficiencies of the optimal configuration at SUT-specific (A^*_{SUT}) and common (A^*_{comm}) parameter settings.

Table 28 presents the results in terms of the average efficiencies across all four SUTs, calculated using the geometric mean as discussed in section 3.2.5. The average efficiency of A^*_{comm} is compared against the baseline configuration, A_{base}, and against A^*_{SUT}.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)
A_{base}	15.1		
A^*_{SUT}	1.34	A_{base} to A^*_{SUT}	11.34
A^*_{comm}	1.86	A^*_{SUT} to A^*_{comm}	0.72
		A_{base} to A^*_{comm}	8.12

Table 28 – The efficiencies of the baseline (A_{base}) and optimal configurations (A^*_{SUT}) using SUT-specific parameters, and the optimal configuration using common parameters (A^*_{comm}).

Discussion and Conclusions As expected, the average efficiency at the common parameters is reduced compared to SUT-specific parameters: the instrumented SUT is executed 40% more times at the common parameters. Although the average efficiency is reduced, the parameter settings predicted from the linear model have *improved* the efficiency for two of the SUTs, `bestMove` and `nsichneu`. This unexpected result provides additional evidence that the rugged response surface limits the ability of RSM to tune the parameters, and justifies the use of the linear model rather than RSM in Experiment VIIIb.

Despite the reduction in average efficiency as a result of using the common parameter settings, the efficiency of A^*_{comm} is neverthless approximately 8 times better than that of the baseline algorithm, A_{base}.

3.8.6 *General Discussion and Conclusions*

The outcome of this experiment is a set of common parameter settings listed in table 26 (RQ-1). The intention is that these settings provide guidance for achieving good efficiency when applying the algorithm to new SUTs. Since the algorithm at the common parameters is 8 times more efficient than the algorithm considered viable in chapter 2, it is likely to demonstrate an acceptable efficiency over a wider range of SUTs.

The algorithm efficiency using these settings is 40% less than that which would be achievable if the parameters were tuned specifically for each SUT (RQ-2). We argue that such SUT-specific tuning would not normally occur in practice: as long as the algorithm has acceptable efficiency at the common parameters, the computing resources and time would be better spent on performing the software testing itself.

Run Time The objective of the work in this chapter is to improve the performance, measured in terms of run time, of the algorithm. Since run time is influenced by the algorithm implementation and computing environment, the efficiency—the number of SUT executions—is used as a more reliable metric of performance in the empirical work. Nevertheless, it is sensible to confirm that the run time is indeed improved by the optimised algorithm configuration and parameter settings.

Table 29 shows the median run times of the algorithm configurations A_{base} using SUT-specific parameter settings, and A^*_{comm} at the common parameter settings. The data is obtained from Experiments V and VIIIc respectively. All the algorithm trials reached the target maximum coverage probability in Experiment VIIIc, but in Experiment V a few trials did not. Therefore the run times were calculated using the procedure used in the analysis of Experiment I (section 2.5.4) to account for the probability that algorithm needs to be re-run should it initially fail to derive a suitable input profile.

SUT	Run Time (min)		Ratio
	A_{base}	A^*_{comm}	r
`bestMove`	35.2	2.83	12.4
`nsichneu`	6.30	0.209	30.2
`cArcsin`	0.568	0.0592	9.59
`fct3`	3.51	0.371	9.45

Table 29 – The median run times of the algorithm configurations A_{base} and A^*_{comm}.

The observed run times are greatly improved, by a factor of at least 9 times, for all SUTs, providing strong evidence that the optimisation of algorithm efficiency has indeed resulted in improved algorithm performance. Since the experiments were run on a computing grid of heterogenous nodes that has a variable user load, there may be substantial noise in the reported times; for this reason we make no further interpretation of this data.

(Although configuration A_{base} is similar to the algorithm used for experiment work in chapter 2, the run times for `bestMove` and `nsichneu` in table 29 are much less than those reported in Experiment I (table 5). Two changes in particular account for this reduction. Firstly, the target minimum coverage probabilities for A_{base} are 20% lower than the equivalent targets in Experiment I which presents an easier problem. Secondly, since the diversity measure is no longer calculated, the overhead of maintaining a large data structure for this purpose—which occurred even if the diversity measure was not used in the fitness metric—is eliminated. The rationale for both changes is explained in section 3.5.2.)

Comparison between Efficiency and Domain Cardinality In the discussion of Experiment V (section 3.5), we noted that when the baseline algorithm is applied to `bestMove` and `nsichneu`, the number of executions of the instrumented SUT was larger than the cardinality of the input domain. Therefore, it would more efficient to systematically run the SUT for each input in the domain and calculate a profile from the results. The equivalent comparison may now be made for the configuration A^*_{comm}: table 30 shows the median number of SUT executions (a restatement of the efficiencies listed in table 27) and the proportion of the domain that the number of executions is equivalent to.

SUT	Domain Cardinality	SUT executions	Proportion of Domain
`bestMove`	2.62×10^5	1.13×10^7	43
`nsichneu`	3.40×10^6	4.40×10^5	0.13
`cArcsin`	∞	9.40×10^5	—
`fct3`	6.71×10^7	2.59×10^6	0.039

Table 30 – A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^*_{comm}.

At the common parameters, the number of SUT executions is now much less than the domain cardinality for `nsichneu`, but remains many multiples of the domain cardinality for `bestMove`. We address this issue in Experiment X.

3.8.7 *Threats to Validity*

Many of the threats to validity identified in Experiments V, VI, and VII also apply to this experiment. In particular, the common parameters may not generalise to a wide range of SUTs given the small number of SUTs used in the experiment. In addition:

Subjective Analysis of SUT-Specific Parameters The process of analysing patterns in the SUT-specific parameters was subjective. As no patterns were proposed, the impact of this threat is likely to be small in practice: no subjective relationships between parameters and SUT characteristics were incorporated into the parameter setting guidance. Nevertheless, beneficial relationships may have been overlooked.

3.9 Experiment IX – Caching Instrumentation Data

3.9.1 Objectives

In the discussion of Experiment VIII (section 3.8.6), we noted that when the optimal algorithms was applied to the `bestMove`, the number of executions of the instrumented SUT was greater than the domain cardinality, and so it would more efficient to systematically run the SUT for each input in the domain.

In the experiment, we propose and evaluate a solution: caching the instrumentation data resulting from executing the SUT. During the course of the algorithm, the second and subsequent execution of the instrumented SUT with the same input would use the instrumentation data from the cache rather than executing the SUT. (This assumes that the SUT is deterministic in the sense that the same instrumentation data is always returned for a given input.) If the execution of the SUT takes a relatively long time, caching may improve the run time of the search algorithm, especially if the cardinality of the domain is small so the likelihood of the same input being sampled more than once is high.

The research questions are:

RQ-1 To what extent does the use of caching reduce the number of SUT executions?

RQ-2 How does the run time of the algorithm change as result of caching?

3.9.2 Preparation

The cache is implemented as an associative array where the key is the input, and the value is the instrumentation data returned by executing the SUT. The instrumentation data from all SUT executions are cached. Caching is enabled by an algorithm parameter, and affects only the mechanism of evaluating candidate input profiles: assuming a deterministic SUT, the algorithm returns *identical* results (i.e. the same input profile) whether or not caching is enabled.

3.9.3 Method

Experiment VIIIc provides a baseline against which to compare the results of caching. The method in this experiment is therefore the same as that of Experiment VIIIc: 64 trials of the optimum algorithm for each SUT, but in this experiment caching of instrumentation data is enabled.

The response measured in this experiment is the number of SUT executions that arise as a result of cache misses, i.e. SUT executions performed because the cache does not currently contain the instrumentation data for the given input.

To ensure the most direct comparison with Experiment VIIIc, the same set of seeds to the pseudo-random number generator are used.

3.9.4 Results

Table 31 lists, for comparison, the median number of SUT executions (the efficiency) and runtime for each SUT from Experiment VIIIc in which the algorithm trials are run *without* caching. Table 32 lists the corresponding results from this experiment in which algorithm trials are run *with* caching.

SUT	Domain Cardinality	SUT Executions	Proportion of Domain	Run Time (min)
`bestMove`	2.62×10^5	1.13×10^7	43	2.83
`nsichneu`	3.40×10^6	4.40×10^5	0.13	0.20
`cArcsin`	∞	9.40×10^5	—	0.06
`fct3`	6.71×10^7	2.59×10^6	0.039	0.37

Table 31 – A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^*_{comm} *without* caching.

SUT	Domain Cardinality	SUT Executions	Prop. Reduct. in SUT Execs	Proportion of Domain	Run Time (min)	Prop. Reduct. in Run time
`bestMove`	2.62×10^5	1.95×10^5	57.8	0.74	1.08	2.6
`nsichneu`	3.40×10^6	1.48×10^5	2.98	0.044	0.25	0.80
`cArcsin`	∞	9.40×10^5	1.00	—	0.19	0.32
`fct3`	6.71×10^7	1.06×10^6	2.43	0.016	0.75	0.49

Table 32 – A comparison of the SUT domain cardinality and median number of SUT executions made by the algorithm A^*_{comm} *with* caching. The columns 'Prop. Reduction in SUT Execs' and 'Prop. Reduction in Run Time' are the factors by which the number of executions and the run time change when caching is enabled; a value greater than 1 indicates an improvement.

3.9.5 *Discussion and Conclusions*

Number of SUT Executions The column 'Prop. Reduction in SUT Execs' in table 32 shows that caching reduces the number of SUT executions for all SUTs, apart from `cArcsin` for which the domain is infinite and the likelihood of executing the SUT with the randomly-sampled input more than once is effectively zero (RQ-1). The largest reduction, by a factor of more than 57, is for the `bestMove`. Without caching, the average number of SUT executions for `bestMove` is 43 times the domain cardinality (table 31). Assuming there is no limit to the cache size, caching will always ensure the number of SUT executions is at most the domain cardinality, but for `bestMove` the number of executions is somewhat less than this limit: approximately 74% of the input domain is sampled (table 32).

The algorithm, with caching, is more efficient than an exhaustive evaluation of the input domain for *all* SUTs. For SUTs `nsichneu` and `fct3`, only a very small proportion of the input domain is evaluated during the derivation of an input profile.

Run Time The run times will be noisy as a result of the heterogeneous nodes forming the computing grid on which these experiment were run and the variable user load on the grid. Moreover, the times are dependent on method chosen for implementing the cache. Nevertheless, there is evidence that caching does reduce the time for `bestMove`, but the times of the other SUTs are increased (RQ-2). We hypothesise that for these SUTs, the overhead of maintaining and accessing the cache of instrumentation data is greater than the time saved in executing the SUT. In support of this hypothesis, we note that the increase in run time is greatest for the SUTs with largest domain cardinalities and the fewest proportion of cache hits: `cArcsin` where no cache hits occur, and `nsichneu` where only approximately 60% of the SUT executions are avoided by cache hits.

A potential solution could be to limit the size cache to the most recent SUT executions: although the number of cache hits may be reduced, the performance overhead of maintaining

a smaller cache would be smaller.

Recommendation As for the other experiments in this chapter, it is difficult to confidently generalise these results given the relatively small number of SUTs. We do, nonetheless, speculate that for SUTs with relatively small domain cardinality (such as `bestMove`), caching may improve performance since the large number of cache hits compensates the overhead of maintaining the cache. However, for SUTs with larger domains caching is likely to decrease performance, and for all SUTs, caching is likely to substantially increase the memory used by the algorithm. For these reasons, we do not implement the use of caching by default, and it is *not* enabled in the subsequent experiments of this thesis.

3.10 Experiment X – Utility of the Optimised Algorithm

As for Experiment VIIIc, we use the symbol A^*_{SUT} in the following to refer to the optimal configuration, $A_{\text{inib}}(1.0)$, when using parameters tuned specifically to each SUT; and A^*_{comm} when using the common parameters.

3.10.1 Objective

Experiments VI and VII optimised the algorithm configuration, and Experiment VIII optimised the common parameter settings. Only four SUTs were considered in these experiments and although the SUTs have diverse characteristics, we do not claim that the resulting algorithm A^*_{comm} is necessarily optimal across a wide range of SUTs. Nevertheless, the improvement in efficiency over the baseline configuration is so large that it is reasonable to expect some improvement when the optimised algorithm is applied to many other SUTs, even if the size of the improvement is less than observed with the examples SUTs used to date.

In this experiment we demonstrate this claim is true for at least one SUT, `tcas`, that differs substantially in some aspects from the four SUTs used to derive the optimised algorithm.

The research question is:

RQ-1 Does the algorithm A^*_{comm} demonstrate an efficiency improvement compared to the baseline algorithm, A_{base}, for the SUT `tcas`?

3.10.2 Preparation

The Traffic Collision Avoidance System (TCAS) is an on-board aircraft system that warns pilots of a potential collision with another aircraft and recommends evasive action. The SUT `tcas` is an implementation of the logic that recommends pilot action in an early version of TCAS. It was developed by Siemens and is made available by the Software-artifact Infrastructure Repository (Do et al., 2005). The instrumented source code is available at: `http://www-users.cs.york.ac.uk/smp/supplemental/`.

The characteristics of `tcas` are shown in table 33 together with those of the other four SUTs for comparison. The first major difference is that we choose to derive an input profile for condition coverage (a stronger adequacy criterion that branch coverage) of `tcas`. There are 31 reachable conditions in the code, and since each has 2 possible outcomes, there are a total of 62 coverage elements. The second difference is the cardinality of the input domain of `tcas`: it is substantially larger than that of the other SUTs taking integer arguments. (In order to calculate this cardinality we make conservative assumptions as to the valid range for input arguments where this was not obvious from the code itself. The calculated cardinality is therefore likely to be an underestimate.)

3.10.3 Method

Target Minimum Coverage Probability During 64 preliminary trials of the algorithm on `tcas`, the highest minimum coverage probability observed was 0.0974. For consistency with the other SUTs, the target minimum coverage probability, $\tau_{p_{\min}}$, is chosen to be 0.078 which is 80% of this highest value.

SUT	`bestMove`	`nsichneu`	`cArcsin`	`fct3`	`tcas`
lines of code	89	1 967	70	135	135
no. loops	7	1	0	0	0
no. conditional branches	42	490	18	19	12
no. condition outcomes					62
no. input arguments	2	11	3	8	12
argument data types	`int`	`int`	`double,bool`	`int`	`int`
domain cardinality	2.62×10^5	3.40×10^6	∞	6.71×10^7	9.83×10^{28}

Table 33 – Characteristics of the SUTs `bestMove`,`nsichneu`, `cArcsin`, `fct3`, and `tcas`.

Procedure Three set of algorithm trials are performed using `tcas` as the SUT:

- Three RSM tuning runs using the algorithm A_{base}. This applies the method of Experiment V in order to estimate the efficiency of the baseline algorithm applied to `tcas`.
- Three RSM tuning runs using the algorithm A^*_{SUT}. This applies the method of Experiment VII in order to estimate the efficiency of the optimised algorithm configuration applied to `tcas` when using SUT-specific parameters.
- 64 algorithm trials of algorithm A^*_{comm}. This applies the method of Experiment VIIIc in order to estimate the efficiency of the optimised algorithm configuration applied to `tcas` when using the common parameter settings .

3.10.4 Results

The median efficiencies are reported in table 34. For A_{base} and A^*_{SUT}, the median efficiency is the best of the three tuning runs, following the same procedure as Experiments V and VII.

Algorithm Configuration	Efficiency (10^6 executions)	Comparison	Ratio (r)
A_{base}	91.5		
A^*_{SUT}	8.37	A_{base} to A^*_{SUT}	10.9
A^*_{comm}	13.5	A^*_{SUT} to A^*_{comm}	1.61
		A_{base} to A^*_{comm}	6.77

Table 34 – The efficiency of the baseline (A_{base}) and optimal configurations (A^*_{SUT}) using SUT-specific parameters, and the optimal configuration using common parameters (A^*_{comm}) when applied to `tcas`.

3.10.5 Discussion and Conclusions

The results show an improvement in efficiency when the optimised algorithm is applied to `tcas` compared to the baseline algorithm (RQ-1). The pattern is similar to that seen for the other four SUTs: there is a substantial improvement as a result of optimising the algorithm configuration, A^*_{SUT}, compared to the baseline, A_{base}. There is an expected decrease, in this case of 61%, when using common parameter settings, A^*_{comm}, compared to the SUT-specific parameters of A^*_{SUT}. Overall, the efficiency change from A_{base} to A^*_{comm} is an improvement by a factor of 6.77, a figure consistent with the average improvement of 8.12 reported in Experiment VIIIc across the four SUTs used to optimise the algorithm configurations and parameters.

We make no claim that this experiment provides evidence of a general result: such an experiment would need to consider a large set of diverse SUTs. However it does illustrate that, for at least one other SUT, the algorithm can be used as it would be in practice—i.e. using the common parameter settings instead of tuning the parameters specifically for the SUT—and that the algorithm demonstrates 'good' efficiency in this situation.

3.11 Conclusion

The objective of the research described in this chapter was to optimise the performance of the algorithm in order to reduce both the cost and time taken when deriving concrete strategies for statistical testing.

The approach was two-fold:

1. Enhancing the algorithm configuration, i.e. the choice of representation, fitness metric and search method. The configuration that demonstrated the best efficiency on average across the SUTs has the following features:

   representation: A binned Bayesian network with constraints on the representation size: a maximum of one parent for each node, and a maximum number of bins per nodes (calculated from the characteristics of the specific SUT).

   fitness metric: An estimate of the minimum coverage probability by multiple executions of the instrument SUT which may be refined by a limited number of re-evaluations.

   search method: Random mutation hill climbing; initialisation by creating the maximum number of bins at each node; and directed mutation: dynamic bias in the mutation operators based on an additional feedback channel.

2. Tuning the algorithm parameters. In order to provide practical guidance to users of the algorithm, a common set of tuned algorithm parameter settings—the same for all four SUTs—was derived.

In combination, the optimised configuration and parameter settings improve the efficiency, i.e. the number of times the instrumented SUT is executed by the algorithm, by a factor of more than 8 on average. The algorithm run time is improved by a similar factor.

The improvement in performance enables the derivation of concrete strategies for statistical testing and therefore contributes to the cost-efficiency of the techniques. Moreover, the improved performance will enable the algorithm to scale to larger and more complex SUTs while continuing to derive profiles in a practical time.

A secondary contribution of this chapter is the empirical method: the use of response surface methodology (RSM) as an objective method to tune algorithm parameters, and the use of bootstrapping to estimate the confidence that any observed differences in algorithm efficiency were not a result of chance.

In Experiment V (section 3.5), we concluded that RSM was an effective tuning method, although our implementation may occasionally fail to accommodate the stochastic noise in the algorithm efficiency. A comparison of figures 21 and 22, illustrating the efficiency of tuned and untuned algorithm configurations respectively, demonstrates that the effectiveness of RSM as a tuning method occurs across all configuration comparison experiments: the efficiency is consistently improved. However, the RSM procedure used for the experiments of this chapter required an average of approximately 1200 trials (algorithm runs) to tune a single algorithm configuration for a single SUT. Given that each configuration was tuned three times for each of the four SUTs, and each trial may take as long as an hour if the configuration is inefficient, the use of RSM would not have been possible without the availability of substantial computing resources in the form of grid of 300 CPU cores.

Unaddressed Issues We note the following unaddressed issues:

Generality: The process of optimising the algorithm configuration and parameter settings required substantial computing resources and so placed a constraint on the number of SUTs that could be used as representative examples in the optimisation process. Although the set of example SUTs was small, the four SUTs were chosen so that their characteristics—such as size, cardinality of input domain, and number of coverage elements—were diverse. We hypothesised that using a diverse set of examples would avoid the process 'overfitting' to any one type of SUT and therefore enable the optimised algorithm to be efficient in deriving input profiles for SUTs other than the four examples. The results of applying the optimised configuration and parameter settings to an additional SUT, `tcas`, in Experiment X is consistent with this hypothesis. However, we cannot confidently claim that the optimised algorithm is efficient more generally, across a wide range of SUTs, without further experimentation.

Diversity: In chapter 2, we noted that input profiles derived by the algorithm could lack diversity: the test sets generated from the profiles contained inputs from a smaller range of values than was necessary. It is possible that the algorithm configuration improvements in this chapter and the choice of optimal parameter settings, while substantially improving algorithm efficiency, may also affect the diversity of the derived profiles.

Chapter 4

Strategy Representation Using Stochastic Grammars

4.1 Introduction

4.1.1 Motivation

The outcome of the experimentation in the previous two chapters is an efficient algorithm for deriving concrete strategies for statistical testing. For the SUTs considered in chapter 3, input profiles with a suitably high minimum coverage probability can be generated in a few minutes using computing resources typical of a desktop PC.

However, the type of SUTs to which the algorithm can currently be applied is restricted by the assumption made in section 2.2.1 that the input to the SUT is a fixed number of numeric values. The objective of the research described in this chapter is to widen applicability of the technique through the use of a new representation for input profiles based on stochastic grammars.

Test data is often generated using grammars when the inputs to the SUT are highly structured; such approaches are discussed in section 4.2 below. We propose here to use a grammar as common representation for both relatively simple input domains—such as the SUTs considered in the previous chapters—as well as highly-structured domains. It is a flexible approach that enables the representation of inputs incorporating categorical as well as numeric arguments, inputs expressed as variable-length sequences, and inputs that satisfy complex validity constraints. In addition, the proposed grammar-based representation shares structural featues with the binned Bayesian network representation (section 4.3.4); this similarity facilitates the re-use of optimal configuration and parameter settings derived in the preceding chapter.

4.1.2 Contributions

The contributions in this chapter are:

- A new representation for input profiles based on stochastic context-free grammars that also retains key aspects of the current binned Bayesian network representation.
- An empirical demonstration that the grammar-based representation is compatible with the SUTs considered in earlier chapters in terms of both applicability and efficiency.
- Two case studies that illustrate the application of the grammar-based representation to SUTs with complex input domains. A binned Bayesian network would not have been a practical representation of a probability distribution over the input domains of these SUTs.

4.1.3 Chapter Outline

Related work on test data generation using grammars is discussed in section 4.2. The proposed grammar-based representation for statistical testing is described in section 4.3, and its implementation discussed in section 4.4.

Experiment XI (section 4.5) demonstrates how the grammar-based representation can be applied to the SUTs considered in chapters 2 and 3, and that the efficiency of the algorithm is broadly equivalent.

In Experiment XII (section 4.6) the grammar-based representation is the applied to two cases studies in order to demonstrate its use on highly structured input domains containing

variable-length sequences. The case studies are the circular buffer container class from the Boost C++ library and a simulation of mobile robots.

4.2 Grammar-Based Testing

Our proposed use of stochastic context-free grammars for representing input profiles is an example of *grammar-based testing*: the use of formal grammars to generate test data. In this section we describe the characteristics of formal grammars that facilitate the generation of test data, and discussed related work on grammar-based testing.

4.2.1 Formal Grammars

Grammar-based testing techniques represent an input to a SUT as a sequence of symbols. In formal language theory, such as that described by Martin (1996) and Hopcroft et al. (2006), a sequence of symbols is termed a *string*, and the set of legal symbols that can be used to form a string is the *alphabet* Σ. However, not every string that can be formed from the alphabet is necessarily a valid input to the SUT: in general the SUT's input domain corresponds to a subset of the all such strings. Such a subset is termed a *language*.

A *grammar* specifies which strings belong to the language using a set of rules that define valid compositions of symbols. Grammars may expressed as a set of symbol rewriting rules, and in this case the language is defined as the set of strings that can generated in this way from a given starting point. The rewriting rules, called *productions*, involve both symbols from the alphabet—referred to as *terminals* in this context—and additional symbols called *variables*. Each production rule specifies a valid rewriting of a subsequence of one or more symbols, at least one of which is a variable, to a new subsequence of zero or more symbols.

The following are examples of productions involving the terminal alphabet $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ and the variables $\mathcal{V} = \{\mathtt{S}, \mathtt{T}\}$. (We follow the convention of using upper case for the first letter of variable names, and lower case for terminals.)

$$
\begin{aligned}
\mathtt{S} &\rightarrow \mathtt{aT} \\
\mathtt{S} &\rightarrow \mathtt{aTb} \\
\mathtt{aT} &\rightarrow \mathtt{aTT} \\
\mathtt{aT} &\rightarrow \mathtt{aa} \\
\mathtt{aTTb} &\rightarrow \mathtt{c}
\end{aligned}
$$

These productions may also be written in a more compact form:

$$
\begin{aligned}
\mathtt{S} &\rightarrow \mathtt{aT} \mid \mathtt{aTb} \\
\mathtt{aT} &\rightarrow \mathtt{aTT} \mid \mathtt{aa} \\
\mathtt{aTTb} &\rightarrow \mathtt{c}
\end{aligned}
$$

where productions with the same left-hand side are combined and | separates each of the right-hand sides.

One of the variables is identified as the start symbol, and is usually denoted S. Generation begins from the string S and proceeds through the iterative application of productions to rewrite subsequences of symbols in the string until the string consists of only terminals.

In all but the simplest grammars non-determinism occurs whenever more than one production may be applied during a rewriting iteration. For example, if the string is currently aTTb, three of the above productions are applicable: $\mathtt{aT} \rightarrow \mathtt{aTT}$, $\mathtt{aT} \rightarrow \mathtt{aa}$, and $\mathtt{aTTb} \rightarrow \mathtt{c}$. In such a situation, any *one* of the productions could be chosen.

If a sequence of productions can be identified that rewrites the start symbol to a given string, then that string is in the language defined by the grammar. For example, the strings aa, aaab, c are members of the language defined by the grammar formed by the above productions since these strings may be generated as follows:

```
S ⇒ aT ⇒ aa
S ⇒ aTb ⇒ aTTb ⇒ aaTb ⇒ aaab
S ⇒ aTb ⇒ aTTb ⇒ c
```

Conversely, the strings `b`, `bc`, and `ab` are examples of strings that are not part of the language since they cannot be generated by any sequence of productions. (In these examples, we implicitly apply *leftmost derivation*: the productions considered at each rewriting step are those that match the leftmost variable in the partially-derived string.)

4.2.2 Context-Free Grammars

Context-free grammars (CFGs) restrict the form of the productions: the left-hand sides consist of only a single variable symbol. This restriction ensures that the same rewriting rules always apply to a variable regardless of what symbols precede or follow it in the string, i.e. free of its context.

The advantage of using CFGs is that the restriction on the form of the productions can simplify both the generation of strings from the grammar and checking that a given string belongs to the language specified by the grammar, the former being the most important process for test data generation. Although this restriction limits the type of languages that may be specified, CFGs can nevertheless represent useful forms of structured input data such as XML, algebraic expressions, and program source code.

For example, the following grammar specifies binary trees in which each node is labelled with an integer between 0 and 9. (In this and subsequent grammars, some of the symbols consists of multiple characters and so, for clarity, we separate consecutive symbols in grammar productions using spaces, and place terminals in single quotes.)

```
           S → Node
        Node → InternalNode | LeafNode
InternalNode → ‘(’ Label Node Node ‘)’
    LeafNode → ‘(’ Label ‘)’
       Label → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
```

Example of strings generated using this grammar are:

`(3(9(5)(5))(4(0)(1)))`

and,

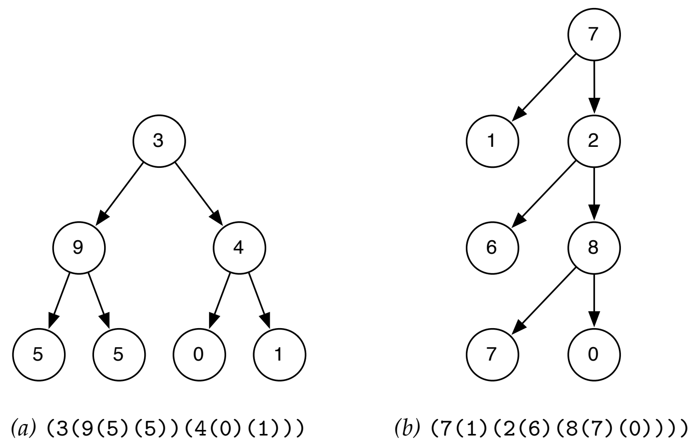
`(7(1)(2(6)(8(7)(0))))`.

The trees corresponding to these two strings are shown in figure 26. In this example the generated strings are an abstract representation of the corresponding trees: the strings would be interpreted by code in the test harness to construct trees using the objects or data types expected by the SUT. An alternative would be to use the grammar to directly construct code that when compiled or interpreted would construct the trees.

The production rules ensure that a valid binary tree is always generated. For example, the production for the variable `InternalNode` ensures that the internal nodes always have two children, and the productions for the variable `Label` ensure that all nodes are labelled with a integer between 0 and 9.

The grammar is recursive in the sense that generation can return to the `Node` variable as a result of the potential derivation:

`Node ⇒ InternalNode ⇒ ‘(’ Label Node Node ‘’)’ .`



(a) `(3(9(5)(5))(4(0)(1)))` *(b)* `(7(1)(2(6)(8(7)(0))))`

Figure 26 – Binary trees represented by two strings from the language defined by the example grammar.

This recursion enables the generation of trees with multiple nodes, and the non-deterministic choice as to whether a node becomes an internal or leaf node permits trees with different number of nodes to be generated.

4.2.3 Stochastic Grammars

A *stochastic grammar* is formed by annotating each production with a weight. At a string rewriting iteration where more than one production is possible, the non-determinism is resolved by choosing one of the productions at random, with the weights specifying a probability distribution over the possible productions.

This formulation is particularly convenient in the case of context-free grammars. For each variable, either none or all of the productions with that variable on the left-hand side may apply at each rewriting iteration, and so the weights of the productions for that variable specify a fixed probability distribution.

The weights in the stochastic grammar define a probability distribution over the language defined by the grammar, and thus over the test inputs generated by the grammar. As an example, consider the binary tree grammar discussed above as a stochastic context-free grammar. The relative weights attached to the productions `Node` $\rightarrow$ `InternalNode` and `Node` $\rightarrow$ `LeafNode` determine the frequency with which trees with a given height are generated: the higher the relative weight of `Node` $\rightarrow$ `InternalNode`, the greater the average height of the generated tree. Similarly, the weights associated with each production for the variable `Label` determine the frequency with which each integer 0 to 9 appears as a label.

4.2.4 Related Work

A number of researchers have described the use of formal grammars for the generation of test data. The motivation is typically to generate test inputs that pass the code in the SUT which checks the validity of the input in order to reach and test the 'core' functionality of the SUT.

We discuss below examples using both deterministic and stochastic grammar for this purpose. The use of a stochastic grammar is the more straightforward: a test set of a suitable size can be constructed by random sampling a chosen number of inputs from the grammar. When using deterministic grammars, it is less obvious how test sets of practical size can

be chosen from all possible strings that the grammar could generate. We discuss stochastic grammars first as they tend to occur earlier in the literature than deterministic equivalents, and we suspect that the more straightforward generation of inputs from stochastic grammars is the reason for this.

Stochastic Generation Maurer (1990) uses a stochastic grammar to test the functionality of VLSI circuits in simulation. The use of grammar to generate random, but structurally correct, test data was motivated by the poor fault-detecting ability of more systematic test data generation strategies. The stochastic grammar used by Maurer is one developed specifically for this purpose, called Data-Generation Language (DGL). Weights were used to bias the generation to test data that—based on the testers' experience—is likely to detect most faults: for example, binary inputs with few zeros, or with few ones, tend to find bugs in VLSI arithmetic circuits. An interesting feature of DGL is the annotation of productions with 'action routines' written in C. These small pieces of code may be used to simplify the representation of the language, or to predict test outputs and therefore act as an automatic oracle.

A natural application of grammar-based testing is the testing of compilers. The test input to the compiler is textual source code, but a random selection of characters is very unlikely to test anything but error detection in the lexical analysis initially performed by the compiler; to reach and test later stages of the compiler, more structured input text is required. McKeeman (1998) describes the use of stochastic grammars for this purpose when testing a C compiler: increasingly sophisticated grammars enabled testing of all parts of the compiler including those requiring semantically correct source code.

To test the verifier and compiler/interpreter of a Java Virtual Machine, Sirer and Bershad (1999) use a grammar to generate Java bytecode. The grammar, called lava, is stochastic and context-free. However, productions may be annotated with, firstly, a limit on the number of times that the production may be applied; secondly, code fragments similar in nature to the 'action routines' of DGL; and thirdly, guards in the form of Boolean expressions. The limit on the number of times a production may be applied is used enforce compliance with bounds on the size of structures within the bytecode. The code fragments and guards allow the implementation of context-sensitive features.

Deterministic Generation Majumdar and Xu (2007) describe a solution to the problem of determining a practical subset of the language to be used as test inputs: a grammar is used to ensure that inputs are structurally valid, but concolic execution (a combination of symbolic and dynamic execution) is used to minimise the number of test cases required to achieve structural coverage in the code. Sets of terminals in the original grammar are replaced by symbolic constants to form a 'symbolic grammar', and input strings are sampled exhaustively from this smaller grammar. For each 'symbolic' input, concolic execution is used to systematically derive a minimal set of concrete test inputs that exercise as many structural elements in the SUT as possible. The authors applied this technique to five SUTs requiring structured inputs and showed that the combination of grammar-based and concolic techniques outperforms naive grammar-based and traditional concolic execution used in isolation in terms of the coverage achieved within a fixed time.

Godefroid et al. (2008) also combine grammar-based specification of structured input data with symbolic execution for the testing of compilers and interpreters; they describe the technique as 'grammar-based whitebox fuzzing'. A context-free grammar is used to generate valid source code that is used as a seed input for symbolic execution. The constraints gener-

ated by symbolic execution are expressed in terms of tokens arising from the lexical analysis in the compiler or interpreter; a custom solver derives solutions to these constraints that also satisfy the context-free grammar. An empirical comparison of this approach against other automated test generation strategies for a JavaScript interpreter showed that grammar-based whitebox fuzzing achieved the best coverage in the 'deepest' module of the interpreter: the module that performs code generation.

Hoffman et al. (2010) use production annotations to reduce the size of an exhaustively enumerated context-free grammar. Firstly, a limit may be placed on the number of times a production is applied when generating strings; this limit is equivalent to the annotation used by Sirer and Bershad (1999). Secondly, a set of independent variables can be identified for which a covering array, rather than the entire cross-product of productions, is enumerated. A covering array of strength n (where n is the less than the number of independent variables) requires only that the cross-product of the productions for each subset of n variables is enumerated: this requires fewer strings than the full cross-product. The use of grammars annotated in this way is demonstrated on two case studies: the 'sanitisation' code of an RSS client for which inputs are XML with a specified format; and a network security appliance for which inputs are TCP packets.

Relationship to Our Proposed Grammar Our proposed grammar has the same purpose as the related work discussed here: to permit the representation of highly-structured input data, and—as for the stochastic grammar-based approaches—to 'bias' the probability distribution over the language of valid inputs in such as way as to improve the fault-detecting efficiency.

It is noticeable that many of the grammar-based testing techniques enhance a formal grammar with code fragments, limits, and other annotations in order to facilitate the representation of valid inputs. As described in the next section, we similarly enhance a formal stochastic context-free grammar with two novel features that we hypothesise will improve the representation of input profiles and the ability of a search-based algorithm to derive them.

4.3 *A Stochastic Context-Free Grammar Representation*

4.3.1 *Overview*

We propose to use a stochastic context-free grammar as a new representation for input profiles in our search algorithm. We claim that such a representation is capable of representing a wider range of input types than the binned Bayesian network representation: the use of a formal grammar enables the representation of highly-structured inputs, and the use of recursion within the grammar facilitates the generation of variable-sized inputs from a compact representation.

The proposed representation is a form of stochastic context-free grammar; similar forms are used by a number of the grammar-based testing techniques described in section 4.2.4. However, there are three novel features to the representation:

- production weights that are conditional on the usage of productions earlier in the string generation;
- an efficient representation for variables that generate scalar numeric data types;
- the application of automated search to the representation.

The following subsections describe the novel features of the SCFG representation in more detail.

(In this remainder of this chapter, we refer to the binned Bayesian network representation concisely as 'BBN', and the representation based on a stochastic context-free grammar as 'SCFG'.)

4.3.2 *Conditional Production Weights*

The distribution of weights that annotate the productions of a given variable are permitted to be conditional on productions used earlier in the generation of the string. Such dependencies are defined by links from parent to child variables: the production weights of a child variable are conditional on which production rules were most recently used for its parent variables.

To illustrate this conditionality, we use the following grammar as an example:

S $\rightarrow$ AS | BS | h
A $\rightarrow$ c (w_1) | d (w_2) | e (w_3)
B $\rightarrow$ f | g

The annotations (w_i) are the weight associated with the production. For clarity, only the weights associated with the variable A are shown.

If variable A were conditionally dependent on parent variable B, then there would normally be two different sets of values for the weights w_1, w_2, w_3 representing different probability distributions over the production rules for A. The first distribution would be used if B $\rightarrow$ f had been the most recently used production for variable B, and the second if B $\rightarrow$ g has been used most recently.

However, there need be no correspondence between the conditional dependencies of variables and the order in which they are used when generating a string from the grammar. For this reason, there is a third distribution for w_1, w_2, w_3 that is used should no production for B have been applied yet. In general, if a child variable is conditionally dependent on single parent variable, and the parent has n productions, then there are $n + 1$ distributions of weights over the productions of the child variable.

We permit a child to have zero, one, or multiple parents, and for the conditional dependencies between variables to form cycles: for example, `B` could be conditionally dependent on `A` even though `A` is conditionally dependent on `B`. We also permit a variable to be conditionally dependent on itself.

The effect of this conditional dependency is to permit a limited form of context-sensitivity in terms of the probabilities of applying particular productions, even though the grammar itself remains context-free.

As an example, consider again the grammar used to generated binary trees in section 4.2. The `Label` variable is defined in a context-free manner: the productions used to generate a label are the same for both internal and leaf nodes. However, an input profile that detects the most faults in the SUT might be one in which leaf nodes take labels close to 0 and internal nodes take labels closer to 9. This can be achieved if the `Label` variable is conditionally dependent on the `Node` variable: the probability distribution over the labels can then change dynamically during generation depending on which one of the productions `Node` → `InternalNode` and `Node` → `LeafNode` was used most recently.

4.3.3 *Scalar Numeric Variables*

In the binary tree grammar of section 4.2.2, the variable `Label` represents an integer value in the interval $[0, 10)$. Since the interval has a small cardinality, it was feasible to represent this using ten production rules:

`Label` → ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

For larger intervals of the integers, and intervals of the reals (even if the reals are approximated by a floating point representation with finite cardinality), the use of one production for each value in the interval is impractical. In addition to increasing the memory used by the representation, the large number of productions would reduce the efficiency of any search method applied to the grammar.

Instead, we propose a representation for such variables similar to the binning mechanism used in the BBN representation. The interval $[a, b)$ of valid values taken by the variable may be partitioned in a small number of bins: $[a, c_1), [c_1, c_2), \cdots, [c_n, b)$. A production is then defined for each *bin* rather than each *value*:

`V` → $[a, c_1)$ | $[c_1, c_2)$ | $\cdots$ | $[c_n, b)$

For example, the variable `Label` may be represented using this mechanism as:

`Label` → $[0, 5)$ | $[5, 7)$ | $[7, 10)$

When such a bin is returned as a terminal when generating a string from the grammar, it is replaced by a concrete value chosen randomly using a uniform distribution over the interval.

We will use the term *binned variables* for grammar variables that use this mechanism, and *bin terminals* for the bins that partition the variable's interval.

4.3.4 *Relationship to Binned Bayesian Network Representation*

The SCFG and BBN representations share many features. We may equate SCFG *variables* to Bayesian network *nodes*: it is between these objects that the conditional dependencies in the probability distribution are defined. The bin terminals of binned variables in the SCFG are essentially identical in form to the bins at Bayesian network nodes. Indeed, any BBN

representation may be rewritten in the SCFG representation using binned variables. We demonstrate this equivalence in Experiment XI later in this chapter.

However, the SCFG representation extends the BBN in a number of ways. The first, and most important, is that the SCFG representation allow standard (i.e. non-binned) variables to refer to zero, one or more other variables in its production rules, and such references may lead to recursion. This permits the generation of structured and variable-length inputs. Secondly, the Bayesian network restricts conditional dependencies in the probability distribution to form directed acyclic graphs; the SCFG representation removes the acyclic restriction. We claim that these additional features enables the SCFG representation to represent many more types of input domain than the binned Bayesian representation.

4.3.5 *The Application of Automated Search*

The stochastic grammar-based testing techniques described in section 4.2.4 adjust the production weights to 'bias' the generation of test inputs to values with desirable properties. For example, McKeeman (1998) *manually* adjusts the weights to avoid non-terminating recursion in the grammar used to construct source code for testing a compiler, and to limit the chance of generating source code causing divide-by-zero errors. With a similar purpose to McKeeman's *manual* adjustments, we wish to allow the search algorithm to *automatically* adjust the weights in order to produce suitable input profiles.

Therefore, the search algorithm is able to modify the following aspects of the SCFG representation:

- for all variables: the production weights;
- for all variables: the conditional dependency between variables;
- for binned variables only: the number and length of bin terminals.

These aspects are modified by the following mutation operators, defined by analogy to the mutation operators applied to the BBN representation that are described in section 2.2.3.

M_{prb} increases or decreases the weight of a production by a factor ρ_{prb}. After mutation, the weights are normalised so that weights across the productions of a single variable form a probability distribution.

M_{len} increases or decreases the length of one bin terminal of a binned variable by multiplying or dividing it by a factor ρ_{len}. After mutation, the bin lengths are normalised so that the total length of the bin terminals are equal to the length of the interval represented by the variable.

M_{spl} splits a bin terminal into two new bins, each of half the length of the original.

M_{joi} joins two adjacent bin terminals to form a single bin with a length equal to the sum of the two original bins' lengths.

M_{add} adds a conditional dependency between two variables.

M_{rem} removes a conditional dependency between two variables.

In this chapter, we apply the same search method and fitness metric—random mutation hill climbing and the estimated minimum coverage probability, respectively—that were found

to be effective for the BBN representation. The following configuration features, which were found to improve the efficiency of the search process in chapter 3, remain available to the search algorithm.

Directed Mutation Directed mutation operates by analogy to that used for the BBN representation. For every string sampled, a record is kept of which *conditional* (i.e. distinguished by the most recently production in the parents of the variable) productions were chosen during the generation process. The strings that exercised the coverage elements having the minimum coverage probability are then used in conjunction with this data to direct mutation to the associated production weights and, for binned variables, bin terminals. As for the BBN representation, directed mutation is implemented using mutation groups, and requires the following three groups:

G_{edge} consists of the operators that modify conditional dependencies: M_{add} and M_{rem};

G_{bins} consists of the remaining operators that modify production weights and bin terminals directly: M_{len}, M_{spl}, M_{joi}, and M_{prb}, and applies to *all* productions and bin terminals;

G_{drct} consists of the same weight- and bin-modifying operators as G_{bins}, but applies them *only* to production weights and bin terminals that contributed strings that executed the elements having the minimum coverage probability.

Limit on the Number of Conditional Dependencies In Experiment VIIa (section 3.7.2), it was found that a limit placed on the number of parents at each node of the Bayesian network improved algorithm efficiency. By analogy, a limit is placed on the number of parent variables on which the production weights of a child variable may be dependent on in the SCFG representation.

Limit on the Number of Terminal Bins In Experiment VIIb (section 3.7.3), it was found that a limit placed on the number bins at each node of the Bayesian network improved algorithm efficiency. By analogy, a limit is placed on the number of bin terminals that a binned variable may have in the SCFG representation. The limit is a multiple of the square root of the number of coverage elements calculated using the equation (43).

Terminal Bin Initialisation In Experiment VIIc (section 3.7.5), it was found that initially creating a fixed number of bins at each node improved algorithm efficiency. By analogy, we allow the initialisation of a fixed number of bin terminals. In both cases the number of bins is expressed as a proportion of the maximum number of bin terminals calculated according to equation (44).

Limit on the Number of Fitness Evaluations In Experiment VIIg (section 3.7.8), it was found that algorithm efficiency was improved by placing a limit on the number of times the fitness of the same candidate profile is re-evaluated. The same limit is applied when searching using the SCFG representation.

Algorithm Parameters The search algorithm parameters when using the SCFG representation are shown in table 35. As a result of the similarity between the SCFG and BBN representations, the SCFG representation algorithm parameters have equivalent interpretations to those

listed for the BBN representation in table 26; moreover, the same number of parameters are used for each representation.

Parameter	Description
K	evaluation sample size
λ	neighbourhood sample size
ρ_{prb}	production weight mutation factor
ρ_{len}	bin terminal length mutation factor
W_{bins}	G_{bins} mutation group weight
W_{edge}	G_{edge} mutation group weight
W_{drct}	G_{drct} mutation group weight
w_{rem}	M_{rem} mutation operator weight
w_{add}	M_{add} mutation operator weight
w_{prb}	M_{prb} mutation operator weight
w_{joi}	M_{joi} mutation operator weight
w_{spl}	M_{spl} mutation operator weight
w_{len}	M_{len} mutation operator weight
μ_{prnt}	Maximum no. parent variables per child variable
μ_{bins}	Maximum no. bin terminals per binned variable
ζ	Initial no. bin terminals (as proportion of μ_{bins})
μ_{eval}	Maximum no. evaluations per profile

Table 35 – Algorithm parameters used when input profiles are represented as an SCFG.

4.4 *Implementation*

This section outlines the most relevant aspects of the implementation of the search algorithm using the SCFG representation.

In the next chapter, we propose the use of commodity GPU cards as a parallel computing environment in which to run the algorithm. Utilising GPU cards in this way is called general-purpose computation on GPUs (GPGPU). The use of GPGPU places constraints on the programming language in which the algorithm is implemented, the language features used, and the memory usage by the algorithm. (We describe these constraints in detail in section 5.2.6.) The algorithm implementation used for the research in chapters 2 and 3 would not satisfy the GPGPU constraints, and so enhancing the implementation to incorporate the SCFG representation would not be practical. Instead, we create a new implementation of the algorithm that satisfies the GPGPU constraints, and can therefore run on both a CPU and a GPU.

4.4.1 *GPGPU Constraints*

We distinguish two categories of source code according to the environment in which it will run:

Host: Code that will only ever run on the CPU. This category includes code that parses the algorithm parameter file, initialises the representation, and writes output to the log file.

Common: Code that runs on either the CPU (for the research described in this chapter) or the GPU (for the research described in chapter 5). This category contains the core parts of the search algorithm: the mutation operators, the generation of strings from the SCFG, fitness evaluation including execution of the SUT itself, the processing of the feedback data for directed mutation, and pseudo-random number generation.

Both host and common code are written in C++. However, GPGPU restricts the C++ language features that can be used by code in the common category. The most relevant of these restrictions are that memory cannot be allocated dynamically and that the C++ Standard Template Library is not available (this library was utilised extensively in the existing algorithm implementation).

One of the results of the empirical investigations in chapter 3 was that the algorithm efficiency is improved if limits are placed on the numbers of bins and parents at each node in the Bayesian network. In section 4.3.5 we proposed to retain equivalent limits on the number of parent variables per child variable and the number of bin terminals per binned variable in the SCFG representation. These limits enable an upper limit to be placed on the memory used by the SCFG representation for a specific SUT. Therefore, it is possible to allocate memory statically at the start of search algorithm for the SCFG representation. The representation is then manipulated directly rather than via the Standard Template Library.

4.4.2 *Assessing Valid Mutations*

In the existing algorithm implementation, the entire set of possible mutations is assessed to determine which of these mutations are valid, and then one atomic mutation is chosen at random from the set of *valid* mutations. For example, some mutations that add an edge between nodes may be invalid because they create cycles in the network: a mutation would be chosen at random only from those that add edges without creating a cycle.

For simplicity, the implementation of the SCFG algorithm chooses a mutation from the set of *possible* (but not necessarily valid) mutations, and whether the chosen mutation is valid is assessed only once it has been selected. If the chosen mutation proves to be invalid, another mutation is chosen at random, up to a maximum of four attempts.

Checking the validity of all possible mutations can be time-consuming, and so by checking the validity of a single mutation after random selection is likely to improve run time. The disadvantage is that, rarely, no valid mutation may be made even after four attempts: in this case the neighbour created by mutation is no different from the current solution.

4.4.3 Representation of Binned Variables

We place a limit of 2^{16} on the length of the interval represented by the variable and permit only integer values for both the total length of the interval and for length of bin terminals that partition the interval.

This implementation decision enables the use of the same code for both real and integer variables, reduces the memory required since lengths are represented by two bytes rather than the four bytes required by a floating point representation, avoids the need to handle floating point precision issues when generating strings from the grammar, and improves performance during string generation. If the variable represents an interval of the reals, or an interval of the integers longer than 2^{16}, the interval specified by the grammar can be scaled to meet these constraints, and then converted to the actual interval when strings sampled by the grammar are interpreted prior to executing the SUT. We additionally impose a minimum value of 1 for the length of bin terminals; if instead a minimum of value of 0 were permitted, the length could never be mutated to a larger value using the M_{len} mutation operator which multiplies the current length by a constant factor.

4.4.4 Representation of Production Weights

Production weights are represented by integer values with a maximum value of 2^{16}.

As for the binned variables, this implementation decision reduces memory required by the representation compared to the alternative of storing weights as floating point values, and both simplifies and speeds up the sampling of strings from the grammar. After initialisation and mutation, weights are normalised so that the sum of weights for a variable is $256n_{\text{prod}}$ where n_{prod} is the number of productions (or bin terminals) for the variable. We additionally impose a minimum value of 1 for any weight; if instead a minimum of value of 0 were permitted, the weight could never be mutated to a larger value using the M_{prb} mutation operator.

4.4.5 Limits on the Length of Generated Strings

When the SCFG representation permits recursion, particular choices of production weights can lead to very long strings being generated, and in some cases the generation may not terminate at all. Therefore limits are placed both on the length of strings generated (where the length is the number of terminal symbols) and on any intermediate string (which may additionally include variable symbols) during the generation; the limits are denoted L_{string} and L_{inter} respectively. Generation of a string terminates as soon as either limit is reached. The code that interprets the generated strings in order to construct test data can choose to either discard such abnormally truncated strings or attempt to interpret them sensibly. For

the experiments described in this chapter, the latter choice is made. Both of these limits are parameters to the algorithm.

4.4.6 Pseudo-Random Number Generation

In order to ensure a fair comparison between algorithm implementations running on the CPU and GPU, we require a common pseudo-random number generator (PRNG). The GNU Scientific Library was used to generate pseudo-random numbers in the existing algorithm implementation. However, this library is incompatible with the GPGPU constraints. Although our chosen implementation of GPGPU, NVIDIA's CUDA framework, provides a PRNG which can be called in both CPU and GPU code, preliminary experiments showed that the generator was very slow when called from the CPU compared to the GPU performance. Instead, a bespoke 64-bit implementation of Marsaglia's XORShift algorithm (Marsaglia, 2003) was used as the PRNG on both the CPU and GPU.

4.5 Experiment XI – Compatibility

4.5.1 Objectives

In section 4.3.4, we argued that any input profile represented using the BBN representation–i.e. profiles over an input domain consisting of a fixed number of scalar numeric arguments—could also be represented using the SCFG representation. In other words, the set of profiles representable using a SCFG is a superset of the profiles representable using a BBN.

In this experiment, the first objective is to demonstrate how the SCFG representation may be used to represent the input profiles of SUTs that were represented using a BBN in previous chapters. The second objective is to assess how the efficiency of search algorithm changes when the SCFG representation is used in place of the BBN representation for these SUTs.

The research questions are therefore:

RQ-1 Can the SCFG representation be applied to SUTs whose input domains were previously represented using a BBN in chapter 3: `bestMove`, `nsichneu`, `cArcsin`, `fct3`, and `tcas`?

RQ-2 How does the change of representation from BBN to SCFG affect the efficiency of the search algorithm?

Normally the choice of parent and child variables for the conditional dependencies proposed in section 4.3.2 is unrestricted: the production weights of any variable may be conditional on any other variable in the grammar (including itself). However, we demonstrate below that grammars representing a fixed number of scalar numeric arguments require no non-determinism, and so the order in which variables are processed (rewritten) when generating a string is predictable. It is therefore possible to restrict conditional dependencies so that a variable may only be conditional on a variable that is processed earlier during the string generation process. This motivates a further research question:

RQ-3 Does restricting conditional dependencies to the variable processing order improve algorithm efficiency?

4.5.2 Preparation

SCFG Representation To construct a suitable SCFG representation for each of the SUTs the following pattern is applied:

$$\begin{aligned} \mathtt{S} &\rightarrow V_1 \ V_2 \ \ldots \ V_n \\ V_1 &\rightarrow [a_1, b_1] \\ V_2 &\rightarrow [a_2, b_2] \\ \vdots &\rightarrow \vdots \\ V_n &\rightarrow [a_n, b_n] \end{aligned}$$

where the V_i are the arguments to the SUT, each one of which would be a node in the BBN representation. Each V_i is a binned variable representing the interval $[a_i, b_i]$.

For example, the SUT `bestMove` has two parameters, `noughts` and `crosses` which both takes value in the range $[0, 511]$ (section 2.4.3). The SCFG representation is therefore:

```
      S → Noughts Crosses
Noughts → [0,511]
Crosses → [0,511]
```

Real-Valued Binned Variables Two of three arguments for the SUT `cArcsin` are real-valued: `inR` and `inC` are the real and imaginary components of the complex angle, and in section 3.4.2 we chose an input domain of $[-10, 10)$ for both. As a result of the implementation decision to represent all binned variables, both integer and real, using 16-bit integers, the following SCFG is used:

```
        S → InR InC realFlag
      InR → [0,65534]
      InC → [0,65534]
 realFlag → [0,1]
```

The integer value, x, returned for either of the variables `inR` and `inC` when a string is generated is converted to a real-valued argument, x' in the range $[-10, 10)$ using the transformation:

$$x' = -10 + 20\frac{x+\gamma}{65535} \tag{48}$$

where γ is a random variable sampled from the interval $[0, 1)$ using an additional call to the PRNG.

Implementation of Conditional Dependency Constraint Research questions RQ-3 requires conditional dependencies to be restricted to the order in which the variable are processed during string derivation. For the SUTs considered in this experiment, the above grammar pattern ensures that the processing order is the same as the order in which the variables are defined in the grammar. Therefore the implementation of this constraint on conditional dependencies uses the *definition* order and so avoids the need for additional processing of the grammar to determine the *processing* order.

Algorithm Parameters Experiment VIIIc (section 3.8.5) evaluated the efficiency of the algorithm using the BBN representation. Therefore the method of Experiment VIIIc is repeated using the SCFG representation, and the result compared against the earlier experiment. We additionally include the SUT `tcas`, using the results of Experiment X for comparison.

The optimal algorithm configuration and parameter settings used in Experiment VIIIc were the outcome of a series of tuning experiments. To ensure a fair comparison of the two algorithms, an equivalent series of tuning experiments would ideally be performed using the SCFG representation. However, undertaking such an equivalent series of experiment would take significant time and computing resources.

Since understanding the effect of representation on algorithm efficiency is a secondary objective of this chapter—the primary objective is to demonstrate the wider applicability of the SCFG representation—we make the pragmatic decision to re-use the configuration and parameter settings of Experiment VIIIc for the SCFG representation. We argue that the two representations are very similar in nature when the SUT input domains, as here, consist of a fixed number of scalar numeric values, and therefore it is reasonable to expect that the configuration and parameter setting that are optimal for the BBN representation are likely to be acceptably near optimal for the SCFG representation.

As a result of this decision, we are unable to make a direct comparison between the two representations: the choice of parameter settings biases the comparison in favour of the BBN representation. However the experiment is able to derive a *bound* on the algorithm efficiency (an upper bound when quantified as the number of SUT executions) when using the SCFG representation: the efficiency of the algorithm using the SCFG representation will be at least

as good as this bound if the parameter settings were tuned to the representation. Since the two representations are similar, we expect that the bound on the algorithm efficiency is sufficiently tight to enable a conclusion to be drawn as to effect of using the SCFG representation.

The parameter settings (duplicating those used for the BBN representation in Experiment VIIIc) are listed in table 36. The target minimum coverage probability for each SUT is the same as used throughout chapter 3.

Should the algorithm fail to attain the target probability—a situation which occurs only very rarely at these near-optimal parameter settings—it is terminated after a fixed number of iterations, τ_{iter}; in chapter 3 a similar mechanism was used, but based on the total number of SUT executions.

Parameter	Description	SUT	Setting
K	evaluation sample size		302
λ	neighbourhood sample size		9
ρ_{prb}	production weight mutation factor		54.002
ρ_{len}	bin terminal length mutation factor		2.755
W_{bins}	G_{bins} mutation group weight		1 000
W_{edge}	G_{edge} mutation group weight		144
W_{drct}	G_{drct} mutation group weight		10 584
w_{rem}	M_{rem} mutation weight		1 000
w_{add}	M_{add} mutation weight		689
w_{prb}	M_{prb} mutation weight		1 000
w_{joi}	M_{joi} mutation weight		886
w_{spl}	M_{spl} mutation weight		1 159
w_{len}	M_{len} mutation weight		1 028
μ_{prnt}	maximum no. parent variables per child variable		1
μ_{bins}	maximum no. bin terminals per binned variable		$(\eta = 2.0)$
ζ	initial no. bin terminals (as proportion of μ_{bins})		1.0
μ_{eval}	maximum no. evaluations per profile		10
τ_{iter}	max no. iterations		1×10^5
$\tau_{p_{\text{min}}}$	target min. coverage probability	`bestMove`	0.112
		`nsichneu`	0.028
		`cArcsin`	0.12
		`fct3`	0.05
		`tcas`	0.078
L_{string}	max. length generated string		32
L_{inter}	max. length intermediate string		32

Table 36 – The algorithm parameter settings used for the SCFG representation.

4.5.3 Method

For each of the five SUTs—`bestMove`, `nsichneu`, `cArcsin`, `fct3`, and `tcas`—an SCFG representation was constructed using the pattern described above. 64 trials were run for each SUT (each with a different seed to the PRNG) using the parameter settings of table 36. For each trial, the number of times the SUT is executed by the algorithm was recorded, and transformed—if the target minimum coverage probability is not attained—using the method described in section 3.3.3.

The procedure was repeated a second time with the conditional dependencies between grammar variables restricted to the order in which the variables are defined (and therefore processed during string generation).

SUT	`bestMove`	`nsichneu`	`cArcsin`	`fct3`	`tcas`	Average
Median Efficiency (10^6 execs)						
SCFG unrestricted dependencies	20.6	2.10	0.735	2.83	65.8	5.23
SCFG restricted dependencies	11.9	2.43	0.628	3.21	18.0	3.87
BBN	11.3	0.440	0.940	2.59	13.5	2.93
Significance (rank-sum p-value)						
SCFG unrestricted v. restricted	0.047 %	92 %	36 %	90 %	$< 10^{-5}$	$< 10^{-5}$
SCFG restricted v. BBN	82 %	$< 10^{-5}$	1.6 %	52 %	11 %	0.19 %
Effect Size (Vargha-Delaney A)						
SCFG unrestricted v. restricted	0.68	0.51	0.55	0.49	0.82	0.73
SCFG restricted v. BBN	0.49	0.80	0.38	0.53	0.58	0.66

Table 37 – Results of Experiment XI.

4.5.4 *Results*

The results are summarised in table 37.

For each SUT, the median algorithm efficiency—in units of 10^6 executions—is calculated over each of 64 trials for both SCFG with both unrestricted and restricted dependencies between variables. The results for the BBN representation are taken from Experiment VIIIc for `bestMove`, `nsichneu`, `cArcsin`, and `fct3`; and from Experiment X for `tcas`.

To estimate the average efficiency over all five SUTs , 64 mean values are first calculated where the nth value is the geometric mean of the efficiency returned by the nth trial of each SUT. (The choice of geometric mean for this purpose is discussed in section 3.2.5.) The median of these 64 geometric means is then reported in the column labelled 'Average'.

To assess the statistical significance of the differences in algorithm efficiencies, the non-parametric Mann-Whitney-Wilcoxon (rank-sum) test was used to compare the SCFG with unrestricted and restricted dependencies, and the SCFG with restricted dependencies to the BBN. The non-parameteric Vargha-Delaney test is applied to calculate effect size for the same sets of responses. (The Vargha-Delaney statistic is described in section 2.6.4.)

4.5.5 *Discussion and Conclusions*

The results show that it is possible to derive input profiles using the SCFG representation: the efficiencies reported in table 37 are much less than if the algorithm were to be terminated only when reaching the maximum number of iterations rather than on finding a suitable profile (RQ-1).

The results also show that there is an improvement in algorithm efficiency when the conditional dependencies in the SCFG representation are restricted to the order in which the variables are processed, compared to unrestricted dependencies: the average efficiency across the five SUTs is improved by approximately 26%, from 5.23×10^6 to 3.87×10^6 executions, as a result of this restriction, and the difference is statistically significant at the 5% level (RQ-3). However the effect of the restriction differs greatly between individual SUTs: for `tcas` the restriction improves the efficiency greatly, while for `nsichneu`, `cArcsin`, and `fct3` there is no statistically significant difference.

The efficiency of the algorithm when using the SCFG representation (with restricted dependencies) is slightly worse than the BBN representation: on average, the use of the SCFG representation decreases efficiency by approximately 32% from 2.93×10^6 to 3.87×10^6 executions. This difference is statistically significant at the 5% level, and the effect size is medium

according to the guidelines suggested by Vargha and Delaney (2000). Again, the effect on individual SUTs is more complex: there is no statistically significant difference between the representations for `bestMove`, `fct3` and `tcas`, while the SCFG representation decreases the efficiency of the algorithm applied to `nsichneu` greatly.

As discussed above, we must treat the efficiency for the SCFG representation as a bound on the tuned efficiency. Thus we conclude that the restricted SCFG representation results in an average algorithm efficiency that is, *at worst*, 32% less efficient than the BBN representation (RQ-2). We argue that this worst-case difference is small enough that it is reasonable to use the SCFG representation as an alternative for the BBN representation for input domains consisting of a fixed number of scalar numeric arguments.

We argued that the representations are very similar in nature when representing domains consisting of a fixed number of scalar numeric arguments, especially when the variable dependencies are restricted to the derivation order. Nevertheless there are differences in both the algorithm itself and its implementation between the two representations:

- There is no equivalent to the start variable in the BBN representation. The neighbourhood of a candidate profile represented as an SCFG is therefore larger than the equivalent BBN: mutations may add dependencies with the start variable as the parent, and the weights of the start variable's production may be changed. Since the start variable has only one production in the pattern used for this experiment, and this production is used prior to any other variable being derived, neither of these mutations has any effect. However, the algorithm efficiency is likely to be decreased as a result.

- The new algorithm implementation (used only with the SCFG representation) assesses mutations for validity after they are selected; this may rarely result in a neighbour that is no different from the current candidate input profile. This implementation decision is designed to improve algorithm run time, but may decrease efficiency measured in terms of the number of SUT executions.

- The new algorithm implementation stores bin terminal lengths and weights as integers and enforces a minimum length and weight of 1. The existing algorithm implementation (used with the BBN representation) uses floats to store bin lengths and probabilities and has no minimum value.

These differences may offer an explanation for why the efficiency of the SCFG representation is not always equivalent to that of the BBN representation at the same parameter settings. However, in the absence of further experimentation, we are unable to provide an explanation for the large differences in efficiency observed in a few specific cases: the very poor efficiency of the unrestricted SCFG representation applied to `tcas`, and the poor efficiency of both restricted and unrestricted SCFG representation when applied to `nsichneu` compared to the BBN representation.

4.5.6 *Threats to Validity*

Generality Although a reasonably diverse set of SUTs have been used for this experiment, we are unable to generalise, with confidence, the results to the wider class of SUTs with a fixed number of scalar numeric arguments.

Parameter Settings For the reason discussed above, the parameter settings were not tuned for SCFG representation, and observed efficiencies for the SCFG representation must be considered a bound on the tuned efficiency of the algorithm. This limits the conclusions we are able to draw. We cannot, for example, state confidently whether there is, in general, a difference in algorithm efficiency between the two representations. Moreover, some of the unexplained observations—such as the very poor performance of the SCFG representation when applied to `nsichneu`—may be artefacts resulting from this decision.

4.6 *Experiment XII – Wider Applicability*

4.6.1 *Objectives*

In this experiment we demonstrate the wider applicability of the SCFG representation by applying it to two new example SUTs whose inputs could not be easily represented using the BBN representation.

The research question addressed is:

RQ-1 Can the SCFG representation be used to derive input profiles for SUTs whose inputs are highly structured, vary in size, and contain a mixture of categorical and numeric scalar components?

4.6.2 *Preparation*

We describe here the two new SUTs and the algorithm parameters used in this experiment. The instrumented source code of both SUTs is available at: `http://www-users.cs.york.ac.uk/smp/supplemental/`.

SUT `circBuff` The first new SUT is the implementation of a cyclic buffer container in the BOOST C++ library, version 1.50 (Gaspar, 2012). The container is random access but has a fixed size: if new entries are inserted beyond this size, they overwrite the entries at the beginning of the container.

The public interface of the SUT is a single class. The testing objective is coverage of all branches in public methods of this public interface class; coverage of private methods in the class, and methods in related helper classes called by these public methods, are not considered to be part of this objective. We additionally exclude the copy constructor and assignment operator from the set of public methods covered in order to simplify the input domain: testing these methods would require the construction and maintenance of multiple buffer objects. For this objective, the number of branches to be covered is 78. The public methods of the class constitute 1855 lines of code, including non-executable code, but not all methods include branched code.

The predicates controlling many of the branches depend on the state of the class, and this state is itself constructed and modified by calls to the class' public methods. For this reason it not possible to cover all the branches using isolated calls to methods: the calls must be part of a sequence that constructs the object and then makes multiple calls to public methods in order to set up state. The method calls may cover branches in their own code as well as set up an appropriate class state for subsequent methods calls.

We regard such a sequence of method calls a single test input, and use the grammar is shown in figure 27 to define valid inputs.

The terminals, such as `'push_front'`, define method calls and may be followed by numeric arguments to the calls. Terminals with suffices '[0]' or '[1]', such as `'insert[0]'` and `'insert[1]'`, are used to distinguish between overloaded methods.

The grammar defines a valid structure for the sequence of method calls: a constructor, method calls to the constructed object that populate the buffer, method calls that operate on the buffer, and finally a call to the destructor. The recursion in productions for the variables `PopulateOperations` and `Operations` enable a varying number of method calls to be made.

```
                  S → Constructor PopulateOperations Operations ‘destructor’
        Constructor → ‘circular_buffer[0]’ BufferCapacity
                    | ‘circular_buffer[1]’ BufferCapacity BufferSize
 PopulateOperations → PopulateOperation PopulateOperations | PopulateOperation
  PopulateOperation → ‘push_front’ | ‘push_back’
         Operations → Operation Operations | Operation
          Operation → ‘push_front’ | ‘push_back’ | ‘linearize’
                    | ‘rotate’ IteratorPos | ‘set_capacity’ BufferCapacity
                    | ‘resize’ BufferSize | ‘rset_capacity’ BufferCapacity
                    | ‘rresize’ BufferSize
                    | ‘insert[0]’ IteratorPos | ‘insert[1]’ IteratorPos Number
                    | ‘rinsert’ IteratorPos
                    | ‘erase[0]’ IteratorPos
                    | ‘erase[1]’ IteratorPos IteratorPos
                    | ‘rerase[0]’ IteratorPos
                    | ‘rerase[1]’ IteratorPos IteratorPos
                    | ‘pop_front’ | ‘pop_back’
                    | ‘front’ | ‘back’ | ‘operator[]’ Number
     BufferCapacity → [0,9]
         BufferSize → [0,9]
        IteratorPos → [0,9]
             Number → [0,9]
```

Figure 27 – A grammar that defines valid inputs for testing `circBuff`.

The four binned variables at the end of the grammar correspond to the four different meanings of the numeric arguments to some of the method calls. Conditional probability distributions over the productions at these four binned variables would permit different distributions over the scalars for different method calls, should this be necessary.

The strings generated by the grammar are interpreted by a test harness and applied to an instrumented version of the SUT. The object returned by a call to a constructor method is stored by the harness and subsequent method calls are made on that object. For terminals that indicate method calls which take arguments, the test harness reads ahead in the string to retrieve the argument values. For methods that take an object to be stored in the buffer as an argument, the test harness creates an object from a fixed ‘dummy’ class.

An alternative approach would be to use a grammar that output C++ source code to call the methods; the code would be compiled and linked with the test harness and then executed in order to test the SUT. For this SUT, the time taken to compile the code would be much greater than the execution of the SUT, and so this approach is not used.

SUT `epuck` ‘E-pucks’ are small, relatively cheap robots used in robotics research. E-pucks move using two independently-controlled wheels, and may be equipped with sensors and actuators such as infrared proximity sensors, communication LEDs, cameras, and microphones (Mondada et al., 2009; École Polytechnique Fédérale de Lausanne, 2012).

The SUT `epuck` is a lightweight simulator of e-pucks developed in C by Paul O’Dowd at the University of Bristol. Features supported by the simulator include proximity detection using IR, communication between robots, the placement and detection of light sources that act as beacons, the placement and detection of coded patches on the floor, and the placement of static obstacles in the environment. The simulator adds a small amount of random noise to the output of sensors and to the movement of the robot wheels in order to simulate realistic imperfections in the robots’ environment and hardware responses.

As discussed in section 4.4.1, the CUDA architecture used for the research in chapter 5 does not support dynamic memory allocation. In order to ensure a reliable comparison between the SUT used here running on a CPU, and that used in the next chapter running on a GPU, the e-puck simulator code was modified to use static memory allocation. The functional impact of this change is an upper limit on the number of e-pucks, obstacles, patches, and light sources.

The testing objective is coverage of a simple controller algorithm which attempts to avoid objects and to remain in the vicinity of a food patch when one is detected. The high-level flowchart of the controller is shown in figure 28. The objective is to exercise each of the four condition branches of the flowchart at least once during each 'mission'. A mission lasts for 500 iterations of the simulator, equivalent to 10 s of real-world time; each mission is a separate test case. The entire simulator consists of approximately 1 500 lines of code; the controller algorithm is implemented in 53 lines of code.

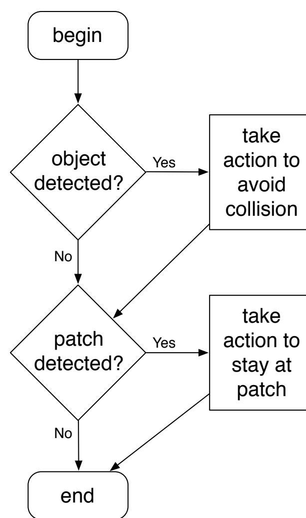


Figure 28 – The high-level control flow structure of the e-puck controller.

The input for a test case is the configuration of robot and its environment for the mission: the position of the robot and the direction in which it is facing, and the placement of obstacles and food patches inside a walled arena of radius 750 mm. Figure 29 shows an example of a mission configuration.

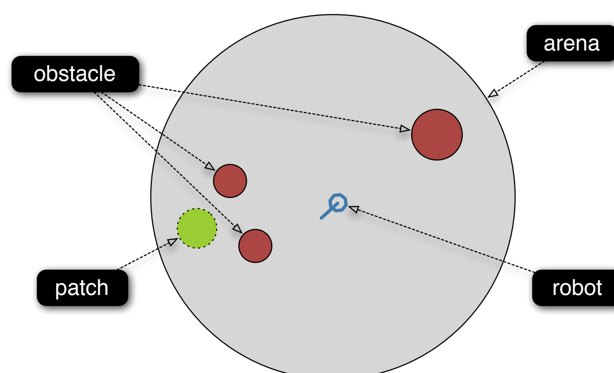


Figure 29 – Plan view of an example mission configuration.

The grammar we use to define a valid test input, i.e. a mission configuration, is shown in

```
                    S → EpuckCluster ObjectClusters
         EpuckCluster → EpuckClusterAzimuth EpuckClusterDistance Epucks
               Epucks → Epuck
                Epuck → EpuckAzimuth EpuckDistance EpuckAngle
        ObjectCluster → ObjectClusterAzimuth ObjectClusterDistance Objects
       ObjectClusters → ObjectCluster | ObjectCluster ObjectClusters
              Objects → Object | Object Objects
               Object → Obstacle | Patch
             Obstacle → ObstacleAzimuth ObstacleDistance ObstacleRadius
                Patch → PatchAzimuth PatchDistance PatchRadius
  EpuckClusterAzimuth → [0,359]
 EpuckClusterDistance → [0,199]
         EpuckAzimuth → [0,0]
        EpuckDistance → [0,0]
           EpuckAngle → [0,359]
 ObjectClusterAzimuth → [0,359]
ObjectClusterDistance → [400,599]
      ObstacleAzimuth → [0,359]
     ObstacleDistance → [0,99]
       ObstacleRadius → [10,99]
         PatchAzimuth → [0,359]
        PatchDistance → [0,99]
          PatchRadius → [10,99]
```

Figure 30 – A grammar that defines valid inputs (mission configurations) for testing epuck.

figure 30.

The grammar defines obstacles and patches in terms of groups we call object clusters. An example of two clusters is shown in figure 31. The object clusters are centered at least 400 mm from the centre of the arena so that the robot is not too near any cluster at the beginning of the mission. The objects in the cluster—obstacles and patches—are defined using distances from centre of the cluster and an angle relative to a line from the centre of the cluster to centre of the arena. The grammar has provision for similarly defining multiple e-pucks in a single cluster, but for this experiment the number of e-pucks is always one, and so this feature is redundant (although the relevant variables and productions are retained).

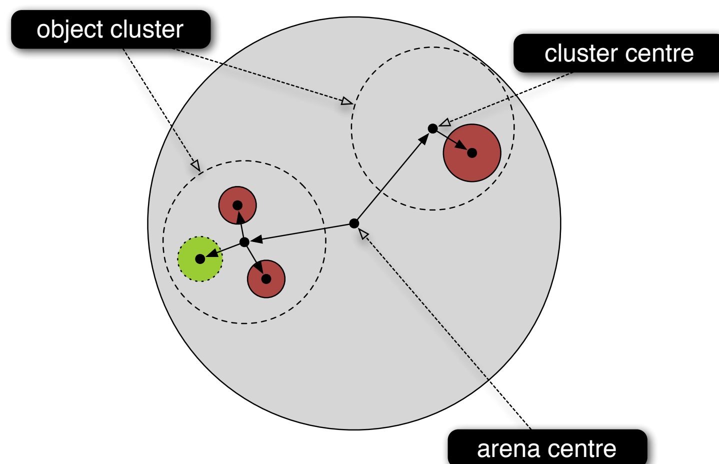


Figure 31 – The construction of the robot's environment using object clusters.

The recursion in the grammar permits different numbers of object clusters, each with

different numbers of obstacles and patches, to be generated. Binned variables are used to generate the positions of clusters, obstacles, patches, and e-pucks; and the radii of obstacles and patches.

The output of the grammar is a string of numbers sampled from the binned variables. The numbers are 'labelled' with the binned variable from which they are generated, and so the strings are unambiguous without the need for additional terminals. Each string is interpreted by the test harness in order to configure the mission. In order to avoid exceeding the statically allocated memory in the simulator, we place a limit on the total number of obstacles and patches created—three in each case—even if the generated string contains more than this number.

Algorithm Parameters For Experiment XI, we argued that since the input domain consisted of a fixed number of scalar numeric arguments, it was reasonable to re-use, for the SCFG representation, the algorithm parameter settings that were found to be optimal for the BBN representation. Since the grammars proposed for `circBuff` and `epuck` do not represent a fixed number of scalar numeric arguments, there is no equivalent argument for using the same parameters for SUTs.

However, the objective is only to demonstrate that the SCFG representation *can* be used for these two SUTs; finding the optimal efficiency of the algorithm is not a goal. Therefore we again re-use the algorithm parameter settings found to be optimal for the BBN representation: in the absence of any other information, these values are the best estimate of viable parameters.

The parameter settings are therefore the same as those used in the previous experiment and listed in table 36. Table 38 lists those parameters specific to the new SUTs used in this experiment.

Parameter	Description	SUT	Setting
τ_{iter}	maximum no. iterations	`circBuff`	1×10^4
		`epuck`	500
$\tau_{p_{\min}}$	target min. coverage probability		1.0
L_{string}	max. length generated string	`circBuff`	64
		`epuck`	128
L_{inter}	max. length intermediate string	`circBuff`	64
		`epuck`	32

Table 38 – The algorithm parameter settings used in Experiment XII.

4.6.3 Method

64 trials of the algorithm using the SCFG representation were run for each SUT. No specific limit was set for the target minimum coverage probability (the algorithm parameter is set to 1.0), and so the algorithm normally ran until the maximum number of iterations has been reached. This limit was 1×10^5 for `circBuff`, but a lower value of 500 was used for `epuck` since each simulated mission takes a relatively long time to complete. In contrast to other experiments in this thesis, the response measured for each trial was the minimum coverage probability of the best profile derived during the algorithm run.

SUT	circBuff	epuck
Proportion of Non-Zero Minimum Coverage Probabilities	0.5	1.0
Lowest Minimum Coverage Probability	0	0.331
Mean Minimum Coverage Probability	0.0430	0.748
Highest Minimum Coverage Probability	0.175	1

Table 39 – Results of Experiment XII.

4.6.4 Results

Table 39 summarises the results. For each SUT, the table shows the proportion of trials for which the response (the minimum coverage probability of the best profile found during each trial) was non-zero; and the lowest, highest, and mean values of response across the 64 trials. (The mean rather than median is used here because for half of the trials for `circBuff`, the minimum coverage probability was zero and so a median value would not meaningfully summarise the non-zero results.)

4.6.5 Discussion and Conclusions

For both SUTs, the algorithm is able to derive input profiles with high minimum coverage probabilities. This supports our hypothesis that the SCFG representation is suitable for the type of input domains used by these two SUTs: highly structured, varying in size, and containing a mixture of categorical and numeric scalar components (RQ-1).

The results show `circBuff` to be a relatively hard problem for the algorithm: even after 1×10^5 iterations of the hill-climb, only half derive a profile with a non-zero minimum coverage probability. Nevertheless, the algorithm is capable of occasionally deriving particularly effective profiles in which each of 78 branches has a probability of least 0.175 of being executed by any one sequence of method calls sampled from the profile.

The SUT `epuck` is an easier problem for the algorithm: in only 500 iterations, all trials derived profiles with an estimated minimum coverage probability of at least 0.331, and one trial achieved a probability of 1.0.

Figures 32 show a sample of mission configurations sampled from the profiles derived by some of the trials for `epuck`. The minimum coverage probability is also shown for each of the selected profiles.

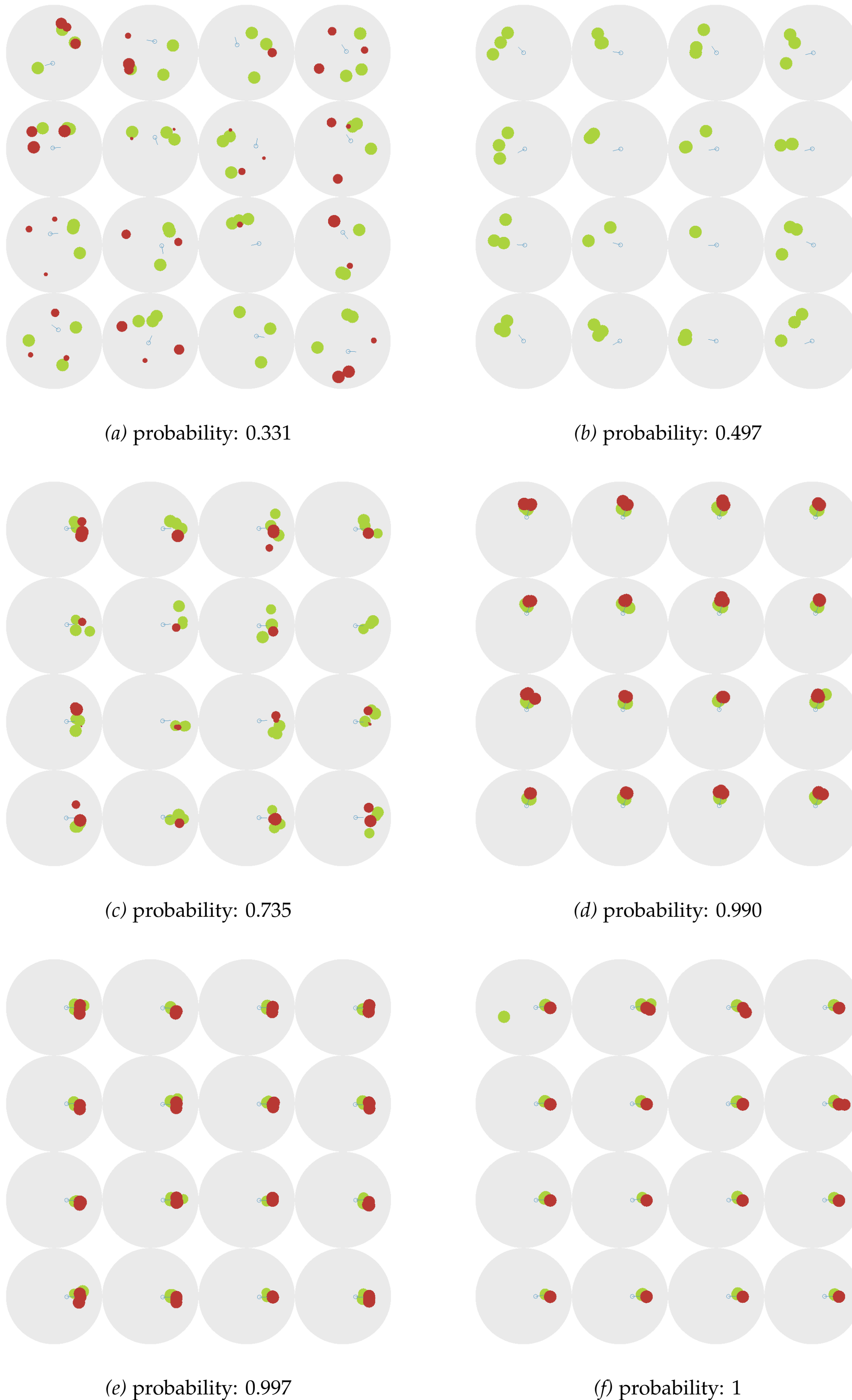
We note two features of these samples. Firstly, the diversity visibly decreases as the minimum coverage probability increases: the mission configurations in sample (c), from a profile with a probability of 0.735, show reasonably large differences; the mission configurations in sample (f), from a profile with a probability of 1.0, are very similar. This pattern of reduced diversity at the very highest minimum coverage probabilities matches the same pattern observed in Experiment III (section 2.7).

Secondly, there appears to be a bias for object clusters to be on the right of the arena: this direction corresponds to an angle of $0°$. We speculate that the cause of this bias is the relatively small number of bin terminals that the variable `ObjectClusterAzimuth`—which specifies the angle of a cluster from the origin—may be partitioned into. Since the SUT has only four branches, the maximum number of bin terminals for this variable is $2 \lceil \sqrt{4} \rceil = 4$. As the diversity decreases, one of these four bins will, by chance, dominate the others by having a higher probability. With only four bins, it is likely that the dominating bin is at the beginning

or end of the partition of the interval $[0, 359]$ for the variable `ObjectClusterAzimuth`. These bins, near $0°$ or near $359°$, both represent object clusters near the right of the arena.

4.6.6 *Threats to Validity*

Generality This experiment considers only two SUTs. It may be reasonable to argue that the SCFG can *represent* many domains that are highly structured, may vary in size, and contain a mixture of categorical and numeric scalar components. However, we cannot argue with confidence that the search algorithm will always be a viable approach for *deriving* suitable input profiles using this representation.



(a) probability: 0.331

(b) probability: 0.497

(c) probability: 0.735

(d) probability: 0.990

(e) probability: 0.997

(f) probability: 1

Figure 32 – Mission configurations sampled from the best input profile derived by six of the 64 trials for `epuck`. Each sample shows 16 mission configurations. The large light grey circle is the arena; the smaller dark grey circles inside the arena are obstacles; the smaller mid-grey circles are patches. The caption for each sample indicates the estimated minimum coverage probability of the profile.

4.7 Conclusion

In this chapter, we proposed and demonstrated a new representation for concrete strategies (input profiles) suitable for statistical testing.

The new representation is based on stochastic context-free grammars (SCFG), and enhances the grammar with two novel features adapted from the binned Bayesian network (BBN) representation used in previous chapters of this thesis:

- The weights of production rules may be conditional on the production rules applied earlier when generating a string from the grammar. This enables a limited degree of context-sensitivity in the representation.
- Variables representing scalar numeric quantities are handled as a special case: the production rules represent a partition over the range of values that the variable may take. A production rule for such a variable is equivalent to a bin in the BBN representation. The number of production rules, and the proportion of the interval represented by each production, may be modified by the search process.

The representation has a number of advantages over the binned Bayesian network representation (BBN) used in previous chapters. It is capable of representing:

- highly-structured inputs and compound data types;
- mixtures of categorical and scalar inputs;
- variable-length sequences.

These capabilities were demonstrated on two case studies in Experiment XII: an object-orientated container class for which inputs were a sequence of method calls, and a robot simulator for which inputs were a highly-structured description of the robot's environment.

If an input profile of a SUT can be represented using the BBN representation, it can also be represented by the SCFG representation. Moreover, Experiment XI demonstrated the algorithm efficiency using the SCFG representation is not substantially worse than that for the BBN representation for such SUTs, even though the parameter settings were not tuned for the SCFG representation.

Unaddressed Issues We note the following unaddressed issues:

Algorithm Configuration and Parameters: Experiment XI demonstrated that the algorithm configuration and parameter settings optimised for the BBN representation were also good choices for the SCFG representation. However, this experiment considered only SUTs with inputs representable as a BBN, i.e. a fixed number of scalar numeric arguments. The case studies Experiment XII suggest the same configuration and parameter settings may be *viable* for SUTs with more complex input domains, but the nature of the *optimal* algorithm configuration and parameter settings for such SUTs has not been addressed.

Diversity: The empirical results of Experiment XII for the robot simulator, illustrated in figure 32, highlight again the tendency for the algorithm to generate input profiles with poor diversity. The observed pattern is consistent with the hypothesis made in section 2.8: that diversity is poorest as the highest values of the minimum coverage probability.

Chapter 5

Strategy Derivation Using Parallel Computation

5.1 Introduction

5.1.1 Motivation

Having widened the applicability of the algorithm in chapter 4 through the use of a grammar-based representation, we return to the issue of algorithm performance.

In chapter 3 the configuration and parameter settings of the search algorithm were optimised so that it executes the SUT—typically the most time-consuming part of the algorithm—as few times as possible. Nevertheless, the search algorithm still requires the SUT to be executed very many times: at the optimal parameter settings derived in Experiment VIII, even the 'simplest' SUT for the algorithm, `nsichneu`, is executed approximately 4.4×10^5 times on average. This large number of executions is practical for the example SUTs only because a single execution of the SUT takes a few *microseconds*. However, using the algorithm for SUTs with execution times of *milliseconds* or longer may not be feasible.

In this chapter, we demonstrate that the performance of the algorithm—measured now in terms of its run time—can be further improved by exploiting parallel computation environments, with the objective of scaling the technique to SUTs that take longer to execute.

The parallelism is exploited in two ways:

- by executing multiple instances of the *instrumented SUT* in parallel, with the objective of reducing the time taken in evaluating candidate strategies; and,
- by executing multiple *search processes* (i.e. multiple instances of the search algorithm) in parallel with the objective of reducing the time taken to derive one input profile.

The parallel environment used for this research is a commodity Graphics Processing Unit (GPU) card. Using toolkits supported by the GPU manufacturers, many modern GPUs can be used as a low-cost environment for executing general purpose code in high-performance parallel configurations - a technique known as *general purpose computation on GPUs* (GPGPU).

5.1.2 Contributions

The contributions in this chapter are:

- A comparison of search algorithm's performance in a single-threaded CPU environment and in a highly-parallel GPU environment.
- An assessment of the performance improvement enabled by taking the solution from the first search process to finish in a set of search processes running in parallel on GPU.
- Evidence that the algorithm performance can be further improved by regularly migrating solutions between the multiple search processes running in parallel.

5.1.3 Chapter Outline

The GPGPU architecture and application development process are explained in section 5.2. The implementation of the search algorithm using GPGPU is discussed in section 5.3.

In Experiment XIII (section 5.4), the search algorithm's performance is compared in the CPU and GPU environments. In the latter, the GPU architecture is exploited at two-levels: by running multiple search processes in parallel, and for each search process, executing multiple instances of the instrumented SUT in parallel.

Experiment XIV (section 5.5) measures the performance improvement when the input profile is taken from the first of multiple parallel search processes to finish.

Experiment XV (section 5.6) investigates the change in algorithm performance when candidate input profiles are regularly migrated between algorithm instances running in parallel.

In Experiment XVI (section 5.7) an alternative implementation is investigated that only exploits the GPU at one level: by running multiple instances of instrumented SUT on the GPU in combination with a single search process running on the CPU.

5.2 *General Purpose Computation on GPUs*

In this section we summarise the relevant architectural features of GPU cards, how they may be used for general purpose computation, and the process by which a GPGPU application is developed.

5.2.1 *Background*

Graphics Processing Units (GPUs) render high resolution, high frame-rate 3D graphics for applications such as gaming, CAD, and film animation. Algorithms for rendering graphics are naturally parallelisable—for example, applying the same transformation to each pixel in a scene—and so manufacturers of GPUs have developed sophisticated parallel computing architectures capable of many trillions of operations per second (NVIDIA Corporation, 2012d).

The performance of GPUs continues to improve, driven by the demand for increasingly realistic graphics. Figure 33, adapted from (NVIDIA Corporation, 2012d), illustrates the recent trend in performance for NVIDIA's GeForce series of GPU cards designed for the high-end gaming market. The graph shows the theoretical maximum rate of single-precision floating operations that the card can perform. The performance of the latest cards is 3 times better than those of only 4 years earlier.

For the same price, the raw performance of a GPU is typically better than that of a CPU. Performance is typically measured in units of floating point operations (FLOP) per second: $1\,\mathrm{GFLOP\,s^{-1}}$ is therefore 10^9 floating point operations per second. The maximum performance of the GTX 680 GPU is $3\,090\,\mathrm{GFLOP\,s^{-1}}$ (NVIDIA Corporation, 2012a) and the card cost approximately $500 on release. For comparison, the maximum theoretical sustained performance of the six-core Intel Core i7-3930 processor is $921.6\,\mathrm{GFLOP\,s^{-1}}$ (Intel Corporation, 2011) with a recommended price of $583 (Intel Corporation, 2012).

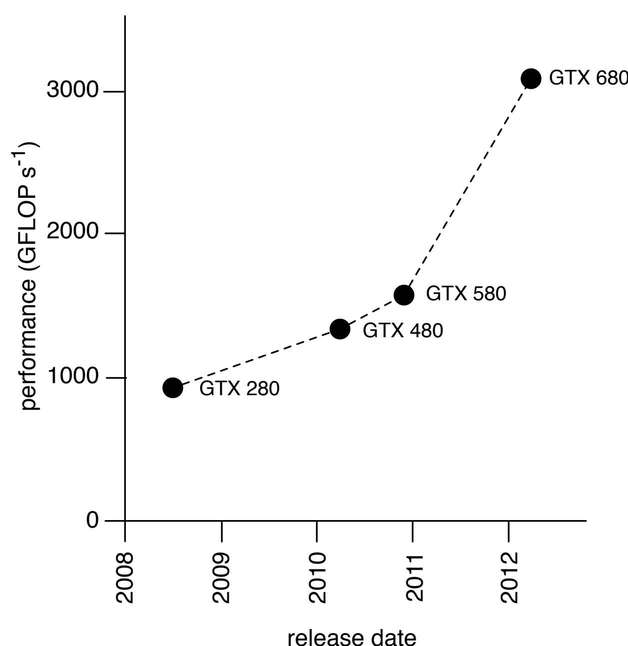


Figure 33 – Single-precision performance of the highest-performance GPU cards of recent NVIDIA GeForce iterations.

Although developed for graphics rendering algorithms, it is possible to utilise the most modern GPUs for other forms of parallel computation, a technique known as *general purpose computation on GPUs* (GPGPU). GPGPU makes available a low-cost high-performance parallel computing environment, potentially utilising GPUs already present in many desktop computers.

5.2.2 CUDA

The use of GPGPU is facilitated by frameworks—languages, development toolkits, and software drivers—provided and supported by GPU manufacturers.

OpenCL (Open Computing Language) is an open-standard framework for GPGPU that is supported on GPUs manufactured by AMD, Intel, NVIDIA, and others, as well as on a limited number of multicore CPUs (Khronos Group, 2012).

CUDA (Compute Unified Device Architecture) is the name of both NVIDIA's GPU architecture and NVIDIA's proprietary GPGPU framework. It is supported by most NVIDIA GPUs released since 2009 (NVIDIA Corporation, 2012b).

The CUDA framework was used for the research described in this chapter, and the descriptions below refer specifically to CUDA.

5.2.3 Physical Architecture

The physical architecture of a CUDA GPU card is shown in figure 34.

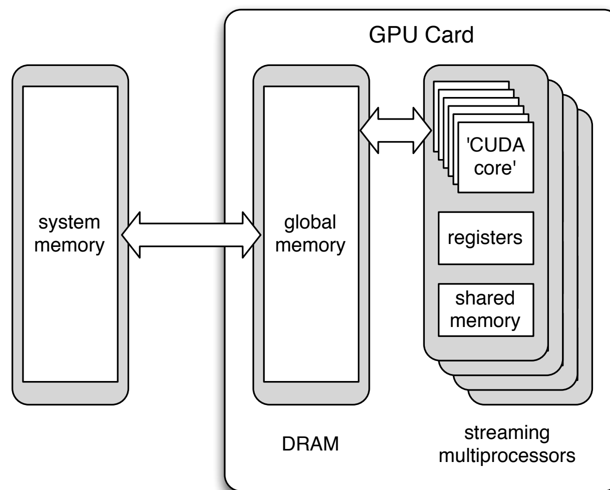


Figure 34 – The physical architecture of a CUDA GPU card.

The processing on the GPU card is performed by a set of streaming multiprocessors. Each multiprocessor itself contains multiple 'CUDA cores'. For example, the research in this chapter uses a NVIDIA Tesla C1060 GPU card which has 30 multiprocessors, each containing 8 cores: a total of 240 cores. A single CUDA core is typically much less powerful than a single CPU core.

Each streaming multiprocessor has a set of registers and a small amount of on-chip ('shared') memory accessible by only the cores on that multiprocessor. The card also has a much larger amount of off-chip ('global') memory accessible by all the cores. On the C1060 card, the global memory is 4 GB.

Data is sent to and retrieved from the card via the global memory: GPGPU applications typically copy data between the main system memory (i.e. the memory accessed by the CPU) and the GPU card global memory.

5.2.4 *Programming Model*

Conceptual Architecture The conceptual CUDA architecture consists of a large number of lightweight threads running in parallel, each thread executing the *same* code—called the *kernel*—but operating on different data.

The threads are grouped into blocks, each block having the same number of threads. On the Tesla C1060 GPU, a block may have up to 512 threads. Each thread and block is assigned a unique id; normally the kernel code reads this id so that the thread can determine the part of the data on which it should operate.

Each thread has a small amount of *local memory* that only it can access. *Shared memory* at block level is accessible by all threads in the block, but not by threads in other blocks. *Global memory* is accessible by threads in all blocks.

This conceptual architecture is illustrated by figure 35.

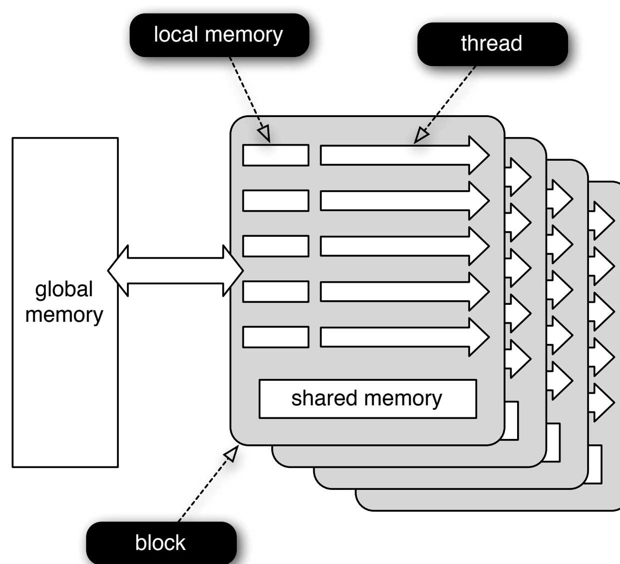


Figure 35 – The conceptual architecture of a CUDA GPU card.

Conceptual to Physical Architecture Mapping Each block is executed by a single streaming processor, and one processor may be assigned more than one block to execute. Although conceptually all threads execute simultaneously, it is small groups of threads—called a *warp*—that are physically executed together by a processor, and the order in which warps are run is dynamically determined by the GPU card at runtime. On a Tesla C1060 GPU, a warp consists of 32 consecutive threads.

Thread-level local memory is a designated area of off-chip global memory, or processor registers, if available. Block level shared memory is on-chip processor shared memory. Conceptual global memory is the off-chip global memory that remains once local memory has been assigned.

5.2.5 Performance Features

The CUDA architecture has a number of features designed to maximise performance when a large number of threads are executing in parallel.

Single-Instruction, Multiple-Thread If a warp of threads is executing in step so that all the threads process the same instructions at the same time, then the multiprocessor is able to utilise a form of instruction-level parallelism called *single-instruction, multiple-thread* (SIMT) to improve performance (NVIDIA Corporation, 2012d).

Each thread is operating on its own part of the data, and so data-dependent conditional branches can cause threads to diverge in their execution paths. The processor handles such divergence automatically, but it does reduce performance. For this reason, kernel code should be designed to avoid divergence within a warp, or synchronisation barriers which permit divergent threads to re-converge should be specified in the code using a CUDA language extension.

Hardware Multithreading If a thread warp is waiting—for example, for data to be returned from global memory—the multiprocessor is able to switch execution to another warp with very little overhead. This is achieved by maintaining the full execution context of every warp on-chip until it has finished executing (NVIDIA Corporation, 2012d). The ability to context switch extremely quickly can 'hide' the relatively high latency of global memory access, but only if there are sufficient ready-to-execute thread warps available for the processor to switch to.

Coalesced Memory Access If threads in a warp access global memory in one of a number of defined patterns, the access is coalesced into a single data transfer resulting in significant performance improvements (NVIDIA Corporation, 2012e). The simplest pattern is for each thread to access a consecutive set of 4 bytes; more recent GPUs—those with a higher compute capability (described below)—are able to coalesce more complex memory access patterns.

5.2.6 Application Development

A CUDA application incorporates both code that executes on the GPU ('device' in CUDA terminology), and code that executes on the CPU ('host'). (We will use the device / host terminology in order to avoid confusion with the purely CPU implementation of the algorithm used in earlier chapters). At a minimum, the host code sets up the data on which the device will operate, copies it to device global memory, launches a kernel on the device to process the data, and retrieves the processed data from device global memory.

For host code, CUDA supplements standard C++ with a small set of language extensions to launch kernel code that runs on the device, and a runtime library for operations such as copying data between host system memory and device global memory.

For device code, a subset of C++ is supported, again with a small set of language extensions to support CUDA features such as the memory hierarchy. The C++ language features omitted depend on the architectural version the GPU hardware (termed the *compute capability*) and the version of the CUDA software framework. For the Tesla C1060 GPU (compute capability 1.3) using version 5.0 of the CUDA framework, the most relevant features that are *not* supported are:

- C++ standard template library;
- dynamic memory allocation;
- input/output;
- C++ exceptions;
- function recursion;
- function pointers;
- runtime type information;
- the `rand()` function: CUDA supplies a more extensive pseudo-random number generation library in its place.

A limitation of earlier GPGPU architectures was the support for, and the accuracy of, floating point operations. However, NVIDIA CUDA GPUs of compute capability 1.3 and greater (and therefore including the Tesla C1060 card used in this research) support double precision floating point data and arithmetic in device code, and the CUDA software framework provides a full implementation of the standard C99 math library. The encodings, rounding, accuracy, and supported operations are compliant with the IEEE 754 standard on floating point computation (Whitehead and Fit-Florea, 2011).

An application may be created from the host and device code in a number of different ways. For the research described in this chapter, the code is compiled and linked by the CUDA tool chain to create a single executable that runs on the host. This host executable stores executable device code in data structures; this code is passed to the device using a call to the CUDA driver.

5.3 Implementation

In this section, we describe the relevant aspects of the implementation, using the CUDA architecture, of our algorithm that derives an input profile.

5.3.1 Architecture

As discussed in section 5.2.6 above, a CUDA application consists of both host (CPU) and device (GPU) code. The host-side code for the algorithm implementation reads and parses the parameter file and initialises the input profile before launching a device-side kernel to perform the hill climbing search. The kernel returns control to the host periodically to enable logging, visualisation, and—in Experiment XV later in this chapter—the migration of profiles between parallel searches.

The host-side code performs initialisation, but the device-side kernel performs all other aspects of the search, including:

- creating neighbour input profiles by applying mutation operators;
- evaluating the fitness of candidate profiles by sampling from the profile, executing the instrumented SUT, and calculating the fitness from the instrumentation data;
- processing feedback data returned during fitness evaluation to enable directed mutation.

Although GPGPU techniques have been previously used for the evaluation of fitness metrics in search-based software engineering (e.g. Yoo et al. (2011) calculate a fitness metric using linear algebra on a GPU), we believe this to be the first time that a GPU has been used to execute an instrumented SUT for this purpose.

Two levels of parallelism are implemented in the device-side kernel:

1. Multiple hill climbs occur simultaneously; each hill climb is implemented in one CUDA block.
2. Within each block, fitness evaluation uses multiple parallel threads: each thread executes the instrumented SUT with a different input sampled from the candidate profile. Multiple threads are also used to collate and calculate the fitness from the instrumentation data, and process the feedback data for directed mutation. (The sampling of the neighbourhood could also have been performed on multiple threads—creating one mutated neighbour per thread—but for some SUTs, the memory required to store the entire sample of mutated neighbours cannot be accommodated in the limited device-side memory.)

Figure 36 illustrates the parallel processing in the device-side kernel.

5.3.2 Code

In order to ensure a reliable comparison, the GPU implementation shares as much code as possible with the CPU implementation described in the previous chapter (see section 4.4).

Shared host-side code—running on the CPU in both implementations—includes:

- parameter file parsing;

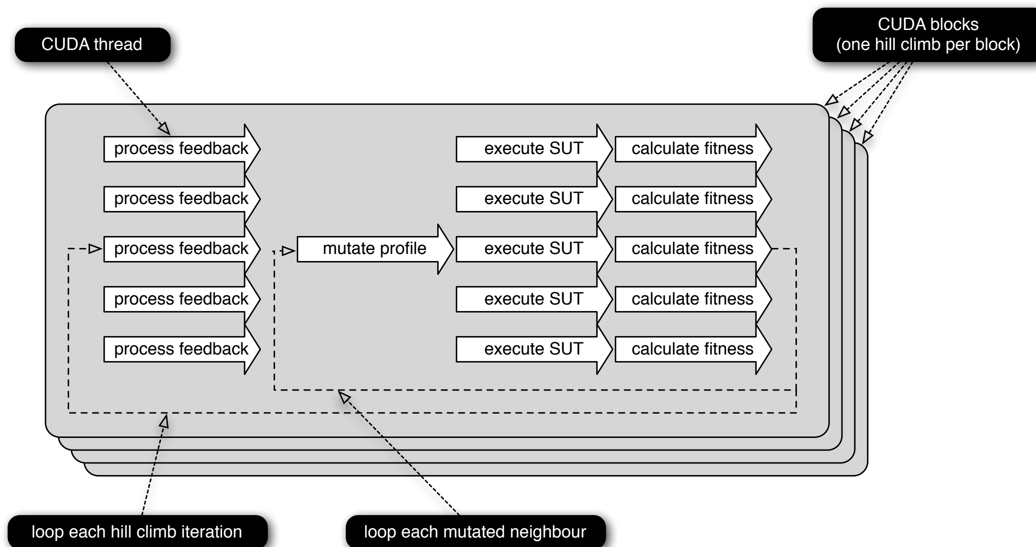


Figure 36 – The implementation of the search algorithm in the device-side (GPU) kernel. Each block executes one instance of the search algorithm, and within each block the SUT is executed in parallel by multiple threads.

- profile initialisation;
- logging and visualisation.

Shared device-side code—running on the CPU or GPU depending on the implementation, and referred to as the 'common' code category in section 4.4.1—includes:

- search mutation operators;
- sampling of inputs from candidate profiles;
- instrumented SUTs;
- pseudo-random number generation (PRNG).

The minor syntax changes required when shared device-side code runs on the GPU (e.g. additional CUDA-specific function qualifiers) are implemented using pre-processor directives.

The most significant pieces of functionality that are not implemented using shared code are the calculation of fitness from instrumented data, and the processing of feedback data for directed mutation. The implementation of this functionality is single-threaded on the CPU but uses multiple interacting threads within each block on the GPU.

To avoid the concurrency and performance issues that would result from a shared PRNG, each thread has its own PRNG instance in the GPU implementation. Although the same issue of concurrency does not arise in the single-threaded CPU implementation, the same number of PRNG instances are created and accessed in the same pattern in the CPU implementation in order to maintain consistency between the CPU and GPU implementations.

5.4 Experiment XIII – GPU Performance

5.4.1 Objectives

In this experiment, the 'raw' performance of the parallel GPU algorithm implementation is compared to that of the single-threaded CPU implementation. We use the term 'raw' here to refer to the rate at which the algorithm performs search iterations rather than the total time taken by the algorithm to derive an input profile. These measurements will be used as a baseline for subsequent experiments.

The research question is:

RQ-1 How do the rates at which the CPU and GPU implementations perform search iterations compare?

5.4.2 Preparation

CPU Environment The computing environment for the CPU implementation (and the host-side code of the GPU implementation) was a single core of a quad-core Intel Xeon E5405 processor running at 2 GHz and sharing 8 GB of system memory. Code was built using `gcc` with `-O2` optimisation options.

GPU Environment The computing environment for the device-side code of the GPU implementation was a NVIDIA Tesla C1060 card which currently costs approximately $1 000. (The server had a second GPU to handle display output, and so the C1060 card was dedicated to running the algorithm.)

The C1060 card has 30 streaming multiprocessors with 8 CUDA cores each, and 4 GB of global memory. The single-precision floating point performance of the card is 933 GFLOP s^{-1} (NVIDIA Corporation, 2012c). The performance of the C1060 is therefore less than a third of the recent GTX 680 card despite its higher cost, but the C1060 has twice the global memory.

The GPU code was built using the CUDA version 5.0 framework.

The number of blocks—and therefore the number of independent parallel searches—was set to 30 so that each streaming processor is assigned one block to process. In preliminary experimentation, this was found to be enough blocks to ensure a sufficient supply of ready-to-execute thread warps for effective hardware multithreading (section 5.2.5): using 60 blocks demonstrated approximately the same per-block performance.

The number of threads per blocks was set to 151. This setting is based on the optimal value of 302 for the algorithm parameter K, the number of inputs sampled when evaluating fitness of candidate input profiles, found in Experiment VIII (section 3.8). Each thread in a block therefore samples two inputs and executes the instrumented SUT twice, once for each sampled input.

SUTs The performance of the CPU and GPU implementations is compared using five of the SUTs used in the preceding chapters: `bestMove`, `nsichneu`, `cArcsin`, `tcas`, and `epuck`. The source code of the SUT `fct3` was not available to us for recompilation using the CUDA tool chain. The SUT `circBuff` could not be used in the GPU implementation since the code contains features not supported by CUDA in device-side code: the C++ standard template library and exceptions.

Two modification were made to the code of the e-puck simulator, `epuck`:

- Dynamic memory allocation was replaced with static allocation in both the CPU and GPU implementations since CUDA does not support the former. (This change is discussed in section 4.6.)

- The simulator code is highly iterative: 500 iterations of the main time step loop are made to simulate 10 s of real-world time, and at each time step all possible interactions between e-pucks and objects in their environment are assessed. Many of these loops contain data-dependent conditional statements, and therefore when executing the instrumented SUT in parallel threads on the GPU, thread divergence is very likely to occur within a warp. Such divergence prevents the full utilisation of SIMT instruction-level parallelism and so reduces performance (see section 5.2.5). For this reason, synchronisation barriers were placed after such loops to force the threads to re-convergence: threads reaching the barrier wait until all other threads in the block have reached the same barrier. The synchronisation barrier code (a single function call) was added manually in this case, but it would be straightforward to automate this process for other SUTs. The barrier code does not change the functionality of the SUT itself.

Server In the experiments of this chapter, the metric used to compare performance is the run time of the algorithm since the metric used previously—the number of SUT executions—cannot be used to compare performance in different computing environments.

However the run time has a number disadvantages as a metric: it is dependent on the quality of the implementation, the hardware used for the computing environment, and the resources used by any other processes running on the same computing environment. The implementation quality and hardware issues are addressed in the threats to validity of this experiment, but we are able to minimise the issue of other processes using the same computing environment by using a standalone server on which no other resource-intensive processes were running during the experiments. The GPU card, in particular, was used only for running the algorithm, and preliminary experimentation demonstrated that timings for multiple trials of the same algorithm configuration using the same PRNG seed were very consistent.

The operating system of the standalone server was 64-bit Ubuntu Linux, version 11.10.

5.4.3 *Method*

Since the objective is to measure the rate at which the implementations perform search iterations, the algorithm is terminated after a fixed number of iterations instead of when it derives a suitable input profile. The algorithm parameter τ_{iter}, the maximum number of search iterations, was set to a SUT-specific value chosen by preliminary experimentation so that one trial took approximately 30 s to run; these values of τ_{iter} are listed in table 40 below. The target minimum coverage probability, $\tau_{p_{\text{min}}}$, was set to 1 for all SUTs so that trials did not terminate because a suitable input profile was found. Otherwise the algorithm parameter settings are the same as those used in the experiments of chapter 4 (tables 36 and 38).

For each of the five SUTs, 8 trials of the CPU algorithm implementation were run, and the run time of the search (including initialisation of the first profile, but excluding setup such as parsing the parameter file and setting up data structures) was recorded.

The process was repeated for the GPU implementation, using the *same* maximum number of search iterations as for the CPU implementation. The run time recorded for each GPU trial was the time to complete *all* 30 parallel search processes. The limit on the number of iterations

(τ_{iter}) applies to each search process individually, so the GPU implementation performed, in total, 30 times more search iterations than the CPU implementation.

5.4.4 Results

Table 40 lists the median time for each algorithm implementation and each SUT across the 8 trials. The 'Per Search' times for the GPU implementation are calculated by dividing the observed run time by 30 since each GPU trial performs 30 independent searches in parallel. The 'GPU Raw Speed-Up' is the ratio of CPU to GPU run times; the 'GPU Raw Per Search Speed-Up' is the ratio of the CPU run time (which is implicitly per search) to the GPU per search time.

SUT	`bestMove`	`nsichneu`	`cArcsin`	`tcas`	`epuck`
Number of Iterations	1 200	1 200	16 000	8 000	8
CPU: Median Run Time (s)	27.21	27.35	25.69	35.92	31.77
GPU: Median Run Time (s)	120.20	248.63	172.31	531.95	21.92
GPU: Median Time Per Search (s)	4.006	8.287	5.744	17.73	0.7307
GPU Raw Speed-Up	0.226	0.110	0.149	0.0675	1.45
GPU Raw Per Search Speed-Up	6.79	3.30	4.47	2.03	43.5

Table 40 – Results of Experiment XIII.

5.4.5 Discussion and Conclusions

The results show that for four of the five SUTs, the GPU implementation takes longer to perform 30 searches in parallel than the CPU takes for a single search: the GPU speed-up for these SUTs is less than 1.

However, for all five SUTs, the *per search* speed-up, and therefore the rate at which search iterations are performed, is at least two times better than the CPU implementation (RQ-1).

The most striking speed-up is for `epuck`: the GPU implementation performs iterations over 40 times faster than the CPU implementation. Moreover, the GPU implementation performs 30 searches in parallel more quickly than the CPU performs a single search.

We explain the large GPU speed-up for `epuck` in terms of the relatively long execution time of the SUT (demonstrated by the CPU implementation performing search iterations for `epuck` at a rate two orders of magnitude slower than for the other SUTs). Within each search iteration, creating neighbours by mutation and executing the instrumented SUT multiple times during fitness evaluation are particularly time-consuming. In the GPU implementation, mutation is single-threaded within each block (as a result of the memory constraints discussed in section 5.3.1) and therefore runs on a single CUDA core that is, in isolation, less powerful than a CPU core; but fitness evaluation is parallelised across multiple CUDA cores. As a result of the long execution time of `epuck`, the slow single-threaded implementation of mutation is insignificant compared to the performance gained by parallelising the fitness evaluation of the SUT. (A more detailed timing analysis of individual parts of the search would be needed to confirm this explanation, but such analysis is difficult to do *inside* a kernel.)

The superior raw performance of the GPU implementation occurs in the context of 30 searches running simultaneously. Since only a single search is needed to derive an input profile, the raw performance advantage of the parallel GPU implementation does not necessarily

result in an input profile being derived more quickly. This issue is considered in the next two experiments.

5.4.6 *Threats to Validity*

Generality As for previous experiments, this experiment has considered a relatively small number of SUTs and therefore we are unable to draw very general conclusions. However, the SUTs are diverse in characteristics that might be expected to be significant for GPU performance: both integer and floating point operations (`cArcsin` uses double precision floating point arithmetic extensively), a range of code sizes (`nsichneu` and `epuck` each have over 1000 lines of code), different degrees of branching (`nsichneu` in particular has many data-dependent branches), and a wide range of execution times.

Coding The relative raw performance of CPU and GPU, measured here by execution time rather than a statistic of the algorithm itself, could result from differences in the quality of each implementation. If, for example, more effort was spent in coding a fast-executing mutation operator in the GPU implementation than the CPU, this could be a cause of observed performance difference. We have minimised this threat by sharing as much as code possible between the two implementations (section 5.3.2).

Hardware The relative performances are very dependent on the choice of hardware, particularly the choices of CPU chip and GPU card. This issue is unavoidable, but we note that both the Xeon chip and Tesla C1060 card are representative of the high-end consumers CPU and GPUs, respectively, at the time that the server used for these experiments was built. Moreover, at the time of writing, NVIDIA GPU cards that exceed the Tesla C1060 in performance are available, and many (such as the GTX 680) are much cheaper than the C1060.

5.5 Experiment XIV – Multiple Parallel Searches

5.5.1 Objectives

If the search method *always* took the same number of iterations to derive a suitable input profile, regardless of the PRNG seed, the previous experiment demonstrated there would be little performance gain in running multiple parallel search processes on the GPU: for many SUTs, a single-threaded search on a CPU would finish before any of the multiple parallel searches on the GPU.

However, the experiments of previous chapters have demonstrated a very large variance in the number of iterations as a result of the stochastic nature of the algorithm. By running multiple search processes in parallel on GPU, and taking the profile from the first search process to finish, we aim to leverage this variance since some of the search processes will, by chance, take much fewer iterations than average to derive a profile.

The research question is:

RQ-1 How does the time taken by the quickest of multiple parallel search processes on the GPU compare to the time taken by a single search process on the CPU?

In contrast to the previous experiment, this research question is expressed in terms of the time taken to derive an input profile rather than the rate at which search iterations are performed.

5.5.2 Preparation

SUTs Three example SUTs are used for this experiment: `epuck`, which demonstrated by far the largest speed up on the GPU in the previous experiment; `bestMove`, which demonstrated the largest speed-up of the remaining SUTs; and, `tcas`, which demonstrated the smallest speed-up.

Parameter Settings The parameter settings for target minimum coverage probability and maximum number of iterations are shown in table 41. The maximum number of iterations is much greater than the average number of iterations required to derive a profile for these SUTs, so most algorithm trials will successfully derive an input profile. The remaining parameters were set to the values used in chapter 4 and shown in tables 36 and table 38.

Parameter	Description	SUT	Setting
τ_{iter}	maximum no. iterations	`bestMove`	2×10^4
		`tcas`	4×10^4
		`epuck`	2×10^2
$\tau_{p_{\min}}$	target min. coverage probability	`bestMove`	0.112
		`tcas`	0.078
		`epuck`	0.5

Table 41 – The algorithm parameter settings used in Experiment XIV.

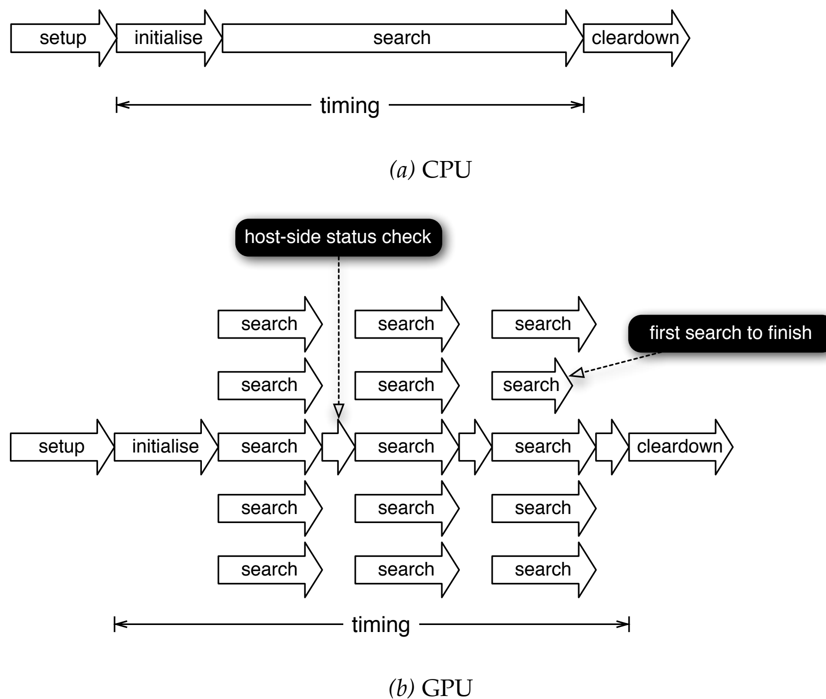
The GPU configuration is the same as that of Experiment XIII: 30 blocks (one search per block), and 151 threads per block. Once the first of the multiple search processes on the GPU has found a suitable input profile, the remaining search processes can be terminated. This logic is implemented in host-side code and therefore the device-side kernel temporarily returns control to the host at regular intervals for this purpose: every 128 search iterations for `bestMove` and `epuck`, and every 2 search iterations for `epuck`.

The configurations of the CPU and GPU computing environments, and the host server, were the same as for Experiment XIII.

5.5.3 *Method*

32 trials of both the CPU and GPU implementations were run, each trial using a different set of PRNG seeds.

As for Experiment XIII, the run time taken by the search process was recorded for each trial. The timing included the search initialisation but not program setup prior to the search (such as parsing the parameter file), nor cleardown once the search has finished. For the GPU implementation, the search process logically ends once the first of the multiple searches finds an input profile with the target minimum coverage probability, but in practice the time recorded is a slight overestimate since timing ends only once control is next returned to the host-side code in order terminate the remaining searches. These start and end points used for timing are illustrated in figure 37.



(a) CPU

(b) GPU

Figure 37 – The timing start and end points for the CPU and GPU implementations.

5.5.4 *Results*

The results are summarised in table 42.

The run times listed in the first two rows of the table are the medians of the 32 trials for each SUT.

The 'GPU First-To-Finish Speed-Up' is the factor by which the GPU median run time is faster than the CPU median run time. The 'GPU Raw Speed-Up' is the raw performance speed-up taken from the results of Experiment XIII, i.e. the ratio of the run time of a single CPU search process to 30 parallel GPU search processes over a fixed number of iterations. 'GPU First-To-Finish v. Raw' is the ratio of these two values: ratios greater than 1 indicates

that the first-to-finish speed up is greater. We calculate this measure in order to isolate any speed-up arising from taking the first of multiple searches to finish from the difference in raw search performance of the CPU and GPU.

The rank-sum p-value estimates the significance of any difference in the 'first-to-finish' and 'raw' speed-ups. It is calculated by pairing the CPU and GPU trials of each SUT in the random order that they were performed: the ith CPU trial is paired with the ith GPU trial. For each pair, the CPU run time is divided by the GPU run time to give 32 values for the first-to-finish speed-up. The same procedure is applied to the results of Experiment XIII to give 8 values for the raw speed-up. For each SUT, the 32 first-to-finish speed-up values are compared to 8 raw speed-up values using the Mann-Whitney-Wilcoxon (rank-sum) test.

The rows 'IQR Time / Median Time' are the ratio of the interquartile range of the run times to the median run time, and so are a standardised measure of the variability across the 32 trials. The purpose of calculating this measure is discussed below.

SUT	`bestMove`	`tcas`	`epuck`
CPU: Median Time (s)	64.12	30.19	364.33
GPU: Median Time (s)	104.72	137.36	158.78
GPU First-To-Finish Speed-Up	0.612	0.220	2.29
GPU Raw Speed-Up	0.226	0.0675	1.45
GPU First-To-Finish v. Raw	2.70	3.25	1.58
rank-sum p-value	4.66×10^{-4}	5.33×10^{-5}	1.24×10^{-3}
CPU: IQR Time / Median Time	1.50	0.859	0.350
GPU: IQR Time / Median Time	0.435	0.458	0.332

Table 42 – Summary of results for Experiment XIV.

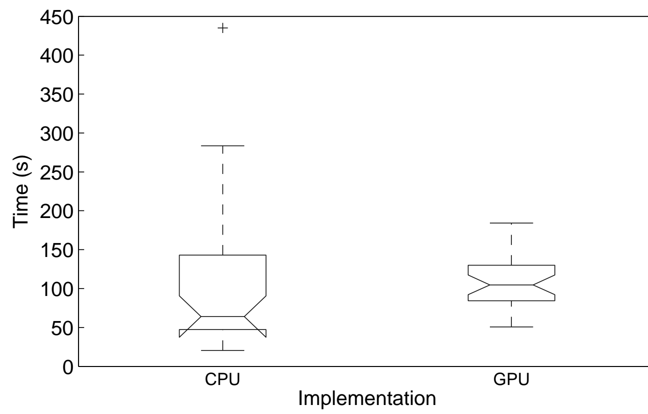
The distributions of timings across the 32 trials for each combination of implementation and SUT are illustrated using boxplots in figure 38. The 'notches' are a visual indication of statistical significance: if they overlap on a pair of boxplots, then any difference in the medians is not statistically significant.

5.5.5 Discussion and Conclusions

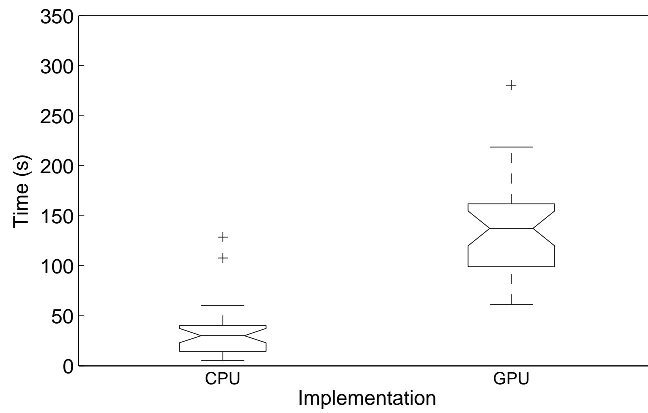
For `bestMove` and `tcas`, the single CPU search process derives an input profile more quickly than multiple parallel GPU search processes. For `epuck`, the situation is reversed: the GPU implementation is the faster by a factor of more than 2 (RQ-1). We speculate again that it is the relatively long execution time of the SUT that is the reason for the much better speed-up observed for `epuck`.

However the first-to-finish speed-ups for all three SUTs are much greater than the raw speed-ups: the times are reduced by a factor of at least 1.58, and these differences are statistically significance at the 5% significance level. Therefore the speed-up in the time taken to derive a profile is not only a result of the raw performance difference between the CPU and GPU implementations, but also of running multiple searches in the GPU implementation and taking the results from the first search to finish.

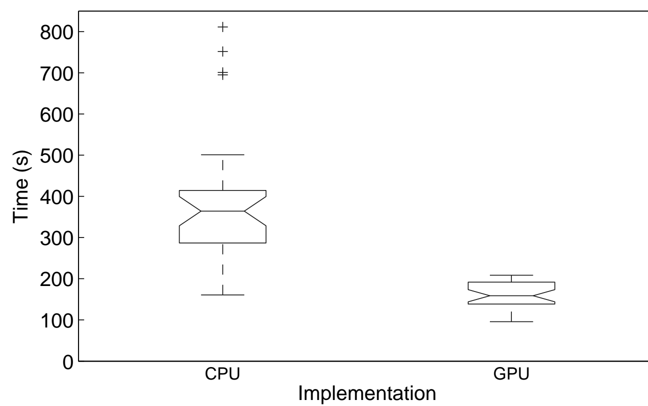
We note also that for all three SUTs, the interquartile range of the run times (calculated as a ratio of the median run time) is lower for the GPU implementation. In other words, the variance in the run times is reduced on the GPU. The effect is particularly noticeable for `bestMove` in the boxplots of figure 38a where the median CPU and GPU run times are fairly similar. We speculate that the reduction in variance on the GPU (which may be useful when



(a) `bestMove`



(b) `tcas`



(c) `epuck`

Figure 38 – Boxplots comparing the distributions of run times for the CPU and GPU implementations for each SUT.

the algorithm is used in practice) is a result of taking an aggregate response—from the first search to finish—of multiple independent search processes.

5.6 *Experiment XV – Spatial Migration*

5.6.1 *Objectives*

In the preceding experiment, the multiple parallel search processes were independent of one another: an 'embarrassingly parallel' configuration. In this experiment we research whether further improvements in algorithm performance may be achieved by regularly migrating the best candidate input profiles between the search processes.

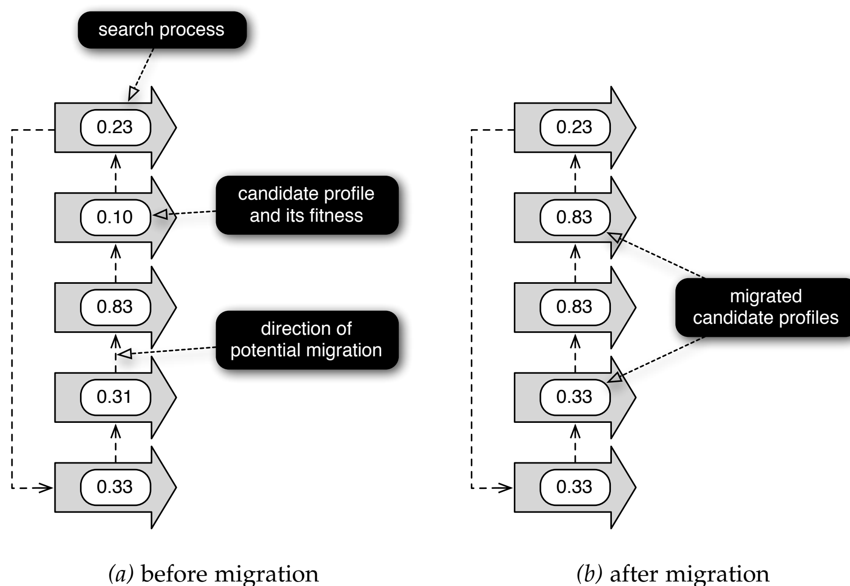
The research question is:

RQ-1 Does regular migration of the best candidate profiles between parallel search processes improve the time taken to derive an input profile?

5.6.2 *Preparation*

The chosen migration method is illustrated in figure 39. The parallel search processes are assigned a fixed ordering at the beginning of the algorithm. Every n_{mig} iterations (where n_{mig} is a new algorithm parameter), each search process is compared to its next highest neighbouring process in turn, starting from the search first in the order. If the candidate input profile of the neighbour is fitter that of the current process, the candidate profile of the neighbour is copied to become the candidate for the current search process.

We choose this spatial migration method since it requires only one additional parameter, and we believe it will avoid premature convergence across the parallel searches: for any one candidate profile to be migrated to all searches would take at least $30n_{\mathrm{mig}}$ iterations, assuming 30 parallel searches.



(a) before migration *(b)* after migration

Figure 39 – An example of the spatial migration method.

The migration method is implemented in host-side code and so the device-side kernel temporarily returns control to the host every n_{mig} iterations

5.6.3 *Method*

The experiment uses the same three SUTs as the previous experiment: `bestMove`, `tcas`, and `epuck`.

For `bestMove` and `tcas`, 32 trials of the GPU implementation were run at 5 values of n_{mig}: 16, 32, 64, 128, and 256. For `epuck`, 32 trials of the GPU implementation are run at 4 values of n_{mig}: 1, 2, 4, and 8. The run time of the algorithm—using the same timing points as Experiment XIV—is recorded for each trial.

5.6.4 *Results*

The distributions of run times at different migration frequencies are summarised by the boxplots of figure 40.

The boxplots provide strong visual evidence that migration has an effect on the time taken by the GPU implementation: that the notches of the best and worst values of n_{mig} for each SUT do not overlap indicates a significant difference in the median timings at the 5% significance level.

In order to compare the effect of migration with the configuration of Experiment XIV that did not use migration, we select the value of n_{mig} that gives the lowest average time for each SUT.

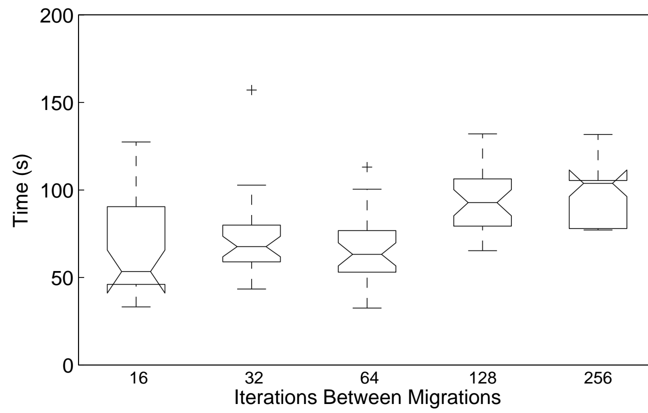
Based on medians shown in the boxplots, the best value for `bestMove` would appear to be $n_{\text{mig}} = 16$. However, two of the trials at this value of n_{mig} did not manage to find an input profile before the (very high) limit on the number of iterations was reached. (For clarity, these extreme outliers are not shown in the boxplot.) Such long running trials did not occur at higher values of n_{mig} nor in the trials of Experiment XIV; and additional experiments suggest such outliers are not abnormal at low values of n_{mig} for this SUT. As a result of these two outliers, while $n_{\text{mig}} = 16$ demonstrates the lowest *median*, $n_{\text{mig}} = 64$ has a much lower *mean* and a median that has no significant difference from that of $n_{\text{mig}} = 16$. We therefore select $n_{\text{mig}} = 64$ as giving the lowest average time for `bestMove`.

For the other SUTs the situation is simpler. The values $n_{\text{mig}} = 16$ and $n_{\text{mig}} = 1$ are selected for `tcas` and `epuck` respectively as they demonstrate both the lowest median *and* mean run times.

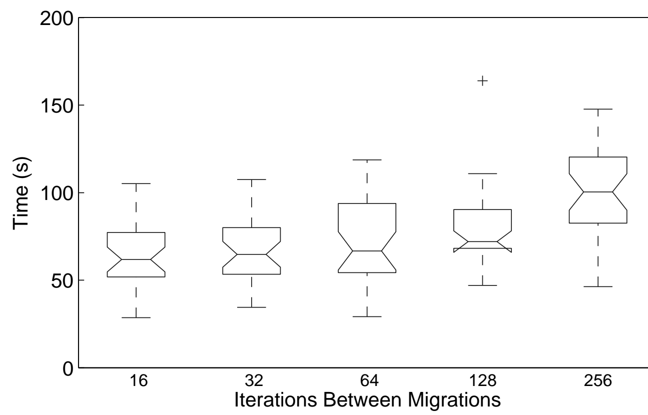
In table 43 the distribution of timings for the best SUT-specific value of n_{mig} is compared to the timings without migration for both CPU and GPU implementations. The latter data are taken from the results of Experiment XIV. The 'Migration Speed-Up' is the ratio of median run times for the GPU implementation with and without migration: values greater than 1 indicate that migration speeds up the algorithm. The rank-sum p-value is calculated from the same data and estimates the statistical significance of any difference in the run times with and without migration.

SUT	`bestMove`	`tcas`	`epuck`
GPU Migration: Median Time (s)	63.19	61.82	92.02
CPU First-To-Finish: Median Time (s)	64.12	30.19	364.33
GPU First-To-Finish: Median Time (s)	104.72	137.36	158.78
Migration Speed-Up v. GPU First-To-Finish	1.66	2.22	1.73
rank-sum p-value v. GPU First-To-Finish	$< 10^{-5}$	$< 10^{-5}$	$< 10^{-5}$

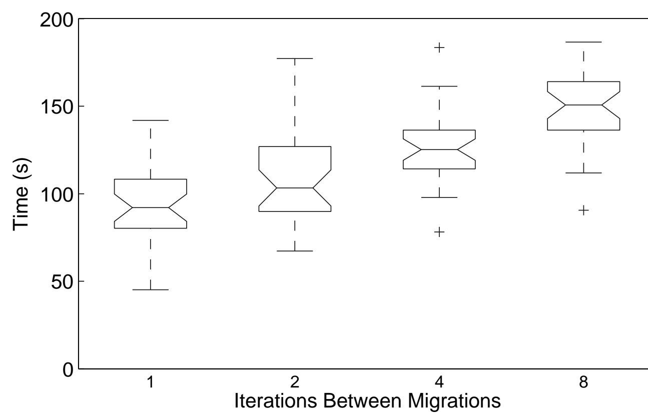
Table 43 – Comparison of algorithm timings with and without migration.



(a) `bestMove`



(b) `tcas`



(c) `epuck`

Figure 40 – Boxplots comparing the distribution of run times for different frequencies of migration.

5.6.5 Discussion and Conclusions

The values for 'Migration Speed-Up' in table 43 are greater than 1 for all three SUTS, and this shows that regular migration can improve the performance of the GPU implementation compared to the same implementation without migration (RQ-1). Migration reduces the median time taken to derive a profile by a factor of at least 1.66 for all three SUTs, and the differences are statistically significant at the 5% significance level.

We note also that GPU implementation using migration derives profiles for `bestMove` as quickly as the CPU implementation, and that the GPU implementation using migration is almost 4 times faster than the CPU implementation for `epuck`. However, these figures will be dependent on the hardware choices for the CPU and GPU.

We speculate that the occasional long running trials observed at the lowest values of n_{mig} for `bestMove` are the result of a process similar to premature convergence: all the parallel search processes operate on similar profiles owing to frequent migrations, but it is particularly difficult to reach a suitable input profile (one with the target minimum coverage probability) from this particular profile.

5.7 *Experiment XVI – One Level of Parallelisation*

5.7.1 *Objectives*

In the earlier experiments of this chapter, *two* levels of parallelism were implemented on the GPU: multiple search processes ran in parallel, and within each search process, parallel executions of the instrumented SUT were performed. In this final experiment, we evaluate the performance improvements possible if only *one* level of parallelism—the execution of the SUT—is implemented on the GPU, and all other parts of the search algorithm occur on the CPU.

The research question is:

RQ-1 What is the change in performance of the search algorithm when (only) the execution of the instrumented SUT is performed on the GPU?

5.7.2 *Preparation*

The revised architecture of the algorithm implementation is illustrated in figure 41.

In contrast to the architecture used in the previous experiments (figure 36), mutation (and the associated processing of feedback to direct the mutation) is performed on the host (CPU). The entire set of neighbours created by mutation is passed to the GPU kernel to be evaluated. The evaluation of each neighbour occurs in one or more CUDA blocks. Each CUDA thread in a block samples an input, executes the instrumented SUT, and accumulates the instrumentation data in order to calculate the fitness.

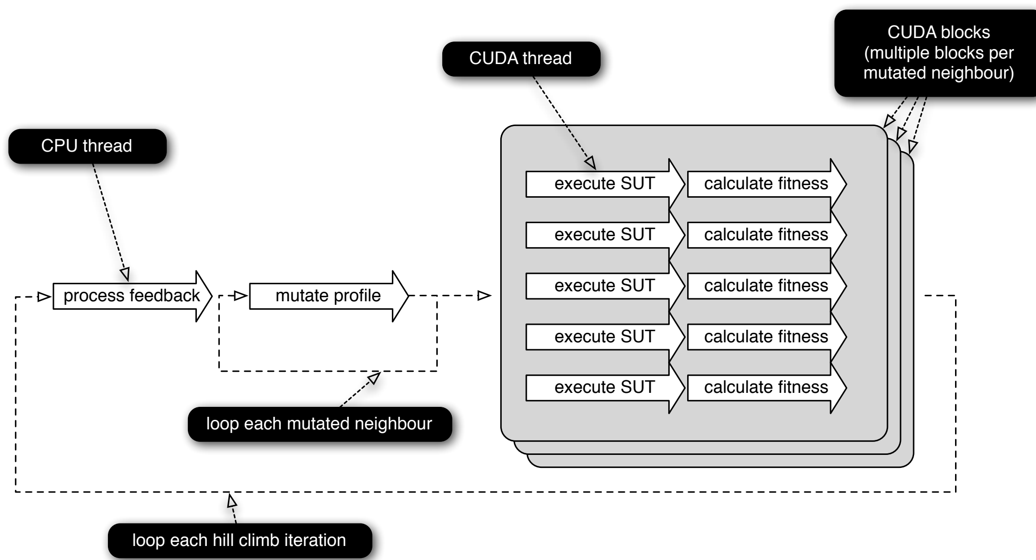


Figure 41 – The revised algorithm architecture that performs only the execution of the SUT in the device-side (GPU) kernel.

5.7.3 *Method*

The experiment considers the five SUTs used earlier in Experiment XIII: `bestMove`, `nsichneu`, `cArcsin`, `tcas`, and `epuck`.

The configurations of the CPU and GPU computing environments were the same as for Experiment XIII. The GPU card has 30 streaming processors, and so the number of CUDA blocks is set to 30 in order to efficiently use this hardware architecture.

The algorithm parameter settings were those derived in chapter 4, listed in tables 36 and 38, and extended in table 41. These parameter settings permit a maximum of 10 candidate profiles to be evaluated at each iteration—9 mutated neighbours plus a re-evaluation of the current profile—and so each candidate is evaluated by 3 of the 30 CUDA blocks. For this reason, the evaluation sample size (K) was rounded up from 302 to 303, the nearest multiple of 3, and thus the number of CUDA threads per block was 101.

32 trials of the algorithm implemented in the revised GPU architecture were performed for each SUT. Since a (small) change had been made to the parameter K, 32 trials using the same parameters were also performed on the CPU.

The response measured was the time taken by the algorithm to derive an input profile.

5.7.4 *Results*

The results are summarised in table 44 and show the median run times for the CPU and GPU implementations. The factor by which the GPU implementation is faster than the CPU is shown in the row 'GPU Speed-Up'. The row 'rank-sum p-value' indicates the statistical significance of any difference in run times, calculated using Mann-Whitney-Wilcoxon (rank-sum) test.

SUT	`bestMove`	`nsichneu`	`cArcsin`	`tcas`	`epuck`
CPU: Median Run Time (s)	72.7	25.6	0.401	23.4	437
GPU: Median Run Time (s)	56.8	18.9	0.160	27.7	10.1
GPU Speed-Up	1.28	1.35	2.51	0.843	43.4
rank-sum p-value	0.570	0.825	1.75×10^{-4}	0.124	$< 10^{-5}$

Table 44 – Results of Experiment XVI.

5.7.5 *Discussion and Conclusions*

For `bestMove`, `nsichneu`, and `tcas`, the run times for the CPU and GPU implementations are similar, and the differences are not statistically significance at the 5% level; for `cArcsin`, there is a statistically significant speed-up on the GPU by a factor of approximately 2.5; for `epuck`, the speed-up is much greater: by a factor of more than 43 (RQ-1).

The observed speed-up for `epuck` is an order of magnitude greater than that achieved in Experiment XV using the previous GPU architecture that parallelises the entire search algorithm on the device. We hypothesise that such a large speed-up is observed for this SUT because its execution time is so long compared to the other parts of the search algorithm: by applying the GPU resources to the parallelisation of the most time-consuming part of the algorithm—the execution of the SUT—a significant speed-up is achieved.

Experiments XIV and XV demonstrated that parallelising the entire search process on a GPU could outperform a single-threaded CPU implementation of the algorithm for some SUTs. However, this experiment suggests that for SUTs with relatively long execution times, such as `epuck`, a better application of GPGPU may be simply to parallelise the execution of the SUT.

5.8 Conclusion

In this chapter we have demonstrated the strategy derivation algorithm executing in the low-cost, highly-parallel computation environment provided by GPU cards.

The parallelism was exploited in two ways:

- to execute multiple copies of the instrumented SUT in parallel for the purpose of evaluating candidate input profiles; and,
- to run multiple search processes in parallel in order (a) that a solution can be taken from the first search process to finish, and (b) that solutions may be migrated between search processes.

Conclusions Specific to the GPU Environment Our first set of conclusions are specific to our chosen example of a parallel computing environment, i.e. commodity GPU cards.

We have demonstrated that GPU cards are a viable execution environment for our algorithm. We believe this environment to be most effective for SUTs with relatively long execution times. We observed that for the SUT with longest execution time, `epuck`, input profiles could be derived almost four times more quickly using the GPU card in our original architecture with two levels of parallelism (Experiment XV), and over forty times more quickly using a revised architecture with one level of parallelism (Experiment XVI). For SUTs with shorter execution times, such as `bestMove` and `tcas`, the difference between the GPU and CPU environments is not as great and we suspect that for SUTs such as these, which of the two environments demonstrates the faster run times will often depend on the raw performance of the underlying hardware rather than difference between parallel and single-threaded architectures. For other SUTs, such as `circBuff`, the constraints on the device-side language features prevent the use of the GPU environment.

Conclusions Relating to Parallel Environments in General Our second set of conclusions are not specific to the GPU environment. These conclusions relate to algorithm improvements enabled by the use of multiple search processes running in parallel, and we suspect qualitatively similar improvements could be realised by exploiting other parallel computing environments such as multicore CPUs, clusters, grids, and cloud architectures.

Experiment XIV demonstrated that running multiple search processes in parallel, and taking the result from the first search process to finish, reduces the time taken to derive an input profile. The run times observed in this experiment were improved by a factor of at least 1.58 over would be expected from the raw performance of the parallel environment. Moreover, the variance in the run time was reduced.

Experiment XV demonstrated that regular migrations of the fittest input profiles between parallel search processes enabled a further improvement in run times. The run times observed in this experiment were improved by a factor of at least 1.66.

Unaddressed Issues We note the following unaddressed issues:

Generality: Our conclusions are drawn from experiments that consider a small number of example SUTs. Experiments on further SUTs, and a more detailed analysis of the timing of the parts of search implemented in the GPU kernel, could improve confidence in the generality of these conclusions.

Algorithm Configuration and Parameters: These experiments have used the configuration and parameter settings that were optimised, in chapter 3, for an algorithm that used a single search process. There is an opportunity to determine the optimal configuration and parameter settings for the parallelised GPU implementation of the algorithm: such optimised settings may demonstrate run times improvements greater than observed in the experiments of this chapter.

Chapter 6

Conclusions

6.1 *Evaluation of the Research Hypothesis*

This thesis was motivated by the need to reduce the cost of deriving concrete strategies—strategies specific to the software-under-test—for abstract testing strategies that are effective at detecting faults. Our exemplar testing strategy was statistical testing, a probabilistic form of structural coverage, with the hypothesis that:

> *Concrete strategies for statistical testing can be derived cost-effectively for a wide range of software using automated computational search.*

We identify three propositions in this hypothesis: the existence of a suitable algorithm based on automated computational search; that the algorithm may be applied to a wide range of software; and that the algorithm is cost-effective in terms of time and resources required to derive a strategy.

The research described in this thesis provides evidence in support of all three propositions, as follows.

Existence The existence of a suitable search-based algorithm was demonstrated in chapter 2. In addition, we verified that strategies derived by the algorithm possess the superior fault-detecting ability previously demonstrated of manually-derived strategies.

Applicability Many real-world SUTs take highly-structured inputs, require a sequence of operations to set their internal state during testing, or take a long time to execute. In chapter 4 we demonstrated a novel representation for input profiles based on a stochastic grammar that enables the algorithm to be applied to SUTs with highly-structured inputs, and to SUTs with state. The use of parallel computation for SUTs with relatively long execution times was illustrated in chapter 5.

Cost-Efficiency In chapter 3, the performance of the algorithm was optimised through enhancements to its configuration, i.e. the combination of representation, fitness metric, and search method. By improving the algorithm performance in this way, the computing resources consumed and time taken to derive a concrete strategy are reduced, and therefore the cost-efficiency of the technique is increased.

Common parameter settings that enable acceptable algorithm performance over a range of SUTs were identified. This guidance on parameter setting reduces or eliminates the need for a test engineer to tune the algorithm parameters to the particular software-under-test, and so avoids a potential cost of using the algorithm in practice.

Commodity GPU cards provide a low-cost, highly-parallel computing environment. The use of GPUs to parallelise the execution of the instrumented SUT during fitness evaluation, and to run multiple searches simultaneously, was demonstrated in chapter 5, and we concluded that for some types of SUT, GPUs provide a cost-efficient environment for running the algorithm.

For all seven real-world SUTs used as case studies in this thesis, concrete strategies may be derived by the algorithm in a matter of minutes using a CPU core, or a GPU card, typical of a desktop computer.

6.2 Opportunities for Further Research

6.2.1 Diversity

In chapter 2, we identified the tendency for the algorithm to generate input profiles with poor diversity: the range of different test inputs sampled from the profile is smaller than would appear to be necessary to achieve the target minimum coverage probability. In section 2.8 we speculated that low diversity is the cause of the poor fault-detecting ability in algorithm-derived profiles with the highest minimum coverage probability.

The addition of a simple diversity measure to the fitness metric (Experiment IV, section 2.9) provided evidence in support of our hypothesis that diversity was lost during the search process, but only partially resolved the issue. Moreover, the calculation of the diversity measure required the maintenance of a large data structure that increased the space and time complexity of the algorithm.

A suitable mechanism for preventing diversity loss would therefore be a valuable research direction. Possible approaches include:

- Exploring the trade-offs between minimum coverage probability and diversity. It is possible that profiles with less than optimal minimum coverage probabilities continue to provide acceptable fault-detecting ability. The results of Experiment III (section 2.7) are consistent with this hypothesis (although the relationship between minimum coverage probability and fault-detecting ability is confounded with the effect of diversity loss). If so, the effect of diversity loss could be minimised by using a lower target minimum coverage probability, while retaining the cost-effectiveness of the testing strategy.

- Identifying more effective diversity measures. The measure used in Experiment IV (section 2.9) was relatively naive and greatly increased the implementation's runtime and memory requirements. There is an opportunity to identify more effective diversity measures that have less overhead in terms of run time and memory.

- Encouraging diversity in the representation. It may be effective to enhance the representation of input profiles to encourage diversity as an alternative to using a diversity measure as a component of the fitness metric. For example, a straightforward approach would be to enforce a suitable minimum value for production/bin probabilities so that *all* inputs have a non-zero probability of being selected. The implementation of the algorithm using the SCFG representation applies a small minimum to probability values as a consequence of the integer representation of weights (section 4.4.4); whether this, or a higher, setting for the minimum encourages diversity could be investigated.

- Using multiple profiles. In Experiment XIV (section 5.5), commodity GPU cards were used to run multiple search processes in parallel, and the input profile returned by the algorithm was from the first of the search processes to derive a profile. Instead of terminating after the first search process, the algorithm could instead terminate after the nth search process, providing n different profiles. The *combination* of these n profiles would provide the concrete strategy: when sampling a test input, one of the n profiles is selected at random and the input is sampled from that profile. The 'combined' profile might demonstrate significantly more diversity than a single profile, but it would cost little extra to derive if a parallel computing environment, such as GPU card, were used.

6.2.2 Novel Adequacy Criteria

In this thesis the algorithm derived a concrete strategy—a SUT-specific input profile—that satisfied the adequacy criterion of statistical testing: that the probability of exercising any given coverage element by a single input chosen at random from the profile is above a chosen minimum.

However the algorithm could be applied to any testing adequacy criterion expressed as a property of a probability distribution over the SUT's input domain. There is therefore an opportunity to investigate the effectiveness of other criteria, particularly those for which it would be impractical to derive concrete strategies using manual techniques.

One class of such novel criteria are those that combine two or more 'traditional' adequacy criteria. Potential examples are: the combination of a structural criterion (such as branch coverage) with a non-functional criterion (such as coverage of a range of execution times); or, the combination of a criterion mandated by certification authorities (e.g. MC/DC coverage of safety-critical avionics software) with another criterion believed to be more effective at detecting faults. The concrete strategy, derived using the automated algorithm, would generate test sets that simultaneously satisfy both criteria, potentially reducing overall testing costs.

6.2.3 Application to System-Level Testing

The real-world SUTs used as case studies in this thesis have been relatively small and self-contained: they would normally be verified by unit testing. However the formulation of the algorithm does not assume testing at the level of units: the fitness metric is expressed in terms of abstract coverage elements which need not be unit-level structural elements such as branches or conditions.

The algorithm, therefore, could be used to derive concrete strategies at the system level for use during integration, system, and acceptance testing. It would be be impractical, however, to execute an instrumented version of system the large number of times required by the algorithm. Instead, the use of a suitable abstraction of the system in the form of an executable model could be viable.

6.2.4 Use of Structural Search-Based Software Testing Approaches

When the algorithm is in a region of the search space where one or more of the SUT's coverage elements are very rarely exercised by inputs sample from the input profiles, the fitness metrics of the current profile and its neighbours is zero and so there is no gradient to guide the search. This was the motivation for algorithm enhancement called 'directed mutation' (section 3.6) which uses feedback from the rare occasions that the least-exercised coverage elements *are* exercised to guide the search by introducing dynamic bias in the mutation operators.

However, directed mutation cannot operate when the least-exercised coverage elements are *never* exercised by the finite sample of inputs used for fitness evaluation. A solution would be to incorporate the approaches taken by search-based software testing techniques for structural testing discussed in section 1.4.4. The approximation level and branch distance metrics could be used to assess how close an input profile came to exercising the never-exercised elements, and thus provide an additional component of the fitness metric that might guide the search. The program stretching approach of (Ghani and Clark, 2009) could also be used to 'stretch' the SUT in order that all coverage elements are exercised; once the search has gained traction, the SUT would be 'relaxed' back to its original form.

6.2.5 *Plans for Future Work*

The research opportunities outlined above are compelling from both an academic and practical point-of-view. I hope to address many of them in my future work.

List of References

E. Alba and F. Chicano. Software project management with GAs. *Information Sciences*, 177 (11):2380–2401, 2007.

S. Ali, L. Briand, H. Hemmati, and R. Panesar-Walawege. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering*, 36(6):742 –762, Nov–Dec 2010.

N. Baskiotis, M. Sebag, M.-C. Gaudel, and S. Gouraud. A machine learning approach for statistical software testing. In *Proc. International Joint Conferences on Artificial Intelligence*, pages 2274–2279, 2007.

P. J. Boland, H. Singh, and B. Cukic. Comparing partition and random testing via majorization and Schur functions. *IEEE Transactions on Software Engineering*, 29(1):88–94, Jan 2003.

P. M. Bueno, M. Jino, and W. E. Wong. Diversity oriented test data generation using metaheuristic search techniques. *Information Sciences*, 2011.

T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

J. Clark, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings — Software*, 150(3):161–175, 2003.

L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, September 1976.

D. R. Cox. Regression models and life tables. *Journal of the Royal Statistical Society, Series B*, 34:187–220, 1972.

L. N. De Castro and J. Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 2002.

M. E. Delamaro and J. C. Maldonado. Proteum/IM 2.0: An integrated mutation testing environment. In *Mutation Testing for the New Century*, pages 91–101. Kluwer Academic Publishers, 2001. ISBN 0-7923-7323-5.

A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A generic method for statistical testing. Technical Report 1386, Laboratoire de Recherche en Informatique (LRI), CNRS - Univ. de Paris Sud, 2004.

H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10 (4):405–435, 2005.

J. Duran and S. Ntafos. An evaluation of random testing. *IEEE Trans.*, SE-10(4):438–444, 1984.

École Polytechnique Fédérale de Lausanne. e-puck educational robot. Available at: `http://www.e-puck.org` [Accessed 3 May 2012], 2012.

P. Emberson and I. Bate. Minimising task migrations and priority changes in mode transitions. In *Proc. 13th IEEE Real-Time And Embedded Technology And Applications Symp. (RTAS 07)*, pages 158–167, 2007.

A. J. Feelders. Statistical concepts. In M. Berthold and D. J. Hand, editors, *Intelligent Data Analysis*, chapter 2. Springer, 2nd edition, 2003.

S. Forrest and M. Mitchell. Relative building-block fitness and the building block hypothesis. In D. L. Whitley, editor, *Foundations of Genetic Algorithms 2*, pages 109–126. Morgan Kaufmann, 1993.

P. G. Frankl and E. J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Transactions on Software Engineering*, 19(3):202–2113, March 1993.

M. Galassi et al. *GNU Scientific Library Reference Manual*. Network Theory, 3rd edition, 2009.

J. Gaspar. Boost C++ Libraries - Templated Circular Buffer Container. Available at: `http://www.boost.org/doc/libs/1_50_0/libs/circular_buffer/doc/circular_buffer.html` [Accessed 30 July 2013], 2012.

K. Ghani and J. A. Clark. Widening the goal posts: Program stretching to aid search based software testing. In *Proc. 1st International Symposium on Search Based Software Engineering (SSBSE 2009)*, pages 122–131, 2009.

P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2008)*, pages 206–215, 2008.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1979.

J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, SE-3, June 1975.

S.-D. Gouraud. AuGuSTe: a tool for statistical testing experimental results. Technical Report 14000, Laboratoire de Recherche en Informatique (LRI), CNRS - Univ. de Paris Sud, 1995.

S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Proc. IEEE International Conference on Automated Software Engineering*, 2001.

J. S. Gourlay. A mathematical framework for the investigation of testing. *IEEE Transactions on Software Engineering*, SE-9(6), November 1983.

J. J. Grefenstette. Optimization of control parameters for genetic algorithms. *IEEE Transactions on Systems, Man and Cybernetics*, 16(1):122 –128, January 1986.

J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In B. Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 137–147, July 2010.

D. Hamlet. When only random testing will do. In *Proc. First International Workshop on Random Testing (RT'06)*, pages 1–9, 2006.

C. Hao, J. A. Clark, and J. L. Jacob. Synthesising efficient and effective security protocols. In *Proc. Workshop Automated Reasoning for Security Protocol Analysis (ARSPA) 2004*, pages 25–40, 2004.

M. Harman. The current state and future of search based software engineering. *Proc. 29th International Conference on Software Engineering (ICSE 2007), Future of Software Engineering (FoSE)*, pages 342–357, 2007.

M. Harman and L. Tratt. Pareto optimal search based refactoring at the design level. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO 2007)*, pages 1106–1113, 2007.

K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. *A Practical Tutorial on Modified Condition/Decision Coverage*. National Aeronautics and Space Administration, 2001.

D. Hoffman, H.-Y. Wang, M. Chang, D. Ly-Gagnon, L. Sobotkiewicz, and P. Strooper. Two case studies in grammar-based test generation. *Journal of Systems and Software*, 83(12):2369 – 2378, 2010.

J. H. Holland. *Adaption in Natural and Artifical Systems*. University of Michigan Press, 1975.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages and computation*. Pearson Addison-Wesley, 3rd edition, 2006.

A. Inselberg and B. Dimsdale. Parallel coordinates: a tool for visualizing multi-dimensional geometry. In *Proc. 1st conference on Visualization (VIS '90)*, pages 361–378, 1990.

Intel Corporation. Intel core i7-3900 desktop processor series export compliance matrix. Available at: `http://download.intel.com/support/processors/corei7/sb/core_i7-3900_d.pdf` [Accessed 23 December 2012], March 2011.

Intel Corporation. Intel core i7-3930k processor. Available at: `http://ark.intel.com/products/63697/Intel-Core-i7-3930K-Processor-12M-Cache-up-to-3_80-GHz` [Accessed 23 December 2012], 2012.

Khronos Group. Conformant Products. Available at: `http://www.khronos.org/conformance/adopters/conformant-products#topencl` [Accessed 23 December 2012], 2012.

S. Kirkpatrick, C. D. Gellatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

K. B. Korb and A. E. Nicholson. *Bayesian Articial Intelligence*. CRC Press, 2011.

B. Korel. Automatic software test data generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

B. Littlewood. The problems of assessing software reliability... when you really need to depend on it. In *Lessons in System Safety: Proc. 8th Safety-Critical System Symposium*, 2000.

M. Lundy and A. Mees. Convergence of an annealing algorithm. *Mathematical Programming*, 34:111–124, 1986.

R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE '07)*, pages 134–143, 2007.

Y. K. Malaiya. Antirandom testing: getting the most out of black-box testing. In *Proc. Int'l Symposium on Software Reliability Engineering (ISSRE)*, pages 86–95, 1995.

G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(14):1–6, 2003.

J. C. Martin. *Introduction to languages and the theory of computation*. McGraw-Hill, 1996.

P. Maurer. Generating test data with enhanced context-free grammars. *IEEE Software*, 7(4):50 –55, July 1990.

W. M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100–107, 1998.

P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering*, SE-2(3):223–226, 1976.

F. Mondada et al. The e-puck, a robot designed for education in engineering. In *Proc. 9th Conference on Autonomous Robot Systems and Competitions*, pages 59–65, 2009.

D. C. Montgomery. *Design and Analysis of Experiments*. John Wiley & Sons, Inc., 6th edition, 2005.

G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, Inc., 2011.

R. H. Myers and D. C. Montgomery. *Response surface methodology: process and product optimization using designed experiments*. Wiley Series in Probability and Statistics. John Wiley & Sons, Inc., 2005.

NVIDIA Corporation. NVIDIA GeForce GTX 680 Whitepaper. Available at: `http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf` [Accessed 23 December 2012], 2012a.

NVIDIA Corporation. CUDA GPUs. Available at: `http://developer.nvidia.com/cuda-gpus` [Accessed 23 December 2012], 2012b.

NVIDIA Corporation. NVIDIA Tesla C1060 Computing Processor - Many Core Supercomputing for Workstations. Available at: `http://www.nvidia.co.uk/object/tesla_c1060_uk.html` [Accessed 25 December 2012], 2012c.

NVIDIA Corporation. *CUDA C Programming Guide*. NVIDIA Corporation, version 5.0 edition, October 2012 2012d.

NVIDIA Corporation. *CUDA C Best Practices Guide*. NVIDIA, version 5.0 edition, October 2012 2012e.

B. Paechter, A. Cumming, M. Norman, and H. Luchian. Extensions to a memetic timetabling system. In E. Burke and P. Ross, editors, *Practice and Theory of Automated Timetabling*, volume 1153 of *Lecture Notes in Computer Science*, pages 251–265. Springer Berlin / Heidelberg, 1996.

R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification and Reliability*, 9:263–282, 1999.

T. J. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Programmers, 2007.

S. Poulding, P. Emberson, I. Bate, and J. Clark. An efficient experimental methodology for configuring search-based design algorithms. In *Proc. 10th IEEE High Assurance Systems Engineering Symposium (HASE '07)*, pages 53–62, 2007.

C. V. Ramamoorthy, S.-B. F. Ho, and W. T. Chen. On the automated generated of program test data. *IEEE Transactions on Software Engineering*, SE-2(4):293–300, December 1976.

S. Rapps and E. J. Weyuker. Data flow analysis techniques for test data selection. In *Proc. 6th International Conference on Software Engineering (ICSE '82)*, pages 272–278, 1982.

Research Triangle Institute. The economic impacts of inadequate infrastructure for software testing. Technical Report RTI Project Number 7007.011, National Institute of Standards and Technology, May 2002.

E. Ridge and D. Kudenko. Analyzing heuristic performance with response surface models: prediction, optimization and robustness. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO 2007)*, pages 150–157, 2007.

P. Ross, D. Corne, and H.-L. Fang. Improving evolutionary timetabling with delta evaluation and directed mutation. In Y. Davidor, H.-P. Schwefel, and R. Männer, editors, *Parallel Problem Solving from Nature — PPSN III*, volume 866 of *Lecture Notes in Computer Science*, pages 556–565. Springer Berlin / Heidelberg, 1994.

H. Scheffé. Experiments with mixtures. *Journal of the Royal Statistical Society. Series B (Methodological)*, 20(2):344–360, 1958.

E. G. Sirer and B. N. Bershad. Using production grammars in software testing. *SIGPLAN Notices*, 35(1):1–13, December 1999.

I. Sommerville. *Software Engineering*. Addison-Wesley, 8th edition, 2007.

Sun Corporation. JDK 1.4 Demo Applets. Available at: `http://java.sun.com/applets/jdk/1.4/index.html` [Accessed 20 Aug 2012], 2006.

R. Tavares, A. Teófilo, P. Silva, and A. C. Rosa. Infected genes evolutionary algorithm. In *Proceedings of the 1999 ACM Symposium on Applied computing (SAC '99)*, pages 333–338, 1999.

P. Thévenod-Fosse and H. Waeselynck. An investigation of statistical software testing. *Software Testing, Verification and Reliability*, 1(2):5–25, 1991.

P. Thévenod-Fosse and H. Waeselynck. Statemate applied to statistical testing. In *Proc. Int'l Symp. Software Testing and Analysis (ISSTA'93)*, pages 99–109, 1993.

P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An experimental study on software structural testing: deterministic versus random input generation. In *Fault-Tolerant Computing, 1991. FTCS-21. Digest of Papers., Twenty-First International Symposium*, pages 410 –417, June 1991.

P. Thévenod-Fosse and H. Waeselynck. Towards a statistical approach to testing object-oriented programs. In *Proc. IEEE Annual International Symposium on Fault Tolerant Computing (FTCS-27)*, pages 99–108, 1997.

P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. Software statistical testing. Technical Report 95178, Laboratoire d'Analyse et d'Architecture des Systèmes du CNRS (LAAS), 1995.

D. Thierens. Adaptive mutation rate control schemes in genetic algorithms. In *Proc. Congress on Evolutionary Computation (CEC '02)*, volume 1, pages 980 –985, May 2002.

J. Tobin. Estimation of relationships for limited dependent variables. *Econometrica*, 26:24–36, Jan 1958.

N. J. Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety-Critical Software*. PhD thesis, Dept. Computer Science, University of York, 2000.

K. K. Vadde, V. R. Syrotiuk, and D. C. Montgomery. Optimizing protocol interaction using response surface methodology. *IEEE Transactions on Mobile Computing*, 5(6):627–639, June 2006.

A. Vargha and H. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2): 101–132, 2000.

J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43:841–854, 2001.

E. Weyuker and T. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, SE-6(3):236–246, May 1980.

D. White, J. A. Clark, J. Jacob, and S. Poulding. Evolving software in the presence of resource constraints. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO 2008)*, pages 1775–1782, 2008.

D. R. White and S. Poulding. A rigorous evaluation of crossover and mutation in genetic programming. In *Proc. 12th European Conference on Genetic Programming (EuroGP '09)*, pages 220–231. Springer-Verlag, 2009.

N. Whitehead and A. Fit-Florea. *Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs*. NVIDIA Corporation, 2011.

F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computer Systems*, 7(3):36:1–36:53, May 2008.

S. Yoo, M. Harman, and S. Ur. Highly scalable multi objective test suite minimisation using graphics cards. In *Proc. 3rd International Symposium on Search-Based Software Eng. (SSBSE 2011)*, pages 219–236, 2011.

Y. Zhang, M. Harman, and S. A. Mansouri. The multi-objective next release problem. In *Proc. Genetic and Evolutionary Computation Conf. (GECCO 2007)*, pages 1129–1137, 2007.

H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.