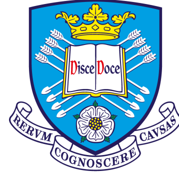
NASSER M. ALBUNIAN

AN INVESTIGATION OF SEARCH BEHAVIOUR IN SEARCH-BASED UNIT TEST GENERATION



The University Of Sheffield.

AN INVESTIGATION OF SEARCH BEHAVIOUR IN SEARCH-BASED UNIT TEST GENERATION

NASSER M. ALBUNIAN

SUPERVISED BY
PROF. GORDON FRASER, UNIVERSITY OF PASSAU, GERMANY
DR. DIRK SUDHOLT, THE UNIVERSITY OF SHEFFIELD, UK

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF SHEFFIELD
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE
SEPTEMBER 2020

Nasser M. Albunian: *An Investigation of Search Behaviour in Search-Based Unit Test Generation,* © September 2020

Dedicated to my beloved parents, my family, my little kids, and foremost my lovely wife.

ABSTRACT

As software testing is a laborious and error-prone task, automation is desirable. Search-based unit test generation applies evolutionary search algorithms to generate software tests and, in the context of unit testing object-oriented software, Genetic Algorithms (GAs) are frequently employed to generate unit tests that maximise code coverage.

Although GAs are effective at generating tests that achieve high code coverage, they are still far from being able to satisfy all test goals (e.g., covering all branches). While some general limitations are known, there is still a lack of understanding of the search behaviour during the optimization, making it difficult to identify the factors that make a search problem difficult.

Therefore, this thesis aims to investigate the search behaviour when GAs are applied to generate object-oriented unit tests and, more specifically, identify the reasons why the search fails to achieve the desired test goals. This is achieved by investigating (1) the fitness landscape structure and the impact of its features on the generation of unit tests and (2) the influence of population diversity on generating potential unit tests. Based on the outcome of this investigation, the impact of test case reduction on the landscape features and population diversity is also investigated.

Our results reveal that classical indicators for rugged fitness landscapes suggest well searchable problems in the case of unit test generation, but the fitness landscape for most problem instances is dominated by detrimental plateaus. However, increasing diversity does not have a beneficial effect on coverage in general, but it may improve coverage when diversity is promoted adaptively. In fact, increasing diversity has a negative impact on the individual length, which can also be mitigated with the adaptive diversity. Applying the test case reduction seems to be promising in improving the landscape structure and reducing the negative side effects of diversity on length, but have no considerable impact on the search performance.

PUBLICATIONS

Albunian, N.M. "Diversity in search-based unit test suite generation". In *Proceedings of the 9th International Symposium on Search Based Software Engineering (SSBSE '17)*. Cham: Springer International Publishing, 2017, pp. 183-189.

Nasser Albunian, Gordon Fraser, and Dirk Sudholt. "Causes and effects of fitness landscapes in unit test generation". In *Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO '20)*. Association for Computing Machinery, 2020, pp. 1204–1212.

Nasser Albunian, Gordon Fraser, and Dirk Sudholt. "Measuring and Maintaining Population Diversity in Search-based Unit Test Generation". In *Proceedings of the 12th International Symposium on Search Based Software Engineering (SSBSE '20)*. Cham: Springer International Publishing, 2020, pp. 153-168.

ACKNOWLEDGMENTS

First and foremost, all praises to almighty Allah for giving me the opportunity, determination and strength in completing this research.

I would like to thank and express my deep and sincere gratitude to my supervisors, Gordon Fraser and Dirk Sudholt, for their patience, continuous support, guidance and encouragement. Without their assistance and dedicated involvement in every step throughout my PhD, this thesis would have never been accomplished.

I should thank my friends for their support and the constant encouragement: Ibrahim Althomali, Abdullah Alsharif, and Abdulaziz Almanea who made my journey a great experience. I am also very thankful to Sina Shamshiri, José Miguel Rojas, and José Campos for their advice and assistance during my Phd. I am grateful to the members of the Verification and Testing groups for their help and encouragement.

Special thanks are due to my best friend and ex-flatmate, Anas Alsuhaibani, for his support, friendship, and for the many enjoyable moments we spent together.

I would like to thank the Institute of Public Administration for funding my scholarship. I am very grateful to Riyadh Alkhudairi for his help and support when needed.

I must express my very profound gratitude to my parents and my family for providing me with unfailing support and continuous encouragement throughout my years of study. My special, profound and affectionate thanks and gratitude are due to the love of my life, my wife Sumaiah, for being kind and supportive to me over the last several years. Last but not least, my deep love and thanks are due to my kids, Mohammed and Taim, who are the only source of inspiration to me.

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation 2
- 1.2 Thesis Aim and Objectives 6
- 1.3 Thesis Structure and Contributions 6

2 LITERATURE REVIEW 11
- 2.1 Software Testing 11
  - 2.1.1 Related Concepts 11
  - 2.1.2 Test Adequacy 13
- 2.2 Automation of Test Generation 16
  - 2.2.1 Random Testing 16
  - 2.2.2 Symbolic Execution based Testing 18
  - 2.2.3 Search-Based Software Testing 19
  - 2.2.4 Test Oracle Problem 32
  - 2.2.5 A Review of Existing Studies 34
- 2.3 Fitness Landscape Analysis 41
  - 2.3.1 Ruggedness 41
  - 2.3.2 Neutrality 42
  - 2.3.3 Fitness Landscape Measurements 43
  - 2.3.4 Fitness Landscape Analysis in Test Generation 47
- 2.4 Population Diversity in Evolutionary Algorithms 50
  - 2.4.1 Population Diversity Measurement 50
  - 2.4.2 Population Diversity Maintenance Techniques 52
  - 2.4.3 Population Diversity Control Techniques 55
  - 2.4.4 Population Diversity in Test Generation 57
- 2.5 Summary 59

3 CAUSES AND EFFECTS OF FITNESS LANDSCAPES IN UNIT TEST GENERATION 61
- 3.1 Introduction 61
- 3.2 Fitness Landscape Analysis with the Whole Suite Approach 63
  - 3.2.1 Experimental Setup 64
  - 3.2.2 Threats to Validity 65
  - 3.2.3 Experimental Results 66
- 3.3 Fitness Landscape Analysis with the Many-Objective Sorting Algorithm 70
  - 3.3.1 Experimental Setup 71
  - 3.3.2 Threats to Validity 72
  - 3.3.3 Experimental Results 73
- 3.4 A comparison of the impact of fitness landscape properties on WSA and MOSA 76
- 3.5 What are the underlying properties of source code that influence the fitness landscape? 79

3.6 Summary 86
4 MEASURING AND MAINTAINING POPULATION DIVERSITY IN UNIT TEST GENERATION 87
4.1 Introduction 87
4.2 An Investigation of Population Diversity with the Whole Suite Approach 88
4.2.1 Measuring Population Diversity in Test Suite Generation 88
4.2.2 Maintaining Population Diversity in Test Suite Generation 91
4.2.3 Empirical Study 96
4.3 An Investigation of Population Diversity with the Many-Objective Sorting Algorithm 112
4.3.1 Measuring Population Diversity in Test Case Generation 112
4.3.2 Maintaining Population Diversity in Test Case Generation 113
4.3.3 Empirical Study 116
4.4 A Comparison of the impact of population diversity on WSA and MOSA 133
4.5 How does the landscape structure affect the population diversity? 139
4.6 Summary 141
5 AN ANALYSIS OF THE EFFECTS OF TEST CASE REDUCTION ON UNIT TEST GENERATION 143
5.1 Introduction 143
5.2 Background 145
5.3 Test Case Reduction Rules 146
5.4 Empirical Study 150
5.4.1 Experimental Setup 150
5.4.2 Experimental Results 153
5.5 Discussion 161
5.6 Summary 162
6 CONCLUSION AND FUTURE WORK 163
6.1 Summary of Contributions 163
6.1.1 Fitness Landscape Analysis 163
6.1.2 Population Diversity Analysis 164
6.1.3 An Analysis of Test Case Reduction Approach 165
6.2 Future Work 165
6.2.1 Fitness Landscape Improvement 165
6.2.2 Improved Versions of MOSA 166
6.2.3 Coverage Criteria 166
6.2.4 Fault Finding Effectiveness 167
6.2.5 Alternative Diversity Techniques 167
6.3 Overall Conclusion 167

BIBLIOGRAPHY 169

LIST OF FIGURES

Figure 2.1 An example of Hill Climbing landscape [110] 21
Figure 2.2 An example of Simulated Annealing landscape [110] with possible movements to worse solutions 22
Figure 2.3 An example of non-dominant fronts in the NSGA [55] 25
Figure 2.4 Two examples of different fitness values resulted from a random walk of six steps . 44
Figure 2.5 The fitness landscape of the objective function when applied on the flag example 48
Figure 3.1 Three different runs of a random walk performed on three classes 66
Figure 3.2 Results of the six fitness landscape measures applied on the 331 classes with the WS approach 67
Figure 3.3 Four classes with runs that result in low AC values 68
Figure 3.4 The Spearman correlation of branch coverage with each of the six measures for all the 331 classes with . The correlation coefficient of branch coverage and AC is -0.16, ND is -0.14, NV is 0.25, IC is 0.16, PIC is 0.16, and DBI is -0.17 69
Figure 3.5 Results of the six fitness landscape measures applied on the branches of the 331 classes 73
Figure 3.6 The Spearman correlation of SR with each of the six measures for all the branches of 331 classes. The correlation coefficient of SR and AC is 0.04, ND is -0.34, NV is 0.41, IC is 0.488, PIC is 0.476, and DBI is -0.481 75
Figure 3.7 Comparison of the six fitness landscape measures applied on the random walks with the objective functions of WSA and MOSA. . 76
Figure 3.8 Comparison of the Spearman correlation of branch coverage/SR with each of the six measures based on the objective functions of WSA and MOSA. 78
Figure 3.9 Four groups of the branches based on their coverage by MOSA and random walk (RW) where a large bubble size indicates a high number of branches 79
Figure 3.10 Number of method executions during the random walk for each branch in the four groups 81

Figure 3.11 Types of methods containing each branch in the four groups 83
Figure 3.12 Number of exceptions thrown by methods containing each branch in the four groups . . 83
Figure 3.13 Number of discrete fitness values obtained by the random walk for each branch in the four groups 84
Figure 3.14 Classifications of the branch types in the four groups 85
Figure 3.15 Examples of branch types classified by Shamshiri et al [150] 86
Figure 4.1 Two automatically generated example test cases to illustrate statement difference 92
Figure 4.2 Average values for coverage and population diversity throughout evolution in Monotonic GA for the four groups of CUTs. 99
Figure 4.3 Diversity throughout the evolution with Monotonic GA and the five diversity maintenance techniques based on different distance measurements. 106
Figure 4.4 Coverage and length over time with Monotonic GA and the five diversity maintenance techniques based on different distance measurements. 108
Figure 4.5 Coverage and length over time with Monotonic GA, fitness sharing (FS), and adaptive fitness sharing (AFS) per four groups of CUTs . 110
Figure 4.6 Average values for coverage and population diversity throughout evolution in MOSA for the four groups of CUTs. 118
Figure 4.7 Diversity throughout the evolution with MOSA and the seven diversity maintenance techniques based on different distance measurements. 125
Figure 4.8 Coverage and length over time with MOSA and the five diversity maintenance techniques based on different distance measurements. . . . 128
Figure 4.9 Coverage and length over time with MOSA, fitness sharing (FS), and adaptive fitness sharing (AFS) per four groups of CUTs 130
Figure 4.10 The average count of executed statements types with MOSA, fitness sharing (FS), and adaptive fitness sharing (AFS) 132

Figure 4.11 Average coverage (Cov.) and population diversity throughout evolution in both Monotonic GA using WSA and MOSA for the four groups of CUTs where a value close to 100% indicates high coverage/diversity. The diversity is measured based on the three measures: Entropy (Ent.), Genotype (Gen.), and Phenotype (Phen.). 134
Figure 4.12 Average population diversity throughout evolution in both Monotonic GA using WSA and MOSA when considering diversity maintenance techniques based on the three measures: Entropy (Ent.), Genotype (Gen.), and Phenotype (Phen.). 136
Figure 4.13 The Spearman correlation of entropy based on default MOSA with each of the six measures for all the branches of 331 classes. The correlation coefficient of entropy and AC is −0.028, ND is −0.58, NV is 0.61, IC is 0.66, PIC is 0.64, and DBI is −0.59. 139
Figure 4.14 The Spearman correlation of entropy based on AFS-fitness with each of the six measures for all the branches of 331 classes. The correlation coefficient of entropy and AC is −0.045, ND is −0.64, NV is 0.65, IC is 0.72, PIC is 0.71, and DBI is −0.70. 140
Figure 5.1 An overview diagram that demonstrates the reduction rules 148
Figure 5.2 Comparison of the six fitness landscape measures with the WSA approach based on the default and reduction versions of the random walk. 153
Figure 5.3 Comparison of the six fitness landscape measures with the MOSA based on the default and reduction versions of the random walk. . 154
Figure 5.4 Coverage, length, and diversity measures over time with Monotonic GA using WSA and adaptive fitness-based sharing (AFS), and their reduction-based versions. 158
Figure 5.5 Coverage, length, and diversity measures over time with MOSA and adaptive fitness-based sharing (AFS), and their reduction-based versions. 159

Figure 5.6 Comparison the performance of the Monotonic GA using WSA and MOSA based on the branch coverage ("Red. Sig. Higher" is the number of classes where reduction-based GA achieved significantly higher coverage than its default version, "Red. Higher" is the number of classes where reduction-based GA achieved higher coverage (but not significantly), "Equivalent" is the number of classes where both versions of the GA result in similar coverage, and both "Red. Sig. Lower" and "Red. Lower" represent the classes where the reduction-based GA results in lower coverage than the default.) . . 160

LIST OF TABLES

Table 2.1 Studies that investigate the performance of the proposed GAs in generating Java unit test suites based on the achieved branch coverage. The algorithms are RS (Random Search), SBS (Single Branch Strategy), WS (Whole Suite) and its archive-based variant (WSA), MOSA, and DynaMOSA. The reported results are either the average branch coverage (shown with percentage) or the number of CUTs/targets (branches). 35
Table 3.1 An example of applying the random walk of 6 steps on a class with 5 branches 71
Table 3.2 Average effect size $\hat{A}_{12}$ with p-values computed for each landscape measure with both objective functions 77
Table 3.3 The average values of the six landscape measures for the branches of the four groups 80
Table 4.1 An example of two test suites (TS_1 and TS_2) containing different test cases (t_i) that vary in how often they execute the predicates (p_j) of the CUT. 89
Table 4.2 An example of the distance values calculated between five test suites 95
Table 4.3 Average diversity over time when applying diversity maintenance techniques based on different distance measures, and the average effect size $\hat{A}_{12}$. 101
Table 4.4 Spearman correlation between the diversity achieved by the five diversity techniques based on different distance measures and length. . . . 107
Table 4.5 Number of classes where adaptive fitness sharing has an increased/decreased/equal coverage compared to Monotonic GA, the average effect size $\hat{A}_{12}$ and the number of classes for which this comparison is statistically significant ($\alpha = 0.05$). 111
Table 4.6 Average diversity over time when applying diversity maintenance techniques based on different distance measures, and the average effect size $\hat{A}_{12}$. 119

Table 4.7 Spearman correlation between the diversity achieved by the seven diversity techniques based on different distance measures and length. . . . 126
Table 4.8 Number of classes where adaptive fitness sharing has an increased/decreased/equal coverage compared to MOSA, the average effect size $\hat{A}_{12}$ and the number of classes for which this comparison is statistically significant ($\alpha = 0.05$).131
Table 4.9 Coverage and Length resulting from Monotonic GA using WSA and MOSA when considering the diversity maintenance techniques. . . 137
Table 5.1 Number of significant classes for the comparison of landscape measures with and without reduction when considering the two algorithms, and the average effect size $\hat{A}_{12}$. 155
Table 5.2 The average effect size $\hat{A}_{12}$ computed for coverage, length, and the diversity measures with the default GA and default AFS/reduction-based GA/reduction-based AFS. 157

1 INTRODUCTION

"Software eventually and necessarily gained the same respect as any other discipline."

Margaret H. Hamilton, 2014 [74]

It all started in the mid of 1800s when the mathematician Ada Lovelace created the first computer program to calculate a sequence of Bernoulli numbers to be carried out on an early computing machine known as Analytical Engine. Later, in 1948, Tom Kilburn [114] wrote the first piece of software that was intended to perform the calculation of the greatest divisor of 2^{18} on a digital computer, despite the fact that the term *software* was not coined until 1958 by John Tukey [167].

From that time onwards, software became an inevitable solution that many businesses and industries rely on to provide efficient services, especially in recent decades where it becomes an ever-present necessity in humankind's daily activities (e.g., education, banking, and communication activities). In fact, in the digital era of today, software plays an important role in safety-critical systems such as air traffic control systems, medical-supporting systems, and nuclear control systems. Such systems require the development of very complex software due to the complexity of their demands. However, software development is not always guaranteed to deliver high-quality software as errors may occur throughout the development process, and in the case of critical systems, this can lead to extremely serious consequences. For example, it has been reported that a bug in the software that controls a radiation therapy machine led to increasing in the quantities of beta radiation that caused the death of more than five patients [99]. Another serious incident that was experienced is when a financial firm lost more than $400 million because of an error in using a software flag that caused undesired code to be executed [143].

Therefore, it is important to ensure that the developed software is high-quality and reliable software, and that can be achieved through *testing* the software. Software testing is a process that is not merely considered to find errors but also to ensure that the software conforms to the requirements. Testing is an important activity during the software development cycle that mainly aims to detect as many bugs as possible in the software. This, in fact, makes it the most expensive activity that consumes up to 50% of the software development resources [118], especially with the increase in software complexity. Furthermore, software testing tends to be a manual process that possibly leads to further development costs and human errors. Thus, automating the testing

process becomes more desirable as it is more cost-effective and reliable than manual testing.

The automated generation of test cases requires the generation of (i) *test data*, i.e., inputs to execute the software, and (ii) *test oracles*, i.e., assertions that are used to verify whether executing test inputs reveals software faults. For over the past decades, several approaches have emerged to automate the generation of test data such as *random* testing where software is exercised with randomly generated data, *symbolic execution* testing where software is tested by executing as many program paths as possible with a set of concrete inputs, and *search-based* testing where metaheuristic search algorithms are applied to generate adequate test inputs. A well-known search algorithm is the Genetic Algorithm (GA) that mimics the process of natural selection and reproduction to generate test inputs with the aid of an objective function.

It has been demonstrated that the search-based approach is often effective in generating test inputs that satisfy testing goals, especially when considering a GA [109]. In the context of unit testing object-oriented software, where tests are sequences of calls on a class under test, GAs have been successfully applied for generating tests. Several studies [62, 128] have shown that GAs are effective at generating tests that achieve high code coverage. When the test goal is based on branch coverage, an effective GA is the one that generates tests that cover as many branches in the source code as possible, and therefore an optimal test is the one that covers *all* the branches.

1.1 MOTIVATION

Despite the success of GAs in generating tests that achieve high code coverage, they are still far from being able to satisfy all test goals (e.g., covering all branches) [149]. To illustrate that, consider the following example of a Java class along with its test case that is automatically generated using a GA.

JCLO is an open-source Java Command Line Option package [1] that parses command line options based on a given class object to assign values to the class's variables. It simply extracts variables from a class object using reflection and then gives values to these variables based on their names and types. The `parse` method (Listing 1.1) is an essential method in the main JCLO class that parses the command line options (i.e., arguments) by assigning the values in these options to the variables of a class. The options cannot be parsed if their format is not valid (i.e., a valid format can be "`--a=x`" or "`-a x`"). Assume a class with three fields; `int a`, `boolean b`, and `float c`, one possible command line would be "`--a=8 --b --c=5.2415`". Therefore, a valid test input that conforms to the format of command line options must be created to test the `parse` method.

1 https://github.com/drsjb80/JCLO

```
public class JCLO {
 ....
  public void parse(String[] args){
    // an example of args: --a=8 --b --c=5.2415
    ....
    for (int i = 0; i < args.length; i++){
      String key = getKey(args[i]); // e.g., a, b, and c
      Field field = getField(key); // e.g. boolean b
      if (field == null){
        throw new IllegalArgumentException("No such option:"
            +key);
      }
      Class type = field.getType(); // e.g. boolean
      String name = type.getName(); // e.g. boolean
      ....
      String value = null;
      if (name.equals("boolean")) {
        value = getBooleanValue(args[i]);
      }
    }
  }
 ....
  private String getBooleanValue(String arg){
    if (hasEquals){
      arg = arg.replaceFirst("[^=]*=", "");
      if ((arg.equalsIgnoreCase("true")) ||
          (arg.equalsIgnoreCase("yes"))) {
        return "true";
      }
      return "false";
    }
    return "true";
  }
}
```

Listing 1.1: `parse` and `getBooleanValue` methods in the `JCLO` class

To automate the generation of test inputs, a test generation tool can be used. Among the popular tools that generate tests for Java programs using a GA is EvoSuite [59]. It generates JUnit test suites for a given Java class under test (CUT) and target coverage criterion (e.g., branch coverage) using different GAs, with the Many-Objective Sorting Algorithm (MOSA) being the most effective algorithm for JUnit test generation [34, 128]. Running EvoSuite on a CUT usually results in a test suite of test cases that are best in covering the branches of the CUT. When running the JCLO class, many test cases are generated, and one test case that achieves the highest branch coverage is shown in Listing 1.2.

```
@Test(timeout = 4000)
public void test05() throws Throwable {
  JCLO jCLO0 = new JCLO("−vNv@%MHagJX {5bEyg",
      "$tyv$y.FZPXB2>A");
  String[] stringArray0 = new String[4];
  stringArray0[0] = "−vNv@%MHagJX {5bEyg";
  try {
    jCLO0.parse(stringArray0);
    fail("Expecting exception: IllegalArgumentException");
  } catch(IllegalArgumentException e) {
    verifyException("edu.mscd.cs.jclo.JCLO", e);
  }
}
```

Listing 1.2: An automatically generated test case that covers 42% of the branches in the `JCLO` class

The test case creates an object with two options (i.e., arguments) where only one option is used to exercise the `parse` method. Although this input covers many branches within the `parse`, `getField`, and `getKey` methods, it still does not cover the other branches, especially those branches in the `getType`, `getName`, and `getBooleanValue` methods because of the thrown exception caused by this test input. In other words, this test input causes an exception to be thrown, and thus any code after that if-statement that throws an exception is not covered. In fact, there are different test inputs in other test cases that cover the rest of the branches in the `getKey` method, however, there is no test input in the whole test suite that executes the branches in the `getBooleanValue` method (Listing 1.1). While many runs of EvoSuite are made on this class, the GA is still unable to find test inputs that lead to the execution of branches in the `getBooleanValue` method.

While some general limitations are known (e.g., the challenges of generating complex parameter objects [149] and the flag problem [109]), there is still uncertainty about what makes it difficult for GAs to satisfy the desired goal (i.e., finding test inputs that achieve high code coverage). In general, GAs do not always behave similarly when they are applied to solve different optimisation problems. It has been argued [82] that the GA performance (i.e., the final branch coverage in our case) is not enough to give insights into how the GA behaves during the search and what makes difficult with certain problems. This is because each optimisation problem has features that influence the behaviour of a GA, and such influence cannot be understood by only looking at the GA performance. Therefore, there is a need for a deep understanding of the optimisation problem features and their influence on the search behaviour. Such an understanding can be provided by investigating the underlying structure of the search space and the influence of its features on the optimisation process.

The concept of *fitness landscape* is a common metaphor that is used to give an intuitive understanding of the search space structure and its influence on the search behaviour. The analysis of the fitness landscape helps in understanding how the problem features relate to the problem difficulty and how such features affect the GA performance. Hence, the fitness landscape analysis becomes an appropriate choice to decide which GA configurations are more suitable to solve a certain optimisation problem. However, previous studies identified several fitness landscape features that influence the search, and two main features that are known to have a great influence on the optimisation process are *ruggedness* and *neutrality* [104]. Ruggedness is a feature that depicts a landscape with many optima and less correlated neighbouring solutions while neutrality is a feature that refers to the amount of neutral areas (i.e., plateaus) in the landscape. In fact, an increase in the ruggedness makes the search for an optimal solution harder as the algorithm might get trapped in local optima and result in sub-optimal solutions. Therefore, in order to ensure that the search space is well explored and to avoid stagnation in local optimum, the individuals of the search population must be *diverse* enough (i.e., not similar individuals).

The loss of population diversity is a well-known issue that occurs during the GAs search. If the individuals of the search population all become very similar and lack diversity, then the search may prematurely converge on a local optimum of the objective function or even a non-optimal point, especially with a rugged landscape. This reduces the effectiveness of the GA, and in the case of search-based test generation, premature convergence would imply a reduced code coverage. Therefore, it is important to maintain diversity in the population during the evolution to avoid such premature convergence and ensure that the search space is well explored [44].

Also, population diversity is strongly related to the size of individuals, and an important issue that has been identified during the evolution of GAs is known as the *bloat* problem [25]. Bloat is a phenomenon that denotes the rapid growth in the size of individuals when evolution progresses with no considerable effect on the fitness, which occurs when a GA considers an individual representation of variable length. In the domain of unit test generation, bloat means more unnecessary statements are added to test cases or even unnecessary test cases are added to an individual test suite. This has a negative impact on the search as evolving longer individuals leads to negative consequences, for example, slowing down the search progress as longer individuals need more evaluation time and thus result in a few generations. Although this problem has been investigated with the problem of unit test generation, it is important to investigate whether the bloat problem has an effect on the fitness landscape features and the level of population diversity.

1.2 THESIS AIM AND OBJECTIVES

The aim of this thesis is to investigate the search behaviour when GAs are applied to generate object-oriented unit tests and, more specifically, identify the reasons of why the search fails to achieve the desired test goals (i.e., finding test inputs that achieve high branch coverage). Therefore, this thesis provides enhanced knowledge and understanding of how the three issues mentioned previously affect the generation of potential unit tests, and this can be fulfilled by the following objectives:

1. To perform an in-depth analysis of the fitness landscape structure and investigate the impact of its two features (i.e., ruggedness and neutrality) on the generation of object-oriented unit tests.
2. To identify the underlying properties of Java source code that influence the features of the fitness landscape.
3. To empirically investigate the impact of population diversity on the generation of unit tests.
4. To empirically evaluate the impact of mitigating the bloat problem (i.e., test case reduction) on fitness landscape features and the level of population diversity, and thus the performance of GAs.

1.3 THESIS STRUCTURE AND CONTRIBUTIONS

This section presents the structure of this thesis along with the contributions of this research.

CHAPTER 2: "LITERATURE REVIEW" presents a literature survey of the research topics studied in this thesis. It begins with introducing the fundamental terminologies of software testing and, more specifically, unit testing. Then, the automation of test data generation problem is presented and its existing approaches are reviewed. In particular, the Single-Objective and Multi-Objective GAs to generate test data for object-oriented program are thoroughly reviewed. The chapter then focuses on the challenges and limitations of the use of GAs from a practical point of view (e.g., the problem of complex parameter objects generation), and from a theoretical point of view as well (e.g., the features of the fitness landscape and the population diversity problem).

CHAPTER 3: "CAUSES AND EFFECTS OF FITNESS LANDSCAPES IN UNIT TEST GENERATION" investigates the fitness landscape structure and the impact of its two features (ruggedness and neutrality) on the generation of unit tests. As an initial step towards understanding

the influence of the fitness landscape on the generation of unit tests, we first analyse the two landscape features when considering each of the two well-known test generation approaches in our domain: the archive-based Whole Suite approach (WSA) and the Many-Objective Sorting Algorithm (MOSA). Then, we investigate how the two landscape features affect the search performance with the two approaches (i.e., how ruggedness and neutrality affect the branch coverage). The results of this analysis based on the two approaches (WSA and MOSA) are statically compared in order to better understand whether an individual representation influences the landscape structure. Moreover, our study investigates the factors that cause the fitness landscape properties by analysing the underlying properties of the Java source code. Therefore, the contributions of this chapter are as follows:

(1) A detailed analysis of the fitness landscape structure and its two features (ruggedness and neutrality) with the WSA and MOSA approaches.

(2) A detailed analysis of the impact of the landscape features on the performance of GAs (WSA and MOSA).

(3) An evaluation of how an individual representation influences the features of the fitness landscape when comparing the influence on the two GAs (WSA and MOSA).

(4) An in-depth analysis of what aspects of the underlying Java source code that affect the landscape features.

CHAPTER 4: "MEASURING AND MAINTAINING POPULATION DIVERSITY IN UNIT TEST GENERATION" empirically investigates the impact of population diversity on the generation of object-oriented unit tests, and see whether maintaining sufficient population diversity level during the evolution improves the generation of tests that achieve high branch coverage. We first adapt common diversity measurements based on phenotypic and genotypic representation to the search space of unit test cases. Measuring diversity is an essential step that helps in understanding its effect on the search behaviour. In order to promote population diversity, we apply well-known diversity maintenance techniques and analyse their influence on population diversity and GAs performance. Furthermore, we adapt the idea of *adaptive* diversity that works by applying a diversity maintenance technique only when the diversity level drops below a certain threshold, and evaluate its effectiveness against the naive diversity approach and default GAs. Similar to the previous chapter, we investigate and compare the impact of population diversity when generating unit tests using the two approaches WSA and MOSA to understand how diversity influences the performance of each algorithm, and see whether the individual

representation affects the diversity during evolution. Therefore, the contributions of this chapter are as follows:

(5) Adapting three diversity measurements based on phenotypic and genotypic representation to the search space of unit tests.

(6) An analysis of how default GAs (WSA and MOSA) maintain population diversity during evolution.

(7) A study into the impact of diversity maintenance techniques on the diversity level and GAs performance.

(8) An empirical evaluation of the effect of adaptive diversity on the maintained diversity level and GAs performance.

(9) An evaluation of how an individual representation influences the population diversity when comparing the influence on the two GAs (WSA and MOSA).

CHAPTER 5: "AN ANALYSIS OF THE EFFECTS OF TEST CASE REDUCTION ON UNIT TEST GENERATION" presents a study that investigates the bloat problem and, more specifically, whether removing redundant statements in unit test cases influences the landscape features, population diversity level, and GAs performance. First, we present the *test case reduction* approach and how it works. Then, we conduct an empirical study that investigates the impact of the reduction approach on the landscape structure where the results of this study are compared to the results of the landscape analysis conducted in Chapter 3. Similarly, we investigate how the reduction approach affects the population diversity level during the evolution and compare if the effect on diversity differs from what is reported in Chapter 4. Finally, we compare the performance of default GAs (WSA and MOSA) to their performance when applying the reduction approach (i.e., does the reduction approach affect the branch coverage?). Therefore, the contributions of this chapter are as follows:

(10) An approach named *test case reduction* that is adapted to object-oriented unit tests.

(11) An analysis of how the test case reduction approach affects the landscape features with the WSA and MOSA approaches.

(12) An analysis of how the test case reduction approach affects the level of population diversity during the evolution with both GAs.

(13) An empirical evaluation of the effectiveness of WSA and MOSA when considering the test case reduction approach.

CHAPTER 6: "CONCLUSION AND FUTURE WORK" concludes the conducted research in this thesis and summarises the findings presented in each chapter. Then, we present possible ideas that extend the existing work of this thesis as future research work.

2 LITERATURE REVIEW

This chapter explores the literature and research relating to the concepts considered in this thesis. We first introduce the concepts and definitions related to the software testing, and discuss why the automation of testing task is needed. Then, we review the previous studies that address the challenges imposed by using Evolutionary Algorithms to solve optimisation problems, more specifically the population diversity loss and the impact of fitness landscape properties.

2.1 SOFTWARE TESTING

During the software development lifecycle, faults occur and defects are inevitably introduced. Testing a software system is an important process to be carried out to detect and remove these faults, and thus to ensure a successful software development. Software testing involves a wide spectrum of tasks that are performed along the development lifecycle, starting from testing individual components of the source code to evaluating the system's compliance with the initial requirements. In the simplest term, finding defects in software requires the creation of different tests that are intended to thoroughly exercise the software with different test inputs. This means that the quality of the created tests substantially affects the fault-revealing ability of software testing. To better understand how tests are created and executed, the following section describes the generic concepts that are related to software testing.

2.1.1 *Related Concepts*

An essential step towards testing software is to create a set of test cases, usually called *test suite*. A *test case* is simply a function that evaluates the behaviour of one or more functions in the software under test using *test inputs* that invoke the function(s) and *test oracles* that indicate whether the expected behaviour is met. A test oracle is often an executable *assertion* that determines the correctness of the test result. The execution of a test case leads to two possible states that are (i) the execution result meets the expected outcome, or (ii) both are not met. The latter case reveals a *defect*, also known as a *bug* or a *fault*, in the software [161]. The concept of defect refers to a design fault or an incorrect implementation that is revealed when a test case throws a *failure* or an *error*. A failure is the observable behaviour of the software that contradicts the expected behaviour (i.e., behaviour contrary to the software specification), which can be observed by a test oracle.

```java
public class Math{
  public int multiply(int x, int y) {
    int result = x + y; // bug in the operator, should be x * y
    return result;
  }
}
```

Listing 2.1: A simple Java method that multiplies two integers

```java
class MathTest{
  @Test()
  public void testMultiply() {
    Math math_0 = new Math();
    assertEquals(6, math_0.multiply(3, 2));
  }
}
```

Listing 2.2: A JUnit test case that tests the `Multiply` method

Simply, a failure is caused because of the software not being able to perform the required functions within the specified requirements due to the occurrence of a defect. An error is a deviation from the actual and the expected behaviour within the software boundary such as reaching an unexpected system state or reporting syntax error during runtime, which is often difficult to be revealed by a simple test oracle (i.e., assertions).

To illustrate the aforementioned concepts, consider the `Multiply` method shown in Listing 2.1 that is supposed to multiply two integers but contains a bug. One possible unit test case to test the `Multiply` method is shown in Listing 2.2 where the test inputs are 3 and 2 that are used to test the correctness of multiplication conducted by the method. The `assertEquals` is used to test the equality of two objects where, in our case, two integers are compared. The order of the parameters within the assertion is the expected value followed by the actual value. In our scenario, the expected value is 6 whereas the actual value is the one that is returned by the `Multiply` method such that A call to `Multiply(3, 2)` returns 5, which is not the expected value. The result of this call represents a *failure* that is due to the *defect* at line 3 in Listing 2.1 caused by the incorrect mathematical sign.

Testing techniques can be divided into two different approaches [118]: black-box approach and white-box approach. The black-box approach, also known as functional testing, is a method of testing the software and its functionalities based on the requirements specifications. This type of testing is independent of the internal structure of the software. Testers, in this case, do not need to be aware of the internal implementation to derive the test cases. On the other hand, the white-box approach, also known as structural testing, is a testing technique that

examines the internal structure and the implementation of software and derives test cases based on the code structure. However, it is obvious that the aim of black-box testing is to test the software behaviour and ensure that the final product of the software meets the users' requirements whereas the white-box testing aims to find the implementation failures and to ensure the implemented source code works as expected.

An important aspect of software testing involves deciding whether enough testing has been performed. This raises questions such as: Are we generating enough tests that ensure the software has been tested thoroughly enough? Are the generated inputs able to reveal the presence of faults, especially with a large input space? Answering such questions has been research interest in recent years, and many techniques have been developed to measure the quality of created test cases [183]. The next section presents a review of these techniques.

2.1.2 *Test Adequacy*

When testing software, there must be some criteria that ensure the generated tests are sufficient enough to terminate the testing process with a level of confidence. In practice, the adequacy criterion focuses on specific features in the software under test that should be exercised by the generated tests. For example, when the focus of the adequacy criterion is to exercise a structural property (e.g., statement or branch), then there must be at least a test case that executes each property to consider the whole test suite as an adequate suite. Several studies have identified and studied different criteria to assess the adequacy of tests [183]. The two most widely-studied test adequacy criteria are *coverage analysis* and *mutation analysis*.

2.1.2.1 *Coverage Analysis*

Coverage analysis is a white box testing technique that measures the effectiveness of a test suite based on the coverage of a specific structural criterion [118]. A structural criterion represents one structural aspect of the source code under test such as a statement or a branch that should be covered by at least one test case. In this case, the ratio of covered aspects to the total number of aspects in the source code represents the coverage value where a high coverage value indicates more of the source code has been executed by the test suite.

One basic and simple structural criterion is the *statement coverage* that requires all the statements in the source code are executed at least once. This coverage criterion is satisfied when all the statements in the source code have been executed by the test suite. The statement coverage is very useful with sequential statements but less useful with decision statements. To explain that, consider the function shown in Listing 2.3 where two possible test cases are needed to achieve 100%

statement coverage, which are ⟨`a=true, b=false, and c=true`⟩ and ⟨`a=false, b=false and c=true`⟩. While these two test cases execute every statement in the source code, the false condition of the second if statement is not yet tested. In this scenario where conditional expressions exist in the source code, the *branch coverage* becomes an important criterion to consider.

```
public boolean foo(boolean a, boolean b, boolean c) {
  boolean result = false;
    if(a || b) {
      result= true;
    } else {
      if(c) {
        result= true;
      }
    }
  return result;
}
```

Listing 2.3: A motivation example to illustrate the concept of different coverage criteria

The branch coverage criterion (also known as decision coverage) aims to ensure that each branch of each control structure is executed at least once by a test case, and thus all reachable source code is tested. In this case, at least one more test case is needed to achieve full branch coverage in the example shown above, which is ⟨`a=false, b=false, c=false`⟩ to execute the false branch of the second if statement that is not executed by the statement coverage.

To improve upon branch coverage, *condition coverage* is used to test sub-expressions in conditional statements where logical operands are considered. This coverage criterion ensures that each condition in a conditional statement *must* be executed at least once to reach full coverage. Applying this criterion to the example shown above requires one more test case to achieve 100% condition coverage, which is ⟨`a=false, b=true`⟩ to execute the other condition in the first if statement.

2.1.2.2 *Mutation Analysis*

As the purpose of software testing is to detect faults in the software under test, it is important to measure how well the test suites are in detecting faults. One popular fault-based technique is Mutation analysis [86] that works by seeding artificial faults into the program under test and check if any test case within the test suite is capable of detecting these faults. The version of the program that contains the faults is called *mutant*. When the output of running test cases with the original version and the mutant version are different, then the mutant is killed, and hence the test cases are powerful in detecting faults.

Otherwise, the mutant is alive, and test cases that kill the mutant must be created. In this case, the mutation score is defined as the percentage of killed mutants to the total number of mutants.

Previous studies show that mutation analysis is very effective in predicting the quality of test suites and more precisely detecting actual software faults [10, 11]. There is empirical evidence that test suites that reveal the artificially-seeded faults can reveal more real faults, which indicates that mutation analysis can be used to evaluate the fault revealing ability of test suites [87]. However, mutation analysis is computationally expensive to perform and very complex to implement; it still leads to high measurement overhead even when it is considered with a small program that possibly might have thousands of mutatns [66]. Researchers therefore most often consider code coverage criteria to evaluate the effectiveness of test suites.

Although coverage criteria evaluate how well test suites are in satisfying testing requirements (e.g., covering branches), it is still important to look at how effective these coverage criteria are in terms of fault detection (i.e., how well they predict the real-fault detection of test suites). This, in fact, motivated several studies to investigate the relationship between code coverage and fault-revelation and analyse the effectiveness of different coverage criteria in estimating mutation scores of generated test suites [36].

Gligoric et al. [66, 67] evaluated the effectiveness of branch and statement coverage in predicting mutation scores. In their study, they considered mutation testing to generate artificial mutants and evaluated the generated test suites for Java and C programs in terms of code coverage and how many mutants are killed. Their results suggest that measuring test coverage (i) has low runtime overhead when compared to mutation testing and (ii) performs well in fault revelation as there is a correlation between coverage and mutation scores (i.e., high coverage correlates to high mutation scores), especially when considering branch coverage. Similarly, Gopinath et al. [70] investigated whether statement, block, branch, and path coverage leads to better estimation of fault detection. Their evaluation of the generated test suites of Java programs reveals that there is a significant increase in fault revelation once a high level of coverage is attained, and more specifically with statement coverage. The conclusion drawn from these studies (i.e., coverage criteria perform well in evaluating the quality of generated test suites) is confirmed by the findings reveals in other studies [36, 58, 87].

2.2 AUTOMATION OF TEST GENERATION

Writing sufficient tests becomes laborious and more difficult with complex software, and more error-prone task when written manually. This, in fact, makes the testing task frequently accounts for 50% of the total cost of software development [118]. To reduce the manual efforts, automating the process of test generation becomes a necessity to ensure the accuracy and the quality of the software under test [53]. However, automating the generation of test cases usually raises two main issues that must be addressed. The first is the test data generation problem (which test inputs cause the software execution to reveal faults?), and on the other hand, the test oracle problem (does the generated test inputs result in a behaviour that is similar to the expected behaviour?). As both issues make the test automation difficult, each has been research interest in the past years [24, 41], and they still impose a challenge to the automated test generation that is not yet solved.

For over the past decades, several approaches have emerged to automate the generation of test data [53, 113]. In general, these approaches fall broadly into three groups: (i) random approaches that rely on the source code to generate random tests, (ii) static approaches that generate tests based on the decision paths in the software under test with no need to the source code, and (iii) dynamic approaches where the software under test needs to be executed to determine how close the generated tests are to satisfy testing requirements.

The rest of this section provides an overview of the most common approaches for the automated test data generation that belong to the aforementioned groups. These are random testing (random approach), symbolic execution-based testing (static approach), and search-based software testing (dynamic approach). Then, the test oracle problem is presented and discussed.

2.2.1 *Random Testing*

Random testing is the simplest and best known test generation approach [18]. Testing software using this approach is simply performed by randomly sampling test inputs from the input domain, i.e., the set of all possible inputs to the software under test, and then using these inputs to execute the software and observe its output. Random testing is often easy to implement and has been demonstrated to detect faults at low cost [17], which makes it the practical choice with incomplete specifications or unexpected security problems as with fuzz testing [69]. This, in fact, makes it often used as a benchmark to be compared with other test generation approaches, as in [150].

However, the random testing approach generates test inputs without following any pre-established testing guidelines, as selecting such

inputs from the input domain is based on a probability. For example, there is a low probability of selecting two equal random inputs from a considerably large input domain to test the equality of two variables such as `if (x == y)`. Therefore, this approach is considered to be sufficient with small software, but not with more complex software.

In the context of object-oriented unit testing where a test case is a sequence of method calls to the class under test, random test generation has been considered by Csallner and Smaragdakis [45] where they proposed the JCrasher tool that generates random unit tests for Java classes. It works by creating random sequences of method calls and then reporting violated method calls (i.e., calls that only throw specific types of exceptions because of faults in the class under test).

Due to the limitations of the random testing approach, Pacheco et al. [123] proposed further improvements over the pure random approach by using a feedback-directed approach, which is implemented in a tool called Randoop [122] to generate unit tests for Java classes. The idea behind this approach is to use the feedback obtained by executing previously generated test inputs to guide the generation of new random inputs to avoid redundant and invalid test inputs. In the case of object-oriented unit testing, method calls are incrementally generated where each method call is selected randomly. The sequence of calls is then executed to provide feedback to (i) avoid the generation of inputs that, e.g., result in illegal behaviour or (ii) generate assertions that validate future changes. To improve upon feedback-directed approach applied in Randoop, the Guided random testing (GRT) [101] extends the feedback-directed approach with static and dynamic analysis performed on the class under test to provide further guidance to the random generation.

Sampling random inputs from a large input domain might result in inputs that are in contiguous areas, which could not be enough to detect faults in the software under test. Therefore, Chen et al. [37] proposed a more systematic random testing approach known as *Adaptive Random Testing* (ART) that guides the generation towards more distributed inputs (i.e., numerical inputs) within the input domain. To achieve that, the test inputs that are far from the previously generated inputs are preferred to be selected to increase the likelihood of detecting more faults. ARTOO is an approach that applies the ART on object-oriented programs [38] where distance is calculated between object inputs rather than numerical inputs. The calculation of the objects distance is based on their direct values, types, and distance to other objects. Despite the success of ART approach over pure random search in generating potential tests, Arcuri and Briand [15] showed that ART imposes high computational cost because of the distance calculation among test inputs, which in practice makes this approach less attractive.

2.2.2 *Symbolic Execution based Testing*

Symbolic Execution (SE) is one popular test generation approach that tests a software program with symbolic values instead of concrete data [89]. The program is presented as an execution tree that consists of different paths where each path, i.e., a conditional branch in the program, is a path constraint to be satisfied. To illustrate that, consider the example shown in Listing 2.4.

```
public boolean compare(int x){
  int y = x / 2;
  if(y == 10)
    return true;
  else
    return false;
}
```

Listing 2.4: A simple method that compares two integers

When using *concrete* data to test the `compare` method, it would read a concrete input value (e.g., 8) and assign it to x. The exeuction, in this case, results in $y = 4$ which makes the conditional branch (line 3) evaluates to false. However, when considering SE, the `compare` method is called with a *symbolic* value (e.g., δ). In this case, the expression δ / 2 is assigned to x, and the condition to be evaluated (line 3) results in two path constraints that are δ / 2 == 10 for the true branch and δ / 2 $\neq$ 10 for the false branch. The two path constraints are then solved by generating two concrete values for δ using a constraint solver such as Z3 [47]. For that, two possible values are $\delta = 20$ to solve the first path constraint and $\delta = 14$ to solve the other constraint.

Despite the success of the SE approach in generating tests that are capable of finding software bugs [134] and improving code coverage [32], it still suffers from several limitations that make it unsuitable in practical scenarios [33]. SE simply executes all the feasible paths of a simple program under test, but that does not scale to large programs. This results in an issue known as *Path Explosion* where the number of paths grows exponentially with an increase of program complexity, and the SE approach, in this case, becomes unable to execute all the feasible paths. The growth in the number of paths can be a result of the loops in the program under test where unbounded loop iterations might possibly make the number of paths infinite. To address this issue, several attempts have been made such as (i) detecting and pruning redundant paths (i.e., paths that produce similar results to the previously explored paths) [26], and (ii) parallelizing the execution of independent paths [158].

Besides the aforementioned issue, the use of a constraint solver makes this approach inefficient as some complex constraints are difficult to satisfy by a constraint solver. Also, symbolic values of external

codes (e.g., third-party libraries) are difficult to execute [9]. Therefore, to improve upon the SE approach, Dynamic Symbolic Execution (DSE) has been proposed [68]. The idea behind this approach is to combine the symbolic execution with the concrete execution where the program under test is executed with symbolic values until these values become difficult to be solved by the constraint solver, which are replaced with concrete values. This approach has been successfully applied to generate tests that achieve remarkable code coverage [31, 181].

The SE and its dynamic version (DSE) approach received attention in the domain of object-oriented unit testing [160, 162, 180]. An example is the *Symstra* approach [180] that generates unit tests using the SE approach where method calls are generated using symbolic values. Another example is the *MSeqGen* approach [160] that relies on both random testing and DSE to generate unit tests.

Although the approaches that rely on the SE and its dynamic version (DSE) are effective in generating potential tests, they face some challenges, especially in the case of object-oriented programs. For example, these approaches may not scale to programs where there is an interaction with public APIs as the state of the program needs to be modified by calls to an API rather than directly setting test inputs.

2.2.3 *Search-Based Software Testing*

In 2001, Harman and Jones [75] introduced the term *Search-Based Software Engineering* that describes the application of search-based optimisation approaches to solve different problems related to the activities of software engineering, which are found to be successful in solving such problems [39, 40], especially the problem of test generation [77]. Applying optimisation approaches to generate software tests, which has become known as *Search-Based Software Testing* (SBST), has received great attention in recent decades [110]. Miller and Spooner [116] were the first to apply an optimisation approach to generate test data where they replaced the Symbolic Execution approach with a numerical maximisation approach to generate floating-point inputs that cover manually-specified program paths. A few years later, Korel [94] extended the approach of Miller and Spooner where random test inputs are used to execute the program under test. When the execution diverges from the desired path, a local search is applied to generate test inputs using a computed branch distance from that desired path. Since then there have been numerous studies published in this direction [109].

2.2.3.1 *Metaheuristic Search Algorithms*

Metaheuristic search algorithms are a set of high-level approaches that apply a heuristic in order to find sufficiently better solutions to a given optimisation problem. They are problem-independent approaches which can be easily adapted to a given problem by changing configurations such as how a solution is encoded. In this case, the search explores a sample of similarly encoded solutions (i.e., search space) to find an optimal solution. However, to efficiently explore the search space, each solution is evaluated using a problem-specific objective function, called *fitness function*. The fitness function evaluates the performance of each candidate solution with regard to the current optimum where better solutions are rewarded better fitness value and vice versa. The selection of individuals for reproduction is based on their assigned fitness values. Therefore, the search prefers fitter solutions that are close to achieving the search goal.

There are many metaheuristic algorithms that have been most widely applied for solving optimization problems, more specifically the test generation problem [109]. They are classified based on the type of search strategy they apply. *Hill Climbing* and *Simulated Annealing* apply a local search (i.e., searching through a local neighbourhood of candidate solutions), whereas *Genetic Algorithms* apply a global search (i.e., searching for a globally optimal solution in the whole population). These three algorithms are reviewed in the following sections.

2.2.3.2 *Hill Climbing*

Hill Climbing algorithm is a well-known local search algorithm that iteratively improves a single solution during the search [76]. The search starts with an arbitrary solution chosen from the search space, and then its neighbourhood is investigated. If one neighbour has better fitness, the search replaces the current solution with that neighbour. In this case, each solution is replaced with its better neighbour until no further better neighbours are found, and that last better solution is considered as the local optimum.

The movement from one solution to another can be based on different strategies [109]. For example, the *steepest ascent* strategy applies the fitness evaluation to all the neighbours and the neighbour with the best fitness is chosen (i.e., if bette than the current solution). Another strategy is the *first ascent* where only one random neighbour is evaluated and chosen if it has better fitness.

However, due to the nature of the search landscape where multiple peaks might exist, the search may end up with a sub-optimal solution. To illustrate that, Figure 2.1 shows an example of a search landscape with many peaks and troughs where Hill Climbing is applied. The search climbed a hill that results in a candidate solution that is locally optimal, which is not the best solution in the search space (i.e., the

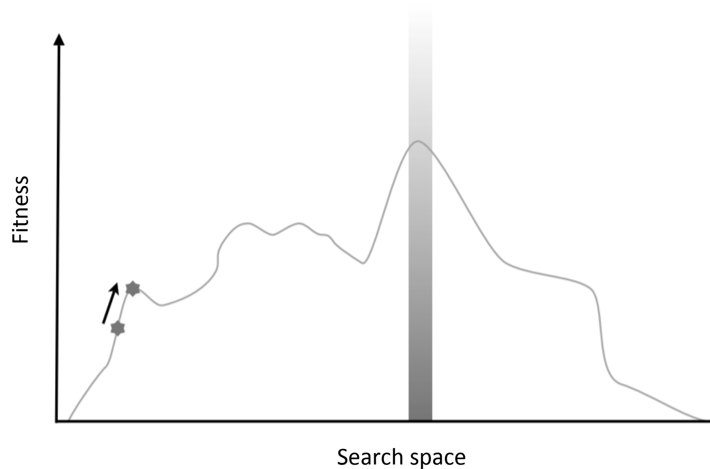


Figure 2.1: An example of Hill Climbing landscape [110]

one that is globally optimal). A possible way to overcome this issue is to restart the search from a different point in the search space to get a better knowledge of the search landscape, and thus increasing the possibility of getting better solutions.

2.2.3.3 *Simulated Annealing*

To reduce the dependency on the initial solution, the Simulated Annealing algorithm is proposed, which works similarly to Hill Climbing [91]. The main difference between the two algorithms is that Simulated Annealing accepts worse solutions during the search, as shown in Figure 2.2, which makes the movements in the search space less restricted. However, the acceptance of worse solutions is determined by a control parameter called *temperature* that depends on the difference in the fitness value between the current solution and the neighbour being considered.

Similar to the process of annealing in metallurgy, the search is started with a relatively high temperature value to allow for more exploration of the search space and avoid being trapped in a local optimum. Then, the temperature is gradually decreased as the search continues to progress. However, if the temperature is rapidly decreased, then the search becomes unable to explore more of the search space, and thus might get trapped in a local optimum as with Hill Climbing.

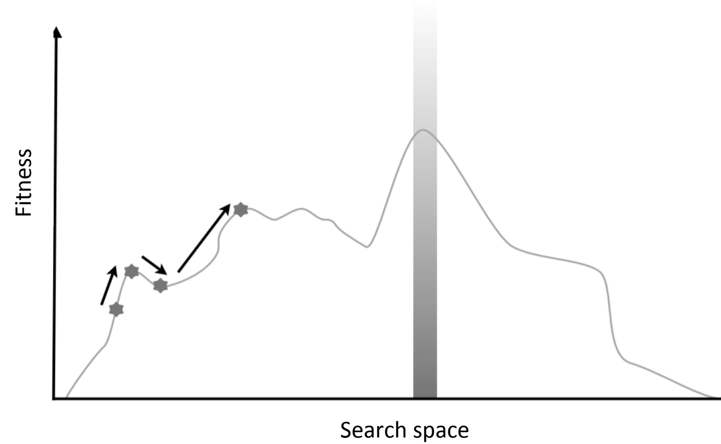


Figure 2.2: An example of Simulated Annealing landscape [110] with possible movements to worse solutions

2.2.3.4 *Single-Objective Genetic Algorithm*

Genetic Algorithms (GAs) are well-known type of Evolutionary Algorithms (EAs) that is inspired by Darwin's theory of natural evolution to solve an optimisation problem [81]. In a GA, a population of potential solutions is gradually evolved toward an optimal solution. The solutions are considered as *individuals* or *chromosomes*, analogous to the biological chromosomes. The algorithm typically starts with a population of random individuals that will be iteratively evolved over many generations.

On each generation, mimicking the process of natural selection, every individual in the population is evaluated by the fitness function, and only two individuals are selected as parents for reproduction. There are different selection processes, which all aim to select the best individuals to be parents in the next generation including (i) *roulette wheel selection* where the probability of an individual being selected for reproduction is proportionate to its fitness value. Individuals with low fitness will have a small probability to be selected. In general, a common problem with this selection algorithm is that an individual with a better fitness value will dominate the selection, and (ii) *rank selection* where the individuals of the population are ranked according to their fitness, and the probability of an individual being selected is calculated based on the rank, rather than the fitness value. This avoids that individuals with higher fitness values dominate selection (i.e., low selective pressure).

Then, the two genetic operators, crossover and mutation, are applied to the selected individuals to form the next generation. Crossover is the process of breeding two parents chromosomes to produce new offspring by combining parts of the two chromosomes, whereas the mutation is the process of randomly modifying different genes in both offspring based on some probability. Mutation is used to maintain genetic diversity within the population, and possibly reduce the similarity between the individuals from one generation to the next. However, the search terminates when either a maximum number of generations is reached, or another pre-defined stopping criterion is met.

2.2.3.5 *Multi-Objective Genetic Algorithm*

Many real-world optimisation problems, especially complex engineering problems, involve multiple objectives that often conflict with each other [93]. These problems are known as *multi-objective optimization problems*. Minimising cost while maximising performance is one example of a multi-objective problem with two objectives to be optimised. In practice, there could be a large number of objectives. With a nontrivial problem, optimising one objective might impair other objectives as there is no one solution that simultaneously satisfies all the objectives. Therefore, an ideal solution to this problem is to find a set of solutions that are the best trade-off solutions between the conflicting objectives, where this is determined by the *dominance* of the solutions in the search space.

To illustrate the dominance concept, a solution x is said to dominate another solution y (also written as $x \succ y$) if x is not worse than y in all the objectives, and x is better than y in at least one objective. In this case, the solution x is said to be *Pareto optimal* if it is not dominated by other solutions, and the set of all non-dominated solutions in the search space are called *Pareto optimal set*. The corresponding fitness values of the Pareto optimal set in the objective space are known as the *Pareto front*.

As GAs are well studied in solving optimisation problems with a single-objective, they also have been extensively applied to solve multi-objective optimisation problems [93, 184] where multi-objective GAs have been proposed. The main difference among different multi-objective GAs relies on how the fitness is assigned to the individuals on the search space. The Vector Evaluated GA (VEGA) [145] was the first multi-objective GA to approximate the Pareto optimal set. It works by randomly dividing the population into M subpopulations of equal size where M is the number of objectives to be optimised. Then, each individual in each subpopulation is assigned a fitness value based on the considered objective function. Once the fitness is assigned, all the individuals of the subpopulations are combined, and the genetic operators are applied on the combined population.

However, this approach is not promising to well explore the search space and thus find the desired Pareto optimal set, as the selected individuals are better in satisfying one objective but not others. In other words, VEGA does only find the extreme individuals of every single objective that makes it difficult to find the best trade-off among all objectives.

Fonseca and Fleming [57] proposed the first multi-objective GA that explicitly utilises the *Pareto ranking* in the fitness assignment, and named their algorithm Multi-Objective Genetic Algorithm (MOGA). In each generation, each individual in the population is assigned a rank according to its dominance, where the individual's rank corresponds to the number of individuals in the population by which it is dominated. In this case, the non-dominated individuals are always assigned a similar rank, whereas the dominated individuals are given ranks based on the surrounded population density. The fitness of each individual is then derived based on the assigned rank. Moreover, a niche formation method (i.e., Fitness sharing) is integrated into the algorithm to ensure enough population diversity is maintained, and thus the search is encouraged toward exploring the solutions in the Pareto-optimal region. However, it has been argued that this type of fitness assignment is more likely to result in a large selection pressure that might make the search converges faster [48].

Later, Srinivas and Deb [157] developed the Non-dominated Sorting Genetic Algorithm (NSGA) that also uses the concept of Pareto dominance for fitness evaluation. In this algorithm, the individuals are ranked based on their non-dominance into *fronts* where each front contains individuals that share the same non-domination level. To illustrate that, Figure 2.3 shows an example of how individuals are sorted into fronts where two objectives are optimised.

Individuals in the *front 1* are the non-dominated individuals in the current population, which are nominated to be the Pareto optimal solutions. For the individuals in the *front 2*, we say that these individuals dominate the other individuals in *front 3* and *front 4* but not those in *front 1*. In this case, the fitness is assigned per each front using the fitness sharing method where better fitness is given to individuals in higher fronts such that the fitness of individuals in front_i is better than of those individuals in front_{i+1}.

Despite the efficiency of NSGA in solving the multi-objective optimization problems, it has been noticed that NSGA is computationally expensive because of the complex computation needed for non-dominated sorting and finding the best sharing parameter to promote population diversity. To overcome these issues, a modified version of NSGA has been proposed, known as NSGA-II [49]. The NSGA-II improves the non-dominated sorting by reducing the computational requirements needed to sort the individuals. It also considers the elitism (i.e., copying the best individuals to the population of next gen-

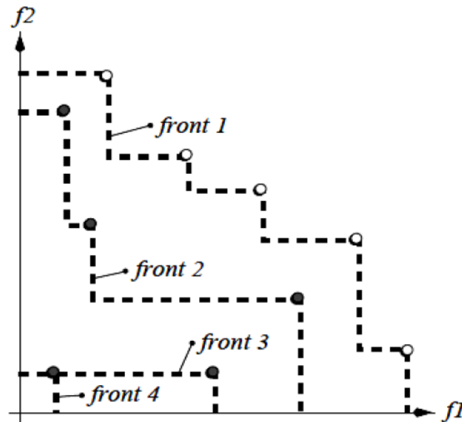


Figure 2.3: An example of non-dominant fronts in the NSGA [55]

eration) to improve the performance of the algorithm, and to reduce the probability of losing the better solutions that are found. Moreover, it does not require any user-defined parameters as it replaces the fitness sharing with a new mechanism called *crowding distance*. The crowding distance provides an estimate of the density of individuals that are in the boundaries of a specific individual. In this case, each individual is assigned a value that is based on how close it is to other individuals in the same front. However, the selection in NSGA-II does not only depend on the fitness, but it also depends on the rank and the crowding distance. Both criteria are used in the tournament selection for reproduction. Therefore, the more desirable individual is the one that has a higher rank and higher crowding distance value.

Besides the aforementioned algorithms, different algorithms have been proposed to enhance the efficiency of the search and obtain a good approximation of an exponentially large Pareto front, especially with an increase in the number of objectives to be optimised. Examples of these algorithms include ε-dominance MOEA (ε-MOEA) [97] and Indicator Based Evolutionary Algorithm (IBEA) [185] that replaces the Pareto dominance with hypervolume indicator during the selection.

2.2.3.6 *Test Data Generation Using Genetic Algorithms*

In the domain of SBST, GAs have been successfully applied to generate test data [109]. An early work that considers GAs to generate test data is the approach proposed by Pargas et al. [129] that targets branch and statement coverage. In their work, the control dependent nodes

of each test goal (i.e., statement of a branch) are used to guide the search to cover the test goal. The objective function, in this case, is the number of executed control dependent nodes to cover a specific goal. However, the formulation of their objective function does not really provide guidance to the search to find test inputs that are closer to cover a specific test goal, as the number of executed control dependent nodes is not enough to tell how close an input is to reach the desired test goal. To overcome this issue, and to further improve the objective function, the branch distance is used [21], which estimates a distance for a given conditional statement to become true or false.

Miller et al. [115] presented a new approach to automatically generate test data using a GA and program dependence graphs. The idea behind this approach is to select each path that leads to the desired branch with the help of the program dependence graph, and then collect all the constraints in the selected path. The GA is then applied to generate test data that satisfy these constraints, and thus reach the branch in the selected path. The approach has been compared to the random testing and other approaches that only use GA, and showed that it can generate potential tests that achieve high branch coverage.

In the case of object-oriented unit-test generation, the study conducted by Tonella [164] introduced GAs to generate unit tests. In this work, an individual is represented as a test case that is a sequence of constructor and method invocations with input values and assertion statements. Each branch in the class under test (CUT) is targeted individually, and the GA generates test cases to cover the targeted branches using the fitness function that evaluates the distance between each test case and the targeted branch. Once the search terminates, all the generated test cases are combined into a single test suite. This approach may work well with small subjects, especially with feasible branches, but it would not scale to large subjects with infeasible goals. This is due to the fact that targeting a difficult or infeasible branch makes the search less effective as most of its effort is wasted on this type of branches. Also, the size of the resulting test suite may grow tremendously when the number of branches in the CUT increases, which is not desirable. The approach, however, is not fully automated as assertions are inserted manually.

WHOLE SUITE APPROACH

To overcome the limitations in Tonella's approach, Fraser and Arcuri [60] proposed the *Whole Suite approach* (WS) that optimises *all* the test goals simultaneously instead of considering one test goal at a time, and applies a single-objective GA (i.e., Monotonic GA) that evolves individuals of test suites. Their approach is implemented in their tool EvoSuite [59] that generates unit tests for Java programs.

The applied GA is described in Algorithm 1. As shown, the GA starts by generating random test suites as an initial population (Line 3).

Algorithm 1: The Monotonic GA applied in EvoSuite

1 **Input**: Population size n, Stopping criterion C, Crossover probability c_p, Mutation probability m_p
2 **Output**: A best individual test suite T
3 $P \leftarrow$ GenerateRandomPopulation(n)
4 **while** $\neg C$ **do**
5     $Z \leftarrow$ Elitism(P)
6   **while** $|Z| \neq |P|$ **do**
7         $p_1, p_2 \leftarrow$ RankSelection(P)
8         $o_1, o_2 \leftarrow$ Crossover(c_p, p_1, p_2)
9   Mutation(m_p, o_1)
10   Mutation(m_p, o_2)
11         $f_p =$ GetMinimumFitness(p_1, p_2)
12         $f_o =$ GetMinimumFitness(o_1, o_2)
13   **if** $f_o \leq f_p$ **then**
14             $Z \leftarrow Z \cup \{o_1, o_2\}$
15   **else**
16             $Z \leftarrow Z \cup \{p_1, p_2\}$
17   **end**
18   **end**
19     $P \leftarrow Z$
20 **end**
21 **return** T

Then, the evolution is performed over successive generations until a desired solution is found or the allocated search budget (e.g., search time or a number of generations) is consumed. In each iteration, the population of the new generation is initialised with the best individuals of the previous generation (Line 5). Following that, two parents are selected from the current population using the rank selection (Line 7) to generate new offspring using the two genetic operators. The crossover is applied to the two parents based on a given probability to generate two offspring (Line 8). Then, the new offspring are mutated according to some mutation probability (Line 9-10). After that, both parents and offspring are evaluated (Line 11-12) to decide which of the two will be added to the population of next generation (Line 13-17).

To better understand this algorithm, its four key elements are defined as follows:

Representation: A solution in the case of unit tests is represented as a test suite τ which is a set of test cases $(t_1, t_2, \ldots, t_n)$. Each test case $t_i = \langle s_1, s_2, \ldots, s_n \rangle$ is a sequence of calls s_j on the CUT. That is, each s_j is an invocation of a constructor of the CUT, a method call on an instance of the CUT, a call on a dependency class in order to generate dependency objects, or it defines a primitive value (e.g.,

number, string, etc.). As the ideal test suite size is not known a priori, the number of tests in a test suite and the number of statements in a test case is variable and can be changed by the search operators.

Fitness function: The fitness function used to guide the search is based on code coverage. Various different criteria as well as combinations of criteria have been previously proposed [139]. One of the most common coverage criteria in practice is the branch coverage [60]. The practical interpretation of this criterion is that, for each conditional statement in the source code, there has to be at least one test case where the condition evaluates to true, and at least one where it evaluates to false. For each condition, it is possible to estimate a distance to it becoming true or false; this *branch distance* [94] is the basis of many coverage-based fitness functions. For example, when the if-condition `if(x == 42)` is executed with `x` equal to 0, then the distance to the condition evaluating to true is $|42 - x| = 42$, whereas if x is 40, then the distance is $|42 - x| = 2$. If the condition evaluates to true, then the distance is 0. Similarly, the distance to the condition evaluating to false can be calculated. The overall fitness value of a test suite is the sum of normalised branch distance values for all the branches B in the CUT, so that a test suite with 100% branch coverage has a fitness value of 0. In this case, the fitness function for a test suite T and a set of branches B to minimise is:

$$f(T, B) = \sum_{b \in B} d(T, b) \tag{2.1}$$

where $d(T, b)$ is the distance between the test suite T and an individual branch b, which is defined as:

$$d(T, b) = \begin{cases} 0 & \text{if branch } b \text{ has been covered,} \\ v(d_{\min}(t \in T, b)) & \text{if the predicate has been} \\ & \text{executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{2.2}$$

where $v(x)$ is the normalization function and $d_{\min}(t \in T, b)$ is the minimum distance from a test case t to a brach b.

Crossover: The crossover operator in the context of test suite optimization works as follows: Given two parent test suites τ_1 and τ_2, a random value x in the range $(0, 1)$ is selected. The first offspring will contain the first $x \times |\tau_1|$ tests from τ_1, followed by the last $(1 - x) \times |\tau_2|$ tests from τ_2. On the other hand, the second offspring will contain the first $x \times |\tau_2|$ tests from τ_2, followed by the last $(1 - x) \times |\tau_1|$ tests from τ_1. As the size of individuals is variable, this operator ensures that offspring do not grow larger than their parents during the crossover.

Mutation: When a test suite τ is mutated, then with a certain probability new test cases are inserted, and with a certain probability the

existing test cases are modified. Each test in τ is mutated with probability $\frac{1}{|\tau|}$. When a test case is mutated, three operations are applied in order: remove, change, and insert statements. Each statement in a test t is deleted or edited with probability $\frac{1}{|t|}$, whereas insertion is applied repeatedly with a certain probability. A particular challenge lies in ensuring that these operations maintain the syntactic validity of the test cases. For example, inserting a method call requires all dependency objects to instantiated before the location of the method call. When dependency objects need to be created, this is typically done by recursively inserting calls that create and modify these objects.

One major issue when generating unit tests is the length of tests, as shown in Tonella's approach. When the search progresses, and after many generations, the length of the test cases becomes longer, which consumes more memory and execution time. Therefore, controlling the test length has been investigated and found to be useful in improving the search performance [61]. Based on that, EvoSuite incorporates the length of a test suite as a secondary objective such that a shorter test suite is preferred over a test suite with a similar fitness but longer length.

To improve upon the performance of the Whole Suite approach, Rojas et al. [140] found out that keeping an *archive* of those goals that are already covered along with their covering tests, and concentrating the search effort on reaching those uncovered goals leads to better coverage results. Their approach is known as *archive-based WS Appraoch* (WSA).

many-objective sorting algorithm

Recently, Panichella et al. [126] proposed a novel many-objective GA to generate JUnit tests, called Many-Objective Sorting Algorithm (MOSA) that targets each test goal (e.g., a branch) as an independent objective to be optimised. The motivation behind this approach is that (i) aggregating multiple target goals into a single objective is detrimental to the search efficiency, and (ii) considering a many-objective approach is more efficient than a single-objective approach when solving a complex problem [92] such as the branch coverage problem. As traditional many-objective algorithms are not scalable to a large number of objectives [126], MOSA considers a different selection scheme known as *preference criterion* that assigns a preference order to non-dominated test cases. The preference criterion simply identifies a subset of test cases that have the lowest fitness values for uncovered branches among all non-dominated test cases. In addition, MOSA considers a second population, called *archive*, that stores the best test cases that cover new uncovered branches, which are used to form the final test suite.

MOSA, as shown in Algorithm 2, starts with an initial population of randomly generated test cases (Line 4), and applies standard genetic operators (Line 7). To generate the next generation, parents and

Algorithm 2: Many-Objective Sorting Algorithm (MOSA)

```
1  Input: Population size n, Stopping criterion C
2  Output: An archive of best test cases T
3  t ← 0                                          ▷ current iteration
4  P_t ← GenerateRandomPopulation(n)
5  T ← GetArchive(P_t)
6  while ¬C do
7      P_o ← GenerateOffspring(P_t)
8      P_u ← P_t ∪ P_o
9      F ← PreferenceSorting(P_u)
10     r ← 0
11     P_{t+1} ← {}
12     while |P_{t+1}| + |F_r| ≤ n do
13         AssignCrowdingDistance(F_r)
14         P_{t+1} ← P_{t+1} ∪ F_r
15         r ← r + 1
16     end
17     CrowdingDistanceSort(F_r)
18     P_{t+1} ← P_{t+1} ∪ F_r                     ▷ size n − P_{t+1}
19     T ← UpdateArchive(T, P_t)
20     t ← t + 1
21 end
22 return T
```

offspring are combined (Line 8) and sorted using the preference criterion and non-dominance relation (Line 9). The test cases that are identified by the preference criterion (i.e., those that are the closest to cover uncovered branches) are assigned rank 0, while the traditional non-dominated sorting used by NSGA-II is applied to rank the remaining test cases into further fronts. Selection is then applied based on the assigned ranks starting at the first front, until reaching the population size n (Lines 12-16). When the number of selected test cases exceeds the population size n, the individuals of the current front F_r are sorted based on the crowding distance (Line 17) and only those individuals with higher distance are selected. At the end of each generation, MOSA updates an archive with test cases that cover uncovered branches with the lowest possible length (Line 19).

Unlike the whole suite approach, MOSA evolves individuals of test cases rather than test suites, and therefore the four key elements of MOSA are defined as follows:

Representation: A solution that is represented as a test case τ consists of a sequence of calls $\tau = \langle s_1, s_2, \ldots, s_n \rangle$ on the CUT. That is, each s_j is an invocation of a constructor of the CUT, a method call on an instance of the CUT, a call on a dependency class in order to generate or modify dependency objects, or it defines a primitive value (e.g.,

number, string, etc.). As the ideal test case size is not known a priori, the number of statements in a test case is variable and can be changed by the search operators.

Fitness function: In the many-objective representation of the unit test generation problem, each branch in the CUT is considered as a single objective to be optimised. In this case, the fitness function of a branch b_i is typically calculated as follows:

$$f(\tau, b_i) = al(b_i, \tau) \ + \ \alpha(bd(b_i, \tau)) \tag{2.3}$$

Here τ is an individual test case to be evaluated, bd is the branch distance, α is a normalisation function that normalises the branch distance in the range $[0, 1]$ [14], and al is the approach level [174]. The approach level is defined as the distance between the closest control dependency of the target node executed by a test and the target node in the control dependency graph. The branch distance for $f(\tau, b_i)$ is calculated for this control dependency. Therefore, the overall fitness of an individual τ is a vector of n fitness values, called fitness vector, such that:

$$f(\tau) \ = < f_1, f_2, ..., f_n > = \begin{Bmatrix} f(\tau, b_1) \\ \vdots \\ f(\tau, b_n) \end{Bmatrix} \tag{2.4}$$

where n is the number of branches in the CUT, and each value in the fitness vector corresponds to a fitness value f_i for a single branch b_i that is calculated using Equation 2.3. In this context, a test case x is said to be better than a test case y if and only if x has a lower approach level + branch distance for one or more branches, and not worse for the rest of the branches.

Crossover: The common crossover operator in the context of test case optimization works as follows: Given two parent test cases τ_1 and τ_2, a random value x in the range $(0, 1)$ is selected. The first offspring will contain the first $x \cdot |\tau_1|$ statements from τ_1, followed by the last $(1 - x) \cdot |\tau_2|$ statements from τ_2. The second offspring will contain the first $x \cdot |\tau_2|$ statements from τ_2, followed by the last $(1 - x) \cdot |\tau_1|$ statements from τ_1. As the size of individuals is not fixed, this operator ensures that offspring do not grow larger than their parents during crossover. Since there can be dependencies between statements within a test, the crossover possibly needs to repair the offspring to ensure validity, e.g., by generating additional statements for missing dependencies.

Mutation: When a test case τ is mutated, each statement in τ is deleted or edited with probability $\frac{1}{|\tau|}$, whereas insertion is applied at a random position with probability σ; if a statement is added, then another one is inserted with probability σ^2, then with σ^3, etc. A challenge lies in ensuring that these operations maintain the syntactic

validity of the statements, for example by recursively inserting calls that create and modify dependency objects.

However, there might be a case where some targets (i.e., branches) have a structural dependency on other targets such that the targets in the former case can only be reached if and only if the targets in the latter case are already satisfied. This case is not considered in MOSA, as it only treats all targets as independent objectives to be optimised. Therefore, the authors extended MOSA to dynamically select coverage targets based on their control dependency, and named their approach DynaMOSA [127]. The search, in the case of DynaMOSA, simply starts by only selecting independent targets that have no dependency on other targets. During the search, when any of these targets is covered, then its uncovered control dependent targets are considered to be optimised in the subsequent generations.

Another approach that uses a multi-objective GA to generate object-oriented unit test cases is *TestFul* that was proposed by Baresi *et al.* [22]. The approach maximises the statement and branch coverage of the CUT. It combines the GA with a hill climber to work at the class and method levels. Unlike other search-based approaches, TestFul considers the internal states of objects to explore the search space. In other words, TestFul generates tests by deriving and reusing useful state configurations and exploring these states to exercise the actual behaviour and reach uncovered branches in CUT. When TestFul was compared against other approaches that work on stateful systems such as jAutoTest (a Java version of AutoTest [112]), Randoop [122], and ETOC [164], TestFul was able to generate tests with higher branch coverage.

The principle of SBST is not only restricted to the test data generation for object-oriented software, but also can be applied to other types of test generation. For example, SBST is applied to automate the testing of Android applications as in [7, 103, 106]. Also, it can be applied to automatically generate tests for Graphical User Interfaces (GUIs) such as EXSYST appraoch [72] that generate system test for the GUIs of Java programs.

2.2.4 *Test Oracle Problem*

The purpose of generating test data is to detect faulty behaviours of software under test. However, exercising the software with the generated test data does not distinguish the correct behaviour from potentially incorrect software behaviour without the use of *test oracles*. Test oracles allow validating the functional correctness of software under test by comparing the output of applying the generated test data on the software with the expected output. In the case of object-oriented unit tests, test oracles are typically assertions that must be included in the generated test cases. However, the construction of test

oracles can be a challenging task to the automation of test generation because of the difficulty in determining the correct output for a given test input. To construct an oracle, there must be a reliable source (e.g., formal specifications) that can be used to derive the expected software behaviour which is compared to the actual behaviour when executing test inputs. When no source is provided, the oracle construction becomes difficult and thus human must be involved to construct manual oracles (i.e., check whether the observed behaviour is correct), which makes it a time-consuming process. This is known as the *test oracle problem* [24].

There have been several attempts to automate the generation of test oracles [24, 146]. Test oracles can simply be generated based on informal software specifications that describe how software should behave, which can be analysed based on natural language analysis [8, 73]. They can also be generated using formal specifications such as software environment constraints and functional properties that are expressed as logical expressions which must be satisfied by software [51]. Z language notations [52] and Abstract Machine Notations [1] are other examples of formal specifications that can be used to derive test oracles. However, the issue with this approach is that software specifications are not always guaranteed to be available with all software systems or they might be available but can be too abstract to be useful.

Another approach is to consider independent versions of the software under test where various implementations of the software under test are developed to apply the same functionalities [105]. One version is known as the gold version that is a trusted version that results in the correct software behaviour. These versions serve as test oracle where they all run with the generated test inputs to check whether the software under test results in similar behaviour to the other versions. If all versions result in similar behaviour to the gold version, then the behaviour of the software under test meets the expected behaviour. Otherwise, the software contains faults that lead to undesired behaviour. However, this approach is very expensive that requires extra efforts, especially with large and complicated software. To reduce the cost of this approach, the M-Model program testing approach [105] is proposed where only versions of specific functions to be tested are implemented instead of implementing the whole software.

Besides the previously mentioned approaches, there are other approaches that consider Artificial Intelligence techniques to derive test oracles based on simulating the correct software behaviour, for example, by using the Info Fuzzy Network that is developed to represent the functional requirements as a directed graph and determine the set of input variables that relevant to the outputs [96]. Although these approaches provide considerable improvements over the generation of test oracles, constructing complete and accurate test oracles that are capable of detecting faulty software behaviours is still a main

challenge that affects the overall test automation. However, it should be mentioned that the focus of this research thesis is only on the generation of test data.

2.2.5 *A Review of Existing Studies*

As of yet, several GAs that generate unit tests for object-oriented programs are introduced and discussed. There are several studies that have systematically investigated their effectiveness in generating potential unit tests. This was done by comparing their performance on a reasonably large number of CUTs with a high number of branches.

Table 2.1 summarises the recent studies that investigate the influence of the state-of-the-art search algorithms on the generation of object-oriented unit tests. To better understand their performance, the algorithms are compared based on the reported results of branch coverage they achieved using different stopping criteria and different corpus of Java projects. Overall, MOSA and its extended version (DynaMOSA) are superior to the other algorithms as they yield better branch coverage, whereas the non-guided random search seems to result in the lowest coverage although Shamshiri et al. [151] reported that random search is more effective than the GA with a certain type of branches.

Initially, Fraser and Arcuri [60] evaluated the Whole Suite (WS) approach against the traditional Single Branch Strategy (SBS) [164] using a large set of CUTs. The results of their evaluation show that the WS approach is able to improve the branch coverage with 6%, as the overall coverage with WS is 83% and SBS is 77%. Besides, the WS has been shown to be effective in (i) generating smaller test suites than the SBS approach, and (ii) reducing the effects of the infeasible branches on the final coverage.

Later, Fraser and Arcuri [62] conducted a larger experiment to confirm that the findings of their previous study can be generalised to other Java programs. To do that, they empirically evaluated EvoSuite based on the WS approach using a corpus of 110 open-source projects (i.e., 100 randomly selected and 10 most popular Java projects), called SF110 corpus, that consists of more than 23,000 Java classes, and more than 800,000 branches. The reason behind the selection of such a high number of classes is to effectively demonstrate the influence of different software artefacts on the experimental results. As a result, EvoSuite was found to be efficient with many CUTs, as the achieved branch coverage was in the range of 20% to 67% per project and average of 71% per class. However, the results revealed that software systems with environmental dependencies (e.g., connecting to networks or databases) have a negative impact on the branch coverage.

To further investigate the capability of the WS approach in generating potential unit tests, Rojas et al. [140] empirically studied the

Table 2.1: Studies that investigate the performance of the proposed GAs in generating Java unit test suites based on the achieved branch coverage. The algorithms are RS (Random Search), SBS (Single Branch Strategy), WS (Whole Suite) and its archive-based variant (WSA), MOSA, and DynaMOSA. The reported results are either the average branch coverage (shown with percentage) or the number of CUTs/targets (branches).

Study	Year	Stopping Criterion	# Projects	# Classes	# Branches	Reported Results	RS	SBS	WS	WSA	MOSA	DynaMOSA
Fraser and Arcuri [60]	2013	1,000,000 statements	19	3165	85,541	Average coverage (per project)	-	77%	83%	-	-	-
Fraser and Arcuri [62]	2014	Two minutes	110	23,886	808,056	Average coverage (per project)	-	-	20% (min) 67% (max)	-	-	-
Panichella et al. [126]	2015	1,000,000 statements	16	64	50 - 1213 (per CUT)	Number of CUTs a	-	-	9	-	42	-
Rojas et al. [140]	2016	Two minutes	110	100	2382	Number of targets	- -	255 -	1410 1149	- 530	- -	- -
				100	34050		-	-	4196	6288	-	-
Panichella et al. [127]	2017	Five minutes	117	346	61,553	Number of CUTs	-	-	-	10	64	93
Shamshiri et al. [151]	2018	Two minutes	110	975	26,258	Average coverage (per CUT)	68.76%	-	-	69.10%	-	-
Panichella et al. [128]	2018	Two minutes	110	180	34,949	Average coverage (per CUT)	-	21%	-	48%	50%	52%
Campos et al. [34]	2018	One minutes	117	346	61,553	Average coverage (per CUT)	73%	-	-	79%	82%	82%

influence of the WS approach on coverage goals such as branch coverage, line coverage, and weak mutation. More specifically, the aim of their study was to (i) characterise which coverage goals are easy to cover by either SBS approach or WS approach, and (ii) to analyse the performance of WS approach when focusing the search for uncovered goals only (i.e., considering an archive). Using a corpus of 100 Java classes, WS results in higher branch coverage (78%) than the SBS (63%), i.e., WS is better on 1410 branches while SBS is better with 255 branches. However, using a test archive (WSA) leads to better results when considering more complex CUTs as WSA is better with 6288 branches while WS is better with 4196 branches. For the other two goals, the WS approach is able to cover more lines and mutants than the SBS approach.

Besides the previously mentioned studies, Shamshiri et al. [150, 151] conducted a study that shed light on the effectiveness of the GA and the random search on the generation of unit test suites. The aim of this study is to investigate the performance of two EAs: the GA using the WSA approach and the Chemical Reaction Optimization (CRO) algorithm (i.e., a metaheuristic algorithm inspired by the process of the chemical reactions), and compare both algorithms to the random search. The authors studied two versions of random search; the default random search (Pure random), and random search that utilises seeds obtained from the CUT (Random+). The four algorithms were empirically evaluated using a subset of 975 classes including more than 26,000 branches. In general, the GA using WSA approach achieves slightly higher branch coverage (69%) than CRO (68.87%), and both algorithms result in better coverage than the two random search versions (68.76% with Pure random and 65% with Random+). For the cases where CRO is significantly better than Random+, the GA mostly outperforms the Random+, and in contrast, when CRO is worse than Random+, both CRO and GA achieve a similar coverage which indicates that both EAs have relatively similar performance. However, it has been noticed that the GA and Random+ achieve the same coverage level with a high number of CUTs (i.e., 648 CUTs), and also both CRO and Random+ result in a similar coverage with 630 CUTs. One possible reason behind that is such classes are mostly trivial and all algorithms can easily achieve full branch coverage. Furthermore, as part of their investigation, the authors analyse how different branch types affect the performance of the algorithms where two groups of branches are observed; gradient branches and plateau branches. The first group includes branches that provide guidance to the search through the branch distance such as comparing two integers, whereas the second group includes branches that do not guide the search to the desired input as not much distance information is given, for example, evaluating boolean predicates. Their findings confirm that the two EAs result in higher coverage than Random+ with the gradient branches, while

Random+ shows better performance with plateau branches than the GA and CRO.

In the study presented by Panichella et al. [126] where MOSA was proposed, the performance of WS was compared to the performance of MOSA in a similar manner to how WS was previously evaluated, except that only 64 Java classes are considered. Their experiment shows that MOSA increases the branch coverage with 42 CUTs, while WS increases the coverage with 9 CUTs. For the remaining 13 classes, the two algorithms result in relatively similar branch coverage. However, the authors reported that when both algorithms result in similar coverage, MOSA reaches the coverage level faster than WS. This confirms that the use of the many-objective algorithm is more effective than using an aggregated single-objective algorithm.

When Panichella et al. [127] extended MOSA with a dynamic selection of coverage goals based on their dependency (DynaMOSA), they assessed the performance of their approach with respect to MOSA and WSA approaches using three coverage goals: branch, statement, and strong mutation. In their empirical study, they considered a corpus of 346 complex and non-trivial Java classes that mostly belong to the SF110 benchmark. The complexity of the selected classes is intended to ensure that their branches are not covered easily in the initial population. The results of the study indicate that DynaMOSA shows better performance in regard to branch coverage with 93 CUTs whereas WSA is better than DynaMOSA on only 10 CUTs. Also, DynaMOSA outperforms its preceding approach (MOSA) on 64 CUTs. However, the authors investigated particular CUTs where no significant difference in the coverage is observed between the three approaches, and found out that DynaMOSA is able to converge to the coverage level in a shorter time than MOSA and WSA, while MOSA is faster than WSA in reaching such a coverage level. For example, the three approaches result in a branch coverage of 96% for one specific CUT, but the observed difference is that DynaMOSA reaches this coverage level in less than 10 seconds of the search, while MOSA consumes nearly 30 seconds to converge to the same coverage level, and WSA consumes almost 50 seconds until it reaches the same coverage level. In the case of statement and mutation coverage, DynaMOSA is superior to the other two approaches in achieving higher statement and mutation coverage.

Later, Panichella et al. [128] conducted a thorough analysis of the performance of the state-of-the-art single-objective and multi-objective search algorithms. This includes the SBS, WSA, and LIPS (Linearly Independent Path-based Search) as single-objective algorithms, and MOSA, DynaMOSA, and MIO (many-objective local search) as multi-objective algorithms. Using a corpus of 180 non-trivial classes with more than 34,000 branches, the single-objective algorithms achieve the lowest branch coverage when compared to the other algorithms.

Overall, the achieved branch coverage is 52%, 50%, 40%, 48%, 48%, and 21% with DynaMOSA, MOSA, LIPS, WSA, MIO, and SBS algorithms, respectively. It is obvious that the multi-objective algorithms are superior to the single-objective ones, more specifically the best performance is achieved by DynaMOSA which confirms the finding in the previous study [127]. In terms of efficiency, the single-objective algorithms are not as efficient as the multi-objective algorithms where the SBS takes a longer time to reach the maximum coverage within the limited search budget. The only case where SBS becomes more efficient than the alternative algorithms is when the branches of a CUT can be covered easily by random individuals of the initial population. This is because the initialization process is faster with SBS than with the others, especially WSA as it evaluates test suites rather than test cases. For the other two single-objective algorithms, LIPS becomes more efficient than WSA when a CUT includes branches that are easy to cover. In the case of multi-objective algorithms, DynaMOSA produces better efficiency than MOSA and MIO as it reaches the maximum coverage faster than the other two algorithms. Moreover, it has been observed that the multi-objective algorithms are more effective and efficient than the single-objective ones when dealing with more complex CUTs.

The analysis of single-objective and multi-objective algorithms in the generation of unit tests has recently received further attention from the study conducted by Campos et al. [34]. In this study, an in-depth analysis is performed by considering more algorithms to be evaluated (total of 13 algorithms), and optimising based on (i) a single-criterion (i.e., branch coverage) and (ii) multi-criteria (i.e., line, branch, exception, weak-mutation, output, method, method-no-exception, and context-dependent branch coverage). Besides the algorithms investigated in the previously mentioned studies, the other algorithms include variants of the standard GA (e.g., Monotonic GA, Steady State GA, $(1 + (\lambda, \lambda))$ GA, etc.), and the random testing algorithm as well. The authors carried out the empirical study on the corpus of 346 classes from the DynaMOSA study [127], and its findings are as follows: First, there is no one single-objective algorithm that is superior to the others when either optimising single-criterion and multi-criteria. Second, when comparing single-objective algorithms against random search and random testing, the GAs are found to perform better than both random approaches, and random search outperforms random testing. Third, the DynaMOSA achieves higher coverage than the other multi-objective algorithms with both single-criterion and multi-criteria optimisation, and also has better performance than single-objective algorithms and the random approaches. As the focus of this thesis is optimising the branch coverage, we only show the coverage of algorithms based on the single-criterion (i.e., branch coverage) in Table 2.1. Although DynaMOSA and MOSA result in the highest branch cover-

age (82%), the DynaMOSA outperforms MOSA with the multi-criteria optimisation.

In summary, GAs are effective at generating tests that achieve high code coverage in which they behave differently, depending on the complexity of the classes and the types of the branches. It has been shown that representing the branch coverage problem as a multi-objective optimisation problem is promising to generate test data that increase the branch coverage, especially with more complex CUTs. On the other hand, the single-objective approach is, in many cases, not as effective as the multi-objective approach but still better than the random search, in particular, the archive-based Whole Suite approach (WSA). However, despite the effectiveness of these algorithms in generating potential tests, there is no algorithm that is always capable of satisfying all coverage goals, as shown in the aforementioned studies. This, in fact, raises the question as to what inhibits the search algorithm from achieving high coverage. The next section discusses the possible answers to this question.

2.2.5.1 *Challenges and Limitations in GAs for Unit Testing*

While investigating the performance of different GAs in the generation of optimal test suites, several challenges arise that have detrimental effects on the code coverage. One well-known challenge is the generation of complex parameter objects [64, 149]. There might be a case where reaching a specific branch requires finding inputs for complex data types, for example, a complex input string that constructs a control flow graph object or requires a certain sequence of calls. Another challenge is the external method call problem [179] where invoking a method to an external library either causes an exception to be thrown or returns a value that is not guaranteed to cover the desired branch. Despite the attempts made to overcome these challenges [63, 179], they are still open challenges for the search algorithms.

Another encountered challenge is known as the *flag problem* [109] that is caused by evaluating a branch predicate when involving a boolean variable (called flag variable). The evaluation of such predicate results in two values that represent two plateaux in the fitness landscape, which in this case makes the search unguided and random as no gradient is provided to guide the search to the true branch. As a solution to this problem, the program transformation approach can be used to remove flag variables from branch predicates by replacing them with expressions that make them flag-free [78]. However, one issue related to this approach is that the flag variables that are in loops make it difficult to transform a program.

Moreover, the performance of the GAs can be affected by features related to the software system as reported by Oliveira et al. [120]. Authors investigated whether different features of Java CUTs influence the effectiveness of different GAs (i.e., Random testing, WSA, and

MOSA). As a result, they found out that the number of methods, the coupling between object classes, and the response for a class have a great impact on the performance, and more specifically the branch coverage.

Although efforts have been made to mitigate these challenges and improve the code coverage, there is still uncertainty about what causes a low code coverage, especially with state-of-art algorithms. A crucial issue when considering a GA to solve an optimisation problem is that there is no indication of why the search fails to reach the desired goal (i.e., finding test inputs that achieve high code coverage in our case), which occurs because of the lack of understanding of the search behaviour during the optimisation. Such an understanding can be provided by investigating the underlying structure of the search space and the influence of its features on the optimization process, and that is by considering the *fitness landscape analysis*. Analysing the fitness landscape helps in identifying the search properties that are related to the problem difficulty [117].

However, the features of the underlying structure are not the only factor that influences the search efficiency, but the search space may not be well explored by the individuals of a population [44]. One reason to why this occurs is the lack of diversity in the population, which leads to a well-known issue known as *premature convergence*. If the individuals of the search population all become homogeneous and lack diversity, then the search may converge on a local optimum of the objective function or even a non-optimal point. This reduces the effectiveness of the GA, and in the case of search-based test generation, premature convergence would imply a reduced code coverage.

In fact, maintaining the population diversity is particularly important with a rugged fitness landscape (i.e., a landscape with many optima) to avoid stagnation in local optimum. GAs tend to lose diversity during the evolution, and in the case of a landscape with multiple optima, it is not guaranteed to reach the global optimum (i.e., the optimal solution among all possible solutions). Therefore, population diversity is beneficial to ensure that all individuals are spread all over the search space and thus explore all landscape optima [177].

As the focus of this thesis is on the investigation of the impact of the fitness landscape and the population diversity on the generation of unit tests, the two topics will be discussed thoroughly in the rest of this chapter.

2.3 FITNESS LANDSCAPE ANALYSIS

A greater understanding of the behaviour of combinatorial optimization algorithms comes from a thorough analysis of the underlying topological structure of the search space. This topological structure over which the search is being executed is known as the *fitness landscape*, a term that was first introduced by Sewall Wright [178]. More formally, a fitness landscape (S, f, N) of a problem instance for a given optimization problem consists of a set of genotypes S that represent the problem solutions, a fitness function $f : S \rightarrow \mathbb{R}$ that maps each genotype to a numerical fitness value, and a genetic operator N that defines the neighbourhood relationship between the genotypes (i.e., assigning a set of neighbours to each solution). Given a specific landscape structure, an optimization algorithm can be thought of as navigating this structure in order to find optimal or near-optimal solutions.

The structure of a fitness landscape is completely defined by several landscape features [104] that could influence the performance of the search algorithm. Studying the features of the fitness landscape (also known as landscape properties) allows assessing the problem difficulty by investigating the relationship between the landscape features and the algorithm behaviour. This, in fact, provides insights on which landscape features have an effect on the algorithm performance, and the suitability of the algorithm to solve a given optimisation problem. However, these features are associated with the optima in terms of their number, size, and distribution across the landscape. For example, the modality is one feature that evaluates the number of optima and their density in the search space. A fitness landscape with a single optimum (i.e., unimodal landscape) is easy to search in which a deterministic hill climbing algorithm is suitable, whereas a landscape with many optima (i.e., multimodal landscape) is more challenging. Looking at the frequency of optima in the landscape is not always an indicator of problem hardness, as Horn and Goldberg [83] reported that there is a case where a multimodal landscape with many local optima is easier to search than the unimodal one.

Unlike the modality, *ruggedness* and *neutrality* provide more information about the landscape, and have an explicit impact on the ability of the optimization algorithm at finding optimal solutions [65].

2.3.1 *Ruggedness*

Ruggedness is an important feature of the fitness landscape that contributes to the problem hardness. A fitness landscape is said to be rugged if the landscape contains multiple local optima surrounded by deep valleys and an isolated global optimum. In a rugged landscape, the neighbouring solutions are less correlated as the difference in their

fitness values is high. In this case, the search for an optimal solution is thought to become harder as the algorithm might get trapped in local optima and result in sub-optimal solutions. Ruggedness can be analyzed based on different types of landscape walks [133], i.e., randomized explorations of the search space. Among these walks is the *random walk*, which is a very efficient way to represent the structure of the fitness landscape, regardless of the starting point [175].

RANDOM WALK

A random walk is a mathematical formalization that describes a path of successive random steps on a mathematical space. The walk in such space is unbiased as it moves in a direction that is independent of other directions explored previously. The random walk was first introduced in 1905 by Karl Pearson [130], and has been applied in various scientific disciplines as a fundamental approach to describe the behaviour of the applied stochastic processes.

In the case of combinatorial optimisation problems, the random walk starts at a randomly initialized solution in the landscape and then arbitrarily moves to a neighbouring solution using the genetic operator N. After that, the same process is repeated at each step of the walk (i.e., moving randomly from one solution to its neighbour) until a required number of steps is reached. As a result, a sequence of M fitness values is obtained when the random walk terminates where M corresponds to the number of steps. Several studies show that the random walk is effective to be used with statistical measures that analyse the features of the fitness landscape [104, 175], i.e., statistical fitness landscape analysis metrics rely on the outcomes of the random walk to analyse the landscape features.

2.3.2 *Neutrality*

Ruggedness alone is not enough to measure the search difficulty if equilibrium periods dominate the process of evolution. Such periods result in a set of neighbouring genotypes that have the same fitness value. The presence of these periods in a landscape defines the concept of neutrality [136]. A neutral fitness landscape can be depicted by a landscape with plateaus. A plateau in the landscape can be described as a flat terrain where all neighbouring solutions are of equal fitness. In this case, the mutation in a neutral fitness landscape produces much more movements in genotype space with no effects on fitness. A solution x is said to be a neutral neighbour of a solution y if $f(x) = f(y)$. The existence of neutral areas in a landscape can be detrimental to the search as no gradients are provided to the search to follow, which possibly makes the search stagnates. In contrast, neutrality can be beneficial as it can help in escaping from nearly local optimal solutions. In order to obtain a comprehensive picture of a

Algorithm 3: The pseudocode of neutral walk

```
1  x_0 ← GetInitialRandomSolution
2  neutral_solutions ← x_0
3  α ← GetNeighbour(x_0)
4  θ ← φ
5  while α ≠ θ do
6  |   if f(x_0) = f(α) then
7  |   |   neutral_solutions ← append(α)
8  |   |   α ← GetNeighbour(α)
9  |   else
10 |   |   α ← θ
11 |   end
12 end
13 return neutral_solutions
```

neutral landscape, a neutral walk can be used, which is a variation of a random walk that remains within a neutral area.

NEUTRAL WALK

Algorithm 3 demonstrates how a neutral walk is applied. Similar to the random walk, a neutral walk starts with an initial random solution (Line 1). Then, a genetic operator is applied on the initial solution in order to find its neighbour (Line 3). If the fitness of the neighbouring solution is similar to the fitness of the initial solution (Line 6), the neighbouring solution is considered as a neutral neighbour of the initial one and thus can be appended to the sequence of neutral neighbouring solutions of the neutral walk (Line 7). This is repeated by finding the neighbour of each newly generated neutral neighbour (Line 8) until no more neutral neighbours can be found.

2.3.3 *Fitness Landscape Measurements*

The sequence of fitness values that are obtained from a landscape walk can be used to analyse the structure of the fitness landscape. Based on that, different statistical measures have been proposed [133] to measure both ruggedness and neutrality:

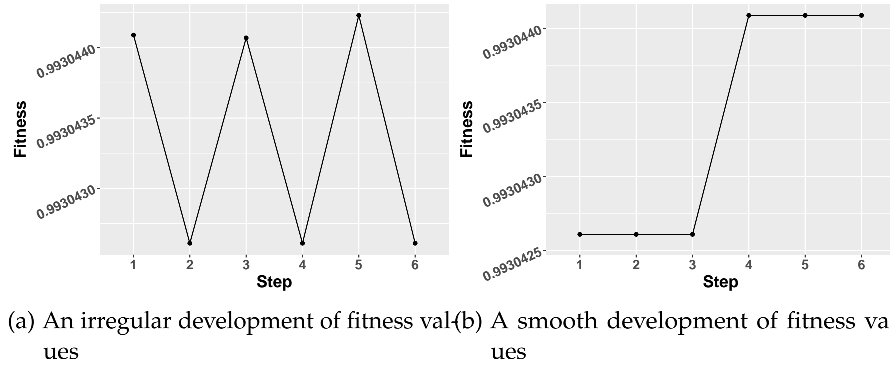
2.3.3.1 *Measure 1: Autocorrelation*

A well-known measure of ruggedness is the autocorrelation function that was introduced by Edward Weinberger [175]. Autocorrelation (AC) is applied to the sequence of fitness values that are obtained from the random walk to measure the correlation between the fitness of each two individuals that are i steps away. A low value of AC results from less similar fitness values, which indicates a rugged landscape.

In contrast, more similar fitness values result in a high AC value and, in this case, neutral areas seem to dominate much of the landscape. The AC is calculated as follows:

$$r(s) = \frac{\sum_{i=1}^{N-s}(f_i - \overline{f})(f_{i+s} - \overline{f})}{\sum_{i=1}^{N}(f_i - \overline{f})^2} \quad (2.5)$$

where N is the total number of the individuals of the random walk, s is the step size, f_i is the fitness of the ith individual, and $\overline{f}$ is the mean fitness of all the individuals. The resulting value is in the range of -1 to 1 where the landscape is more rugged when the AC value is close to -1.



(a) An irregular development of fitness values

(b) A smooth development of fitness values

Figure 2.4: Two examples of different fitness values resulted from a random walk of six steps

To illustrate how the AC is applied on a series of fitness values that are obtained by the random walk, consider the two examples of random walks that are shown in Figure 2.4. Looking at the case shown in Figure 2.4a, it indicates a rugged landscape as the fitness values are more fluctuated. This is confirmed by the AC function that returns a value of -0.83, which is very close to -1. On the other hand, the landscape in the case shown in Figure 2.4b is flatter as many of the individuals of the random walk result in similar fitness values, which indicate the presence of plateaus. In this case, the resulted value of the AC is 0.65 that can be interpreted as many neutral areas dominate the landscape.

2.3.3.2 *Measure 2: Neutrality Distance*

The Neutrality Distance (ND) is one measure of neutrality in a landscape. It measures the number of neutral steps made at the start of the walk as it continues only within a neutral area trying to continuously increase the distance to the starting individual. More formally, for a walk $x_1, x_2, \ldots$, ND is the largest t such that $f(x_1) = f(x_2) = \cdots = f(x_t)$. For example, consider the following two sequences of fitness values that are obtained by a walk on two dif-

ferent landscapes: $S1 = \{f_t\}_{t=0}^{7} = \{0.3, 0.3, 0.3, 0.2, 0.2, 0.7, 0.7\}$ and $S2 = \{f_t\}_{t=0}^{7} = \{0.3, 0.3, 0.3, 0.3, 0.3, 0.3, 0.4\}$. The output of calculating the ND with the first sequence $S1$ is 3 as the first 3 individuals in the walk are considered as neutral neighbors since they result in a similar fitness. On the other hand, the ND in the case of the second sequence $S2$ is 6. The interpretation of the two results is that the landscape of $S2$ seems to be dominated by neutral paths according to the result of ND.

2.3.3.3 *Measure 3: Neutrality Volume*

The Neutrality Volume (NV) is another measure of neutrality based on the number of neighbouring areas of individuals with equal fitness during the random walk. This measure provides more information about the neutrality in the landscape as it estimates the size of neutral areas in the landscape. For example, the NV of the sequence of fitness values $\{f_t\}_{t=0}^{7} = \{0.3, 0.3, 0.3, 0.2, 0.2, 0.7, 0.7\}$ is 3 as there are 3 areas of equal fitness with values $0.3, 0.2$, and 0.7. The NV of $\{f_t\}_{t=0}^{7} = \{0.3, 0.3, 0.1, 0.2, 0.2, 0.7, 0.4\}$ is 5. The interpretation of the two cases is that the landscape in the first example is expected to be flatter than of the second example as more of the fitness values are equal.

Besides these measures, additional measures to gain further information about the structure of the landscape have been proposed [171] based on information analysis. These measures, known as information-based measures, depend on the sequence of the fitness values that are obtained from the random walk. However, instead of directly using the fitness values of the random walk, the following steps are applied:

Step 1: The sequence of fitness values $\{f_t\}_{t=1}^{n}$ is first transformed into a series of fitness changes:

$$\Delta \{f_t\}_{t=1}^{n} := \{f_t - f_{t-1}\}_{t=2}^{n} \tag{2.6}$$

Step 2: The series of fitness changes is represented as an ensemble of objects that can be defined as a string $S(\epsilon) = s_1, s_2, \ldots, s_n$ of symbols $s_i \in \{\bar{1}, 0, 1\}$ given by:

$$s_i = \begin{cases} \bar{1}, & \text{if } x < -\epsilon \\ 0, & \text{if } |x| \leq \epsilon \\ 1, & \text{if } x > \epsilon \end{cases} \tag{2.7}$$

where x corresponds to each of the fitness changes that are resulted from equation 2.6. The parameter ϵ is a real number that is taken from the interval $[0, l_n]$, where l_n is the length of the interval of the fitness values that are obtained by the random walk.

To illustrate how the last two steps are applied, consider the following sequence of fitness values [171] $\{f_t\}_{t=0}^{5} = \{0, 0.01, 0.05, 0.2, 0.21, 0.9\}$. The ϵ value should be in the interval $[0, 0.9]$. Assuming $\epsilon = 0.01$, the string $S(\epsilon)$ resulting from Step 2 is $\{01101\}$. Then, the obtained string $S(\epsilon)$ is further analysed by applying the following measures:

2.3.3.4 *Measure 4: Information content*

The Information content (IC) is designed to capture the variety of shapes in the string $S(\epsilon)$ in order to analyse the ruggedness of the landscape. It is an entropy measure of the number of consecutive symbols that are not equal in the string $S(\epsilon)$. It can be calculated using the formula:

$$H(\epsilon) = -\sum_{p \neq q} P_{[pq]} \log_6 P_{[pq]} \tag{2.8}$$

The probabilities $P_{[pq]}$ are frequencies of the possible blocks pq of elements from the set $\{\bar{1}, 0, 1\}$, and are defined as:

$$P_{[pq]} = \frac{n_{[pq]}}{n} \tag{2.9}$$

where $n_{[pq]}$ is the number of occurrences of each pq in the string $S(\epsilon)$. The value of $H(\epsilon)$ increases with an increase in the number of peaks in the landscape (i.e., a rugged landscape), and, in contrast, it decreases when plateaus dominate the landscape. Applying this measure on the example shown earlier, the only possible sub-blocks of the string symbols are 01 and 10 since the symbol $\bar{1}$ is not shown in the string $S(0.01)$. In this case, the probabilities are $P_{[01]} = 2/5$ and $P_{[10]} = 2/5$. Note that the first number in the string $S(\epsilon)$ is considered as the last string as well since the string is taken with periodic boundary conditions. The result of $H(\epsilon)$ is approximately 0.4091, which indicates that the landscape path is not entirely rugged.

2.3.3.5 *Measure 5: Partial Information content*

The purpose of the Partial Information Content (PIC) measure is to analyse the modality of the landscape by filtering the string $S(\epsilon)$ into $S'(\epsilon)$ removing all zeros and all symbols that equal their preceding symbol. In this case, the new string $S'(\epsilon)$ has the form $\{\bar{1}, 1, \bar{1}, \dots\}$. The PIC can then be calculated as:

$$M(\epsilon) = \frac{\mu}{n} \tag{2.10}$$

where μ is the length of the string $S'(\epsilon)$ and n is the length of the string $S(\epsilon)$. If the landscape path is maximally multimodal, $M(\epsilon)$ is 1 as the string $S'(\epsilon)$ is identical to $S(\epsilon)$ (i.e., $S(\epsilon)$ cannot be modified). In contrast, the landscape path is flat when the $M(\epsilon)$ is 0 as there

are no slopes in the landscape path. When the PIC is applied on the example above, the $M(0.01) = 1/5$, which implies that the landscape is partially flat.

2.3.3.6 *Measure 6: Density-basin Information*

In contrast to the IC measure, Density-basin Information (DBI) measure estimates the variety of flat areas in the landscape. It captures the information of smooth points by only considering the equal consecutive symbols in the string $S(\epsilon)$. In this context, the only possible sub-blocks of the string symbols are $00, 11, \bar{1}\bar{1}$, and the entropic measure is defined as:

$$h(\epsilon) = -\sum_{p=q} P_{[pp]} \log_3 P_{[pp]} \tag{2.11}$$

Therefore, a high value of $h(\epsilon)$ indicates a low density of peaks in the landscape, and thus that the landscape structure is dominated by flat areas.

2.3.4 *Fitness Landscape Analysis in Test Generation*

There have been several studies that attempt to investigate the fitness landscape in the domain of SBST in order to understand how the landscape features affect the success of search algorithms in generating potential test inputs.

An early study that implicitly studied the fitness landscape of structural testing was conducted by McMinn [109] where he investigated the fitness landscape of different objective functions, and identified certain problems that lead to the existence of plateaus in the landscape. More specifically, analysing the fitness landscape of different objective functions that target the branch coverage demonstrates that the objective function that considers the number of executed control-dependent nodes results in more plateaus in the objective fitness landscape. This occurs because such an objective function does not give guidance to the search, especially when diverging away from the target node, as no distance information is provided. When the branch distance is incorporated into the objective function, the formation of many plateaus in the landscape is prevented.

However, a plateau can be introduced in the landscape when the branch predicate involves a boolean variable (i.e., flag variable), as mentioned in section 2.2.5.1. To illustrate the flag problem, consider the example shown in Listing 2.5 where covering the true branch is only achieved when the `flag` is true. In this case, when x is set to any other value than zero, the `flag` will always be false, and the fitness value will be higher than zero (i.e., minimised objective function). This, in fact, results in large plateaus in the fitness landscape, as shown in

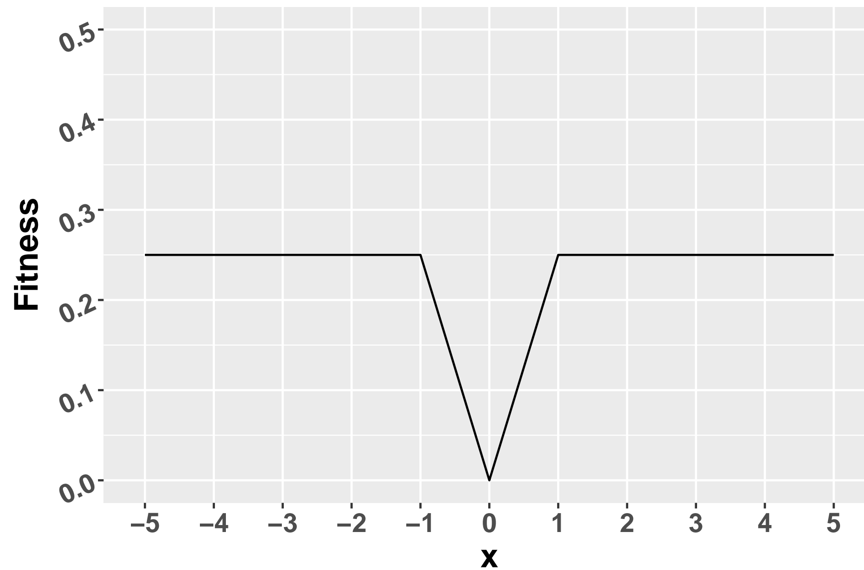


Figure 2.5: The fitness landscape of the objective function when applied on the flag example

Figure 2.5, since the objective function does not provide guidance to the search as to how far the true value is from current false value.

```
boolean flag = (x == 0);

if(flag)
  output = 0;
else
  output = x * y;
```

Listing 2.5: Flag problem example

There are many approaches that have been proposed to overcome the flag problem such as using a program transformation [78] and extracting the sequence of nodes that must be executed to reach the branch predicate containing the flag [20]. Despite their success in showing better performance when dealing with the flag problem, they are not always capable for solving this problem (e.g., when the flag variable is involved in a loop).

In a very related study, Aleti et al. [6] investigated the fitness landscape in the whole test suite generation (WSA), and study how the two genetic operators (i.e., crossover and mutation) influence the test generation. In this study, the properties of the fitness landscape were analysed using information acquired during the evolution, such as the sequence of fitness values of the best individuals and the number of fitness improvements, and then correlated with the resulting branch and method coverage of two versions of the GA; a GA with the mutation only and a GA with both crossover and mutation. The study results

suggest that the fitness landscape is dominated by plateaus that are detrimental to the search, and using crossover in such a landscape is not beneficial as it fails to escape local optima. An important finding is that adding a test case to the individual test suite that does not lead to better coverage induces a plateau in the landscape. On the other hand, adding a test case that improves the coverage might conflict with an existing test case, which negatively affects the individual test suite and results in a local optimum. It was also observed that many iterations of the search explore the flat areas in the landscape, which thus worsens the search efficiency.

Recently, Vogel, Tran, and Grunske [172] studied the fitness landscape of test suite generation for mobile applications using a multi-objective evolutionary search algorithm known as Sapienz. Their fitness landscape analysis focuses on the global topology of the landscape, i.e., how individuals are distributed over the search space, and not on local structure, i.e., ruggedness and neutrality. The analysis is based on 11 metrics that characterize the Pareto-optimal solutions, population, and connectedness of Pareto-optimal solutions with respect to the genotypic similarity among all individuals. These metrics are applied after every generation of the algorithm, revealing that the search convergences because of the lack of diversity. This is observed when the search after a few generations (i.e., almost 25 generations) becomes unable to find new non-dominated individuals that dominate existing local non-dominated ones. Also, the search stagnates because of the duplicity of solutions in the population that leads to the loss of genetic variation among the individuals. Most importantly, the distribution of Pareto-optimal solutions over the search space is another reason of low population diversity, as these solutions tend to group in one cluster in the search space and are not spread in many areas of the space. This, in fact, led the authors to extend their study to investigate the problem of population diversity.

However, the study conducted in [172] showed that the search stagnates after almost 25 generations, which is observed when running Sapienz with a limited number of apps (i.e., 10 apps), few repetitions (i.e., 20 repetitions), and a limited number of generations (i.e., 10 generations was used only to experiment Sapienz when enabling diversity promotion). This motivated the authors to extend their study [173] in order to gain a thorough understanding of the fitness landscape by considering extra apps under test (i.e., 34 apps), a high number of repetitions (i.e., 30 repetitions), and performing the search over 40 generations. Using the same 11 metrics in the previous study with the new configurations, the results still confirm that the search stagnates after nearly 25 generations.

2.4 POPULATION DIVERSITY IN EVOLUTIONARY ALGORITHMS

Despite the success of Evolutionary Algorithms (EAs) in solving complex optimisation problems, they still suffer from a well-known problem known as premature convergence. This problem occurs when the search prematurely converges to local optima because of the lack of population diversity. Maintaining population diversity during the evolution of EAs is crucial for avoiding premature convergence [107]. The term "diversity" refers to the variety in a population based on the differences at the genotype (i.e. structural) or phenotype (i.e. behavioural) levels. The term genotype refers to genetic characteristics of an organism (individual) while the phenotype refers to the observable characteristics in individuals (e.g., the fitness value of an individual). The relationship between genotype and phenotype can be seen as any change in organism's genes (genotype) will cause an observable change in the observable characteristics of the organism (phenotype). Population diversity has been seen as an important research topic in the area of EAs, because of its impact on the EA performance [44, 159]. In general, the genetic search is motivated when the individuals of a population are more diverse. A diverse population is also beneficial during the exploration phase to avoid premature convergence and to escape local optima, and thus ensure that the search space is adequately explored. The rest of this section covers different diversity measures, maintenance, and control techniques.

2.4.1 *Population Diversity Measurement*

Population diversity measures are intended to quantify the variety of a population's individuals based on structural (i.e., syntactic) or behavioural (i.e., semantic) levels. These levels differ among different domains [29], e.g., the structure of an individual in the case of Genetic Programming (GP) is not similar to the one with other EAs. One important benefit of considering diversity measures is to understand the behaviour of an EA during the evolution, especially when any of the diversity control techniques is applied. Moreover, diversity measures can be used to guide the search in EAs; an example is the Diversity-Control-Oriented Genetic Algorithm [152] that uses a diversity measure based on Hamming distance to calculate a survival probability for the individuals. The selection in this algorithm is based on the survival probability; the low distance between an individual and the current best individual leads to a low survival probability of this individual, and vice versa. In contrast, Burke et al. [29] showed that diversity measures do not always improve the search process as there is not always a positive correlation between diversity measures and improving the fitness in GP.

In general, there are three different levels of diversity measurement [44]: Genotype level, Phenotype level, and Composite measures. The genotypic diversity measures the structural (i.e., syntactic) differences among the individuals of a population. In contrast, the phenotypic diversity is based on the behavioural (i.e., semantic) differences in the population's individuals. Finally, composite measures are a combination of genotypic and phenotypic measures. However, the performance of the genotype-based and phenotype-based measures is not always consistent. Their performance varies when they are applied to different types of problems [166]. In fact, phenotype-based measures are found to be more promising than the genotype-based measures. For example, Tsutsui et al. [166] noticed that their phenotypic measure maintained a proper balance in the exploration and exploitation of different populations, and showed better performance than the genotypic measure. In GP, Burke et al. [29] showed that the phenotype-based measure is easier and less costly than the genotype-based measure.

However, several diversity measures have been proposed in the literature for different EAs problems [30, 44, 80] and classified into the following two categories:

2.4.1.1 *Phenotype Measures*

The phenotypic diversity measures aim to measure the differences between individuals of a population based on their behaviour, which in practice is measured based on the fitness value of each individual in the population; the diversity rate is based on the spread of fitness values [80]. A well-known fitness-based measure is the entropy measure, first proposed by Rosca [142]. The entropy represents the amount of disorder in a population, where an increase in entropy leads to an increase in diversity in the population. Rosca defines diversity based on the entropy as follows:

$$E(P) = -\sum_{k} p_k \cdot \log p_k \tag{2.12}$$

where the population P is partitioned according to the fitness value which will result in a proportion of the population p_k that is occupied by the partition k.

However, behaviour can also be measured without the need to consider the fitness values. For example, in GP the phenotypic diversity can be measured based on the observed behaviour of the individuals that are represented as programs when they are executed during the evolution [84]. Another example can be seen in the Diversity-Guided EA that was proposed by Ursem [168], which utilises a semantic measure to alternate between the exploration and exploitation. The measure considers the distance of each individual in the population to the average point of the population.

2.4.1.2 *Genotype Measures*

In contrast to phenotypic diversity measures, genotype-based measures attempt to measure the structural differences between individuals of a population. Counting different genotypes is one way to measure the structural differences [44]. For example, Koza [95] considered the number of different programs in a GP (i.e., a program is represented as syntactic tree) as a structural difference among the individuals. Angeline et al. [12] extended the previous approach to consider the variety of subtrees by counting unique subtrees to measure the diversity. Another way to measure the genotypic diversity is to measure the differences between two individuals based on an edit distance [80]. When this measure is applied to GAs, the distance between two individuals is measured by how many bits are needed to transform one individual to another (i.e. Hamming distance can be used here). In the case of GPs, the representation of individuals (e.g. trees and graphs) is more complicated than GAs. Therefore, previous studies attempted to use different distance-based measures to calculate the genotypic diversity [80]. For example, O'Reilly [119] used Levenshtein distance to measure the amount of syntactic differences between two trees based on the shortest cost sequence of three operations (i.e. single node insertions, deletions and substitutions) to transform two trees to be equal in their structure and content. De Jong et al. [46] also used a similar edit distance in a multiobjective method that calculates the syntactic distance between two trees as the sum of the distance of the corresponding nodes. The corresponding nodes are the nodes in both trees that are in the common area when they are aligned.

As mentioned previously, the diversity measures can be used to guide the evolutionary search as they provide feedback to improve the search. However, there are several diversity techniques that utilise the provided feedback to guide the evolution process, which are known as *diversity control techniques*. Also, there exist other techniques that do not rely on the feedback provided by the diversity measures, and only maintain a proper level of diversity during the search that are known as *diversity maintenance techniques* [44].

2.4.2 *Population Diversity Maintenance Techniques*

Many diversity maintenance techniques have been proposed in the past for different EAs [54]. A recent survey by Črepinšek et al. [44] classified these techniques into two categories: niching techniques and non-niching techniques. Niching techniques aim at maintaining enough diversity in the population and reducing the effects of genetic drift by segmenting the population into subpopulations to locate multiple optimal solutions. Non-niching techniques maintain diversity without the need to maintain sub-populations, for example, increasing

population size, changing selection pressure, or applying replacement restrictions. Both niching and non-niching techniques are capable of preserving diversity in a population.

2.4.2.1 *Niching Techniques*

In this section, two well-known niching techniques, i.e., Fitness Sharing and Clearing, are presented:

FITNESS SHARING

Fitness sharing is the most popular approach among the niching techniques [144] and theoretical studies have proven it to be effective on pseudo-Boolean benchmark problems [121]. It aims to find multiple peaks in the solution space, where each subpopulation around a peak represents a niche where individuals share the same resource (i.e., fitness value). The idea behind fitness sharing is to decrease the value of the resource that is shared by the individuals of a niche when the number of individuals is high, and increase it when there are few individuals in a niche, which gives these individuals a higher probability to be selected for next generations. The shared fitness of each individual is calculated as follows:

$$f_i' = \frac{f_i}{m_i} \tag{2.13}$$

where m_i is the niche count which is the number of individuals that share the fitness f_i. The niche count m_i is calculated by summing the sharing function (sh) for all the individuals in the population

$$m_i = \sum_{j=1}^{\mu} sh(d_{ij}) \tag{2.14}$$

where μ represents the population size and d_{ij} is the distance between individual i and individual j (e.g. Euclidean distance [144]). The sh measures the distance between each two individuals in the population as follows:

$$sh(d_{ij}) = \begin{cases} 1 - (d_{ij}/\sigma_s)^{\alpha}, & d < \sigma_s \\ 0, & \text{otherwise} \end{cases} \tag{2.15}$$

where σ_s is the peak radius (i.e., sharing radius) and α is the parameter that regulates the form of the sharing function, commonly equal to 1.

CLEARING

Clearing is similar to fitness sharing, except that it shares the available resources among the best individuals of each subpopulation rather than among all individuals of a subpopulation. That is, it keeps the

fitness of the best individuals (i.e., *dominants*) in each subpopulation as they are, and the fitness of the other individuals is cleared (e.g., set to zero) when maximising. The dominants take all rather than sharing resources with the other individuals of the same niche as is done in the sharing method. This technique is found to be promising for solving challenging multimodal functions [43] and, in addition, it outperforms fitness sharing in dealing with genetic drift [132]. In recent empirical work [42] it outperformed other diversity mechanisms and it was the only mechanism found able to tunnel through fitness valleys.

2.4.2.2 *Non-Niching Techniques*

There are different non-niching techniques that are used to promote population diversity during evolution. Among these techniques are the following:

INCREASING POPULATION SIZE

One simple technique is to increase the population size to allow for more new individuals that more likely enhance the diversity level. Despite its simplicity, increasing the population size is not always guaranteed to maintain the diversity [111].

INFUSION APPROACH

New individuals are randomly generated and inserted after a certain number of generations. One simple approach that is used by Grefenstette [71] is to seed the population with new randomly generated individuals every generation. Koza [95] presented an infusion technique, called *decimation*, where a high proportion of a population is replaced by new random individuals at regular intervals. However, there are infusion approaches that do not insert new random individuals, for example, the concept *opposition* that was introduced by Rahnamayan et al. [135] where the opposite individuals of the current individuals are inserted rather than inserting new random individuals.

DUPLICATE ELIMINATION

The purpose of this technique is to remove the similarity between individuals of the population, which has been shown to enhance population diversity and GA performance [35, 141]. Two individuals are considered similar when their distance to each other is zero. To ensure enough diversity in the population, the eliminated individual is either mutated [107] or replaced with a new generated individual.

DIVERSE INITIAL POPULATION

The initial population is known to have an impact on convergence [102, 165], and its diversity can potentially enhance the performance of the GA [50]. The initial population is diversified by generating a population of random individuals with size m larger than the intended pop-

ulation size n, and then selecting the most distant n individuals from the population of m individuals based on a diversity measure [172].

2.4.3 *Population Diversity Control Techniques*

As mentioned earlier, the diversity control techniques rely on the feedback that is provided by the population diversity measures to steer the evolution towards better exploration and exploitation of the search space. Population diversity can be controlled through the selection and the two genetic operators (i.e. crossover and mutation).

An early attempt to adapt the diversity control through selection is the selection criterion in the Diversity-Control-Oriented Genetic Algorithm (DCGA) [153]. It depends on measuring the distance between an individual and the best individual to determine the selection probability for this particular individual (i.e., an individual with higher distance is preferable). Later, Chaiyaratana et al. [35] extended the DCGA to include a fitness-based measure to the selection criterion. In this case, an individual has a higher selection probability if its fitness is high enough and has a higher structural difference from the best individual.

Healthy population diversity (HPD) is another approach that is presented in the work of McGinley et al. [108], which is a measure of the diversity of healthy (high-fitness) individuals in a population that is used to control the diversity. This measure is used to adaptively control the selection pressure through adapting the tournament size (i.e. the tournament size is increased to promote the survival probabilities of the fittest individuals). The selection, in this case, is based on the healthy diversity contribution and the fitness of each individual.

Adaptively changing mutation and crossover rates [44, 131] based on the diversity level is thought to help to avoid convergence to a local optimum. The idea is to set the crossover and mutation rates based on a current diversity level. One simple approach that is applied by Whitley and Starkweather [176] is to adaptively change the mutation rate according to the genotypic measure of Hamming distance between individuals to preserve the population diversity.

Srinivas and Patnaik [156] presented a customised GA, known as Adaptive Genetic Algorithm, that adapts the crossover and mutation rates depending on the diversity level that is based on the difference in fitness values of the individuals. In this case, the crossover and mutation rates are increased when the population converges to local optima and, on the other hand, rates are decreased when the population is scattered within the search space.

Another approach that utilises the fitness values to dynamically adapt the two rates is applied by Vasconcelos et al. [170]. Their diversity measure is calculated as the ratio between the mean and maximum values of the fitness for each generation, which results in a value in the

range [0,1]. When the resulted value is one, most of the individuals tend to have the same genetic structure (i.e. low diverse individuals) and, consequently, the GA converges too quickly. In this case, the mutation rate is increased to allow for more exploration and the crossover rate is decreased. In contrast, when the diversity is high, the crossover rate is increased to allow for more exploitation and the mutation rate is decreased.

Using a diversity measure such as fitness entropy measure is also used to control the two genetic operator rates, as with the adaptive genetic algorithm with diversity-guided mutation [100]. This algorithm combines adaptive probabilities of crossover and mutation, which shows better performance with multimodal test functions, and was found to be capable of avoiding premature convergence.

Another recent approach that dynamically controls crossover and mutation rates based on a phenotypic measure that utilizes the Euclidean distance is proposed by McGinley et al. [108]. Here the crossover rate is increased when diversity is high to allow for more exploitation, whereas the mutation rate is increased when diversity is low to allow for more exploration. This approach demonstrates that the dynamic control of the two operators leads to better search performance, and maintain a proper level of diversity is beneficial to the search as well.

The aforementioned techniques work towards achieving a proper level of population diversity when the target is a single objective to be optimised. However, in the case of multi-objective optimisation, diversity techniques are applied to achieve a well-distributed tradeoff front. For example, the Multi-Objective Genetic Algorithm (MOGA) [57] applies fitness sharing on the population of individuals in every generation. Similarly, fitness sharing is also considered in the Non-dominated Sorting Genetic Algorithm (NSGA) [157] but is applied to each front's individuals rather than the whole population. To overcome the issue of the user-defined sharing parameter in fitness sharing, NSGA-II [49] replaces fitness sharing with the crowding distance approach that provides an estimate of the density of individuals that are in the boundaries of a specific individual (i.e., how close an individual to others in the same front).

Although these multi-objective GAs apply the diversity techniques implicitly for the purpose of maintaining diversity during evolution, there are other algorithms that consider diversity as an explicit objective to be optimised. For example, the Genetic Diversity Evaluation Method (GeDEM) [163] uses a distance-based diversity to be considered as an objective during the optimisation process. GeDEM considers a two-objective non-dominated sorting to rank individuals that are based on the rank with respect to the objective function values and the assigned diversity values. The diversity is measured based on Euclidean distance in the objective function space and the Hamming distance in the string space.

2.4.4 *Population Diversity in Test Generation*

The topic of diversity in EAs has been extensively investigated with different domains, as shown in the previous sections. In the context of test generation, there have been several studies on generating diverse test cases [147]. Some of these studies aim for the tests within the final test suite to be diverse, rather than the individuals in the search population. The rest of this section presents different studies that investigate the population diversity in the domain of test generation.

Feldt et al. [56] presented test diversity metrics to find diverse and meaningful tests to human software developers. The metrics depend on a theoretical model for test variability (VAT) with points of variations (i.e., aspects on which two tests might differ). One metric is the Universal test diversity that measures the diversity based on information distance between each two test cases; information about the actual execution of a test for all the variation points in the VAT model. The second metric measures the diversity based on the Normalized Compression Distance (i.e., depending on test information) between tests. The authors conducted an initial experiment to evaluate the second metric against human subjects to cluster 25 Ruby test cases and fount that the metric is able to provide better clustering of test cases in a way that is intuitive to humans.

Kifetew et al. [88] aim to control the diversity of the generated tests by considering the orthogonal exploration mechanism of the search space, which is the estimation of the evolution directions via Singular Value Decomposition (SVD). Evolution directions help in guiding the evolution towards promising directions (i.e. directions that best individuals follow) and exploring new regions in the search space. Based on that, they presented three different schemes of SVD-based GA; one scheme considers the history of populations encountered during evolution, another scheme considers a random direction instead of an orthogonal direction when the search gets stuck in a local optimum, and the combination of the last two schemes is presented as a separate scheme. When evaluating the three schemes against the basic GA with a variety of Java classes, the three schemes are more effective (i.e., achieve better coverage) and more efficient (i.e., consume less search time) than the basic GA, and the best among the three schemes is the combined application of the first two schemes.

Bueno et al. [28] proposed a new test data generation technique, called Diversity Oriented Test Data Generation, that attempts to generate diverse test sets by considering GA, Simulated Annealing, and a proposed metaheuristic called Simulated Repulsion. The later approach generates test data by "simulating particle systems subject to electrical repulsion forces". The diversity measure that is applied considers the distance (i.e., Euclidean distance) among the test data in the program input domain to compute the diversity for test sets. To

evaluate their technique, they performed a Monte Carlo simulation to assess how the test set size and failure rate affect the effectiveness of the proposed technique. In addition, they evaluated the coverage and mutation scores that are achieved by the technique. The evaluation has been conducted using the three metaheuristics against random testing and, as a result, the proposed technique with the Simulated Repulsion improves the coverage in most cases.

Recently, Palomba et al. [124] presented a study to measure the textual similarity between test cases for the purpose of reducing the test coupling (i.e., higher diversity) and increasing the test cohesion (i.e., lower test length). To achieve that, they proposed two quality metrics to detect cohesion and coupling issues at the test case level; the first is the coupling between test methods and the other is the lack of cohesion of a test method. Both metrics rely on Information Retrieval methods to measure test cohesion and test coupling. The two metrics have been incorporated into the MOSA algorithm within the selection mechanism as a secondary objective. When they experimented MOSA including the two metrics against the default MOSA and the default strategy in EvoSuite (i.e., WSA), they found that the proposed metrics are able to improve the branch coverage and reduce the size of tests. More importantly, the generated test cases using the two metrics are more cohesive and less coupled.

In the study presented by Shahbazi et al. [148], the fitness function in a multiobjective GA is modified to be based on the diversity of black-box string test cases. The fitness function, in this case, measures the diversity of a test set as the distance between every test case and its nearest test case (i.e., higher fitness value indicates more diverse test cases). The authors examined different string distance functions such as Levenshtein, Hamming, Cosine, Manhattan, Cartesian, and LSH distance functions. They found that the LSH distance function performs better in measuring the string distance between test cases. However, when comparing the performance of the considered GA (i.e., with diversity-based fitness function) against the random testing, the former is found to be promising in increasing the diversity of test cases, which is indicated by its performance in producing effective tests that increase the fault detection ability.

When Vogel et al. [172] studied the fitness landscape of test suite generation for mobile applications, the analysis of the fitness landscape indicates that the search converges after few generations, and thus the loss of population diversity occurs. Therefore, they extended the study to examine the behaviour of the considered multiobjective GA when promoting population diversity during the evolution. To do that, they incorporate four approaches into the algorithm. The first approach is the Diverse Initial Population that is applied at the initialisation. The second is the adaptive diversity control that aims to control the generation of offspring based on the current diversity

level such that (i) if the diversity level drops below a certain diversity threshold, new individuals are generated as offspring and only most distant individuals are selected from the combination of population and offspring, whereas (ii) the offspring is obtained using the default crossover and mutation operators when the diversity level is higher than the threshold. To overcome the issue of solutions duplicate identified by the landscape analysis, the Duplicate Elimination approach is applied after reproduction and before selection. In addition, the selection is extended to make the diverse individuals are preferable to be selected. However, the diversity is measured based on the average of pairwise genotypic distances between all the individuals.

The diversity-enabled version of their GA (Sapienz^{div}) is evaluated in terms of the achieved coverage and the faults found, and as a result, promoting diversity does not have an effect on coverage, but it finds more faults. Moreover, increasing diversity is found to have a negative impact on the length of tests (i.e., longer test sequences). In terms of efficiency, the diversity-based GA (Sapienz^{div}) leads to an extra runtime when compared to the standard GA (Sapienz). These findings are confirmed by the extended empirical study [173] where longer runs of search (i.e., 40 generations) are given in order to investigate the impact of Sapienz^{div} on the obtained results when the Sapienz search stagnates.

2.5 SUMMARY

This chapter gave a literature review of the research topics studied in this thesis. The focus of the review was on the automation of software testing, and, more specifically, the test data generation problem. We reviewed the existing approaches for the automated test generation, and showed how such approaches achieve the testing goals. As the Search-Based Software Testing (SBST) approach demonstrates better performance in terms of branch coverage, we thoroughly reviewed the existing Single-Objective and Multi-Objective Genetic Algorithms (GAs) that are applied to generate test data for object-oriented programs, and discussed their performance with a different corpus of Java classes. Despite the success of these algorithms in generating potential tests that achieve high branch coverage, there are still cases where they do not perform well, and thus does not always generate test data that achieve full branch coverage. Therefore, we shed the light on the challenges and limitations of the use of GAs from a practical point of view (e.g., the problem of complex parameter objects generation), and from a theoretical point of view as well (e.g., the features of the fitness landscape and the population diversity problem).

As the focus of this thesis is to understand the search behaviour when optimising unit tests, we dedicated the rest of this chapter to the main two topics that have a great influence on the search behaviour;

fitness landscape analysis and population diversity. We first introduced the fitness landscape and its the two features of the landscape (i.e., ruggedness and neutrality). Then, we presented possible measurements of these two features, and how they affect the search. After that, we provided an overview of the population diversity problem, and showed how maintaining enough diversity level during the evolution is beneficial to the search. For that, we looked at several diversity measurements and diversity promoting mechanisms.

Although much research has been performed examining the impact of the fitness landscape features and the population diversity problem on test generation, they received little attention in the domain of object-oriented unit test generation, and there remain open questions that need further research. For instance, how do the two features of fitness landscape influence the search ability in finding test data that achieves full branch coverage, especially with fine-grained objective function on a branch level? How do the underlying properties of the Java source code affect the landscape features? When considering the state-of-the-art algorithms to generate unit tests, what effect does the population diversity have on the search performance? How effective are diversity maintenance and control techniques when applied during the evolution?

In order to answer these fundamental questions, we conducted two main empirical studies that aim to (i) investigate the causes and effects of fitness landscape features in unit test generation, and (ii) investigate the evolution of unit tests and whether maintaining the population diversity during the search has an influence on the performance of GAs. Both studies are presented in the following chapters; the study of the fitness landscape is shown in Chapter 3 and the study of population diversity is shown in Chapter 4.

3 CAUSES AND EFFECTS OF FITNESS LANDSCAPES IN UNIT TEST GENERATION

The content of this chapter is based on work undertaken during this PhD by the author, which has been published at The Genetic and Evolutionary Computation Conference (GECCO) 2020 [4]. The work presented in this chapter extends the published work with an extra experiment on WSA along with its results (Section 3.2).

3.1 INTRODUCTION

As discussed in the previous chapter, Genetic Algorithms (GAs) have been successfully applied for generating object-oriented unit tests, and several studies [34, 62, 128] have shown that GAs are effective at generating tests that achieve high code coverage. However, they are still far from being able to satisfy all test goals (e.g., covering all branches) [34, 149]. While some general limitations are known (e.g., the challenges of generating complex parameter objects [64, 149]), there is a lack of understanding of the search behaviour during the optimisation, making it difficult to identify the factors that make a search problem difficult.

In theory, GAs usually behave differently with different optimisation problems and their behaviour cannot be understood by only looking at their performance (i.e., the final branch coverage in our case), as the algorithm performance is not enough to give insights into how it behaves during the search and what makes it difficult with a given problem. The reason behind that is each optimisation problem has features that influence the behaviour of a GA and its performance. In this case, the GA performance does not advance our understanding and knowledge of how such features affect its hehaviour [82]. Therefore, there is a need for a deep understanding of the optimisation problem features and their impact on the search behaviour. Such an understanding can be provided by investigating the underlying structure of the search space and the influence of its features on the optimisation process.

The concept of the fitness landscape is among the most commonly used metaphors to give an intuitive understanding of the search space structure and help in predicting search behaviour with different search problems. Analyzing the fitness landscape helps in identifying the properties that are related to the problem difficulty [6]. The two main

properties of fitness landscapes that are known to have a great influence on the optimisation process are *ruggedness* and *neutrality* [104]. The interplay of these properties has motivated the development of several techniques that study the structure of fitness landscapes.

The aim of this chapter is to analyse the fitness landscape and to investigate the impact of its properties on the generation of unit tests. More specifically, we study the influence of the two landscape properties, ruggedness and neutrality, on unit test generation. To gain a better understanding of the influence the fitness landscape on the generation of unit tests, we consider the two well-known approaches in our domain that are the archive-based Whole Suite approach (WSA) and the Many-Objective Sorting Algorithm (MOSA). In particular, we analyse the fitness landscape with the objective function that is defined for WSA (i.e., aggregating multiple target goals into a single objective) to gain insights of what makes it less effective to the search when compared to the fine-grained objective function on a branch level, which is defined for MOSA. Also, we intend to investigate whether the representation of individuals influences the properties of the search landscape; an individual of a test suite with WSA and a single test case with MOSA. Moreover, we extend our investigation to understand what underlying properties of source code that influence the fitness landscape features.

Fitness landscape analysis uses different proxy measurements to gather evidence on these properties, usually by analyzing the way fitness values change while randomly walking across the search space. In this chapter, we apply the six most common such measurements, defined in Section 2.3.3, to investigate random walks on a selection of 331 Java classes. By contrasting the resulting metrics with the performance of a GA on generating tests for these problem instances, we can identify how they affect the search, and what aspects of the underlying source code causes these properties.

Our experiments suggest that the landscape structure is mostly dominated by neutral areas, i.e., plateaus, which makes it harder for the search to find test inputs. Although ruggedness is often considered a negative property of the fitness landscape, in the case of unit test generation and the scale of ruggedness observed there, we find that higher ruggedness is an indicator of more informative landscapes, resulting in better performance of the search. A closer look at the causes of neutrality suggests that influential factors are (1) whether the target code is contained in private methods, for which there is no direct guidance provided by the fitness function; (2) whether the code has preconditions that are difficult to satisfy and cause exceptions when violated; and (3) the prevalence of boolean flags, which provide no guidance to the search. This suggests that the search could be improved by enhancing the existing fitness functions to consider inter-procedural

distance information, by addressing the problem of generating *valid* complex objects, and by applying testability transformations.

Our findings conform to the findings of Aleti et al. [6], more specifically the fact that the search space has many plateaus that are detrimental to the search. However, we considered a more fine-grained objective function on a branch level, rather than aggregating all the branches into a single objective function. Moreover, our study investigates the factors that cause the fitness landscape properties such as the underlying properties of the source code.

As the study of the fitness landscape is conducted on two different approaches, the investigation of the fitness landscape when considering the WSA approach is presented in Section 3.2, while the investigation of the fitness landscape when considering MOSA is presented in Section 3.3. In both sections, we describe the experimental setup and procedure followed to conduct the experiment. Then, we present and discuss the experimental results and the answers to the research questions. To better understand the impact of both approaches on the fitness landscape, we compare and present an overall discussion of the results in the previous two sections in Section 3.4. Finally, we provide a detailed analysis of the underlying properties of source code that influence the fitness landscape properties in Section 3.5.

3.2 FITNESS LANDSCAPE ANALYSIS WITH THE WHOLE SUITE APPROACH

The purpose of this study is to analyse how unit test generation using WSA approach is influenced by the fitness landscape. This requires investigating the landscape properties (i.e., ruggedness and neutrality) when the GA considers this specific approach, and then examining how these properties affect the search performance (i.e., the final coverage). Therefore, we design the study to answer the following research questions:

RQ 1.1: What are the properties of the fitness landscape for the JUnit test generation problem when considering the WSA?

RQ 1.2: How do the fitness landscape properties affect the search behaviour when the GA considers the WSA?

Our assumption is that the underlying landscape structure is dominated by plateaus, as confirmed by the findings of Aleti et al. [6], since mutating an individual of a test suite is expected to result in few changes in fitness values during the random walk, which indicates the existence of plateaus in the landscape. A landscape that highly dominated by plateaus is expected to have a negative effect on the GA search as plateaus do not offer enough guidance to the search to find better test inputs in the search space [109].

3.2.1 *Experimental Setup*

3.2.1.1 *Selection of Classes Under Test*

Choosing a diverse set of classes is important in studying the properties of the fitness landscape since the features of Java classes might have an impact on the landscape properties. Therefore, we used the selection of 346 complex and non-trivial classes from the DynaMOSA study [127] where the complexity of classes ranges from 2 to 7939 branches. The complexity of the selected classes is intended to ensure that their branches are not covered easily in the initial population.

3.2.1.2 *Unit Test Generation Tool*

Among the popular tools that generate tests for Java programs using an evolutionary algorithm is EvoSuite [59]. It generates JUnit test suites for a given Java CUT and target coverage criterion using different evolutionary algorithms, with the MOSA algorithm, described in Section 21, being the most effective algorithm for JUnit test generation [34, 128]. By default, EvoSuite applies a Monotonic GA that considers the archive-based WS approach (WSA), as described in Section 2.2.3.6.

3.2.1.3 *Experiment Procedure*

To better understand the influence of the fitness landscape properties on the generation of JUnit tests, we conducted an experiment that involves (i) applying random walks on each CUT, and then (ii) applying all the six fitness landscape measures (described in Section 2.3.3) on the sequence of fitness values obtained by the landscape walks. To perform a walk on a landscape, we applied the corresponding mutation operator in order to move from one landscape point to another where each point in a landscape corresponds to one step of the walk.

In order to perform the experiment, we implemented and run random walk in EvoSuite. We also ran the Monotonic GA in order to compare its performance against the fitness landscape measures. To minimise the influence of other optimizations, we used a "vanilla" configuration [59] and default settings [16] with only branch coverage as target criterion. The search stopping criterion was set to be a one minute timeout, which is EvoSuite's default search budget. As a required step to run a random walk, we consider the most commonly used number of random walk steps in the literature, which is 1000 [23]. We ran EvoSuite 30 times on each class in order to account for the randomness of the algorithm under consideration and the landscape walk.

Running this experiment on the corpus of 346 classes resulted in data for only 331 classes. This is due to the environmental dependencies of 8 classes that are difficult to fulfil by EvoSuite, and the search

timeout was reached for 7 classes because of constraints that cannot be solved within a specific time [62].

3.2.1.4 *Experiment Analysis*

RQ1.1 Analysis: Applying a landscape walk of m steps on a class X is defined as a sequence $x_1, x_2, \ldots$ such that x_{i+1} is the outcome of a mutation applied to x_i where an initial individual x_1 is created randomly by applying the insertion mutation repeatedly until the fixed number of test cases is reached (i.e., 100 test cases). In this case, the final outcome of the walk consists of m fitness values, which are used by each landscape measure to analyse the properties of the fitness landscape. To answer RQ1, we consider the distribution of the average values per each measure (i.e., the average of values resulting from 30 runs) across all the classes.
RQ1.2 Analysis: The impact of fitness landscape properties on the search behaviour can be understood by analysing the correlation between the search performance and the landscape measures. The search performance is usually measured in terms of the resulting branch coverage where a high coverage indicates better performance and vice versa. Therefore, we apply the Spearman correlation on both branch coverage and each of the landscape measures for every single run of a class.

3.2.2 *Threats to Validity*

To control threats of the stochastic behaviour of both techniques, i.e., Monotonic GA and a random walk, we repeated the experiment 30 times. Although we used a selection of 331 complex classes with a diverse number of branches, which was also used by previous studies [127], our results may not generalize to other classes. The search budget used in running Monotonic GA is based on EvoSuite's default search budget of one minute, which is examined previously to assess the performance of Monotonic GA [34]. We chose EvoSuite as a test generation tool as it is the most effective and state of the art JUnit generation tool [125] although the results may not generalise to other coverage-driven test generation tools.

In regards to the random walk, choosing the number of steps of a random walk as 1000 is common practice [23]. It should be noted that a high number of random walk steps (i.e., > 1000) has been tested and, as a result, there is no difference in the obtained results when compared to the random walk of 1000 steps. Since the random walk is performed based on the mutation operator defined in EvoSuite, the outcomes of the random walk may vary when considering different mutation probabilities (i.e., insert, change, and remove test cases/statements probabilities) or even considering a different mutation operator. To investigate the impact of the fitness landscape properties on the

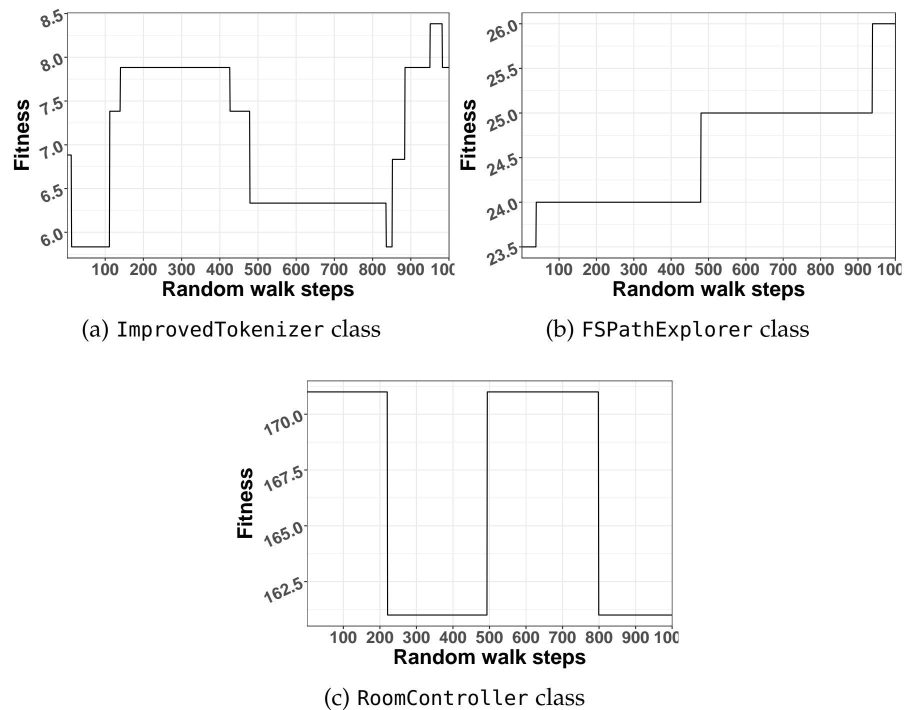
test generation, we considered the branch coverage as a proxy measurement of the quality of the generated tests, however, it is likely that results would not be similar when considering different coverage criteria. Since the objective function influences the landscape structure, any further improvements upon the current objective function might lead to change in the obtained results. Furthermore, the results reported in this study are thoroughly dependent on the six statistical landscape measurements, described in Section 2.3.3, which are sufficient and very effective in analysing the structure of the fitness landscape. However, the obtained results in this study may not generalise to other landscape measurements although we believe other measurements will most likely confirm our findings.

3.2.3 *Experimental Results*

This section presents the results of the conducted experiments and discusses the answers to the two research questions.

3.2.3.1 *RQ1.1 — What are the properties of the fitness landscape for the JUnit test generation problem when considering the WSA?*

As our investigation of the fitness landscape properties relies on the performed random walk on each CUT, we first demonstrate the results of applying a random walk on Java classes by showing various examples of random walk runs with some classes.



(a) `ImprovedTokenizer` class

(b) `FSPathExplorer` class

(c) `RoomController` class

Figure 3.1: Three different runs of a random walk performed on three classes

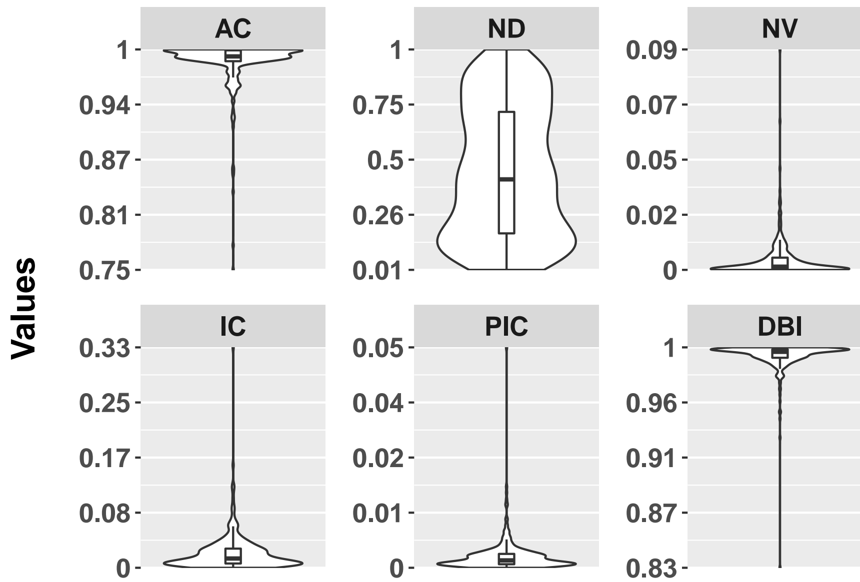


Figure 3.2: Results of the six fitness landscape measures applied on the 331 classes with the WS approach

Figure 3.1 shows three different runs of a random walk for three classes where each run represents the change in the fitness within 1000 steps of a random walk. We observe that the fitness trajectory of each random walk for each class is unlike the fitness trajectories with the other classes. In fact, this is the case when a random walk is applied on the same class several times (i.e., applying the random walk on one class more than once will most likely not result in a similar fitness trajectory). This can be interpreted as every random walk explores a different path in the landscape, which certainly results in a different sequence of fitness values except on a flat landscape. Therefore, applying multiple random walks (e.g., 30 runs) on every single class possibly gives an indication of the landscape properties.

The series of fitness values that are obtained by the random walk are analysed using the six landscape measures, and their results are shown in Figure 3.2. In general, all the measures indicate that the fitness landscape is mostly dominated by plateaus (i.e., flat areas dominate most of the landscape structure). First, the AC measure with most of the classes results in values that are higher than 0.9, which is interpreted as highly correlated fitness values of the random walk, and thus indicate a flat landscape.

In the case of neutrality measures, the ND measure shows that nearly the first 45% of the random walk steps are all neutral steps (i.e., the median is nearly 45%), which supports the evidence that plateaus dominate the landscape. Looking at the NV measure, we clearly see that there is a small number of neighbouring areas of individuals with equal fitness during the random walk, i.e., NV ≈ 3, which is interpreted as the landscape is highly dominated by plateaus since a

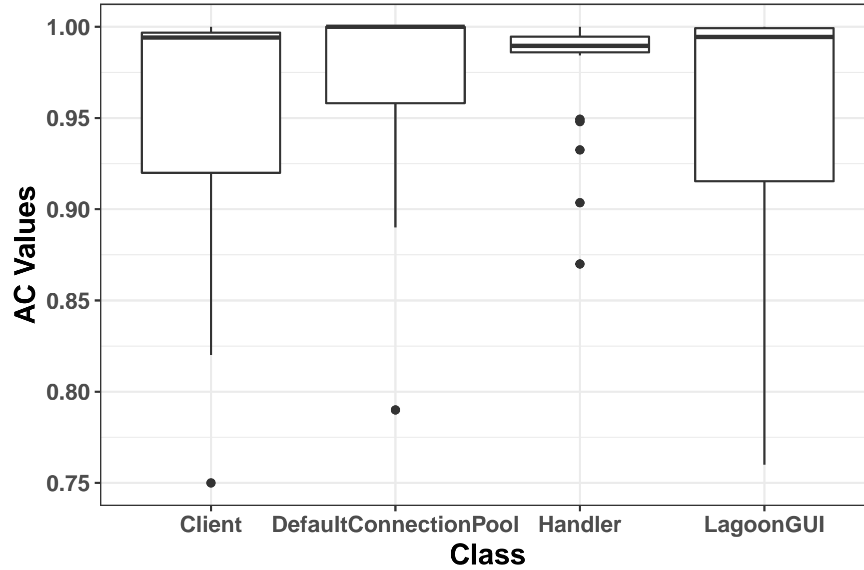


Figure 3.3: Four classes with runs that result in low AC values

low number of neighbouring areas of neutral individuals indicates the existence of plateaus in a landscape.

For the information-based measures, the IC measure is designed to analyse the ruggedness of the landscape where a value close to 1 indicates an increase in the number of peaks in the landscape, i.e., a rugged landscape. Our results show that the IC with most of our classes is almost 0.01, which means that the landscape in our scenario has a few peaks, and thus many flat areas. This is also confirmed by the PIC measure that estimates the modality in a landscape where a high PIC value is a result of a multimodal landscape, i.e., a high number of optima, whereas a landscape with many plateaus results in a low PIC value. In our case, the results reveal that the PIC values with many classes are lower than 0.005, which indicates that the landscape is highly dominated by plateaus and has few slopes. In contrast to IC and PIC, the DBI measure that analyses the variety of flat areas in the landscape where the density of peaks in the landscape is low and the flat areas are more prominent when the DBI is high, i.e., close to 1. The results of DBI measure in our scenario confirm the results of the other measures as the landscape seems to be dominated with flat areas as most of the classes result in DBI higher than 0.95.

Although the fitness landscape of a large number of classes is dominated by plateaus, several classes seemed to point to the existence of rugged areas in their fitness landscape. This can be seen with the classes where the landscape measures result in lower values such as the case with the AC (< 0.8), and higher values such as the case with the IC (> 0.3). Figure 3.3 shows an example of four classes with runs that result in low AC values. It is obvious that a high number of runs with the four classes achieve high AC values, i.e., AC > 0.95, and very

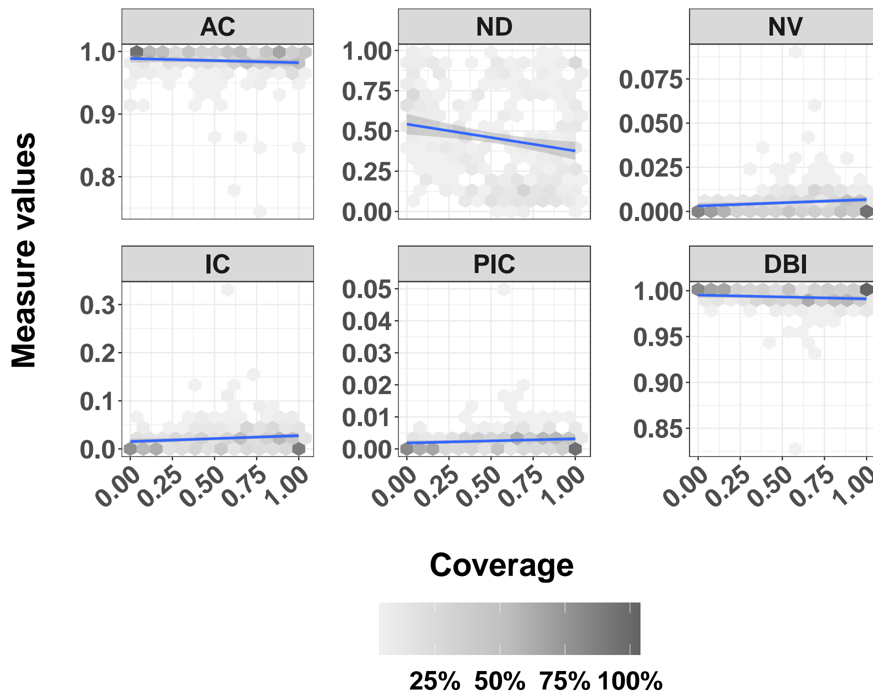


Figure 3.4: The Spearman correlation of branch coverage with each of the six measures for all the 331 classes with . The correlation coefficient of branch coverage and AC is -0.16, ND is -0.14, NV is 0.25, IC is 0.16, PIC is 0.16, and DBI is -0.17

few runs that result in AC values < 0.85. This, in fact, indicates that the landscape is still flat with these classes, and these few runs with slightly low AC values do not confirm that the landscape structure is highly dominated by rugged areas.

> ***RQ 1.1***: *Neutrality seems to dominate much of the fitness landscape for most of the classes, although there are some exceptions of few runs that indicate the presence of rugged areas in the fitness landscapes.*

3.2.3.2 *RQ1.2 — How do the fitness landscape properties affect the search behaviour when the GA considers the WSA?*

In order to understand the impact of the fitness landscape properties on the test generation, we investigate the Spearman correlation of the branch coverage and each of the landscape measures, as shown in Figure 3.4. Each hexagon represents a set of runs of different classes in which the hexagon density increases with an increase in the number of runs in the same hexagon.

There is always a significant correlation between the branch coverage and each of the measures with p-value < 0.001, but the difference lies in the strength of the correlation (i.e., the correlation coefficient). The strongest correlation, although it is a weak correlation, is observed between the branch coverage and NV (0.25) where a high coverage

value corresponds to a slightly high NV value. A high NV value means that there are more neighbouring areas of neutral individuals in the landscape, and thus few flat areas in the landscape. Based on that, and the correlation of branch coverage and NV, there is a slight possibility that the branch coverage increases with the increase of the number of neighbouring areas of neutral individuals in the landscape (i.e., classes with high branch coverage tend to have slightly higher degrees of neutrality in the landscape structure).

Looking at the correlation of the branch coverage and IC (0.16), a high branch coverage value corresponds to a slightly high IC value. Since a high IC value indicates a large number of peaks in the landscape, this suggests there is a possibility that a landscape with more rugged areas and few plateaus makes it easier for the search to cover branches. This is also shown in the correlation between the branch coverage and PIC (0.16) where a high branch coverage value somewhat corresponds to a high PIC value. Based on this correlation, the branch coverage possibly increases with a multimodal landscape as a high PIC value indicates a high landscape modality.

A negative correlation can be seen with the AC, ND, and DBI where a high branch coverage corresponds to low measure value. In the case of AC, the negative correlation between the branch coverage and AC (-0.16) suggests there is a slight possibility that the branch coverage increases with a rugged landscape, i.e., a landscape of less correlated fitness values. The correlation of branch coverage and ND (-0.14) shows that a large neutrality distance (that is, long sequences of neutral steps in the random walk) might make it difficult to improve the branch coverage. This also can be seen with the correlation of branch coverage and DBI (-0.17) where a high branch coverage slightly corresponds to a low DBI value. According to the definition of DBI, a low DBI value is an indicator of a high density of peaks and few flat areas in the landscape. Then, the negative correlation between branch coverage and DBI suggests that it is possible that branches can be covered easily with a more rugged landscape.

RQ 1.2*: Neutrality seems to have a negative effect on the GA performance, while ruggedness does not seem to decrease the GA performance.*

3.3 FITNESS LANDSCAPE ANALYSIS WITH THE MANY-OBJECTIVE SORTING ALGORITHM

In this section, we present the study of how unit test generation using MOSA is influenced by the fitness landscape. We investigate the two landscape properties (i.e., ruggedness and neutrality) when considering the objective function that is used by MOSA, and analyse their impact on the search performance. Similar to the study from the previous section, we design the study to answer the following research questions:

Table 3.1: An example of applying the random walk of 6 steps on a class with 5 branches

Step	b1	b2	b3	b4	b5
1	1.5	0.99304409	0.5	0.9375	0.2235
2	1.5	0.99304409	0.5	0.9315	0.2233
3	1.4	0.99304261	0.4	0.9315	0.2229
4	1.4	0.99304261	0.4	0.9363	0.2229
5	1.4	0.99304409	0.5	0.9363	0.2229
6	1.5	0.99304409	0.5	0.9315	0.2225

RQ 2.1: What are the properties of the fitness landscape for the JUnit test generation problem when considering the objective function used by MOSA?

RQ 2.2: How do the fitness landscape properties affect search behaviour when considering the objective function used by MOSA?

Similar to our assumption shown in Section 3.2, the landscape structure is still expected to be dominated by plateaus but they are not as large as with the WSA approach (i.e., MOSA's objective function reduces the neutrality degree in the landscape) since mutating an individual of a test case is expected to result in more changes in fitness values than with an individual of a test suite. The existence of plateaus in the fitness landscape is still anticipated to be detrimental to the GA search (i.e., reducing code coverage).

3.3.1 *Experimental Setup*

In this study, we consider a similar set of classes under test to those mentioned in Section 3.2.1.1 where the complexity of the non-trivial classes ranges from 2 to 7939 branches. We also used the EvoSuite tool as a unit test generation tool that is described in Section 3.2.1.2. The experiment procedure that we followed in this study is similar to the procedure described in Section 3.2.1.3 except that we ran MOSA instead of Monotonic GA to compare its performance against the fitness landscape measures. However, the experimental analysis in this study differs from the analysis in Section 3.2.1.4 as the objective function under consideration in this study is of a multiobjective form (i.e., each branch is considered as a single objective function), which is explained in the next section.

3.3.1.1 *Experiment Analysis*

RQ2.1 Analysis: Given a class X with n branches, a landscape walk of m steps on X is defined as a sequence $x_1, x_2, \ldots$ such that x_{i+1} is the outcome of a mutation applied to x_i where an initial individual x_1 is created randomly by applying the insertion mutation repeatedly. For each step in the walk, there will be n fitness values as there are n branches in the CUT. Table 3.1 contains an example random walk of 6 steps on a class with 5 branches, resulting in 5 fitness values for each step. Each of the landscape measures is applied to the sequence of 6 fitness values for each branch. For example, applying the AC measure on the sequence of fitness values results in 0.1668 for branch 1, 0.166415 for branch 2, 1.667 for branch 3, -0.20905 for branch 4, and 0.3064 for branch 5. To answer RQ1, we consider the distribution of these values across all branches.
RQ2.2 Analysis: In order to understand the influence of the landscape properties on the search behaviour, we want to understand how it affects the ability of the GA to cover the branches. While the overall performance of the search is usually measured in terms of the resulting branch coverage, we need to consider individual branches, where the outcome is dichotomous (i.e., either the branch is covered, or it is not). We define a *Success Rate (SR)* for MOSA for each branch as the fraction of runs in which MOSA covers the branch at least once. For example, if we run MOSA five times and in two cases branch b_i is covered by the resulting test suite, then the SR equals $2/5 = 0.4$. However, correlating the SR value for one branch with the m values of a landscape measure of that specific branch requires the use of an appropriate measure of central tendency such as the average of the m values. This results in a single value of the landscape measure that can be correlated with the SR value. For example, consider the following results of the AC measure with the two branches where each of the five results represents the AC value of a single run of random walk: $b_1 \rightarrow \{0.90, 0.92, 0.84, 0.89, 0.90\}$ and $b_2 \rightarrow \{0.84, 0.90, 0.76, 0.56, 0.97\}$. In this case, the average of the five runs for each branch is correlated with the SR value of that branch: $b_1 \rightarrow \{\text{SR} : 0.2, \text{AC} : 0.89\}$ and $b_2 \rightarrow \{\text{SR} : 1, \text{AC} : 0.81\}$.

3.3.2 *Threats to Validity*

Controlling the threats to the internal and external validity is similar to Section 3.2.2 except that the considered GA in this study is MOSA instead of Monotonic GA.

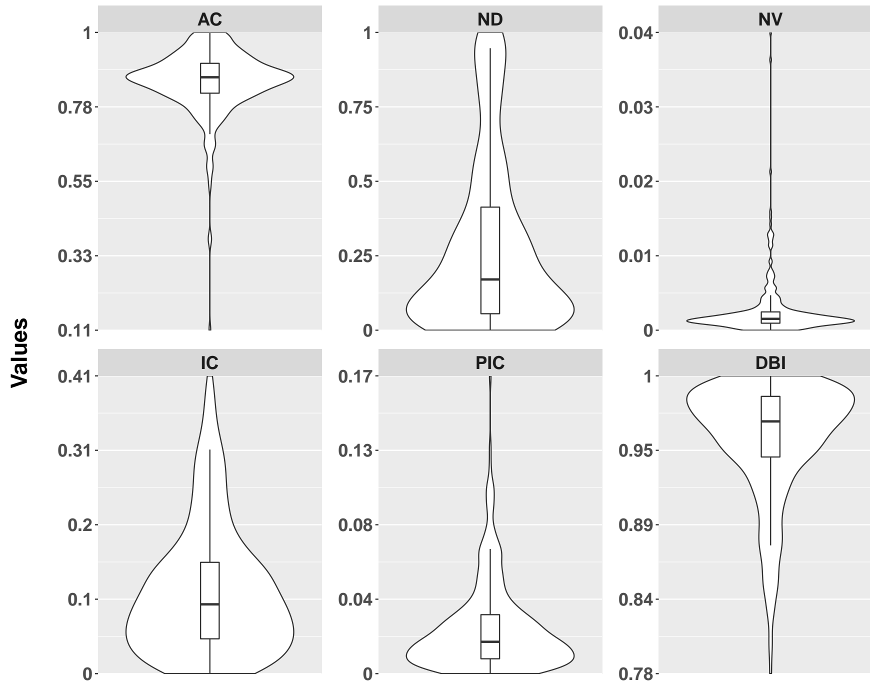


Figure 3.5: Results of the six fitness landscape measures applied on the branches of the 331 classes

3.3.3 *Experimental Results*

In this section, we present the results of the conducted experiments and discuss the answers to the two research questions.

3.3.3.1 *RQ2.1 — What are the properties of the fitness landscape for the JUnit test generation problem when considering the objective function used by MOSA?*

The results of applying the six fitness landscape measures on the series of fitness values obtained by the random walk are shown in Figure 3.5. In general, all the measures indicate that the fitness landscape is mostly dominated by plateaus, i.e., that the landscape is flat. Looking at the results of the AC measure, the AC values for most of the branches are higher than 0.6, which is interpreted as highly correlated fitness values of the random walk, and thus indicate a flat landscape.

The ND measure indicates that, on average, the first 20% steps of the random walk are all neutral steps, which is strong evidence of plateaus in the landscape. The NV measure indicates a small number of neighbouring areas of individuals with equal fitness during the random walk with most of the branches, i.e., the median of NV is ≈ 5, which also indicates a landscape with flat areas.

For the information-based measures, the IC measure is meant to characterize the ruggedness of the landscape where a value close to 1 indicates a large number of peaks in the landscape, i.e., a rugged

landscape. Our results show that the IC with many branches is close to 0.1. This indicates a landscape with a low number of peaks, and thus many flat areas. The PIC measure is an estimate of modality in the landscape where PIC $= 0$ when the landscape is flat and has no slopes, whereas PIC $= 1$ when the landscape is maximally multimodal, i.e., the number of optima is high. Our results reveal that the PIC values with most of the branches are lower than 0.04, indicating that the landscape is mostly flat and has few slopes. In contrast, the DBI measure estimates the variety of flat areas where the density of peaks in the landscape is low and the flat areas are more prominent when the DBI is high, i.e., close to 1. Our results show that most of the branches result in DBI higher than 0.9, indicating a landscape with a low density of peaks.

Although the fitness landscape of a large number of branches is dominated by plateaus, several branches seemed to point to the existence of rugged areas in their fitness landscape. This can be seen with the branches where the landscape measures result in lower values such as the case with the AC ($<$ 0.4), and higher values such as the case with the IC ($>$ 0.4), although they do not indicate a fully rugged landscape [171].

> ***RQ 2.1****: Neutrality seems to dominate much of the fitness landscape for most of the branches, although there are some exceptions of branches with more rugged fitness landscapes.*

3.3.3.2 *RQ2.2 — How do the fitness landscape properties affect the search behaviour when considering the objective function used by MOSA?*

In order to understand the impact of the fitness landscape properties on the test generation, we investigate the Spearman correlation of the SR and each of the landscape measures, as shown in Figure 3.6. Each hexagon represents a set of runs of different branches in which the hexagon density increases with an increase in the number of runs in the same hexagon.

There is always a significant correlation between the SR and each of the measures with p-value $<$ 0.001, but the difference lies in the strength of the correlation (i.e., the correlation coefficient). The strongest correlation, although it is a moderate correlation, is observed between the SR and IC (0.488); a high SR value corresponds to a high IC value. Since a high IC value indicates a large number of peaks in the landscape, this suggests that branches with slightly rugged landscape tend to be covered easily. This is also shown in the correlation between the SR and PIC (0.476) as a high SR value corresponds to a high PIC value. A large PIC value indicates a high landscape modality. This correlation between SR and PIC indicates that on a multimodal landscape it might be easier to find the test input that covers a branch.

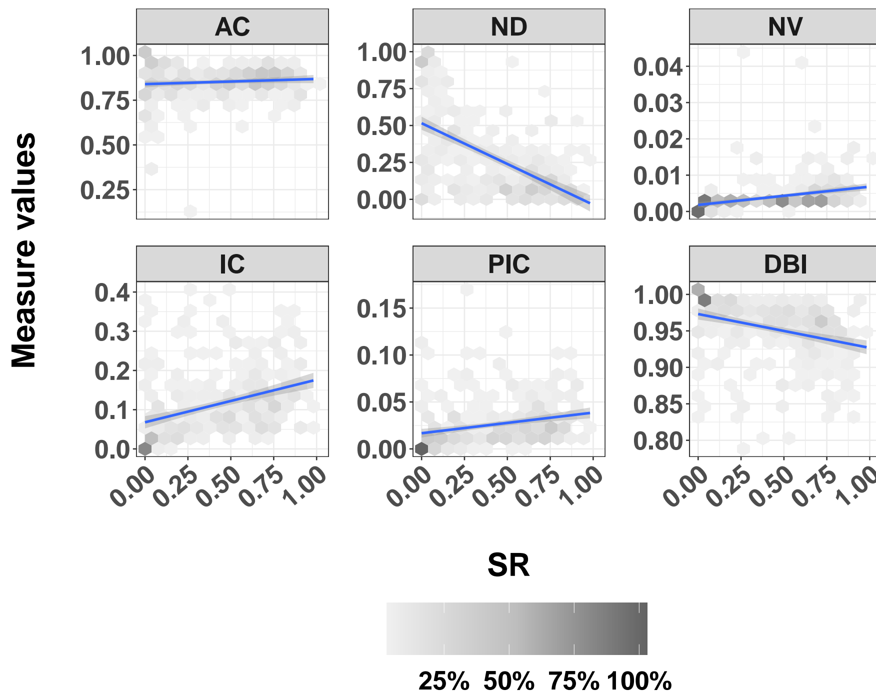


Figure 3.6: The Spearman correlation of SR with each of the six measures for all the branches of 331 classes. The correlation coefficient of SR and AC is 0.04, ND is -0.34, NV is 0.41, IC is 0.488, PIC is 0.476, and DBI is -0.481

The third measure that shows a moderate correlation with SR is the NV (0.41) where a high SR value corresponds to a high NV value. A high NV value means that there are more neighbouring areas of neutral individuals in the landscape, and thus few flat areas in the landscape. Based on that, and the correlation of SR and NV, the possibility of covering a branch becomes higher when the number of neighbouring areas of neutral individuals in the landscape is high.

A negative correlation can be seen with the two measures that estimate the variety of flat areas in the landscape, ND and DBI. A negative correlation means a high SR value corresponds to low measure value. In the case of ND, the negative correlation between SR and ND (-0.34) suggests that a large neutrality distance (that is, long sequences of neutral steps in the random walk) slightly makes it difficult to cover a branch. However, this correlation is weaker than the correlation between the SR and each of IC, PIC, and NV measures. The negative correlation between SR and DBI (-0.481) indicates that a high SR value corresponds to a low DBI value. According to the definition of DBI, a low DBI value is an indicator of a high density of peaks and few flat areas in the landscape. The negative correlation between SR and DBI suggests that such branches might become easier to cover.

Note that the correlation between the SR and the AC measure (0.04) is weaker than the correlation between SR and the other measures.

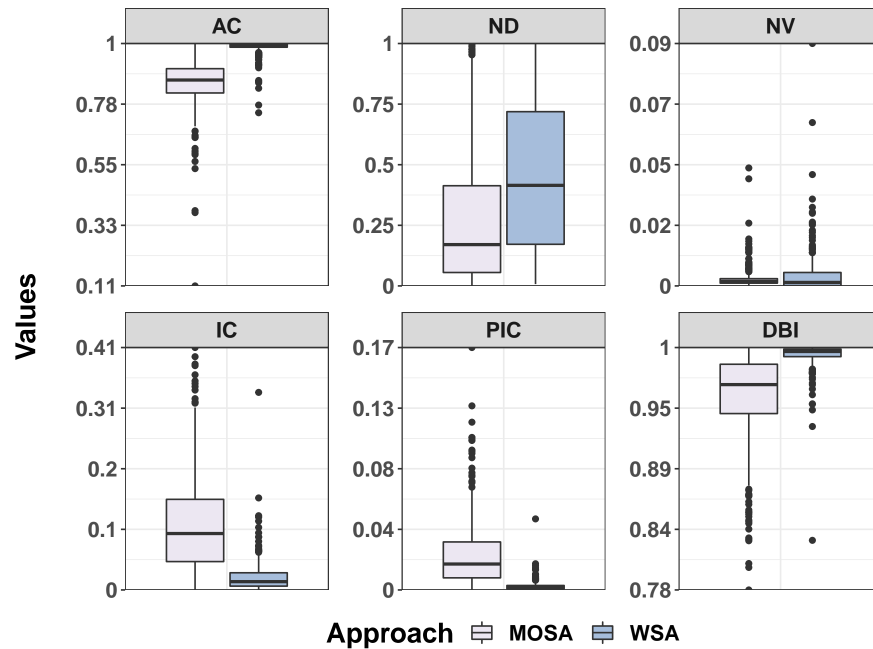


Figure 3.7: Comparison of the six fitness landscape measures applied on the random walks with the objective functions of WSA and MOSA.

The reason behind that is that measuring the correlation between the fitness values of the random walk is not always helpful in predicting the problem difficulty [90, 133], i.e., the correlation between the fitness values of the random walk does not always anticipate whether a branch is easy to cover.

The above evidence suggests that neutrality makes the search harder to find test inputs, while ruggedness seems to have a more positive effect on the search. A possible explanation is that there might be gradients in the fitness landscape that lead to the high variability of fitness during the random walk. Many fitness landscape measures might be reporting a high degree of ruggedness since gradients may appear like a rugged landscape to a blind random walk.

> ***RQ 2.2****: While neutrality seems harmful for search performance, ruggedness does not seem to decrease search performance.*

3.4 A COMPARISON OF THE IMPACT OF FITNESS LANDSCAPE PROPERTIES ON WSA AND MOSA

The study of the two fitness landscape properties (i.e., ruggedness and neutrality) with two different objective functions (i.e., functions used by WSA and MOSA) reveals that the degree of each of the two properties of the landscape structure differs with each objective function. This is confirmed by the results of analysing the landscape measures based on the series of fitness values that are obtained by the

Table 3.2: Average effect size $\hat{A}_{12}$ with p-values computed for each landscape measure with both objective functions

Measure	$\hat{A}_{12}$	p-value
AC	0.81	≤ 0.001
ND	0.67	≤ 0.001
NV	0.51	≈ 0.05
IC	0.83	≤ 0.001
PIC	0.75	≤ 0.001
DBI	0.77	≤ 0.001

random walk, as shown in Figure 3.7. In order to determine whether there is significant difference between the landscape measures with the two objective functions, we computed the Vargha-Delaney's $\hat{A}_{12}$ effect size measure [169] for each class and then calculated the average effect size for each measure as $\hat{A}_{xy}$ where x is the landscape measure with MOSA and y is with WSA. When $\hat{A}_{xy} > 0.5$, then MOSA is better in reducing landscape plateaus than WSA, and vice versa. We also considered the Wilcoxon Mann-Whitney statistical test at a level of $\alpha = 0.05$ to determine if there is statically significant difference in the landscape structure with the two functions. Table 3.2 shows the results of this statistical analysis.

Overall, the outcomes of the landscape measures indicate that plateaus dominate the landscape structure with both objective functions, and only a few cases show a possible increase in the degree of ruggedness. However, the degree of neutrality (i.e., presence of plateaus in the landscape) varies with each of the objective functions as the use of an aggregated single-objective function (WSA-based function) results in larger plateaus in the landscape than the fine-grained objective function on a branch level (MOSA-based function). This is confirmed by the results shown in Figure 3.7 and Table 3.2 where all the measures with MOSA-based function indicate a significant decrease in the presence of plateaus in the landscape when compared to the measures with the WSA-based function, except the NV measure that indicates both functions have nearly similar effect on the landscape.

The strongest difference is observed with the IC measure where the IC with MOSA-based function (≈ 0.1) is higher than its output with the WSA-based function (≈ 0.01) where a high IC indicates low flat areas in the landscape. As a neutrality measure, the ND measure shows that the number of neutral steps that are made with the WSA-based function ($\approx 45\%$) is higher than the number of neutral steps with the MOSA-based function ($\approx 20\%$). One possible conjecture of why MOSA-based function does not produce more plateaus is that applying

Figure 3.8: Comparison of the Spearman correlation of branch coverage/SR with each of the six measures based on the objective functions of WSA and MOSA.

the mutation on an individual of a test case is more likely to result in a change that affects the fitness. This is less likely when mutating an individual of a test suite as with the WSA-based function, since the chance of mutating one statement in a test case that leads to a change in the fitness is less likely with this representation. Therefore, the consequent mutations (i.e., mutations applied on successive steps of the random walk) that result in equal fitness become neutral mutations that produce plateaus in the landscape.

Investigating the impact of the landscape properties on the search performance demonstrates that a landscape with a high degree of neutrality has a more negative effect on the search, and such effect is reduced with the increase of the degree of ruggedness in the landscape. This is shown by the correlation of the branch coverage and the landscape measures, as discussed in Section 3.2.3.2 and Section 3.3.3.2 where an increase in the landscape neutrality mostly leads to a decrease in the branch coverage, and vice versa. This correlation is, in fact, stronger when considering MOSA-based function and weaker with the WSA-based function, as shown in Figure 3.8.

It is obvious that there is a high correlation between branch coverage and each of the landscape measures with MOSA-based function, except the AC measure the correlation is slightly higher with WSA-based function. The strongest correlation coefficient is observed with the IC measure where the correlation in the case of MOSA-based function (0.488) is considerably higher than the correlation coefficient in the

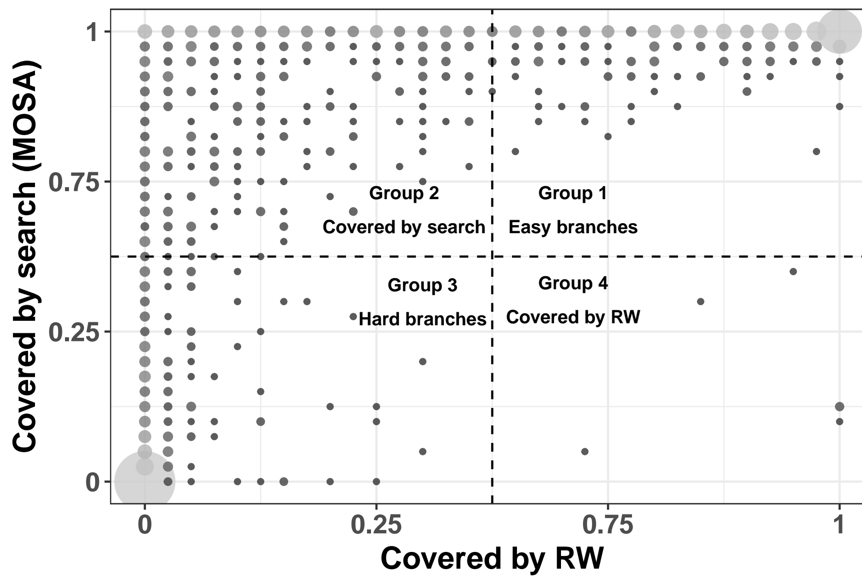


Figure 3.9: Four groups of the branches based on their coverage by MOSA and random walk (RW) where a large bubble size indicates a high number of branches

case of WSA-based function (0.16). From a landscape point of view, one possible explanation of why the search performance with the WSA approach is not as efficient as with MOSA is that the WSA-based function results in more plateaus in the landscape that negatively affect the branch coverage, and therefore the aggregation of all objective functions into a single function is detrimental to the search.

Although MOSA-based objective function is more effective in alleviating the presence of plateaus in the landscape structure, it still produces plateaus in the fitness landscape with most of the branches. This leads to the question of what causes such plateaus in the fitness landscape, and one possibe way to answer this question is by looking closely at the aspects of the code under test that possibly influence the landscape properties. Therefore, we conduct a further investigation that aims to understand how the landscape properties relate to features of Java classes, and more specifically what aspects of the underlying source code causes these properties. This is thoroughly discussed in the next section.

3.5 WHAT ARE THE UNDERLYING PROPERTIES OF SOURCE CODE THAT INFLUENCE THE FITNESS LANDSCAPE?

Having seen that landscape properties can influence the effectiveness of the search, the question now is what aspects of the code under test influence these landscape properties. In order to distinguish between cases where the search is successful simply because the problem is easy, and cases where the reason is the effectiveness of the search

Table 3.3: The average values of the six landscape measures for the branches of the four groups

Group	AC	ND	NV	IC	PIC	DBI
Easy	0.651833	114	9	0.401357	0.092291	0.871883
Search	0.828804	129	5	0.125164	0.057825	0.903901
Hard	0.898022	516	2	0.075532	0.027528	0.960161
RW	0.851833	258	4	0.098439	0.039814	0.928281

algorithm, Figure 3.9 plots the success rate of the search (MOSA) for each branch vs. the number of times that branch was covered by the random walk within the 30 repetitions. That is, the value 0 means a branch is never covered and 1 means a branch is covered by all 30 runs of either of the two techniques. Notably, a large share of the branches is either always covered (top right corner) or never covered (bottom left corner). However, there is also a substantial share of the branches on which the search is effective but the random walk is not (top left corner) – these are cases with a benign fitness landscape. Surprisingly, there are a few cases also in the bottom right corner of the plot, which were covered during the random walks, but not by the search. Based on these observations, we partition the branches into four groups based on whether they were covered by more than 50% of the runs of the search and random walk, illustrated in Figure 3.9.

Table 3.3 shows the mean values of the fitness landscape metrics for the four partitions of Figure 3.9. The metrics show that branches that are always covered (*easy* group) result in a more rugged landscape than branches that are never covered (*hard* group), where the fitness landscape seems to be dominated by plateaus. Branches covered only by the search (*search* group) appear to result in a substantially more challenging fitness landscape than those always covered (*easy*), yet the landscape metrics confirm there are fewer plateaus than in the most challenging *hard* group. Branches in the odd *RW* group are somewhere in between according to the metrics, and there are likely reasons unrelated to the fitness landscape that cause the search to fail here.

There can be multiple reasons for plateaus in the fitness landscape. A fundamental question is whether the methods containing the branches were executed in the first place – as the fitness function only considers intra-procedural information, the fitness landscape would by definition represent a plateau as long as a method is not called. Figure 3.10 shows how often the method containing the branch was actually executed during the random walk. Very clearly, methods in the *easy* group (covered by both, search and random walk) are executed far more often than in the other groups. The methods containing branches

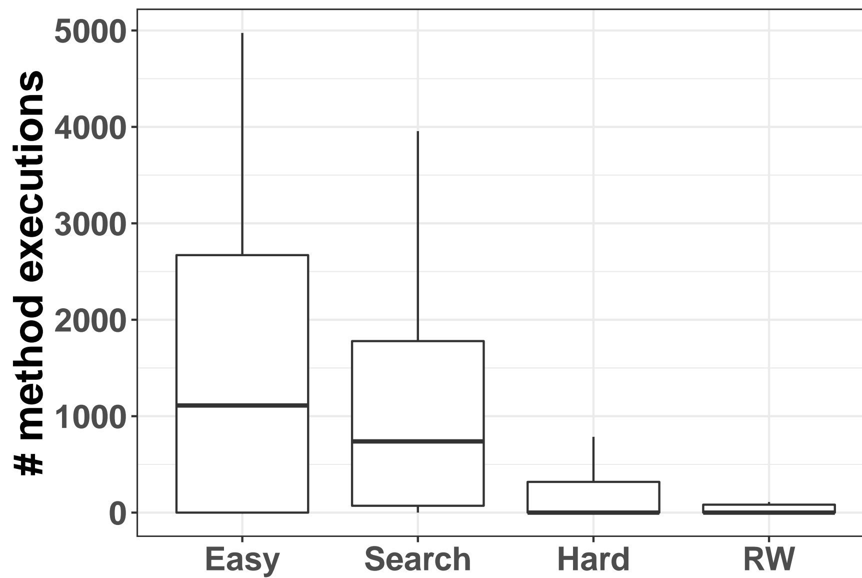


Figure 3.10: Number of method executions during the random walk for each branch in the four groups

covered by the search are executed substantially less often, but still more often than those that are hard to cover.

To understand better why methods are not called, we look at their accessibility, and whether they are methods or constructors (Figure 3.11): Notably, the *easy* branches contain substantially more constructors and public methods than branches in the *hard* and *search* groups. Interestingly, the few cases in the fourth group are all in public methods. Very notably, private methods are predominantly in the *hard* group, and thus not covered at all. Consequently, accessibility is a primary influential factor for the fitness landscape. This also suggests that a refined fitness function that considers inter-procedural distance information could transform the fitness landscape into a more benign one and thus improve the performance of search-based algorithms. An example of this case is the `getBooleanValue` method in Listing 1.1 where all its branches belong to the *hard* group.

```
public void process(Node externs, Node root) {
    Preconditions.checkArgument(
        NodeUtil.isValidCfgRoot(root), "Unexpected control flow
            graph root %s", root);
    .....
   // The following branches are not executed
    if (shouldTraverseFunctions) {
      for (DiGraphNode<Node, Branch> candidate :
          cfg.getNodes()) {
        Node value = candidate.getValue();
        if (value != null && value.isFunction()) {
          prioritizeFromEntryNode(candidate);}
      }
    }
  }
```

Listing 3.1: A method with complex paramter objects that cause an exception to be thrown in the `ControlFlowAnalysis` class

In those cases where methods are actually called, the branch distances could in principle provide a more nuanced fitness landscape. Since plateaus nevertheless dominate, there are two possible conjectures: Either the executions never even reach relevant branches that could provide a gradient but instead cause exceptions to be thrown by invalid complex parameter objects [149], or the source code is dominated by branches comparing references or boolean flags [79] which, by definition, do not provide gradients. As an example of the first case, consider the method `process` that is part of the `ControlFlowAnalysis` class from the `Closure Compiler` project in Listing 3.1. In order to reach the branches within the method, the `root` parameter requires the creation of a complex control flow graph object, and must be validated by the `checkArgument` method. This, in fact, is difficult to be created automatically as the test generator cannot initialise such a complex data structure. In this case, when the method is executed, the `checkArgument` method throws an exception, and thus the code afterwards is not reached (i.e., covering the branches becomes impossible).

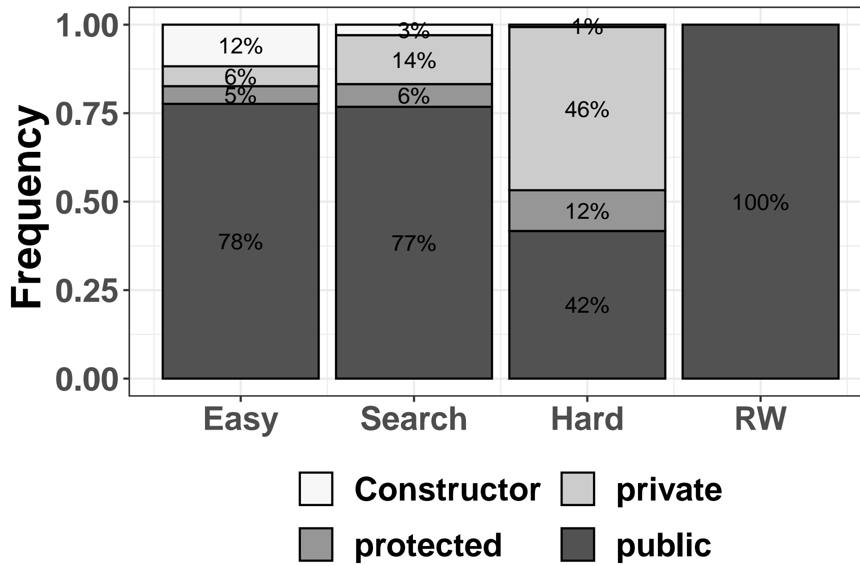


Figure 3.11: Types of methods containing each branch in the four groups

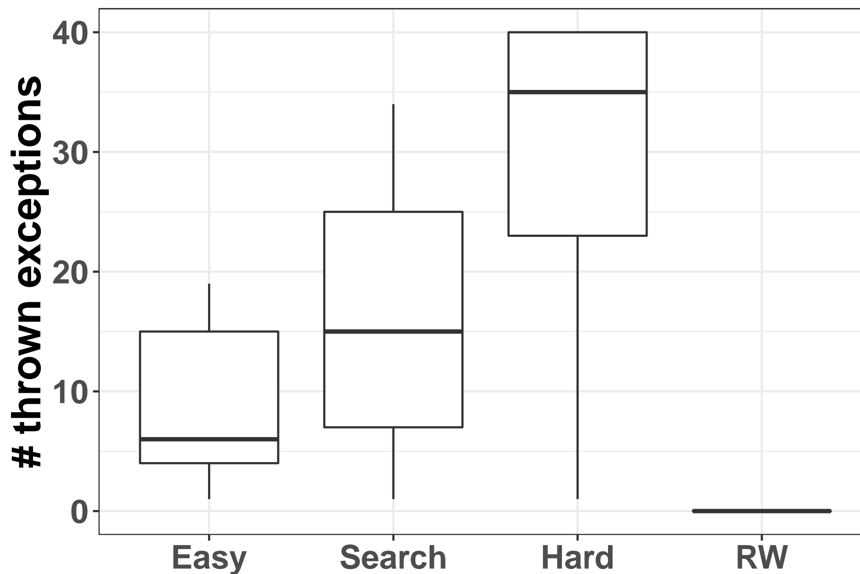


Figure 3.12: Number of exceptions thrown by methods containing each branch in the four groups

Figure 3.12 shows the number of exceptions thrown by the methods containing the branches during the random walk. As expected, the *hard* branches are in methods that are much more likely to result in exceptions (42% of methods calls), while the *easy* branches hardly result in exceptions (8% of methods calls). Branches in group *search* lie in between these two groups (28% of methods calls), and no exceptions at all were observed for the few methods only called by the random walks. Thus, exceptional behaviour clearly is an important factor. A possible cause for such exceptions are dependencies on

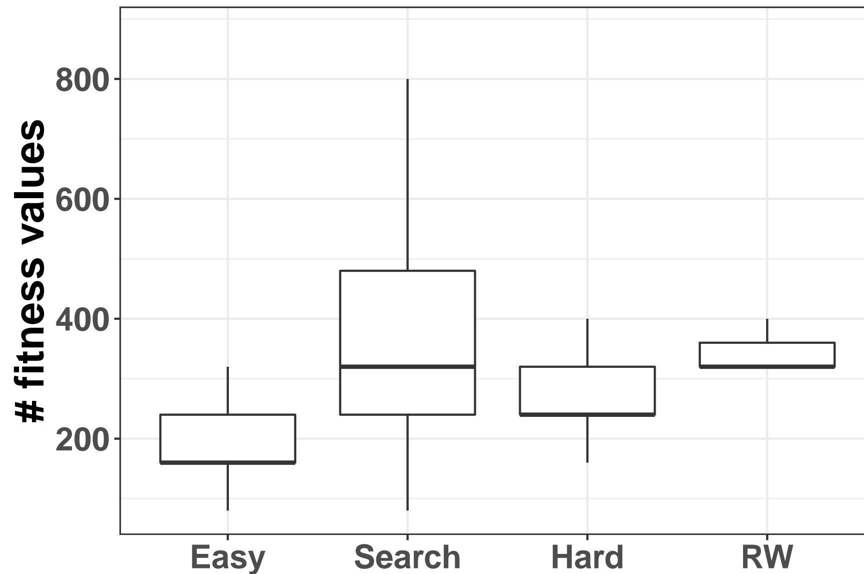


Figure 3.13: Number of discrete fitness values obtained by the random walk for each branch in the four groups

complex objects that are notoriously difficult to configure into valid configurations [149]. Methods may often have implicit preconditions on particular configurations of such valid complex objects, and the fitness function usually provides no guidance for reaching this. Fitness is typically measured only directly on the CUT and not dependency classes; a possible way to improve the fitness landscape would thus be to also consider the code underlying the dependencies, such that there is guidance towards producing valid object configurations. Alternative strategies could include improving the search operators to increase chances of producing valid object configurations, or seeding [138] valid object configurations [85, 160].

To investigate the influence of the branch types, we first look at the number of discrete fitness values observed (Figure 3.13). Intuitively, any gradients along the execution to a target branch would lead to many small variations in the fitness values. This, in fact, is observed with the branches of the *search* group, which explains why the search performs well on these branches, but the random walk does not. Interestingly, however, the number of discrete fitness values is also relatively high for branches in the group that are only covered by the random walk. A possible conjecture is that these are branches requiring specific object configurations that are very difficult to produce, and only happen by chance. Since the search tries to minimise test cases as a secondary criterion while the random walk is likely to invoke many more methods on individual objects, the chances of accidentally producing a valid object configuration then simply are higher for the random walk. It is interesting to see that branches in the *easy* group result in very few distinct fitness values; it is likely that they

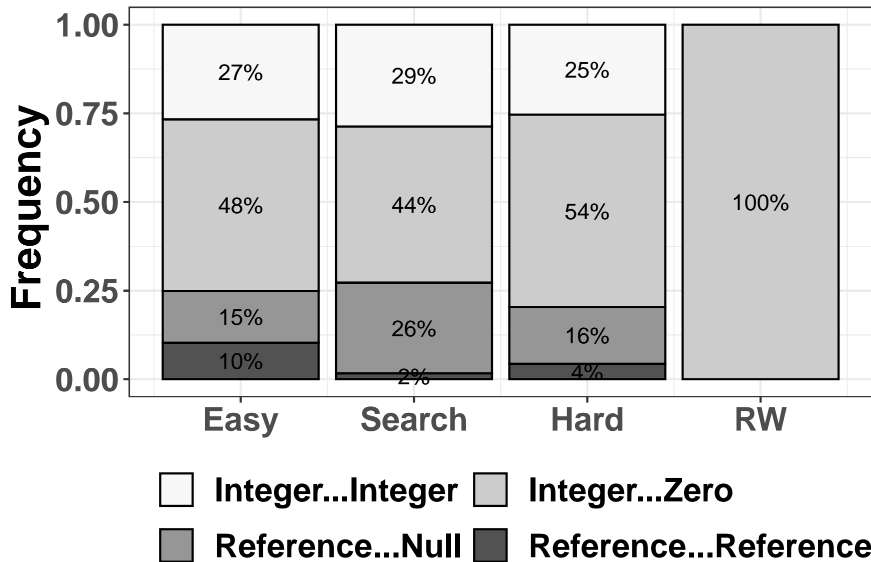


Figure 3.14: Classifications of the branch types in the four groups

are not embedded within complex conditional code constructs, and depend on well explored parameters. The ruggedness suggested by the fitness landscape analysis in these cases thus likely is not the result of gradients, but of frequently flipping if-conditions, such as easy reference or null comparisons.

To investigate this hypothesis, Figure 3.14 shows the types of if-conditions, based on their underlying Java bytecode instructions, using the classification by Shamshiri et al [150]: The most common branch type among all four groups is the "Integer-Zero" category, which is produced by the Java compiler mainly for boolean predicates such as `if(x)`, where *x* is a boolean variable (Figure 3.15a). It is well known that such boolean predicates result in plateaus in the fitness landscape [109]. The *search* group contains slightly more "Integer-Integer" branches that compare two integers (Figure 3.15a), which is the only category of branches that can possibly result in gradients. As expected, the *easy* group contains the most "Reference-Reference" branches that compare two object references (Figure 3.15c) and "Reference-Null" branches that involve the comparison of an object reference to null comparisons (Figure 3.15d), thus contributing to their low difficulty and a low number of discrete fitness values. The branches covered only by the random walk consist of only "Integer-Null" (i.e., boolean) branches, supporting the conjecture that these are if-conditions querying properties of complex objects that are difficult to produce. Consequently, many of the difficult aspects of the fitness landscape could thus potentially be overcome using testability transformations [79] to remove the boolean flags.

```
public void foo(int x){
  if(x == 5){
    // perform an operation
  }
}
```

(a) Integer-Integer branch example

```
public void foo(int x){
  boolean y = false;
  if(x == 5) y = true;
  if(y){
    // perform an operation
  }
}
```

(b) Integer-Zero branch example

```
public void foo(int x){
  Object y = null;
  if(x == 5) y = this;
  if(this == y){
    // perform an operation
  }
}
```

(c) Reference-Reference branch example

```
public void foo(int x){
  Object y = null;
  if(x == 5) y = new
      Object();
  if(y == null){
    // perform an operation
  }
}
```

(d) Reference-Null branch example

Figure 3.15: Examples of branch types classified by Shamshiri et al [150]

As a conclusion, plateaus in the fitness landscape are caused by lack of inter-procedural guidance, the difficulty of satisfying preconditions on complex objects, and the prevalence of boolean flags.

3.6 SUMMARY

Understanding the performance of evolutionary algorithms in generating unit tests requires understanding the underlying structure of the fitness landscape. To this purpose, we studied the fitness landscape in terms of its ruggedness and neutrality. Our study showed that the fitness landscape is highly dominated by neutral areas, i.e., plateaus. Branches that have a large degree of neutrality in their landscape seem to be harder to cover, whereas branches that have a small degree of neutrality in their landscape seem to be easy to cover. Indeed, for this particular search problem, ruggedness does not seem to be detrimental to the search as it indicates the existence of gradients that make a branch easy to cover by GA, and possibly harder to cover by a random walk. The main causes for the often neutral fitness landscapes we identified in our analysis are (1) accessibility of the methods that contain the branches (i.e., private methods are difficult to cover), (2) the difficulty of satisfying the preconditions of methods (i.e., calling them without causing exceptions), but also (3) the classic flag problem (i.e., boolean comparisons offering no guidance) in search-based software testing, which conforms to the findings of previous studies [109]. These insights offer a potential avenue to improving the fitness landscape, for example by adding inter-procedural distance information and testability transformations.

4 MEASURING AND MAINTAINING POPULATION DIVERSITY IN UNIT TEST GENERATION

The content of this chapter is based on work that has been published at The Symposium on Search-based Software Engineering (SSBSE) 2017 [3] and recently in SSBSE 2020 [5].

4.1 INTRODUCTION

As software testing is a laborious and error-prone task, automation is desirable. Genetic Algorithms (GAs) are frequently employed to generate tests, especially in the context of unit testing object oriented software. However, a common general issue when applying GAs is premature convergence: If the individuals of the search population all become very similar and lack diversity [44, 154, 155], then the search may converge on a local optimum of the objective function. This reduces the effectiveness of the GA, and in the case of search-based test generation, premature convergence would imply a reduced code coverage.

To avoid such premature convergence, it is important to maintain diversity in the population. Different techniques have been proposed to achieve this at the genotype and the phenotype levels [44, 155]. For example, diversity can be achieved by scaling an individual's fitness based on the density of its niche or by eliminating duplicate individuals from the population. While diversity maintenance has been extensively investigated within different domains of evolutionary algorithms (e.g., [44]), much less is known about diversity in search-based unit test generation.

In this chapter, we empirically investigate the impact of population diversity on the generation of unit tests for Java programs. More specifically, we aim to see whether increasing population diversity leads to a better GA performance, i.e., generating unit tests that achieve higher code coverage. In order to achieve that, we first adapt common diversity measurements based on phenotypic and genotypic representation to the search space of unit test cases. We then study the effects of different diversity maintenance techniques on population diversity and code coverage. Similar to the studies presented in the previous chapter, we investigate the impact of population diversity when generating unit tests using the two approaches WSA and MOSA.

Our experiments confirm that the effects of different diversity maintenance techniques on population diversity are in-line with other search domains as they lead to improve diversity during the evolution. However, higher population diversity does not lead to higher code coverage but, surprisingly, higher individual length. This negative effect can be mitigated when diversity is adapetively maintained (i.e., applying a diversity maintenance technique only when diversity drops below a certain threshold).

The chapter is organised as follows. First, we present the empirical study of the effects of population diversity on the search-based unit test generation when considering the WSA approach in Section 4.2. Then, we present a more thorough analysis and extension of studying the effects of population diversity when considering MOSA in Section 4.3. In both sections, we describe the diversity measurements that are applied to measure the diversity level during the evolution, and the diversity maintenance techniques that are used to promote population diversity during the evolution. Thereafter, we detail our experimental setup and procedure, and then discuss the experimental results and the answers to the research questions.

4.2 AN INVESTIGATION OF POPULATION DIVERSITY WITH THE WHOLE SUITE APPROACH

In this section, we focus exclusively on investigating the impact of population diversity on the generation of unit tests when using the WSA approach. In order to determine the influence of the diversity of populations of test suites, a prerequisite is to measure diversity. For this, we adapted three diversity measures that are based on the phenotypic and genotypic levels: We measure the phenotypic diversity based on the fitness entropy and test execution traces, and we define a genotypic measurement based on the syntactic representation of test suites. These diversity measures are thoroughly discussed in the following section.

4.2.1 *Measuring Population Diversity in Test Suite Generation*

4.2.1.1 *Fitness Entropy*

The entropy measure adapts the aforementioned principle of fitness entropy. It constructs *buckets* that correspond to the proportions of population that are partitioned based on the fitness values of test cases τ_s in the population μ as:

$$\text{Bucket}(f) \leftarrow |\left\{\tau_i \mid \text{fitness}(\tau_i) \in \left[f', f''\right]\right\}| \tag{4.1}$$

where f is the fitness value that partitioning is based on and τ_i is each individual in the population whose fitness value in the same fitness

Table 4.1: An example of two test suites (TS_1 and TS_2) containing different test cases (t_i) that vary in how often they execute the predicates (p_j) of the CUT.

TS_1		p_1	p_2	p_3	p_4	TS_2		p_1	p_2	p_3	p_4
	t_1	2	1	0	3		t_1	3	5	2	1
	t_2	1	3	0	2		t_2	2	3	0	2
	t_3	0	1	0	1		t_3	2	1	2	3
	P_1	3	5	0	6		P_2	7	9	4	6

interval of f (i.e., the interval of fitness values that are the same in the first five decimal points). In this case, each bucket of fitness holds the number of individuals that are in the same fitness interval. The entropy is then calculated based on each bucket of fitness as:

$$\text{Entropy} = \sum_{i=1}^{B} \frac{\text{Bucket}_i}{\mu} \cdot \log\left(\frac{\text{Bucket}_i}{\mu}\right) \tag{4.2}$$

where B is the number of buckets. As a result, a high value of entropy indicates a high diversity of the population.

4.2.1.2 *Predicate Diversity*

As the fitness value in test suite generation is mainly based on the branch distance (and other similar measurements), there is the potential issue that fitness entropy is dominated by a few statements that achieve the best coverage. For example, the fitness value considers only the minimal branch distance for each branch, but ignores all other executions of the same branch. Therefore, we define an alternative phenotypic diversity measurement that takes more execution details into account. The idea behind this measure is to quantify the diversity of the individuals based in terms of an execution profile of the conditional statements in the class under test. To illustrate this, assume two individuals (TS_1 and TS_2), where each individual consists of three different test cases (t_i) and the class under test (CUT) has four conditional statements (p_j). Each test case covers each predicate in the CUT as shown in Table 4.1.

The diversity in this case is measured based on how often each predicate is executed by each individual, e.g. p_3 is covered 4 times by TS_2, while it is not covered by any test case in TS_1. Predicate diversity is calculated by counting the number of times each predicate p_j in CUT is covered by all the test cases t_i in TS_i, resulting in vectors P_1 and P_2. The distance between TS_1 and TS_2 is calculated using the Euclidean distance between P_1 and P_2 for each pair of individuals in

the population. The use of Euclidean distance as a population diversity measure is shown to be effective in measuring the behavioural diversity, and controlling the evolution process [2, 137]. Therefore, the overall population diversity is the average of all pairwise distances between all individuals, and is calculated as follows:

$$\text{diversity}(P) = \frac{\sum_{i=0}^{|P|} \sum_{j=0, j \neq i}^{|P|} \text{dist}(T_i, T_j)}{|P| \, (|P| - 1)} \tag{4.3}$$

where P is the population of individual test suites and dist is the Euclidean distance between a pair of test suites. A high value of diversity(P) indicates a high predicate diversity level among the individuals of the population P. However, the overall diversity in the case shown in the example above is 6.92.

4.2.1.3 *Statement Diversity*

Genotypic diversity aims to measure the structural differences among the individuals of a population. In our case, the genotype are the sets of sequences of statements. As the tests in an individual are not ordered and the representation has a variable size, direct similarity measurement based on edit distance are difficult. We therefore measure syntactic difference based on the profile of statements, with normalised variable names. This is important since identical statements at different positions of tests will have different variable names. To illustrate that, consider the two test cases of two different test suites in Figures 4.1a,4.1c: Line 5 and 7 in TC_1 and Line 9 in TC_2 are identical but have different variable names; similarly Line 4 in TC_1 and Line 8 in TC_2, Line 6 in TC_1 and Line 10 in TC_2, and Line 8 in TC_1 and Line 11 in TC_2 are the same except for variable names. To normalise a statement, all variable names are replaced with a placeholder, as shown in Figures 4.1b,4.1d. For example, both Line 5 and 7 in TC_1 become identical as the variable name in Line 7 (i.e., `boolean1`) is replaced with a placeholder (i.e., `boolean0`). This is also the case with all the statements of both test cases where variable names become identical, and the difference relies on the assigned values (e.g., all the statements the define an integer variable in TC_2 have a similar variable name and they differ in the assigned values).

To calculate the distance between two test suites TS_1 and TS_2, we determine the set of normalised statements contained in both test suites, and then create two vectors representing the number of occurrences of each statement. For the two test cases in the example shown in Figure 4.1, the two vectors are: $V_1 \rightarrow$ {3:1, 4:1, 5:2, 6:1, 7:2, 8:1, 9:0, 10:0, 11:0}, $V_2 \rightarrow$ {3:1, 4:1, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1, 11:1}) where the number preceding the colon is the line number in a test case and the number after that is the number of occurrences of this statement in the test case (e.g., 5:2 in V_1 indicates that the statement in Line 5 appears twice in TC_1; both Line 5 and 7). However, the statements of

one test case that do not appear in the second test case are assigned a number of occurences of zero, for example, Line 9-11 in V_1 represent the first three lines in TC_2 that do not appear in TC_1. The distance between two test suites is calculated as Euclidean distance between these two vectors (i.e. 2.23), and the overall diversity is the average distance between all pairs of test suites in the population.

4.2.2 *Maintaining Population Diversity in Test Suite Generation*

In order to understand the impact of population diversity on the test generation, the population diversity should be maintained during the evolution process. Increasing diversity can be achieved by using different diversity maintenance and control techniques, as discussed in Chapter 2. In general, the diversity maintenance techniques are classified into non-niching and niching techniques [44]. The purpose of niching techniques is to alleviate the effects of genetic drift by segmenting the population into subpopulations to locate multiple optimal solutions. On the other hand, the non-niching techniques maintain diversity in other ways, for example, by increasing the population size, changing the selection pressure, or applying replacement restrictions. Morover, the population diversity can be controlled based on the feedback provided by the population diversity measures, for example, adaptively changing the mutation and crossover rates based on the maintained diversity level to avoid convergence to a local optimum. In our study, we consider a combination of maintenance and control techniques that are described as follows:

4.2.2.1 *Fitness Sharing*

To promote the diversity in the generated test suites, we applied the concept of fitness sharing as described in Section 2.4.2.1. The main idea behind fitness sharing is to penalise individuals that have multiple copies in the population, by dividing its raw fitness value by its niche count. This assumes that the fitness of the individuals with less number of copies in the population will be maximised to survive. However, in our setting the fitness function is minimised, therefore we replaced the division in Equation 2.13 with a multiplication, i.e., $f_i' = f_i \cdot m_i$. Replacing the division by multiplication will maximise the shared fitness of the individuals that are dominant in the population and make them less attracted for selection. On the other hand, the shared fitness of fewer individuals will be minimised and encouraged to be selected.

The niche count can be based on any type of distance measurement. In the basic version, the distance is defined as the difference between fitness values (FS-fitness); for predicate diversity the distance between the predicate execution vectors is used to determine niches (FS-predicate), and for statement diversity the distance in statement

```
1. @Test
2. public void TC1() {
3.   ArtClassA artClassA0 = new ArtClassA();
4.   int int0 = 1114;
5.   boolean boolean0 = artClassA0.foo(int0);
6.   int int1 = 392;
7.   boolean boolean1 = artClassA0.foo(int0);
8.   boolean boolean2 = artClassA0.zoo(int1);
9. }
```

(a) First test case with non-normalised statements

```
1. @Test
2. public void TC1() {
3.   ArtClassA artClassA0 = new ArtClassA();
4.   int int0 = 1114;
5.   boolean boolean0 = artClassA0.foo(int0);
6.   int int0 = 392;
7.   boolean boolean0 = artClassA0.foo(int0);
8.   boolean boolean0 = artClassA0.zoo(int0);
9. }
```

(b) First test case with normalised statements

```
1. @Test
2. public void TC2() {
3.   int int0 = (-1651);
4.   int int1 = 726;
5.   int int2 = 2438;
6.   int int3 = (-431);
7.   ArtClassA artClassA0 = new ArtClassA();
8.   int int4 = 1114;
9.   boolean boolean0 = artClassA0.foo(int4);
10.  int int5 = 392;
11. boolean boolean1 = artClassA0.zoo(int5);
12. }
```

(c) Second test case with non-normalised statements

```
1. @Test
2. public void TC2() {
3.   int int0 = (-1651);
4.   int int0 = 726;
5.   int int0 = 2438;
6.   int int0 = (-431);
7.   ArtClassA artClassA0 = new ArtClassA();
8.   int int0 = 1114;
9.   boolean boolean0 = artClassA0.foo(int0);
10.  int int0 = 392;
11.  boolean boolean0 = artClassA0.zoo(int0);
12. }
```

(d) Second test case with normalised statements

Figure 4.1: Two automatically generated example test cases to illustrate statement difference

Algorithm 4: The modified Monotonic GA applied in EvoSuite to incorporate diversity techniques

```
1  Input: Population size n, Stopping crietrion C, Crossover
     probability c_p, Mutation probability m_p, Large population
     size m
2  Output: A best individual test suite T
3  P_i ← GenerateRandomPopulation(m)
4  P ← getMostDistantIndividuals(P_i , n)
5  D_i ← MeasureDiversity(P)              ▷ initial diversity level
6  while ¬C do
7  |  D_p ← MeasureDiversity(P)           ▷ current diversity level
8  |  P ← ApplySharing(P)                 ▷ fitness sharing or clearing
9  |  c_p ← CalculateAdaptiveCrossoverRate(D_p)
10 |  m_p ← CalculateAdaptiveMutationRate(D_p)
11 |  Z ← Elitism(P)
12 |  while |Z| ≠ |P| do
13 |  |  p_1, p_2 ← RankSelection(P)
14 |  |  o_1, o_2 ← Crossover(c_p, p_1, p_2)
15 |  |  Mutation(m_p, o_1)
16 |  |  Mutation(m_p, o_2)
17 |  |  f_p = GetMinimumFitness(p_1, p_2)
18 |  |  f_o = GetMinimumFitness(o_1, o_2)
19 |  |  if f_o ≤ f_p then
20 |  |  |  Z ← Z ∪ {o_1, o_2}
21 |  |  else
22 |  |  |  Z ← Z ∪ {p_1, p_2}
23 |  |  end
24 |  end
25 |  P ← EliminateSimilarIndividuals(Z)
26 end
27 return T
```

counts (FS-statement). To apply the fitness sharing, we modify the Monotonic GA by applying fitness sharing on the population of each generation (Line 8 in Algorithm 4). This ensures that sharing is applied on the fitness of the individuals in the initial population and the population after each generation.

4.2.2.2 *Clearing*

Clearing is another niching technique that works similar to fitness sharing except that sharing is limited to the best individuals of a subpopulation. In practice, there are k individuals in a niche that are considered as winners (i.e., dominant individuals) since they result in the highest fitness values. In this case, clearing preserves the fitness

of the dominant individuals and reset the fitness of the remianing inidividuals in the same niche. Since we are minimising and the optimal fitness is zero, we set the fitness of cleared individual to a higher fitness value other than zero (e.g., `Integer.MAX_INT`). In order to define a niche, we use the distance between the predicate execution vectors for predicate diversity (CL-predicate), and the distance in statement counts for statement diversity (CL-statement). Note that the distance cannot be defined as the difference between fitness values since a niche in this case contains only individuals with a similar fitness, and applying the clearing on such individuals leads to keep all the inidividuals as there are no k individuals with higher fitness values that the others. Similar to fitness sharing, clearing is applied on the population of each generation including the initial population (Line 8 in Algorithm 4).

4.2.2.3 *Diverse Initial Population*

Population diversity is not only important to be maintained during the evolution, it is also important to maintain the diversity of the initial population. Therefore, we consider to generate a diverse initial population by generating a population of random individuals with a size m larger than the intended population size n, and then selecting the most distant n individuals from the population of m individuals based on a diversity measure. To achieve that, the distance is calculated from each individual to the others in the population, and then the overall distance is calculated for each individual as the sum of its distances from $m - 1$ individuals. For example, consider the five test suites in Table 4.2 that are generated randomly as an initial population (i.e., $m = 5$) along with their distance to each other. Assume that the intended population size is set to 3 individuals (i.e., $n = 3$), therefore we are only interested to select 3 individuals that result in the highest distance values. Looking at the total distance of each test suite in the table, we can clearly see that TS_1, TS_3, and TS_5 are the most distant individuals where TS_5 has the highest distance from the other test suites (20.6), and in contrast TS_2 is the most similar test suite to the others (10.4).

In order to calculate the distance between individuals, we use the distance between the predicate execution vectors for predicate diversity (DIP-predicate), and the distance in statement counts for statement diversity (DIP-statement). In Algorithm 4, instead of generating n random initial individuals, we generate random individuals of size m (Line 3) and then select only the most n distant individuals to form P (Line 4).

Table 4.2: An example of the distance values calculated between five test suites

	TS1	TS2	TS3	TS4	TS5
TS1	-	3.5	5	2.8	8
TS2	3.5	-	1.9	3	2
TS3	5	1.9	-	2.6	7
TS4	2.8	3	2.6	-	3.6
TS5	8	2	7	3.6	-
Total	19.3	10.4	16.5	12	20.6

4.2.2.4 *Adaptive Crossover and Mutation Rates*

Adaptively changing mutation and crossover rates based on the diversity level is thought to help avoiding convergence to a local optimum [44]. This technique is an important diversity control technique that relies on the feedback provided by the diversity measures. The crossover probability is increased when diversity is high to allow for more exploitation, whereas the mutation probability is increased when diversity is low to allow for more exploration [108]. The crossover probability is adapted (Line 9 in Algorithm 4) as:

$$P_c = \left[\left(\frac{PD}{PD_{\max}} \cdot (K_2 - K_1) \right) + K_1 \right] \tag{4.4}$$

where K_2 and K_1 define the range of P_c, PD is the current diversity level, and $PD_{\max}$ is the possible maximum diversity level. The mutation probability is adjusted (Line 10 in Algorithm 4) using the following equation:

$$P_m = \frac{PD_{\max} - PD}{PD_{\max}} \cdot K, \tag{4.5}$$

where K is an upper bound on P_m.

Since a variable size representation tends to have multiple different mutation types, i.e., adding, changing, removing statements, the probability of these three operations is adapted by increasing by a random value when the mutation probability increases, and vice versa. The two rates can be adaptively changed based on the predicate diversity measure (AR-predicate), the statement diversity measure (AR-statement), and the entropy measure (AR-entropy).

4.2.2.5 *Duplicate Elimination*

The lack of population diversity is primarily caused by a high similarity between the individuals of a population. Two individuals in the population are considered similar when their distance to each other is zero. Therefore, to ensure enough diversity in the population, one of the two similar individuals is eliminated and replaced with a new generated individual. However, the two similar individuals are evaluated, and the one with the best fitness is kept. The similarity between the individuals can be calculated based on the differences in predicates executions (DE-predicate) and the differences in the sequences of statements (DE-statement). This technique is applied after generating the offspring, and more specifically on the union of population and offspring (Line 25 in Algorithm 4).

4.2.3 *Empirical Study*

Our goal in this study is to analyse how unit test generation is influenced by test suite diversity, and whether maintaining diversity during the search has an influence on the branch coverage. We therefore designed our study to answer the following research questions:

RQ 1.1: How does population diversity change throughout evolution when considering the WSA approach?

RQ 1.2: How effective are diversity maintenance techniques with the WSA?

RQ 1.3: What are the effects of increasing population diversity in the WSA?

As our hypothesis, we predict that the default Monotonic GA does not maintain enough population diversity during evolution since the convergence behaviour seems to happen when it is applied to generate unit tests (i.e., certain maximum branch coverage is reached at an early stage of the search that indicates a possible convergence). As mentioned in Section 4.1, the convergence occurs when there is a lack of population diversity which possibly have a negative impact on the code coverage. However, applying diversity maintenance techniques with the WSA is expected to increase population diversity during the evolution, which possibly leads to better code coverage (i.e., increasing diversity motivates better exploration of the search space which might result in better test inputs that increase the code coverage).

4.2.3.1 *Experimental Setup*

As the scope of our study considers the diversity of test suites that are generated for Java programs, we used EvoSuite to generate JUnit test suites for a given Java CUT and target coverage criterion using the Monotonic GA, which applies the WSA approach. To minimise the influence of recent optimisations, we use a "vanilla" configuration [59] with only branch coverage as target criterion [139] and no test archive [140]. Besides, we used all the default settings in EvoSuite [16], including elitism (copying the best individual). Since open source Java code contains many classes that are either trivially easy to cover, or impossible to cover by EvoSuite [62], we used the selection of 346 complex classes from the DynaMOSA study [127].

4.2.3.2 *Experiment Procedure*

To better understand the influence of the population diversity on the generation of JUnit tests, we conducted an experiment that involves (i) applying each of the three diversity measures defined in Section 4.2.1 on each CUT to measure the diversity level throughout the evolution, (ii) applying each of the diversity maintaining techniques defined in Section 4.2.2 on each CUT to promote the diversity throughout the evolution. Each of diversity maintaining techniques is integrated into Monotonic GA where its performance compared to the performance of the default Monotonic GA, i.e., without using diversity techniques. Note that each of the diversity techniques is run separately, and therefore the total number of runs is 6 (i.e., a single Monotonic GA run with each of the 5 techniques and one run of default Monotonic GA). During each of the executions of EvoSuite, we tracked the three diversity measures at 30s intervals throughout the search, together with other relevant measurements (size and coverage).

Running this experiment on the corpus of 346 classes resulted in data for only 311 classes. Besides the environmental dependencies of 8 classes that are difficult to fulfil by EvoSuite, search timeout was reached with 27 classes because of the constraints that cannot be solved with 7 classes and the extra computation time needed for diversity metrics with 20 classes that makes it difficult to complete 30 runs.

4.2.3.3 *Parameter Setting*

We used the following values for the parameters of the diversity techniques based on preliminary experiments: The sharing radius σ_s = 0.1; the number of dominants for clearing is set to 1; the parameters of the diverse initial population are m = 80 and n = 50; the parameters of adaptive mutation rate are K_1 = 0.6 and K_2 = 0.8 and the adaptive crossover rate was set to K = 0.8.

4.2.3.4 *Threats to Validity*

One threat to the internal validity of this study is the random behaviour of the Monotonic GA, which is mitigated by repeating each experiment 30 times and applying rigorous statistical procedures to evaluate the obtained results. While we used EvoSuite as a JUnit test generation tool to perform the experiments, the results may differ if the considered GA is applied to other tools. Another threat to internal validity stems from the classification of the classes under test based on the development of coverage described in Section 4.2.3.5. To mitigate this threat, we validated that all the classes are classified correctly with many repetitions of the experiment.

To cope with possible threats to external validity, we used a selection of 346 complex classes from 117 open-source Java projects that are used by previous studies [127] although results may not generalise to other subjects. Threats to construct validity may possibly result from only considering the branch coverage as an indicator of how increasing population diversity affects the GA performance. The use of other criteria may not lead to similar results as to branch coverage. Past experiments with EvoSuite considered search durations of 1-2 minutes, however, this might not be enough to investigate the effects on convergence. Therefore, we used a substantially larger search budget of 30 minutes. It should be noted that we considered running an experiment with the search budget higher than 30 minutes (e.g., 60 minutes), and as a result, there is no obvious difference in the obtained results.

In regards to population diversity maintenance, the results reported are limited to the previously mentioned diversity maintenance techniques used in the experiments. These techniques are found to be effective in improving population diversity during the evolution when considered with different problem domains [44]. They are also representative of the general classification of diversity techniques (i.e., niching and non-niching techniques). However, the results obtained from applying these techniques on the considered corpus of Java classes with the Monotonic GA may not be generalised to other diversity maintenance techniques.

4.2.3.5 *How does population diversity change throughout evolution when considering the WSA approach?*

A first step towards understanding the influence of population diversity on the evolution is to measure how the diversity is changed throughout the evolution. For that, we measure the diversity in the Monotonic GA to get an idea of whether Monotonic GA is able to maintain a high level of diversity during the search. However, as different CUTs result in different patterns of coverage during the evolution,

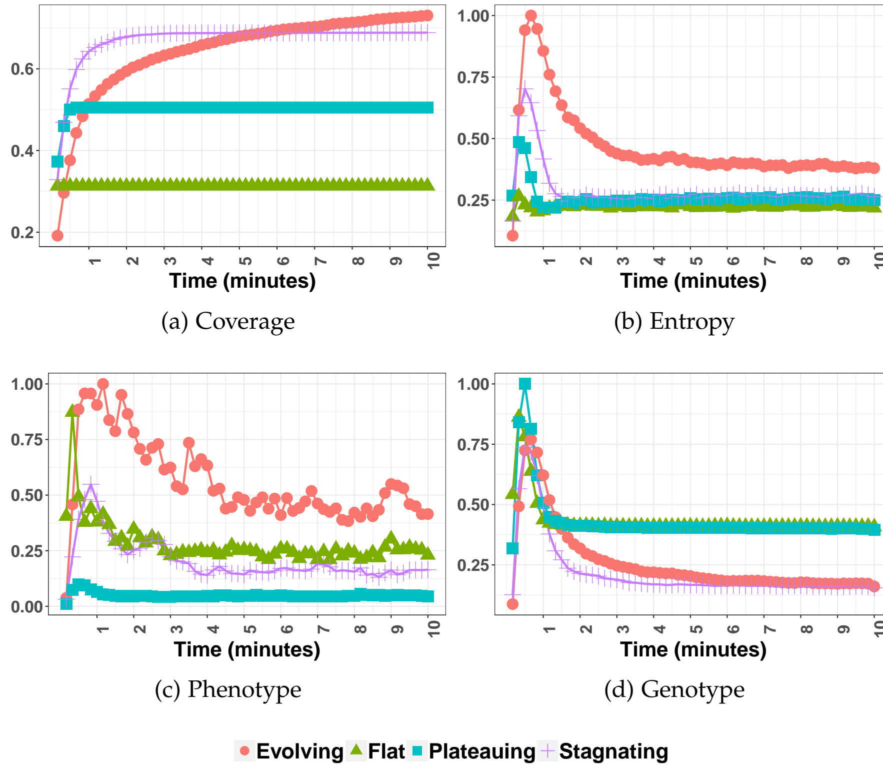


Figure 4.2: Average values for coverage and population diversity throughout evolution in Monotonic GA for the four groups of CUTs.

it is necessary to look at the diversity based on the different coverage patterns.

Past experiments with EvoSuite considered search durations of 1-2 minutes; in order to study the effects on convergence we use a substantially larger search budget. We first performed runs of 30 minutes to observe the development of coverage (the best coverage achieved in the current population) over a long period of time and we used this data to classify CUTs into four disjoint groups as follows:

- **Evolving** contains CUTs where the coverage after 30 minutes is higher (by more than 0.01) than after 10 minutes (84 CUTs).
- **Flat** is the set of CUTs where the coverage never changes (45 CUTs).
- **Stagnating** contains CUTs for which the coverage stagnates after 10 minutes: it is higher than after 2 minutes, but increases by less than 0.01 from 10 minutes to 30 minutes (36 CUTs).
- **Plateauing** contains the remaining CUTs for which the coverage after two minutes is constant (146 CUTs).

We then started 30 runs for each CUT with a search budget of 10 minutes and applied our proposed measures defined in Section 4.2.1

to measure the diversity level for each of the four groups, as shown in Figure 4.2. It is obvious that the diversity behaviour is different among the four groups, and the difference can be seen with the three diversity measures. Overall, there is a decrease in the diversity after one minute search time regardless to the state of the coverage; whether coverage keeps growing or stagnates.

For the *evolving* group, the coverage keeps growing throughout the entire 10 minutes, but the population diversity drops sharply after one minute according to the three diversity measures. Despite the decrease in diversity, this group still achieves better semantic diversity level when compared to the other groups as shown by the entropy and phenotye diversity, but less genotype diversity level than other groups except the *stagnating* group.

In the case of the *flat* group, the coverage is the lowest among all other groups. Similar to the *evolving* group, the population diversity does not improve after one minute although the entropy measure shows almost a constant diversity level during the entire search. As the branch coverage indicates few predicates are covered with this group, we expect a low behavioural diversity (i.e., low entropy and phenotype diversity). This is confirmed by the entropy measure despite the slight difference in entropy diversity between this group and the *stagnating* and *plateauing* groups, but not by the phenotype measure. In the genotype diversity, the structural differences among the individuals becomes constant after one minute.

For the *stagnating* group, we can clearly see that the diversity level with all the diversity measures drops once the coverage increase slows down. A possible reason to the drastical decrease in diversity is that when the coverage stagnates, individuals become moslty similar in terms of fitness and predicates execution that lead to low entropy and phenotype diversity. In this case, when two individuals result in a similar fitness value, the EvoSuite's ranking mechanism prefers the shorter one (i.e., the one with few number of statements), which thus results in low genotype diversity.

Looking at the *plateauing* group, the diversity level becomes constant during the search once the coverage convergences, as shown with the three diversity measures. In terms of behavioural diversity, this group is the lowest in the phenotype diversity while it is almost the lowest in the entropy diversity. This indicates that only very few predicates are covered when compared to the predicates execution by other groups. Also, the structural differences between individuals is similar to the *flat* group, which indicates that the constant coverage leads to a constant genotype diversity.

RQ1.1: *Monotonic GA using WSA approach does not maintain sufficient diversity in the population during the evolution.*

Table 4.3: Average diversity over time when applying diversity maintenance techniques based on different distance measures, and the average effect size $\hat{A}_{12}$.

Technique	Phenotype	$\hat{A}_{12}$	Genotype	$\hat{A}_{12}$	Entropy	$\hat{A}_{12}$
AR (fitness)	0.367	0.49	0.3496	0.46	0.4626	0.49
AR (predicate)	0.3367	0.48	0.3429	0.44	0.4537	0.43
AR (statement)	0.3694	0.52	0.3628	0.54	0.4853	0.54
CL (predicate)	0.5001	0.53	0.3667	0.63	0.4701	0.54
CL (statement)	0.5507	0.58	0.3922	0.74	0.4911	0.61
DIP (predicate)	0.533	0.56	0.3689	0.51	0.4481	0.46
DIP (statement)	0.4413	0.53	0.4095	0.72	0.4611	0.49
DE (predicate)	0.8616	0.57	0.9594	0.82	0.9452	0.82
DE (statement)	0.8627	0.61	0.9437	0.85	0.9519	0.83
FS (fitness)	0.858	0.71	0.9498	0.80	0.9383	0.90
FS (predicate)	0.627	0.59	0.7089	0.68	0.8192	0.81
FS (statement)	0.4701	0.55	0.5666	0.69	0.8295	0.82
Monotonic GA	0.3009	-	0.2178	-	0.3537	-

4.2.3.6 *How effective are diversity maintenance techniques with the WSA?*

We observed in RQ1.1 that the Monotonic GA using WSA does not maintain well diversity level throughout the evolution. This raises a question of whether applying the diversity maintenance techniques leads to an increase in the population diversity. Therefore, we apply the diversity maintenance techniques presented in Section 4.2.2 on each CUT.

Table 4.3 summarises the results of the average diversity among the four groups with each technique based on different distance measures (i.e., predicate, statement, and fitness). To observe the similarity in the diversity achieved by the Monotonic GA and each of the diversity techniques, we consider the effect sizes computed with Vargha-Delaney's $\hat{A}_{12}$ measure [169]. The effect size estimates the probability that a single run of Monotonic GA with a diversity technique results in higher diversity than the default Monotonic GA (i.e., with no diversity techniques). The computation of this measure is conducted using the diversity that is calculated as the average diversity achieved in each run of the technique. Since the population diversity is measured using three different measurements, there are three different $\hat{A}_{12}$ values with each technique. However, a value of $\hat{A}_{12} = 0.5$ indicates that there is no difference in the diversity level achieved by both techniques, whereas a value of $\hat{A}_{12} = 1$ indicates all the runs of the Monotonic GA with a diversity technique results in better diversity level than the

default Monotonic GA. In contrast, a value of $\hat{A}_{12} = 0$ is an indicator of a better diveristy level with the default Monotonic GA.

From the table, we can obviously see that all the diversity techniques are able to promote higher population diversity than the default Monotonic GA, as confirmed by the three diversity measures. However, the increase in diversity level differs among different techniques where the fitness sharing (FS) and the Duplicate Eliminate (DE) achieve the highest diversity among all the diversity techniques. Looking at the entropy measure, the two versions of DE (i.e., DE-predicate and DE-statement) increase diversity the most although the fitness-based FS (FS-fitness) result in the largest effect size. This is also the case with the phenotype measure where the two versions of DE result in the highest diversity level but the FS-fitness has the largest effect size. The genotype measure also confirms that the highest diversity is achieved by the DE-predicate, DE-statement, and FS-fitness where the DE-statement has the largest effect size.

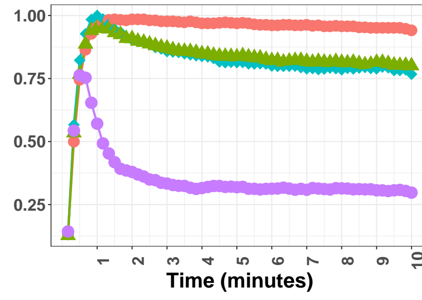
To investigate the diversity behaviour with each technique, Figure 4.3 shows the diversity level achieved throughout the evolution for each of the five techniques. For each technique, the resulting diversity is shown by the three diversity measures (i.e., entropy, phenotype, and genotype). Looking at the niching techniques, the FS is obviuosly better than Clearing (CL), which is also confirmed by the results in Table 4.3. The FS-fitness maintains better diveristy level during the evolution than the other two versions of FS (i.e., FS-predicate and FS-statement). This can be clearly seen with the three diversity measures in Figures 4.3a $-$4.3c where the population diversity with FS-fitness is higher than FS-predicate and FS-statement, especially with the phenotype and genotype measures where diversity keeps increasing with FS-fitness. For the other two versions of FS, the use of predicate-based distance (FS-predicate) leads to better diversity level during the evolution than the use of statement-based distance (FS-statement), except the entropy measure that shows almost a similar diversity level with both versions. This is also the case in CL where the entropy and genotype measures (Figures 4.3d and 4.3f) show that there is a trivial difference between the diversity achieved by CL-predicate and CL-statement where the latter results in slightly higher diversity during the evolution. The phenotype measure (Figures 4.3e), however, indicates that the population diversity with CL-statement is almost better than with CL-predicate.

Similar to CL, the Diverse Initial Population (DIP) demonstrates a similar diversity behaviour during the evolution where DIP-statement results in almost better diversity level than DIP-predicate, as shown by the entropy and genotype measures (Figures 4.3h and 4.3j). However, the phenotype measure (Figures 4.3i) indicates that DIP-predicate is to some extent able to achieve higher diversity than DIP-statement. For the Adaptive Crossover and Mutation Rates (AR) where the rates of

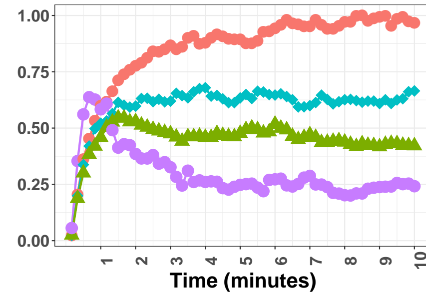
crossover and mutation are adaptively changed based on the diversity level during the evolution, there is no obvious difference between the diversity achieved by using the three measures (i.e., AR-entropy, AR-predicate, and AR-statement) although the results in Table 4.3 shows that the diversity with AR-statement is slightly higher than the other two versions of AR.

More interestingly, the difference in the diversity achieved by the default Monotonic GA and the two versions of DE is clearly high. One possible reason of why this technique promotes higher diversity when compared to the other techniques is the removal of similar individuals that result from selecting shorter tests when the coverage convergence and replacing them with newly generated individuals; EvoSuite prefers to select shorter tests when coverage convergences, which leads to decrease in the diversity level as shown with default Monotonic GA. However, the three diversity measures (Figures 4.30 – 4.3q) indicate that there is no obvious difference in the diversity level when considering both distances; DE-predicate and DE-statement.

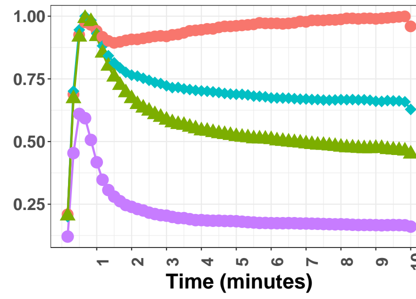
RQ1.2: *Diversity maintenance techniques are able to increase diversity during the evolution, and Fitness Sharing and Duplicate Eliminate achieve the highest diversity level.*



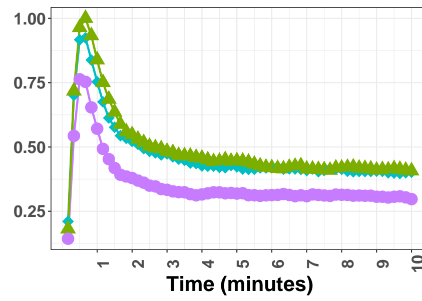
(a) FS – Entropy



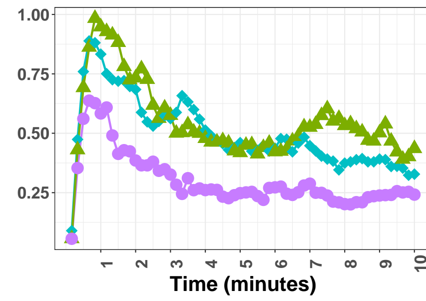
(b) FS – Phenotype



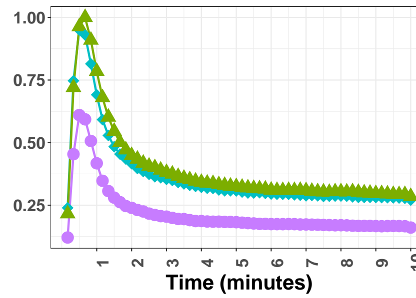
(c) FS – Genotype



(d) CL – Entropy

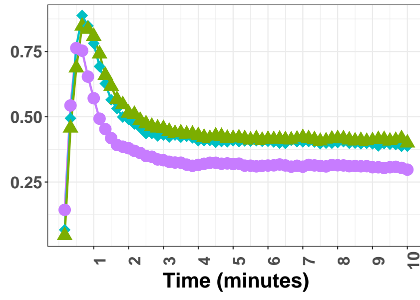


(e) CL – Phenotype

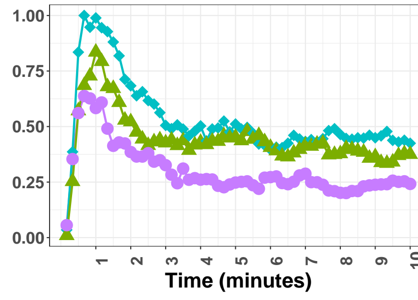


(f) CL – Genotype

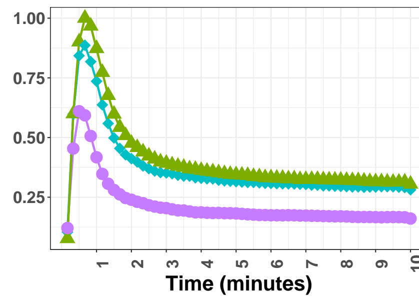
● Fitness ● Monotonic GA ◆ Predicate ▲ Statement



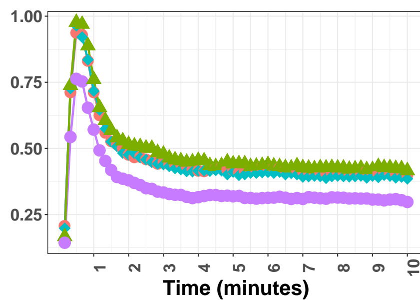
(h) DIP – Entropy



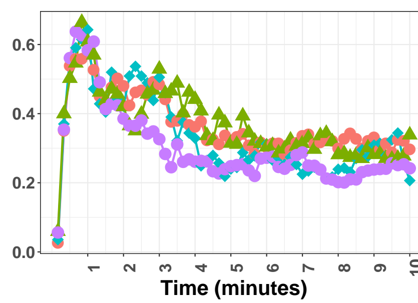
(i) DIP – Phenotype



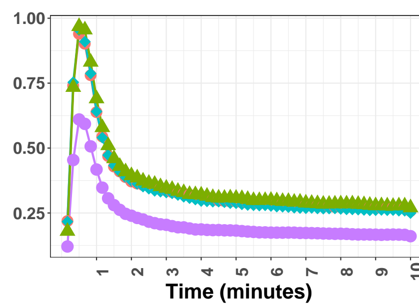
(j) DIP – Genotype



(k) AR – Entropy

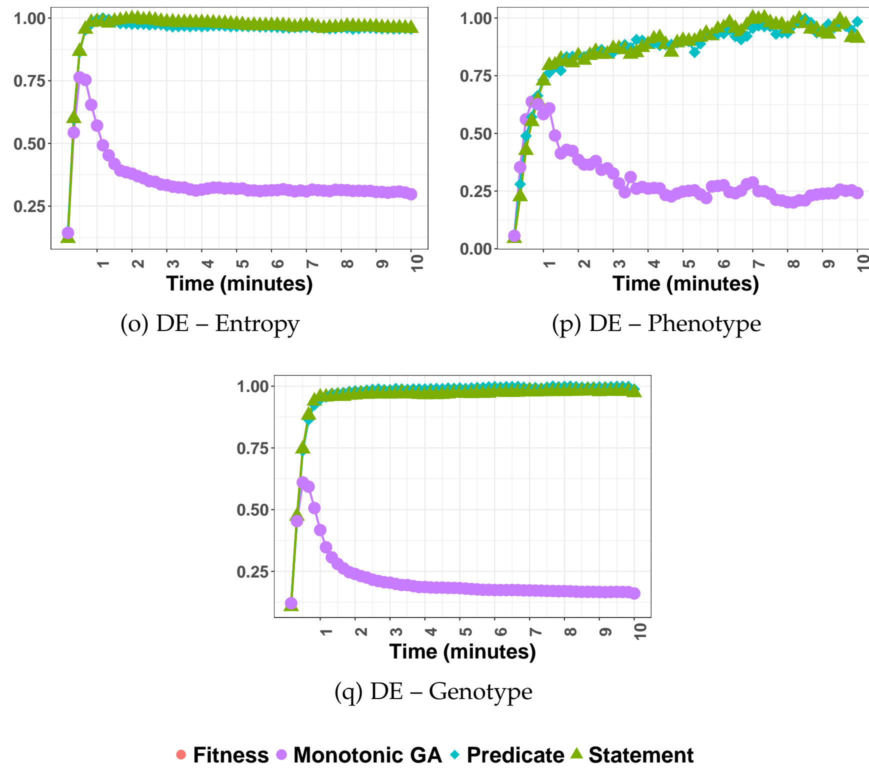


(l) AR – Phenotype



(m) AR – Genotype

● **Fitness** ● **Monotonic GA** ◆ **Predicate** ▲ **Statement**



(o) DE – Entropy

(p) DE – Phenotype

(q) DE – Genotype

Figure 4.3: Diversity throughout the evolution with Monotonic GA and the five diversity maintenance techniques based on different distance measurements.

4.2.3.7 *What are the effects of increasing population diversity in the WSA?*

Ultimately, code coverage is the goal of the search in our context, and thus the main point of interest is whether increasing diversity leads to the desired effect on higher coverage. Therefore, we look at the achieved coverage throughout the evolution and also the average size of the individuals in the population to gain a better understanding on the impact of diversity on the performance of Monotonic GA. Figure 4.4 shows the results of the best coverage in the population and the average length of all test suites in the population for Monotonic GA with and without each of the five diversity maintenance techniques during the evolution. In general, the Monotonic GA with all diversity techniques either result in lower or similar coverage to the default Monotonic GA. Figure 4.4a and 4.4c demonstrate that both niching techniques (i.e., FS and CL) result in lower branch coverage than the default Monotonic GA where the $\hat{A}_{12}$ values with the FS-entropy, FS-predicate, and FS-statement are $0.46, 0.44$, and 0.38, respectively, whereas the $\hat{A}_{12}$ value is 0.45 with CL-predicate and 0.39 with CL-statement.

For the remaining techniques, there is no obvious difference between the coverage achieved by the default Monotonic GA and the coverage

Table 4.4: Spearman correlation between the diversity achieved by the five diversity techniques based on different distance measures and length.

Technique	Phenotype	Genotype	Entropy
AR (fitness)	0.4999	0.6175	0.7168
AR (predicate)	0.5117	0.6278	0.6810
AR (statement)	0.5104	0.6390	0.6690
CL (predicate)	0.5589	0.6228	0.6723
CL (statement)	0.5546	0.6315	0.6620
DIP (predicate)	0.5378	0.5815	0.7059
DIP (statement)	0.5506	0.5783	0.7025
DE (predicate)	0.4406	0.6890	0.7176
DE (statement)	0.4526	0.6376	0.7300
FS (fitness)	0.5502	0.5915	0.7426
FS (predicate)	0.4602	0.6196	0.6915
FS (statement)	0.5265	0.6007	0.6603
Monotonic GA	0.5740	0.6195	0.7200

achieved by the GA with each of the diversity techniques, as shown in Figures 4.4e, 4.4g, and 4.4i. In the case of DIP technique, both DIP-predicate and DIP-statement have a similar $\hat{A}_{12}$ value that is 0.49. While the three versions of AR have the $\hat{A}_{12}$ values of 0.51 with AR-entropy, 0.48 with AR-predicate, and 0.47 with AR-statement. The $\hat{A}_{12}$ value is 0.51 with DE-predicate and 0.49 with DE-statement.

Looking at Figure 4.4, we can clearly see that increasing diversity leads to an increase in the length (i.e., the total number of statements in a test suite). In fact, the length for the techniques with higher diversity show longer individuals, as shown by FS (Figure 4.4b) and DE (Figure 4.4j). This conjecture is supported by looking at the correlation between diversity and length; Table 4.4 presents the Spearman correlation between the average diversity throughout evolution and average test suite length in the population, averaged per class, for each diversity technique with different distance measures. The entropy measure shows a strong correlation between the diversity achieved by each technique and the length where the highest correlation is observed with FS-fitness and DE-statement. However, the correlation is slightly lower between diversity and length in terms of genotype measure, and it is even lower but still moderate correlation in the case of phenotype measure. The larger individuals affect the exploration, since on average only one statement in a test case is mutated, independently of the length.

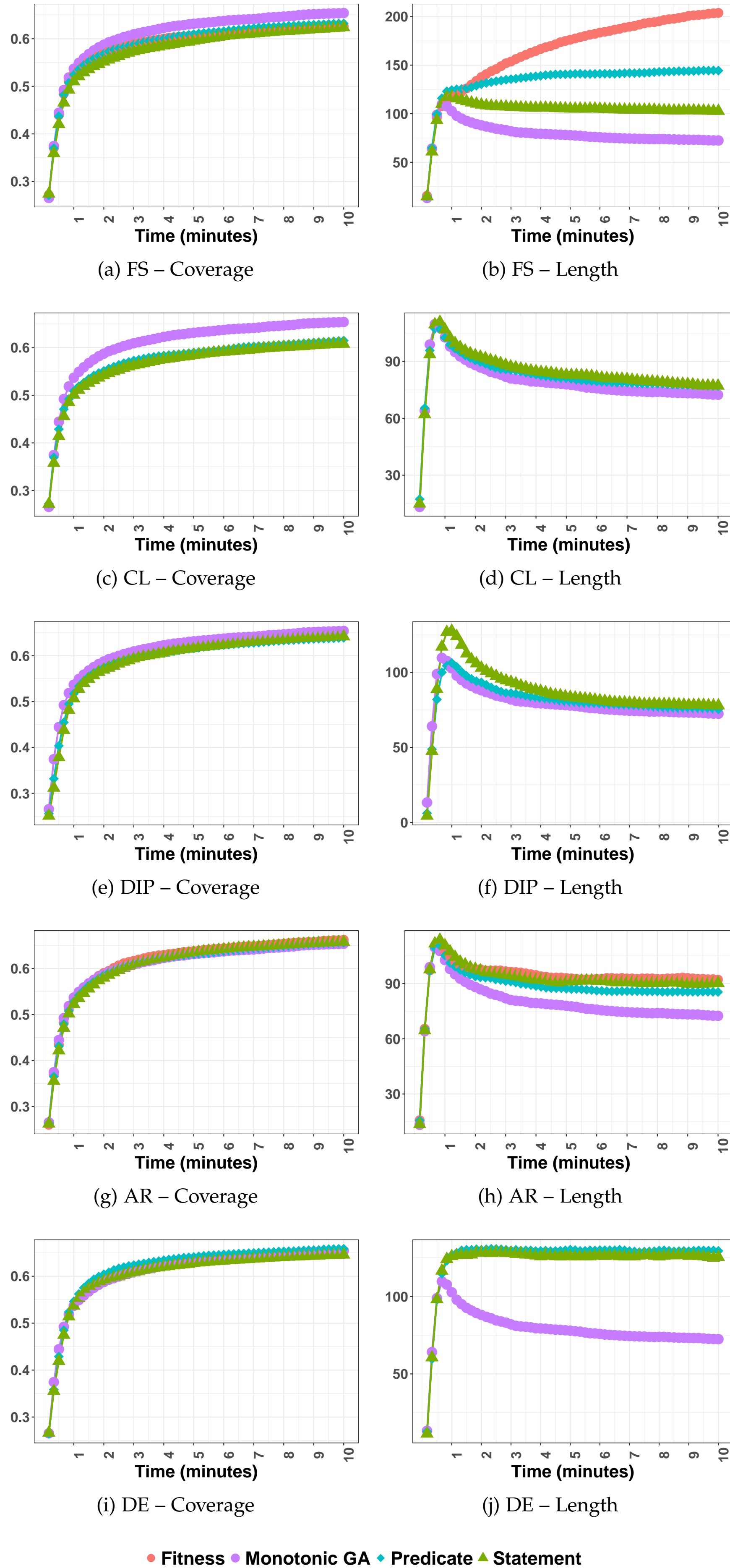


Figure 4.4: Coverage and length over time with Monotonic GA and the five diversity maintenance techniques based on different distance measurements.

As applying a diversity maintenance technique has a negative impact on the achieved branch coverage, we consider to apply an *adaptive* version of a naive application of a diversity technique. That is by applying the diversity technique only when diversity drops below a certain threshold, and once the diversity level exceeds the threshold, the diversity technique is not applied. We empirically determined a threshold of 60%. The idea behind this approach is to measure the population diversity level within each generation (Line 6 in Algorithm 4), and whenever the diversity level drops below 60% of the initial population diversity level (i.e., $D_p \leq 0.6 * D_i$), the diversity technique is then applied until the level exceeds the threshold (i.e., $D_p > 0.6 * D_i$). As a diversity technique, we focus on the fitness sharing (using all three diversity measures), since RQ2 and the previously discussed results in this RQ suggested that this is the most effective technique to increase diversity. Therefore, we compare the performance of the Monotonic GA when considering the three versions of naive FS and, in addition, three versions of adaptvie FS (i.e., AFS-fitness, AFS-predicate, and AFS-statement).

To better understand the difference in the impact of the naive and adaptive FS on the performance of the Monotonic GA, we look at the coverage and length during the evolution for the Monotonic GA with and without the naive and adaptive application of FS for each of the four groups of CUTs, as shown in Figure 4.5. For the *flat* group, the three adaptive versions of FS, especially the AFS-predicate, result in higher coverage than the default Monotonic GA and the naive FS. The negative effect of increasing diversity on the length is reduced when applying the adaptive approach with this group; the non-adaptive versions of FS lead to longer individuals than the adaptive versions where FS-predicate results in the largest length and, in contrast, AFS-fitness has the smallest effect on length.

For the *evolving* group, it is obvious that the difference in the achieved coverage among all the configurations is trivial although the default Monotonic GA results in the highest coverage. The length plot shows that the largest size is achieved by the FS-predicate whereas the default Monotonic GA results in the smallest size during the evolution. However, the small effect of the adaptive diversity on length (i.e., reducing length more than the naive diversity) is not distinguishable with this group, as the FS-statement leads to slightly lower length than the three versions of the adaptive FS.

Looking at the *stagnating* group, the default Monotonic GA leads to a higher coverage and smaller size than the naive and adaptive versions of FS. The three versions of the adaptive FS achieve slightly higher coverage than the naive FS. However, the effect on the length is slightly similar to the effect observed in the *evolving* group except that the AFS-statement leads to a little lower length than the FS-statement.

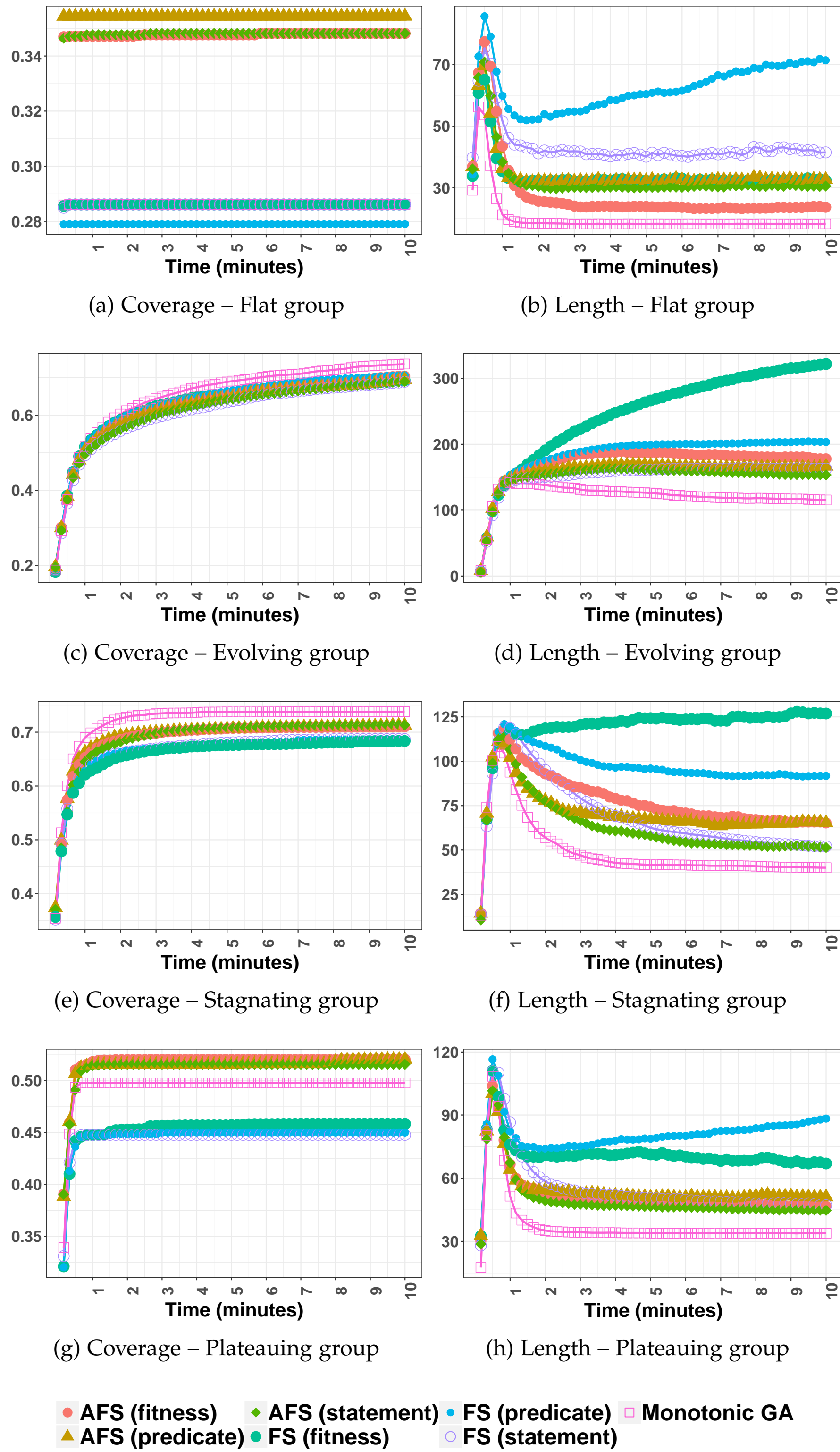


Figure 4.5: Coverage and length over time with Monotonic GA, fitness sharing (FS), and adaptive fitness sharing (AFS) per four groups of CUTs

Table 4.5: Number of classes where adaptive fitness sharing has an increased/decreased/equal coverage compared to Monotonic GA, the average effect size $\hat{A}_{12}$ and the number of classes for which this comparison is statistically significant ($\alpha = 0.05$).

Technique	increased coverage			decreased coverage			equal
	#classes	#sig.	$\hat{A}_{12}$	#classes	#sig.	$\hat{A}_{12}$	#classes
Fitness	51	5	0.52	112	29	0.47	148
Predicate	37	1	0.51	116	53	0.49	158
Statement	43	3	0.51	138	41	0.46	130

For the *plateauing* group, the three versions of the adaptive FS result in better coverage than the default Monotonic GA, which also results in higher coverage than the three versions of the naive FS. In this group, a similar observation to the effect on length in *flat* group is found with this group where the adaptive versions of FS leads to smaller size than the naive FS except that the FS-predicate and FS-fitness result in larger length than other versions, and FS-fitness leads to slightly similar length to the AFS-predicate.

We have seen so far that there are cases where the adaptive FS leads to higher coverage than the default Monotonic GA. To see whether adaptive FS is always beneficial, Table 4.5 shows the number of classes where it increases, decreases, or results in equal coverage with Monotonic GA. In general, the number of classes that decrease coverage is higher than the number of classes that increase coverage, but the number differs among the three techniques. It is obvious that the adaptive fitness-based sharing (AFS-fitness) has the highest number of classes that increase coverage (51 classes) and also the lowest number of classes that decrease coverage (112 classes). This is also the case when comparing the number of classes that show significant increase where AFS-fitness has the highest number of classes with significant increase (5 classes) when compared to the other two techniques, and also the lowest number of classes with significant decrease (29 classes). However, both cases show a small difference between the performance of Monotonic GA with and without this technique (i.e., $\hat{A}_{12} = 0.52$ with increased coverage and $\hat{A}_{12} = 0.47$ with decreased coverage). For the other two techinques, the adaptive statement-based sharing (AFS-statement) has more increasing classes, especially with significant increase, than the adaptive predicate-based sharing (AFS-predicate), but the number of decreases is higher in the latter.

The negative effect of diversity on coverage can be explained by the increase in length: Execution of longer tests takes more time, thus slowing down the evolution. Within the time limit of 10 minutes, Monotonic GA executed 764 generations on average. Always applying

fitness sharing decreases the average number of generations (479, 456, and 369 generations with FS-fitness, FS-predicate, and FS-statement respectively), but these figures improve when using the adaptive approach (668, 569, and 531 generations with AFS-fitness, AFS-predicate and AFS-statement, respectively).

> ***RQ1.3****: Promoting population diversity constantly leads to an increase in the inidividual size and a decrease in the coverage. However, promoting diversity adapetively reduces the negative impact on length and possibly improve coverage.*

4.3 AN INVESTIGATION OF POPULATION DIVERSITY WITH THE MANY-OBJECTIVE SORTING ALGORITHM

This section provides the study of the impact of population diversity on the generation of unit tests when considering MOSA. Similar to the previous section, we measure the population diversity level during the evolution in order to understand the effect of diversity on the performance of MOSA. Also, we apply different diversity maintenance techniques to see whether increasing diversity leads to similar performance as with the Monotonic GA. The rest of this section follows the same structure in Section 4.2 where we first present the diversity measures and the diversity maintenance techniques we consider in the experiment, and then we provide the experiment procedure along with the results.

4.3.1 *Measuring Population Diversity in Test Case Generation*

In this study, we consider similar diversity measures to the three measures presented in Section 4.2.1 that measure the phenotypic diversity based on the fitness entropy and test execution traces, and the genotypic diversity based on the syntactic representation of test cases. As MOSA is a multiobjective algorithm and has different individual representation (i.e., test case rather than test suite), there are some modifications that must be applied to the measures, which are:

- In the fitness entropy measure, the fitness entropy is applied on a set of buckets that is constructed for each objective, and then all entropies are added up to calculate the overall entropy.
- The calculation of predicate diversity is based on the execution of a test case rather than a test suite. In this case, the predicate diversity is calculated by counting the number of times each predicate in a CUT is covered by an *individual* test case, and not by *all* test cases of a test suite.

- In the statement diversity, statements of an individual test case are considered to measure syntactic difference between the individuals.

4.3.2 *Maintaining Population Diversity in Test Case Generation*

To study the impact of population diversity on the test case generation, we consider similar diversity maintenance techniques to those we used in Section 4.2.2. Besides the five techinuqes we presented previously, we consider two more techniques that are designed specifically for MOSA; Diversity-based Ranking (Section 4.3.2.6) and Diversity-based Selection (Section 4.3.2.7). In this section, we review the techniques as to how they are adapted in a multiobjective context and how they are applied in MOSA.

4.3.2.1 *Fitness Sharing*

Recall that the idea behind fitness sharing is to maximise the fitness value that is shared by the individuals of a niche when the number of individuals is high, and minimise it when there are few individuals in a niche, which gives these individuals higher probability to be selected for next generations. In a multiobjective context, fitness sharing is calculated for each objective, and when the niche count is based on fitness (i.e., fitness-based sharing), the distance is computed based on fitness values of each objective. To incorporate fitness sharing into MOSA, it is applied on the initial population (Line 6), and on the union of parents and offspring population (Line 16) in Algorithm 5.

4.3.2.2 *Clearing*

In a multiobjective scenraio, clearing is calculated for each objective, and applied on the initial population (Line 6), and on the union of parents and offspring population (Line 16) in Algorithm 5.

4.3.2.3 *Diverse Initial Population*

Similar to the idea presented in Section 4.2.2.3, in order to generate a diverse initial population with MOSA, we modify Line 4 in Algorithm 5 to generate random individuals of size m and then select only the most n distant individuals to form P_t (Line 5).

4.3.2.4 *Adaptive Crossover and Mutation Rates*

The crossover probability is increased when diversity is high to allow for more exploitation, whereas the mutation probability is increased when diversity is low to allow for more exploration. In MOSA, the two probabilities are adaptively set based on the current diversity level (Line 11 and Line 12 in Algorithm 5).

Algorithm 5: The modified MOSA applied in EvoSuite to incorporate diversity techniques

```
1  Input: Population size n, Stopping criterion C, Large
     population size m
2  Output: An archive of best test cases T
3  t ← 0                                          ▷ current iteration
4  P_i ← GenerateRandomPopulation(m)
5  P_t ← getMostDistantIndividuals(P_i, n)
6  P_t ← ApplySharing(P_t)                ▷ fitness sharing or clearing
7  D_i ← MeasureDiversity(P_t)               ▷ initial diversity level
8  T ← GetArchive(P_t)
9  while ¬C do
10   |  D_p ← MeasureDiversity(P_t)           ▷ current diversity level
11   |  c_p ← CalculateAdaptiveCrossoverRate(D_p)
12   |  m_p ← CalculateAdaptiveMutationRate(D_p)
13   |  P_o ← GenerateOffspring(P_t)
14   |  P_u ← P_t ∪ P_o
15   |  P_{t+1} ← EliminateSimilarIndividuals(P_{t+1})
16   |  P_u ← ApplySharing(P_u)            ▷ fitness sharing or clearing
17   |  F ← PreferenceSorting(P_u)   ▷ ranking based on diversity
18   |  r ← 0
19   |  P_{t+1} ← {}
20   |  while |P_{t+1}| + |F_r| ≤ n do
21   |  |  AssignDistance(F_r)         ▷ predicate/statement distance
22   |  |  P_{t+1} ← P_{t+1} ∪ F_r
23   |  |  r ← r + 1
24   |  end
25   |  CrowdingDistanceSort(F_r)
26   |  P_{t+1} ← P_{t+1} ∪ F_r                       ▷ size n − P_{t+1}
27   |  T ← GetArchive(T, P_{t+1})
28   |  t ← t + 1
29 end
30 return T
```

4.3.2.5 *Duplicate Elimination*

As presented in Section 4.2.2.5, the Duplicate Elimination (DE) is applied after generating the offspring, and more specifically on the union of parents and offspring (Line 15 in Algorithm 5).

4.3.2.6 *Diversity-based Ranking*

In the ranking assignment, test cases are selected to form the first non-dominated front based on their objective values such that a test case x is preferred over a test case y if $f_i(x) < f_i(y)$ where $f_i(x)$ denotes the objective score of test case x_i for branch b_i. When two test cases result in a similar lowest fitness value for a given branch b_i, one of them is chosen randomly to be included in the first non-dominated front. Instead of the random selection, we modify the selection to be based on the diversity such that the test case with high distance from other individuals in the population is preferred to be selected. The distance can be based on the predicate execution vectors (DR-predicate) or the statement counts (DR-statement). To apply this technique, we modify the `PreferenceSorting` function in Line 17 in Algorithm 5 to be as shown in Algorithm 6. When selecting the best test case that covers uncovered branch (Line 5) results in more than one best test case (i.e., test cases with the lowest fitness values) , only the test case with higher distance from other test cases is selected (Line 8).

4.3.2.7 *Diversity-based Selection*

Once a rank is assigned to all candidate test cases, the crowding distance is used to make a decision about which test case to select. The basic idea behind the crowing distance is to compute the Euclidean distance between each pair of individuals in a front based on their objective value. In this case, the test cases having a higher distance from the rest of the population are given higher probability of being selected. To investigate the influence of distance-based measures on the selection, we replace the crowding distance with our two measures, i.e., statement-based (DS-statement) and predicate-based (DS-predicate) measures, to calculate the distance between any pair of individuals in each front. Therefore, we replace the `AssignCrowdingDistance` in Line 21 in Algorithm 5 with a new function called `AssignDistance` that calculates the statement/predicate distance from each individual test cased to the other individuals.

Algorithm 6: The modified `PreferenceSorting`

```
1  Input: A set of candidate test cases T
2  Output: Non-dominated ranking assignment 𝔽
3  𝔽_0                                        ▷ first non-dominated front
4  for b_i ∈ B and b_i is uncovered do
5      t_candidate ← SelectBestTestCase(T)            ▷ to cover b_i
6      t_best      ▷ test case in T with minimum fitness for b_i
7      if |t_candidate| > 1 then
8          t_best ← SelectMostDistantTestCase(t_candidate)
9      else
10         t_best ← t_candidate
11     end
12     𝔽_0 ← 𝔽_0 ∪ {t_best}
13 end
14 T ← T − {t_best}
15 if T is not empty then
16     𝔾 ←
        Fast-Nondominated-Sort(T, {b ∈ B | b is uncovered})
17     d ← 0                                   ▷ first front in 𝔾
18     for All non-dominated fronts in 𝔾 do
19         𝔽_{d+1} ← 𝔾_d
20     end
21 end
```

4.3.3 *Empirical Study*

The purpose of this study is similar to the study conducted in Section 4.2.3 which is to investigate the influence of test case diversity on the generation of unit tests. In particular, we aim to analyse the influence of population diversity on the performance of MOSA, and whether maintaining diversity during the search has an influence on the branch coverage. We therefore designed our study to answer the following research questions:

RQ 2.1: How does population diversity change throughout evolution when considering MOSA?

RQ 2.2: How effective are diversity maintenance techniques with MOSA?

RQ 2.3: What are the effects of increasing population diversity in MOSA?

In regards to the ability of MOSA in maintaining population diversity, our assumption is that MOSA is more likely to maintain better diversity level than the WSA approach during the evolution. This is because of its properties that are explicitly designed to maintain better population diversity (e.g., the use of crowding distance). Also,

we expect that the individual representation of a test case increases the population diversity as the difference among test cases can be more obvious than the difference among test suites. However, we still anticipate that applying diversity maintenance techniques with MOSA leads to improve the population diversity level during the evolution, which possibly affects the code coverage negatively.

4.3.3.1 *Experimental Setup and Procedure*

In this study, we consider similar experimental setup to the one mentioned in the previous section (Section 4.2.3.1) where EvoSuite is used to generate JUnit test suites for a given Java CUT using MOSA, the corpus of 311 complex classes from DynaMOSA study [127] is used (i.e., similar to the classes were run in the previous study).

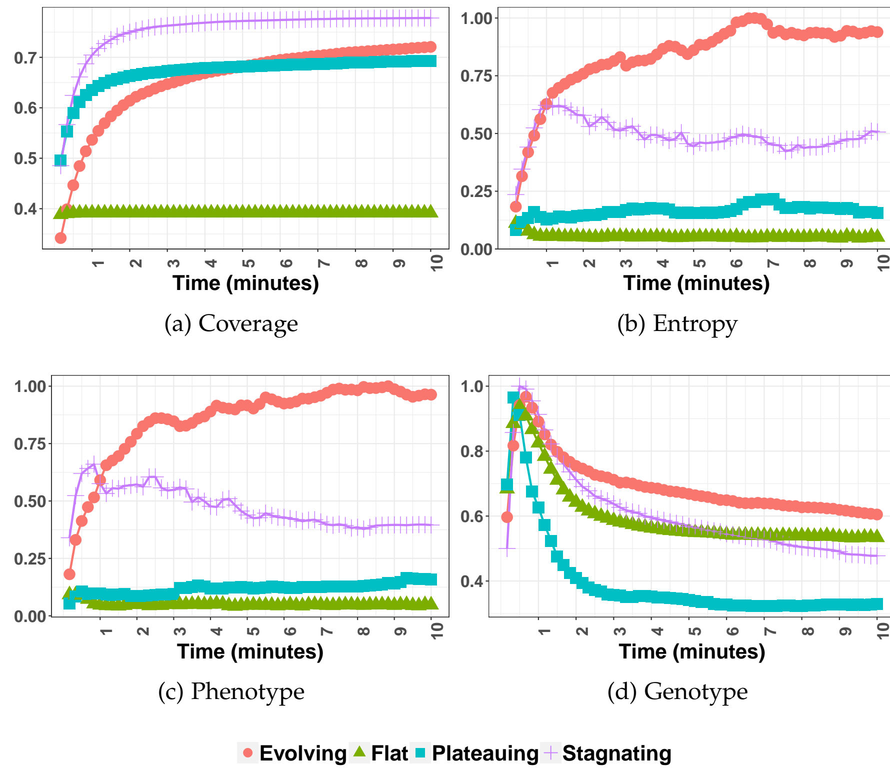
Also, we follow the experiment procedure that is applied in the study of population diversity using WSA approach (Section 4.2.3.2) where we conduct an experiment that involves applying each of the three diversity measures on each CUT to measure the diversity level and then applying each of the diversity maintenance techniques defined in Section 4.3.2 to promote the population diversity throughout the evolution. In a similar manner to WSA, we integrate each diversity technique into MOSA and compare its performance to the default version. However, the total number of MOSA runs is 8 (i.e., a single MOSA run with each of the 7 techniques and one run of default MOSA). For the parameters setting, we used similar values for the parameters of the diversity techniques as in Section 4.2.3.3.

4.3.3.2 *Threats to Validity*

Controlling the threats to the internal and external validity is similar to Section 4.2.3.4 except that the considered GA in this study is MOSA instead of Monotonic GA.

4.3.3.3 *How does population diversity change throughout evolution when considering MOSA?*

In order to understand the influence of population diversity on the test generation, we measure the diversity level that is maintained by MOSA during the evolution. That is to get an idea of whether MOSA is able to maintain a proper level of diversity during the search. To achieve that, we look at the diversity based on the different coverage patterns that previously introduced in Section 4.2.3.5. We therefore followed a similar procedure to classify the CUTs into different groups by performing runs of 30 minutes to observe the development of coverage with MOSA. This results in similar four groups as with the previous study except the difference in the number of CUTS:



(a) Coverage (b) Entropy

(c) Phenotype (d) Genotype

Figure 4.6: Average values for coverage and population diversity throughout evolution in MOSA for the four groups of CUTs.

- **Evolving** contains CUTs where the coverage after 30 minutes is higher (by more than 0.01) than after 10 minutes (93 CUTs).
- **Flat** is the set of CUTs where the coverage never changes (60 CUTs).
- **Stagnating** contains CUTs for which the coverage stagnates after 10 minutes: it is higher than after 2 minutes, but increases by less than 0.01 from 10 minutes to 30 minutes (43 CUTs).
- **Plateauing** contains the remaining CUTs for which the coverage after two minutes is constant (115 CUTs).

Then, we started 30 runs for each CUT with a search budget of 10 minutes and applied the three diversity measures defined in Section 4.3.1 to measure the diversity level for each of the four groups, as shown in Figure 4.6. It is obvious that the diversity behaviour is different among the four groups, and the difference can be seen with the three diversity measures. For the *evolving* group, coverage keeps growing throughout the entire 10 minutes, and this group also shows a continuous growth of entropy and phenotype diversity. In terms of genotype diversity, there is a reduction after an initial sharp growth phase, but less than in all other groups. For the *flat* group,

Table 4.6: Average diversity over time when applying diversity maintenance techniques based on different distance measures, and the average effect size $\hat{A}_{12}$.

Technique	Phenotype	$\hat{A}_{12}$	Genotype	$\hat{A}_{12}$	Entropy	$\hat{A}_{12}$
AR (fitness)	0.70	0.51	0.79	0.61	0.91	0.59
AR (predicate)	0.57	0.45	0.78	0.55	0.91	0.58
AR (statement)	0.67	0.48	0.77	0.53	0.70	0.52
CL (predicate)	0.87	0.61	0.94	0.81	0.87	0.65
CL (statement)	0.83	0.59	0.90	0.79	0.85	0.63
DE (predicate)	0.76	0.53	0.93	0.80	0.86	0.53
DE (statement)	0.70	0.52	0.90	0.79	0.80	0.52
DIP (predicate)	0.76	0.52	0.65	0.49	0.69	0.51
DIP (statement)	0.71	0.49	0.68	0.48	0.70	0.52
DR (predicate)	0.79	0.51	0.77	0.63	0.73	0.52
DR (statement)	0.61	0.46	0.77	0.62	0.67	0.48
DS (predicate)	0.87	0.67	0.93	0.82	0.74	0.51
DS (statement)	0.83	0.63	0.90	0.80	0.85	0.55
FS (fitness)	0.88	0.63	0.74	0.51	0.95	0.76
FS (predicate)	0.88	0.61	0.90	0.78	0.93	0.75
FS (statement)	0.88	0.61	0.93	0.79	0.88	0.73
MOSA	0.72	-	0.68	-	0.67	-

the phenotype diversity is overall lowest; this is because only very few predicates are covered in the first place, as shown in the coverage plot, and the low entropy. For the *stagnating* group, once the coverage increase slows down, all three diversity measurements go down as well. For the *plateauing* group, once the search converges all diversity measurements drop sharply, and notably the genotypic diversity is lowest of all groups. Overall it seems that, as long as coverage grows, MOSA does well at maintaining diversity. Once coverage stagnates, the population loses diversity. To some extent, this can be explained by EvoSuite's ranking mechanism: If two individuals have the same fitness value, then the shorter of the two is preferred; this is also used in MOSA's rank-based preference sorting. Shorter individuals by construction will have less diversity.

> ***RQ2.1****: MOSA maintains high diversity while coverage increases, but diversity drops once a maximum coverage has been reached.*

4.3.3.4 *How effective are diversity maintenance techniques with MOSA?*

In RQ1 we saw a general tendency that diversity drops once the coverage stops increasing. We therefore would like to see which diversity maintenance techniques succeed at increasing the population diversity during evolution. For that, and similar to the procedure followed in Section 4.2.3.6, we apply the techniques mentioned in Section 4.3.2 on all classes. Table 4.6 summarises the results of the average diversity among the four groups with each technique based on different distance measures (i.e., predicate, statement, and fitness). The effect size estimates the probability that a single run of MOSA with a diversity technique results in higher diversity than the default MOSA (i.e., with no diversity techniques).

We can clearly see that the three diversity measures indicate that many of the diversity maintenance techniques are able to promote diversity higher than MOSA. The entropy measure indicates that diversity is the lowest with MOSA compared to all other techniques although the difference is negligible when compared to the statement-based Diversity-based Ranking (DR-statement) with the smallest effect size, and fitness-based fitness sharing (FS-fitness) increases entropy the most with the largest effect size. A similar trend can be observed for phenotypic diversity, where FS-fitness results in the overall highest diversity although predicate-based Diversity-based Selection (DS-predicate) has the largest effect size. In contrast to entropy, not all techniques succeed in increasing phenotypic diversity; for example, predicate-based Adaptive Rates (AR-predicate) results in the lowest diversity and the smallest effect size. The genotype measure indicates a wide range of effectiveness, with Clearing based on predicate distance (CL-predicate), the statement-based fitness sharing (FS-statement), and DS-predicate as the most successful in promoting genotypic diversity, especially the DS-predicate as it results in the largest effect size. However, several other techniques lead to a reduction on genotypic diversity, surprisingly in particular the two versions of Diverse Initial Population (DIP) where DIP-statement has the smallest effect size.

To investigate how each technique affects the diversity level during evolution, Figure 4.7 shows the diversity level achieved throughout the evolution for each of the five techniques. For the two niching techniques (i.e., FS and CL), the resulting diversity by both techniques differs among the three diversity measures. The entropy measure (Figures 4.7a and 4.3d) shows that FS is better than CL where the three versions of FS result in higher diversity than the two versions of CL, which is confirmed by the results in Table 4.6. However, the phenotype measure (Figures 4.7b and 4.7e) indicates that the difference between the two techniques is negligible except the FS-fitness that is slightly better than the two versions of CL. The genotype measure (Figures 4.7c and 4.7f) shows that both techniques lead to high increase in diversity

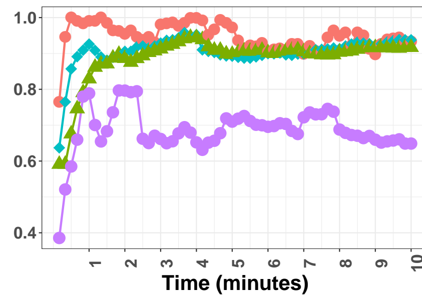
when compared to the default MOSA, which is confirmed by the largest effect size, except the drastic drop with FS-fitness.

In the case of DIP measure, the three measures (Figures 4.7h $-$ 4.7j) demonstrate no obvious difference between the diversity achieved by the two versions of DIP and the default MOSA except the potential increase in diversity by DIP-predicate in the phenotype measure. Looking at the AR measure, the entropy measure (Figures 4.7k) shows that both AR-fitness and AR-predicate lead to higher diversity than AR-statement and even more higher than the default MOSA. This is also the case with genotype measure (Figures 4.7m) except that the AR-fitness is slightly better than the AR-predicate and the latter is very similar to AS-statement. However, the phenotype measure (Figures 4.7l) indicates a drop in the diversity after 50% of the search time with the three versions of AR although AR-fitness reaches almost similar diversity level to the default MOSA.

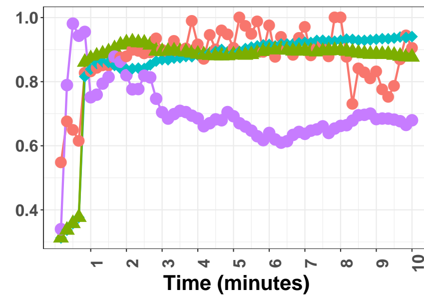
The three diversity measures demonstrate different diversity level with the two versions of Duplicate Eliminate (DE) technique. In the entropy measure (Figures 4.7o), the DE-predicate increases diversity more than DE-statement and default MOSA, except after 75% of the search time where DE-statement inconstantly increases diversity. The phenotype measure (Figures 4.7p) shows that both versions of DE lead to similar increase in diversity, especially after 50% of the search time. However, there is obvious increase in diversity by both versions that is shown by the genotype measure (Figures 4.7q) where the DE-predicate results in the considerable increase in diversity, which is also confirmed by the largest effect size ($\hat{A}_{12} = 0.80$).

For the two techniques that are adapted specifically for MOSA, the Diversity-based Ranking (DR) and Diversity-based Selection (DS) result in different effect on diversity during the evolution. The entropy measure (Figures 4.7r and 4.7v) demonstrates that the DS technique is slightly better than the DR where DS-statement results in the highest diversity despite the drop in the last 20% of the search time. The DR-predicate, however, is quite similar to the default MOSA except in that period of time where DS-statement drops. In the case of DR, the DR-predicate leads to many improvements in diversity when compared to the DR-statement, which is very similar to default MOSA. Looking at the phenotype measure (Figures 4.7s and 4.7w), DS results in higher diversity than DR where the highest increase is observed with the DS-predicate followed by DS-statement. This is also the case with the two versions of DR where DR-predicate achieves better diversity than the default MOSA, but DR-statement is worse than default MOSA. The genotype measure (Figures 4.7t and 4.7x) shows that both versions of DS keeps increasing diversity while the two versions of DR lead to decrease in diversity, but still better than default MOSA.

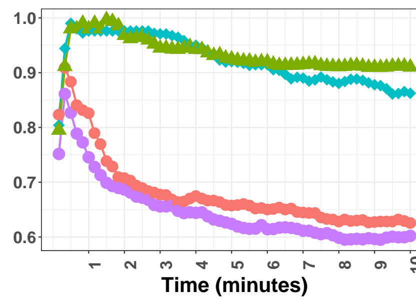
***RQ2.2**: Most diversity maintenance techniques succeed at increasing diversity, but there are exceptions. Fitness sharing achieves the most consistent increase.*



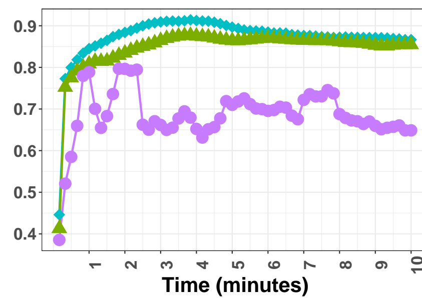
(a) FS – Entropy



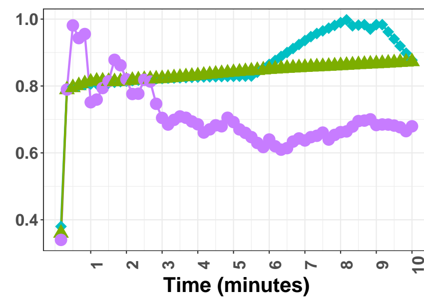
(b) FS – Phenotype



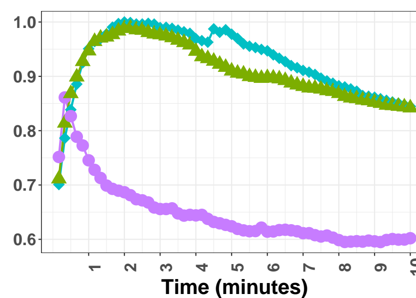
(c) FS – Genotype



(d) CL – Entropy

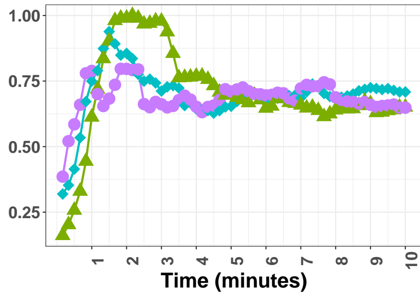


(e) CL – Phenotype

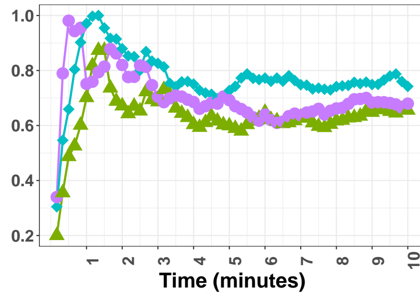


(f) CL – Genotype

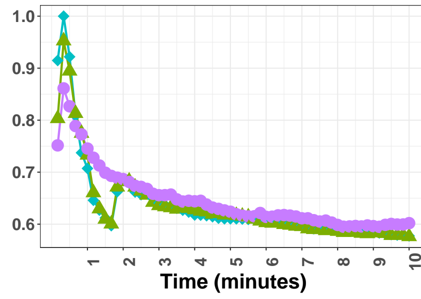
● Fitness ● MOSA ◆ Predicate ▲ Statement



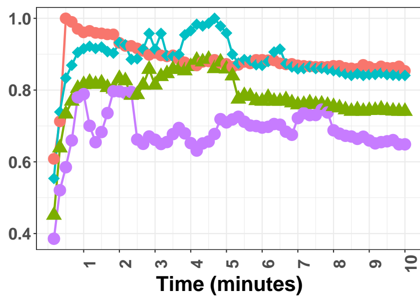
(h) DIP – Entropy



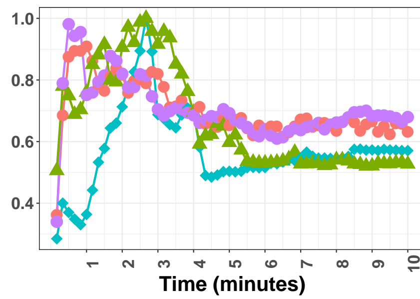
(i) DIP – Phenotype



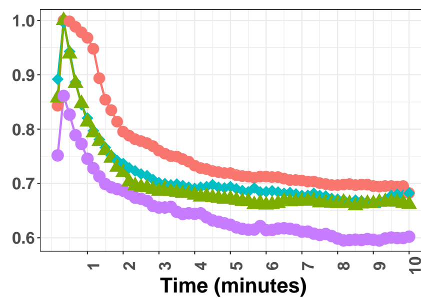
(j) DIP – Genotype



(k) AR – Entropy

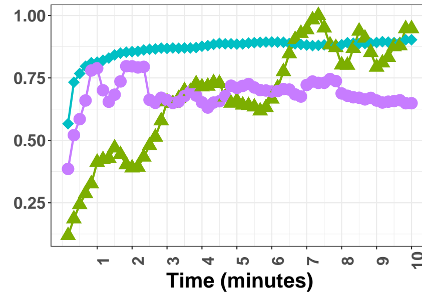


(l) AR – Phenotype

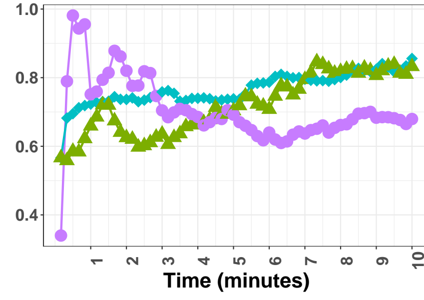


(m) AR – Genotype

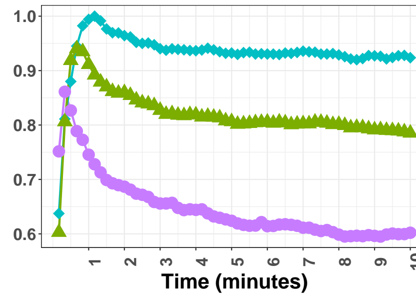
● Fitness ● MOSA ◆ Predicate ▲ Statement



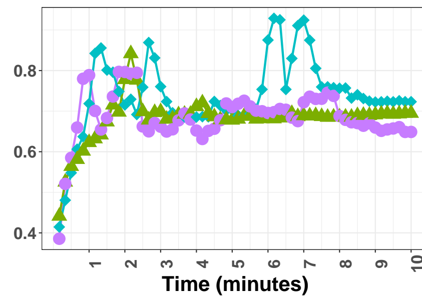
(o) DE – Entropy



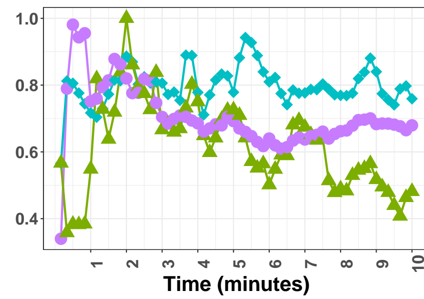
(p) DE – Phenotype



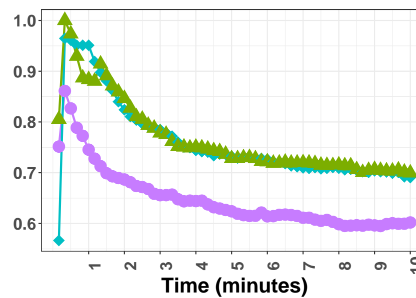
(q) DE – Genotype



(r) DR – Entropy

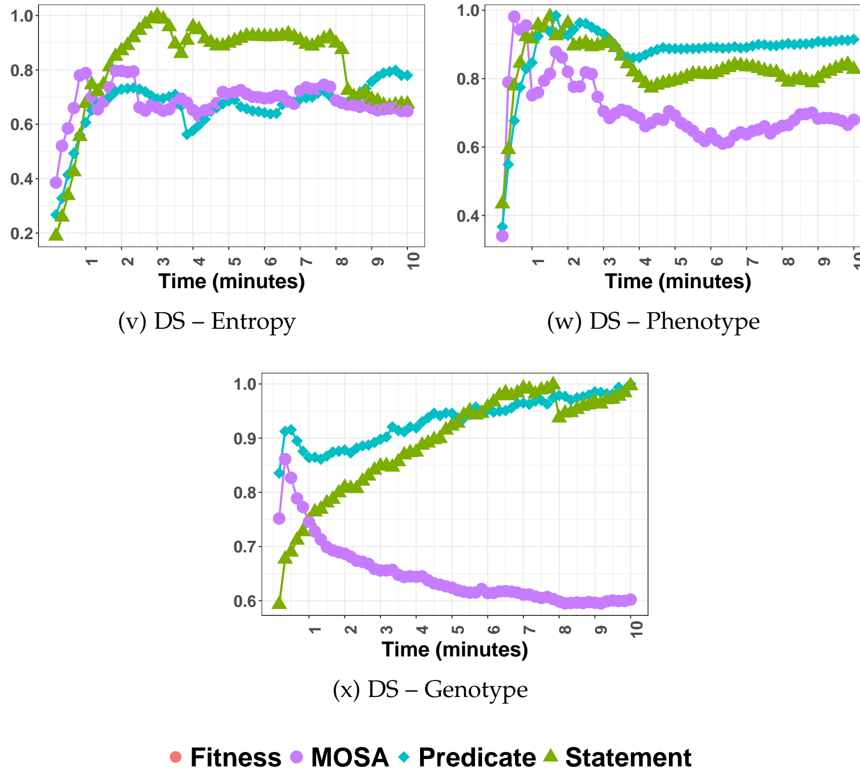


(s) DR – Phenotype



(t) DR – Genotype

● Fitness ● MOSA ◆ Predicate ▲ Statement



(v) DS – Entropy

(w) DS – Phenotype

(x) DS – Genotype

Figure 4.7: Diversity throughout the evolution with MOSA and the seven diversity maintenance techniques based on different distance measurements.

4.3.3.5 *What are the effects of increasing population diversity in MOSA?*

To investigate the impact of diversity on the performance of MOSA, we look at the achieved coverage and the average size of the individuals in the population throughout the evolution. Figure 4.8 shows the results of the best coverage in the population and the average length of all individuals in the population for MOSA with and without each of the seven diversity maintenance techniques during the evolution. Overall, the performance of MOSA with the diversity techniques either similar or lower than the defaull MOSA in terms of branch coverage. We can clearly see that the diversity techniques that result in high increase in diversity lead to lower branch coverage than the default MOSA such as CL and DS.

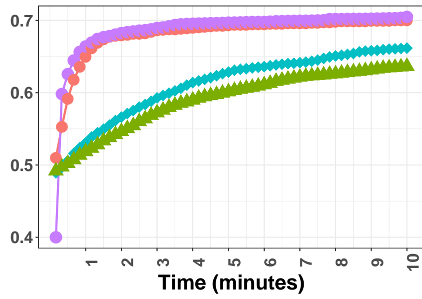
In the case of FS, two versions of FS (i.e., FS-predicate and FS-statement) reduce the coverage, but the FS-fitness results in slighlty similar coverage to default MOSA where the $\hat{A}_{12}$ values with the FS-entropy, FS-predicate, and FS-statement are $0.49, 0.47$, and 0.46, respectively. However, the coverage with the two version of CL is considerably lower than default MOSA where $\hat{A}_{12}$ is 0.39 with both versions. The two versions of DIP perform similarly as they result in lower coverage than the default MOSA in almost the first five minutes

Table 4.7: Spearman correlation between the diversity achieved by the seven diversity techniques based on different distance measures and length.

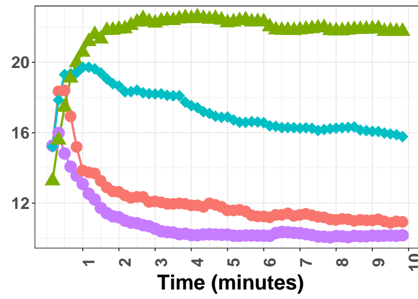
Technique	Phenotype	Genotype	Entropy
AR (fitness)	0.3338	0.5060	0.6075
AR (predicate)	0.2496	0.4008	0.6857
AR (statement)	0.3098	0.4541	0.7203
CL (predicate)	0.4301	0.4272	0.6403
CL (statement)	0.4527	0.2249	0.6373
DE (predicate)	0.3195	0.3165	0.7744
DE (statement)	0.3113	0.3534	0.6120
DIP (predicate)	0.3444	0.3398	0.5624
DIP (statement)	0.3880	0.4286	0.5248
DR (predicate)	0.4383	0.3647	0.3489
DR (statement)	0.4158	0.3594	0.3669
DS (predicate)	0.5729	0.3684	0.5970
DS (statement)	0.5023	0.4677	0.5353
FS (fitness)	0.2564	0.1301	0.2348
FS (predicate)	0.4393	0.5504	0.5839
FS (statement)	0.3791	0.3376	0.5581
MOSA	0.3211	0.4444	0.4957

of the search and then achieve similar coverage to default MOSA; $\hat{A}_{12} = 0.49$ with DIP-predicate and $\hat{A}_{12} = 0.485$ with DIP-statement. For the AR technique, Figure 4.8g suggests that the adaptive crossover and mutation rates seem to have a negative impact on coverage as the three versions of AR lead to lower coverage than default MOSA where the $\hat{A}_{12}$ with the three versions is almost 0.45. This is also the case with the DE and DS where there is an obvious difference in the achieved coverage by the two versions of each technique and default MOSA; $\hat{A}_{12} = 0.47$ with DE-statement, $\hat{A}_{12} = 0.46$ with DE-predicate, DS-predicate, and DS-statement. However, the DR technique performs similar to default MOSA except that the latter achieves higher coverage in the first two minutes of the search; $\hat{A}_{12} \approx 0.49$ with both versions of DR.

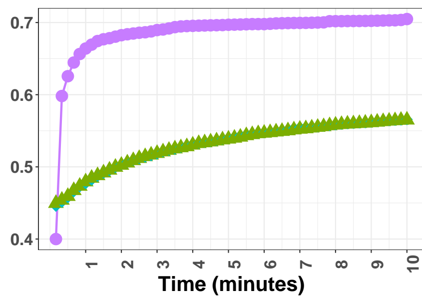
The negative effect of diversity on the length is observed with all the techniques, as shown in Figure 4.8. We can obviously see that increasing diversity leads to longer test cases. In this case, we expect that those techniques that lead to higher diversity also lead to longer individuals. To validate this conjecture, and similar to what we consider in Section 4.2.3.7, we look at the Spearman correlation between the



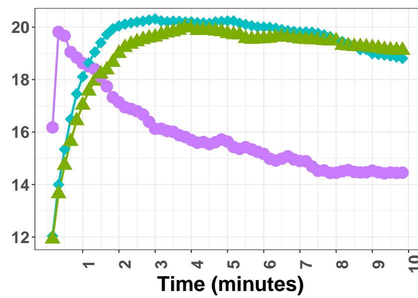
(a) FS – Coverage



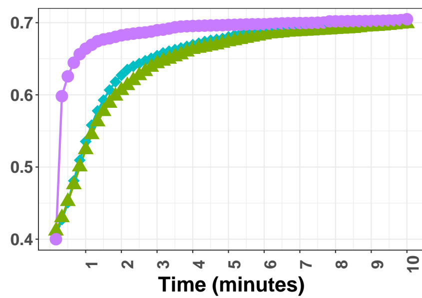
(b) FS – Length



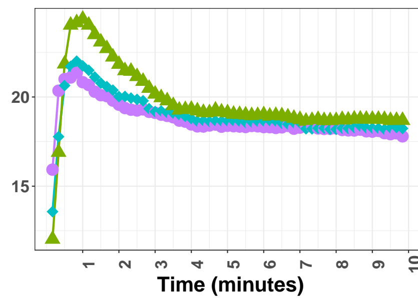
(c) CL – Coverage



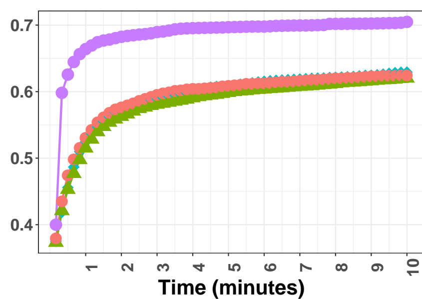
(d) CL – Length



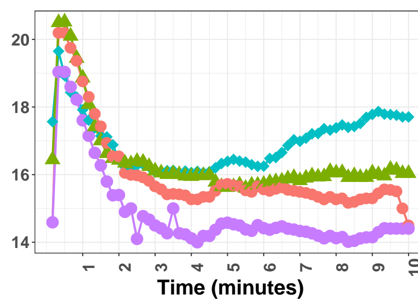
(e) DIP – Coverage



(f) DIP – Length



(g) AR – Coverage



(h) AR – Length

Fitness MOSA Predicate Statement

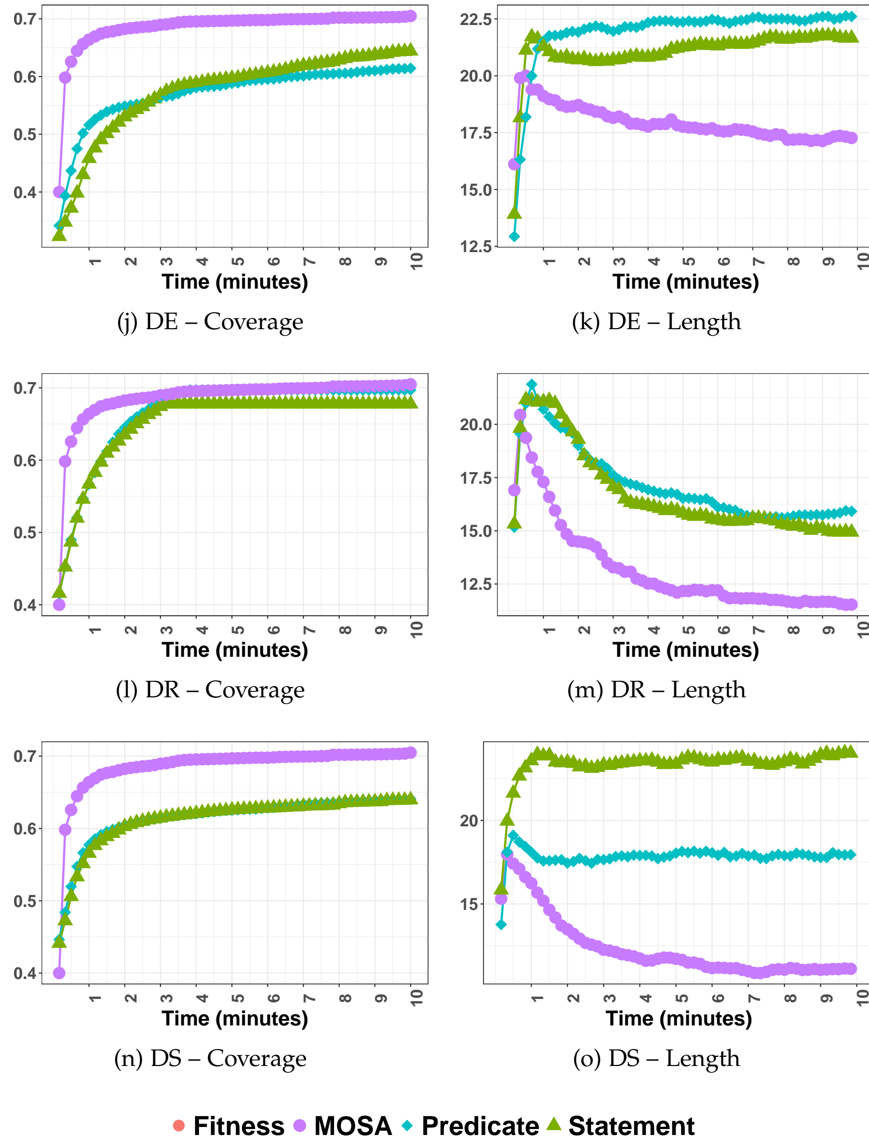


Figure 4.8: Coverage and length over time with MOSA and the five diversity maintenance techniques based on different distance measurements.

average diversity throughout evolution and average test case length in the population, which is shown in Table 4.7. Based on the entropy measure, we can observe that there is a strong correlation between diversity and length with some techniques where the strongest correlation is with DE-predicate, but however the correlation with other techniques is not as strong as with FS-fitness, which indicates that increasing diversity does not necessarily always lead to an increase in length. The phenotype and genotype measures show slighlty weaker correlation between diversity and length, especially the FS-fitness with genotype measure that indicates higher diversity does not result in longer tests. This, in fact, is confirmed by the genotype diversity shown in Figure 4.7c and the length in Figure 4.8b.

To follow the procedure we apply in Section 4.2.3.7, we consider to apply the adaptive version of fitness sharing and compare its performance to the naive application of fitness sharing and the default MOSA. The adaptive diversity approach work by applying the fitness sharing only when diversity drops below a certain threshold (i.e., 60% of the initial population diversity level), and once the diversity level exceeds the threshold, the diversity technique is not applied. Figure 4.9 shows the results of the best coverage in the population and the average length of all test cases in the population for MOSA with and without fitness sharing (FS (fitness/predicate/statement)) during the evolution.

For the *flat* group, there is a small difference in coverage as MOSA and fitness-based fitness sharing result in slighlty higher coverage than the other techniques. The length plot shows how MOSA removes all redundancy from the population, while adding diversity leads to larger individuals. In particular, genotype-based fitness sharing has quite dramatic effects on size. Fitness-based sharing (adaptive and non-adaptive) has the smallest effects on size.

For the *evolving* group, MOSA achieves the highest coverage, while maintaining a somewhat constant population size. Generally, fitness sharing slightly increases size, quite dramatically so for genotype-based fitness sharing. Adaptive fitness-based sharing even leads to smaller individuals than MOSA. Non-adaptive fitness sharing using genotype and phenotype diversity leads to a notably lower coverage.

For the *stagnating* group, the non-adaptive fitness sharing using genotype and phenotype diversity again lead to a notably lower coverage and larger size. This time, however, adaptive fitness-based sharing consistently leads to a higher average coverage than MOSA, and even smaller individuals than MOSA.

The *plateauing* group shows similar results to the *stagnating* group, with larger coverage improvement of adaptive phenotype-based sharing. While genotype and phenotype based non-adaptive sharing again lead to lower coverage initially, in this group the size remains large

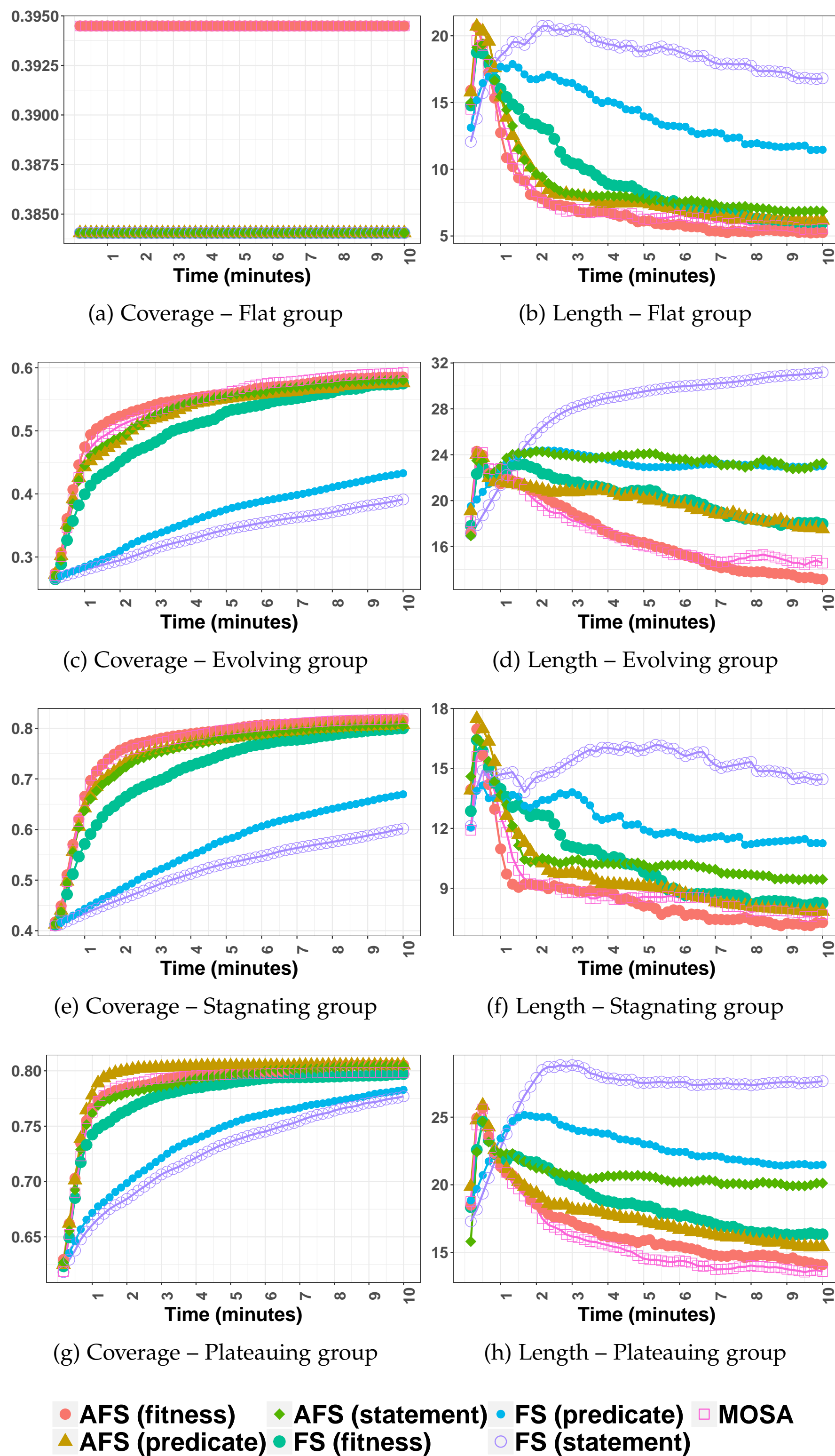


Figure 4.9: Coverage and length over time with MOSA, fitness sharing (FS), and adaptive fitness sharing (AFS) per four groups of CUTs

Table 4.8: Number of classes where adaptive fitness sharing has an increased/decreased/equal coverage compared to MOSA, the average effect size $\hat{A}_{12}$ and the number of classes for which this comparison is statistically significant ($\alpha = 0.05$).

Technique	increased coverage			decreased coverage			equal
	#classes	#sig.	$\hat{A}_{12}$	#classes	#sig.	$\hat{A}_{12}$	#classes
Fitness	99	6	0.52	87	27	0.48	125
Predicate	52	2	0.51	141	39	0.48	118
Statement	77	4	0.51	116	33	0.49	118

but constant, and the coverage catches up and even overtakes MOSA in the end.

Adaptive fitness sharing leads to higher coverage when the search in MOSA stagnates (e.g., when coverage does not increase). To see whether adaptive fitness sharing is always beneficial, Table 4.8 shows the number of classes where it increases, decreases, or results in equal coverage with MOSA. Adaptive fitness-based sharing (AFS-fitness) increases coverage on 99 classes and decreases it on 87 classes, albeit having only 6 significant increases as opposed to 27 significant decreases. For adaptive sharing based on predicate (AFS-predicate) and statement (AFS-statement) differences we found more decreases than increases.

The negative effect of diversity on coverage can be explained by the increase in length where the execution of longer tests takes more time, thus slowing down the evolution. During the 10 minutes search limit, MOSA executed 629 generations on average. The naive fitness sharing decreases the average number of generations (457, 344, and 339 generations with FS-fitness, FS-predicate, and FS-statement respectively), but these figures improve when using the adaptive approach (631, 538, and 519 generations with AFS-fitness, AFS-predicate and AFS-statement, respectively). Our conjecture is that the low number of generations is a result of a high number of statements that take long time to execute. To validate this conjecture, we take a step further in our investigation and look at the average number of executed statements with all generations, and surprisingly we found out that the number of executed statements with naive fitness sharing techniques is lower than MOSA (206710, 181844, 153249, and 22451 statements with MOSA, FS-predicate, FS-fitness, and FS-statement respectively), whereas the adaptive approach results in more executed statements than MOSA (300934 with AFS-statement, 268043 with AFS-predicate, and 217835 with AFS-fitness).

To gain a further understanding of the impact of increasing diversity on the individual length, we look at the types of the executed statements with each diversity technique. This allows us to see the

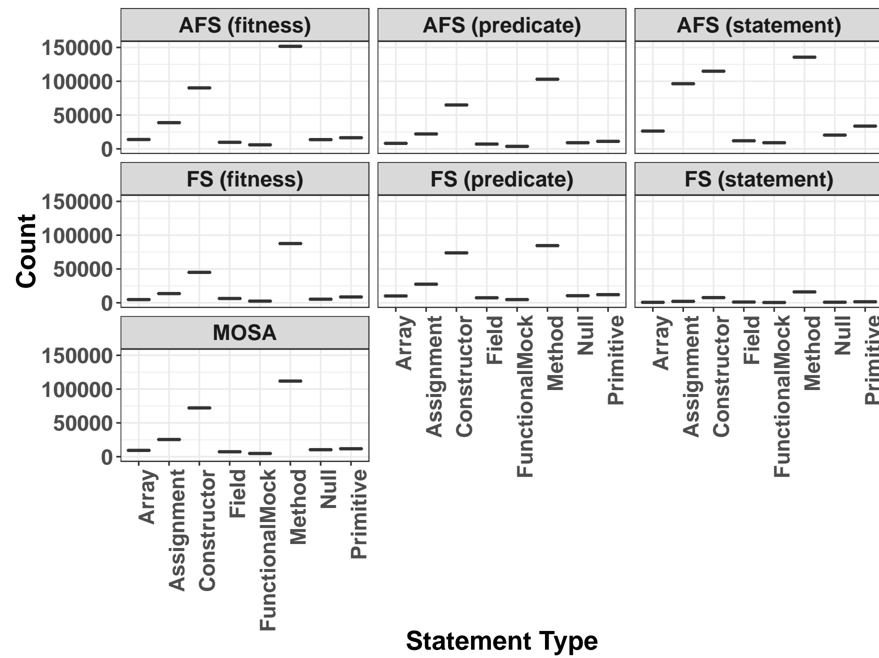


Figure 4.10: The average count of executed statements types with MOSA, fitness sharing (FS), and adaptive fitness sharing (AFS)

distribution of different statement types over different techniques, and whether there is a biased increase of certain types of statements that happens with enforced diversity (i.e., does promoting diversity lead to an increase in specific statement types?). Therefore, and based on the well-known Java statement types, we first classify the statements into the following types:

- **Array** statement is a statement that declares an array such as `byte[] byteArray0 = new byte[5]`.
- **Assignment** statement is to assign a value to a specific variable such as `byteArray0[0] = byte0`.
- **Constructor** statement is to initialize an object such as `ImprovedTokenizer improvedTokenizer0 = new ImprovedTokenizer(string0)`.
- **Field** statement is to initialize a field such as `StringBuffer stringBuffer0 = improvedTokenizer0.myBuffer`
- **Functional Mock** statement is to create a mock object that simulates the behaviour of class or the behavior of its dependencies. For example, initializing the constructor of the `Response` class requires the use of an abstract class (i.e., `HttpURLConnection`) such as: `Response(HttpURLConnection connection)`. In this case, a mock object can be used to provide a simulation for every method in the mocked `HttpURLConnection` class such as: `HttpURLConnection httpURLConnection0 = mock(HttpURLConnection.class)`.

- **Method** statement is to invoke a method in the CUT such as `String string0 = improvedTokenizer0.next()`.
- **Null** statement is to assign a null value to a variable such as `String string0 = null`.
- **Primitive** statement is to initialize a variable of primitive type such as `char char0 = 'a'`.

We then apply this classification on the executed statements resutling from the conducted experiment and count the number of statements with each type as shown in Figure 4.10. It is obvious that the number of statements of each type differs among all the techniques where higher number of statements is observed with the adaptive FS and, in contrast, lower number of statements is with the naive FS whereas the default MOSA is in the middle. In general, the dominant statement type among all the techniques is the Method type followed by the Constructor and Assignment types although the number of statement that belong to these three types is different with each technique. The highest number of Method statements is observed with AFS-fitness while AFS-statement has the highest number of Constructor statements and Assignment statements. In fact, the AFS-statement leads to the highest number of statements that belong to the Array, Field, Functional Mock, Null, and Primitive types when compared to the other techniques. Therefore, we conclude that there is no obvious statement type that dominates the executed statements when promoting population diversity, and thus the increase of individual length seems to be a general increase that is independent of certain statement types.

> ***RQ2.3***: *Promoting diversity generally leads to larger tests and reduces coverage. However, when coverage stops growing, adding diversity may improve MOSA.*

4.4 A COMPARISON OF THE IMPACT OF POPULATION DIVERSITY ON WSA AND MOSA

Investigating the influence of population diversity on the generation of unit tests reveals that there is a difference in the degree of how population diversity affects the search when considering the two algorithms; Monotonic GA using WSA approach and MOSA. Therefore, this section reviews the difference in the impact of population diversity on the search when using the two algorithms by comparing how diversity influences the performance of each algorithm.

As an initial step towards understanding how population diversity affects the search, we investigate whether each of the two algorithms is able to maintain sufficient population diversity level during the

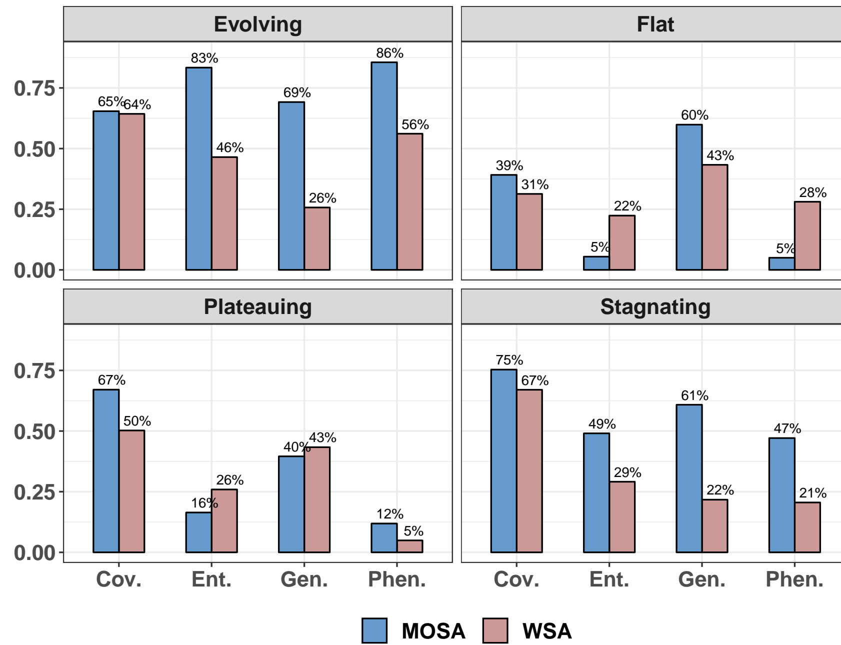


Figure 4.11: Average coverage (Cov.) and population diversity throughout evolution in both Monotonic GA using WSA and MOSA for the four groups of CUTs where a value close to 100% indicates high coverage/diversity. The diversity is measured based on the three measures: Entropy (Ent.), Genotype (Gen.), and Phenotype (Phen.).

evlution, as shown in Section 4.2.3.5 and Section 4.3.3.3. For better understanding, Figure 4.11 compares the average coverage and diversity based on the three measures (i.e., phenotype, genotype, and entropy) throughout the evolution with both algorithms.

We can clearly see that the achieved coverage and diversity by both algorithms during the evolution differs among the four groups. For the coverage, MOSA results in higher coverage than Monotonic GA using the WSA approach with the four groups. The highest difference between the coverage achieved by both algorithms is noticed with the *plateauing* group ($\hat{A}_{12} = 0.42$) whereas the smallest difference is with the *evolving* group ($\hat{A}_{12} = 0.49$).

In terms of diversity, it is obvious that each of the three measures indicates different results of the achieved diversity as there are cases where Monotonic GA is able to reach higher diversity level than MOSA. For example, the entropy measure shows that the diversity level achieved by Monotonic GA with the *flat* and *evolving* groups is higher than the level achieved by MOSA ($\hat{A}_{12} = 0.62$ and $\hat{A}_{12} = 0.54$ respectively). In the case of genotype measure, all the groups demonstrate that MOSA maintains considerably higher diversity level than Monotonic GA except the *plateauing* group although there is no significant difference between the two algorithms with this group ($\hat{A}_{12} = 0.51$). Also, the phenotype measure shows that MOSA

promotes higher diversity during the evolution than the Monotonic GA with all the groups except the *flat* group where Monotonic GA significantly increases diversity more than MOSA ($\hat{A}_{12} = 0.65$).

Based on the results of the three measures, we see that MOSA seems to be superior to Monotonic GA in maintaining higher diversity level during the evolution, and a possible reason behind that is MOSA has its own properties that are explicitly designed to maintain better population diversity such as the crowding distance that is used to decide which test case to be selected for the next generation (i.e., the test case whigh higher distance from other test cases has higher probability to be selected). Also, the individual representation plays an important rule as the similarity between individual test suites is expected to be higher than with individual test cases (i.e., the test suite representation is bound to have more redundancy).

In Section 4.2.3.6 and Section 4.3.3.4, we investigate whether applying the diversity maintenance techniques with each of the two algorithms can improve the diversity level during the evolution. To compare the impact of each of these diversity techniques on the two algorithms, Figure 4.12 presents a comparison of the diversity level achieved by the two algorithms when applying the diversity maintenance techniques.

Overall, we observe that both algorithms achieve different diversity level with every diversity technique, and no one algorithm that always maintaines higher diversity than the other algorithm. In fact, applying many of the diversity techniques with MOSA leads to an increase in the diversity more than when they are applied to the Monotonic GA. This can be seen with all the versions of AR, CL, and DIP techniques where the highest difference between the diversity achieved by both algorithms is observed with the CL-predicate based on the genotype diversity measure ($\hat{A}_{12} = 0.73$). In contrast, the two versions of the DE technique maintains higher diversity level when they are applied to Monotonic GA where the DE-statement based on phenotype diversity measure shows that highest diversity between the two algorithms ($\hat{A}_{12} = 0.57$). The fitness sharing based on predicate and statement distances achieves higher diversity with MOSA except the fitness-based version where the genotype diversity measure indicates an increase in diversity is achieved with the Monotonic GA ($\hat{A}_{12} = 0.55$).

To better understand how the diversity maintenance techniques affect the diversity level during evolution, we compare the diversity achieved by each algorithm (i.e., the default GA) with the resulting diversity level when incorporating each of the diversity techniques with the algorithm. In the case of Monotonic GA, we observe that the diversity techniques always lead to an increase in the diversity where the highest significant increase is achieved by the DE-predicate based on the genotype diversity measure ($\hat{A}_{12} = 0.82$). However, applying the diversity maintenance techniques with MOSA does not always

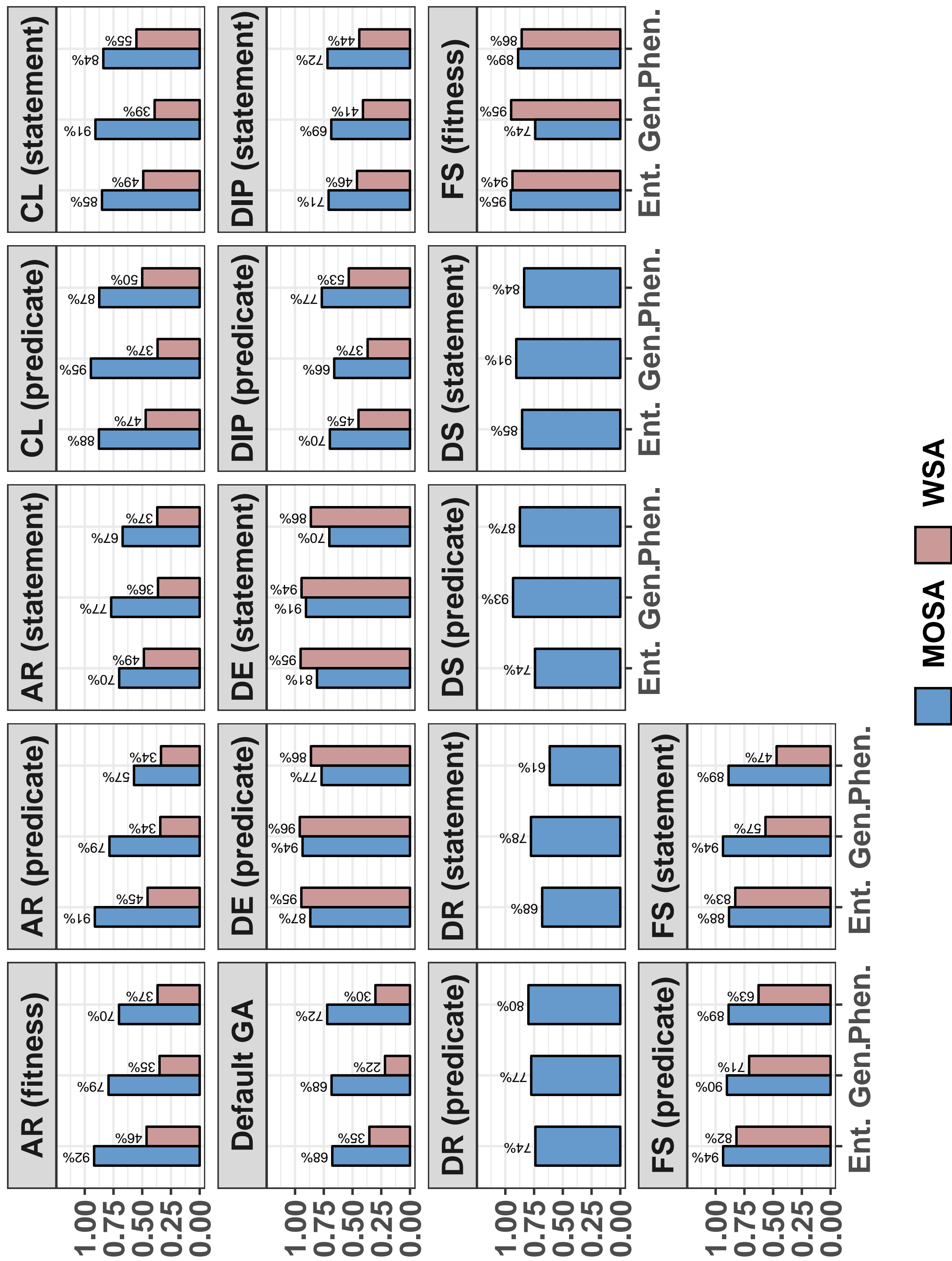


Figure 4.12: Average population diversity throughout evolution in both Monotonic GA using WSA and MOSA when considering diversity maintenance techniques based on the three measures: Entropy (Ent.), Genotype (Gen.), and Phenotype (Phen.).

Table 4.9: Coverage and Length resulting from Monotonic GA using WSA and MOSA when considering the diversity maintenance techniques.

Technique	Coverage		Length	
	WSA	MOSA	WSA	MOSA
AR (fitness)	64.67%	62.42%	135	15
AR (predicate)	64.16%	62.93%	125	18
AR (statement)	64.07%	62.04%	127	16
CL (predicate)	60.84%	56.57%	101	19
CL (statement)	60.18%	56.50%	92	19
DE (predicate)	65.06%	61.43%	114	22
DE (statement)	63.87%	64.42%	110	21
DIP (predicate)	63.43%	70.38%	111	18
DIP (statement)	63.72%	69.87%	118	19
DR (predicate)	-	69.70%	-	16
DR (statement)	-	67.77%	-	15
DS (predicate)	-	64.10%	-	18
DS (statement)	-	63.96%	-	23
FS (fitness)	62.10%	69.97%	108	11
FS (predicate)	62.63%	66.17%	112	15
FS (statement)	61.87%	63.64%	97	28
AFS (fitness)	64.36%	70.11%	91	10
AFS (predicate)	63.92%	69.57%	89	11
AFS (statement)	63.18%	68.93%	90	13
Default GA	64.64%	70.48%	81	9

lead to an increase in the diversity as the genotype diversity measure indicates a decrease in diversity when applying techniques such as the three versions of AR. Moreover, when comparing the increase in diversity that results from applying the diversity techniques with MOSA to the increase resulting from Monotonic GA, we can clearly see that the increase with MOSA is not as high as with Monotonic GA. A possible reason behind that is the default Monotonic GA is not as efficient as the default MOSA in promoting higher diversity, and therefore applying a diversity maintenance technique will definitely result in a better diversity level than the level achieved by the default one.

In order to understand the impact of population diversity on the performance of the two algorithms, we compare the final coverage and length that are resulted from applying each diversity maintenace

technique on both algorithms, as shown in Table 4.9. In general, we clearly see that applying diversity techniques to both algorithms reduce coverage and increase length although there are techniques that result in slightly similar or better coverage (e.g., DE-predicate achieves trivial increase in coverage when compared to the default Monotonic GA). When comparing the performance of each technique with both algorithms, there are techniques that lead to better branch coverage when applied with the Monotonic GA such as AR and CL despite the small difference in the coverage with both algorithms, while the other techniques result in higher coverage when applied with MOSA. However, it is not possible to compare the resulting length from both algorithms as the length in the WSA refers to the statements of an individual test suite while the length in MOSA is the number of statements of an individual test case.

One important part of our investigation is to look at the influence of the adaptive diversity on the performance on both algorithms. In Table 4.9, it is obvious that the adaptive diversity reduces the negative impact on length and possibly improve the coverage. The fitness-based adaptive version leads to slightly similar coverage to the default GA although there is no considerable difference between the coverage achieved by this version and the others, but still result in better coverage than the naive diversity. In terms of length, applying the diversity adaptively does not increase the length as with the naive diversity approach, and this is the case with both algorithms.

Our findings regarding the effect of diversity on coverage and length conform to those obtained in the study by Vogel et al. [172, 173] where increasing diversity does not have an effect on coverage, but it finds more faults. Also, increasing diversity is found to have a negative impact on the length of tests (i.e., longer test sequences).

Despite the importance of understanding the influence of population diversity on the generation of unit tests, it is also important to understand how the landscape structure affects the population diversity. As mentioned in Section 2.2.5.1, maintaining population diversity can be beneficial, especially with a rugged landscape, to avoid stagnation in local optimum and ensure that the search space is well explored. This leads to the question of whether the landscape structure has an impact on the diversity level (i.e., does an increase in landscape ruggedness/neutrality negatively affect the diversity of population?). Therefore, the next section investigates how the landscape features influence the diversity level, especially with an increase in ruggedness.

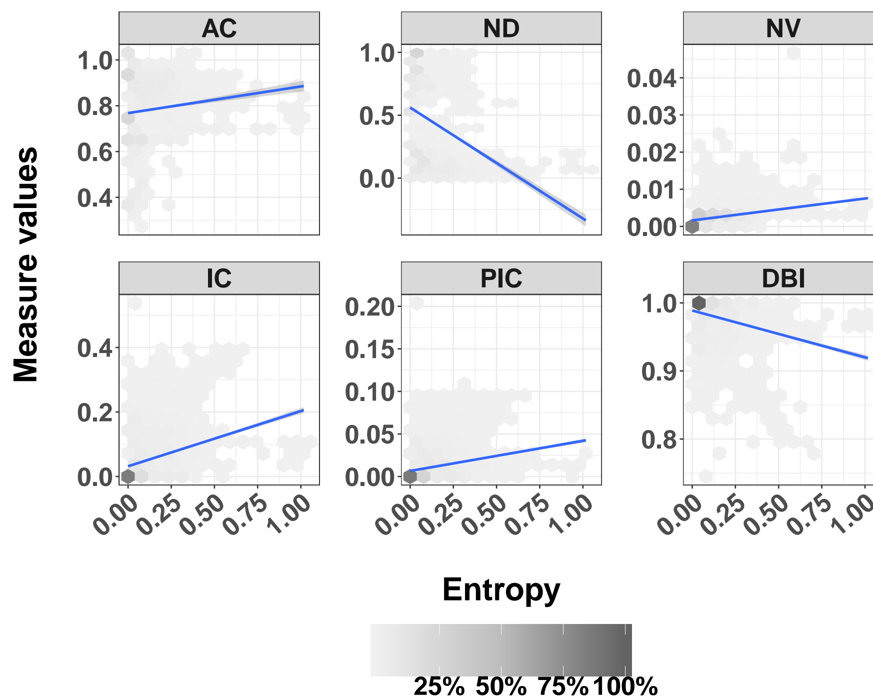


Figure 4.13: The Spearman correlation of entropy based on default MOSA with each of the six measures for all the branches of 331 classes. The correlation coefficient of entropy and AC is −0.028, ND is −0.58, NV is 0.61, IC is 0.66, PIC is 0.64, and DBI is −0.59.

4.5 HOW DOES THE LANDSCAPE STRUCTURE AFFECT THE POPULATION DIVERSITY?

In order to understand the impact of the landscape structure on the population diversity, we investigate the Spearman correlation of the entropy measure and each of the six landscape measures (Section 3.3.3.1) when considering MOSA. In this case, the entropy and each landscape measure are calculated for each objective (i.e., branch) and the two resulting values are then correlated, as shown in Figure 4.13.

There is always a significant correlation between the entropy and each of the measures with p-value < 0.001, but the difference lies in the strength of the correlation (i.e., the correlation coefficient) where the strongest correlation is observed with the IC measure (0.66) and the weakest correlation is observed with the AC (−0.028). In general, we clearly see that an increase in population diversity tend to correlate to an increase in landscape ruggedness (i.e., high entropy indicates high diversity) which is confirmed by all the measures except AC. It should be noted that an increase in landscape ruggedness does not necessarily cause the population to be more diverse. Looking at the ND measure, the negative correlation between entropy and ND (−0.58) suggests that cases with a large neutrality distance (that is, long sequences of neutral steps in the random walk) tend to have a

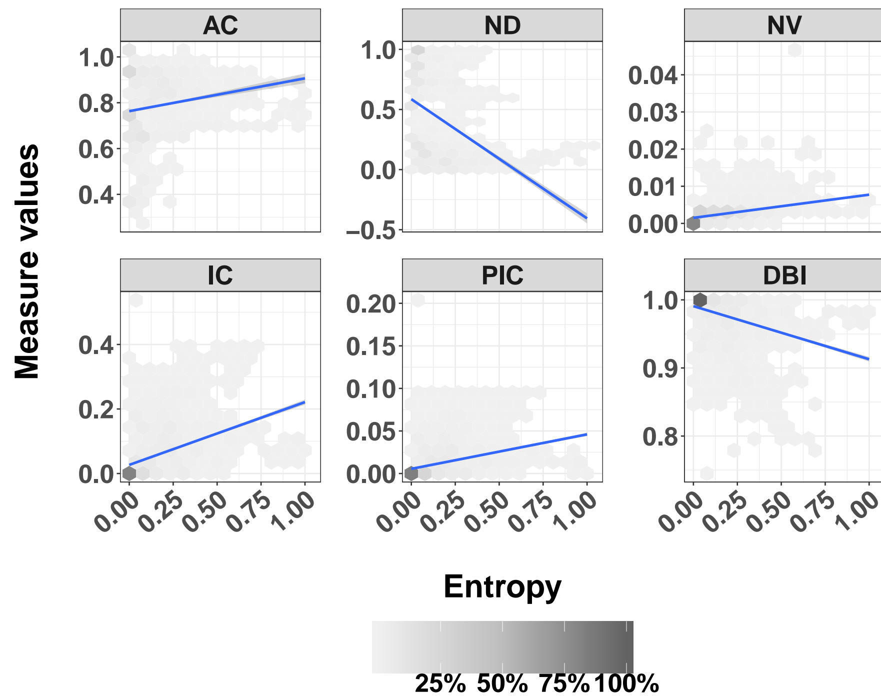


Figure 4.14: The Spearman correlation of entropy based on AFS-fitness with each of the six measures for all the branches of 331 classes. The correlation coefficient of entropy and AC is −0.045, ND is −0.64, NV is 0.65, IC is 0.72, PIC is 0.71, and DBI is −0.70.

lower level of population diversity. The correlation between entropy and NV (0.61) indicates that the increase in neighbouring areas of neutral individuals (i.e., few flat areas in the landscape) relates to higher diversity level.

For the information-based measures, the correlation between entropy and IC (0.66) indicates that a high entropy value corresponds to a high IC value, which means that an increase in landscape ruggedness relates to an increase in population diversity. This is also shown in the correlation between the entropy and PIC (0.64) as a high entropy value corresponds to a high PIC value. A large PIC value indicates a high landscape modality. This correlation between entropy and PIC indicates that population can be more diverse on a multimodal landscape. The negative correlation between entropy and DBI (−0.59) indicates that a high entropy value corresponds to a low DBI value. According to the definition of DBI, a low DBI value is an indicator of a high density of peaks and few flat areas in the landscape. The negative correlation between entropy and DBI suggests that there might be a possibility that population diversity increases with an increase in rugged areas in the landscape.

As the correlated entropy is based on the default MOSA (i.e., without incorporating any diversity maintenance techniques), we ask a question of whether promoting diversity makes any difference in the

correlation between the landscape measures and entropy. Therefore, we consider the entropy when applying the adaptive fitness sharing based on fitness differences (AFS-fitness) and correlate it with the landscape measures, as shown in Figure 4.14.

It is obvious that there is still a correlation between entropy and each of the landscape measures as the correlation still indicates that an increase in landscape ruggedness correlates to higher population diversity. In fact, promoting diversity shows a stronger correlation between entropy and the landscape measures, which is observed with all the measures. For example, the correlation coefficient between entropy and IC when promoting diversity is 0.72 while the correlation coefficient with the default MOSA is 0.66.

Therefore, based on these results, landscape plateaus seem to decrease the population diversity and ruggedness can be beneficial to diversity. A possible reason of why ruggedness leads to an increase in population diversity is that ruggedness indicates the existence of gradients that usually lead to many small variations in the fitness values, which is the case that is observed with the branches that are easy to cover by GAs (Section 3.5). Thus, it is expected that the frequent change in fitness values caused by the existence of gradients leads to high entropy.

4.6 summary

The loss of population diversity is a common problem that occurs during the search of a GA. This often leads to prematurely converge on suboptimal solutions that might not even be local optimal solutions. This reduces the effectiveness of the GA, and in the case of search-based test generation, the code coverage is reduced. Therefore, it is important to maintain the diversity of population to avoid this issue. In this chapter, we investigate the impact of population diversity on the generation of unit tests by (i) measuring the diversity level during the evolution with both algorithms (Monotonic GA using WSA and MOSA), (ii) applying diversity maintenance techniques to enhance the population diversity, (iii) studying the impact of increasing diversity on the generation.

Measuring the diversity of generated unit tests based on entropy, genotypic, and phenotypic levels suggest that (i) the default Monotonic GA is not as efficient as the default MOSA in maintaining higher diversity and (ii) applying diversity maintenance techniques on the two algorithms are very effective at promoting diversity throughout the evolution, which is more obvious with Monotonic GA as the increase in diversity caused by these techniques with MOSA is not as high as with Monotonic GA.

Looking at their effect on the performance of GAs, we see that increasing diversity leads to (i) reduced coverage although there are

techniques that result in slightly similar coverage, and (ii) a possible increase in the length. However, the results of applying the adaptive approach suggest that adaptive diversity reduces the negative impact on length and, to some extent, improves the coverage, especially with the adaptive fitness-based sharing (AFS-fitness). The increase in the individual length is caused by adding more statements to an individual test that eventually leads to a decrease in the number of executed generations during the search, which is more obvious with the naive diversity approach. Investigating the type of statements that are added when increasing diversity reveals that there is no a specific statement type that happens with enforced diversity.

Finally, investigating whether the landscape structure influences population diversity reveals that an increase in landscape ruggedness leads to higher population diversity.

5 AN ANALYSIS OF THE EFFECTS OF TEST CASE REDUCTION ON UNIT TEST GENERATION

5.1 INTRODUCTION

In the last two chapters, we observe that the individual representation is an important constituent of the GA that influences the generation of unit tests. This can be seen when analysing the fitness landscape where an individual of a test suite leads to generate more plateaus in the fitness landscape than an individual of a test case. Also, the individual representation has an impact on the population diversity where the GA with a test case representation (MOSA) promotes higher diversity level than the GA with a test suite representation (Monotonic GA using WSA).

However, despite the success of the test case representation in reducing the presence of plateaus in the fitness landscape, it still produces plateaus that are detrimental to the search. This, in fact, led us to conduct further investigation of what makes consequent mutations during the random walk result in equal fitness, which indicates the presence of plateaus in the landscape. This implies that applying the mutation on an individual test case does not lead to an effective change that affects the fitness. Investigating the reasons behind this issue reveals that the test case representation has structurally non-effective code. This non-effective code is a result of the mutation operator that, for example, inserts statements that might seem to be useless, and have no positive impact on the final fitness value. Such statements are just duplicates of other statements in a test case in a way that they both perform similarly (e.g., both call one method with the exact test input). We call these statements as *redundant* statements in the test case. To illustrate that, consider the test case in Listing 5.1 that is a result of a mutated test case.

Looking at the statements of the test case, it is obvious that each of the four test inputs (i.e., `int0`, `int1`, `int2`, and `int3`) is used multiple times to test the `foo` method. This means that the mutation either inserts or changes statements that test the `foo` method with a test input that is already used before to test the same method. For example, the statement in Line 4 performs exactly to the statement in Line 3 where they both test the `foo` method with the input `int0`. This is also the case with the statements in Line 9, 12, and 13, which are the redundant statements in this test case.

```
1.  ArtClass artClass0 = new ArtClass();
2.  int int0 = (-4643);
3.  boolean boolean0 = artClass0.foo(int0);
4.  boolean boolean1 = artClass0.foo(int0);
5.  int int1 = 2115;
6.  boolean boolean2 = artClass0.foo(int1);
7.  int int2 = (-2139);
8.  boolean boolean3 = artClass0.foo(int2);
9.  boolean boolean4 = artClass0.foo(int0);
10. int int3 = (-2220);
11. boolean boolean5 = artClass0.foo(int3);
12. boolean boolean6 = artClass0.foo(int1);
13. boolean boolean7 = artClass0.foo(int2);
```

Listing 5.1: A test case with redundant statements

```
1.  ArtClass artClass0 = new ArtClass();
2.  int int0 = (-4643);
3.  boolean boolean0 = artClass0.foo(int0);
4.  int int1 = 2115;
5.  boolean boolean2 = artClass0.foo(int1);
6.  int int2 = (-2139);
7.  boolean boolean3 = artClass0.foo(int2);
8.  int int3 = (-2220);
9.  boolean boolean5 = artClass0.foo(int3);
```

Listing 5.2: A test case without redundant statements

Moreover, this issue is observed when applying a diversity maintenance technique where higher population diversity leads to an increase in the individual length. Investigating such increase in the length demonstrates that there are many redundant statements that are just duplicates of other statements in the test case, similar to the case shown in the example above. Besides the negative impact of these statements on the length, their existence in the test case makes it harder for the mutation to apply the change operation effectively. In other words, there is a low probability that the mutation changes a statement that possibly improves the fitness, for example, changing the test input in one of the four statements (Line 2, 5, 7, and 10) to test the `foo` method with a new test input that might lead to cover an uncovered branch. This, in fact, leads to the intuition that such statements should be removed in order to reduce the negative impact on length and mutation during the evolution. Applying this intuition to the test case shown above results in the updated test case shown in Listing 5.2.

Therefore, and based on the idea mentioned above, we adapt the approach of *test case reduction* that removes redundant statements after each test case mutation during the evolution. In particular, the aim of this chapter is to investigate whether removing redundant statements leads to (i) decrease the presence of plateaus in the fitness landscape, (ii) reduce the negative effect on length when promoting population diversity, and thus (iii) improve the performance of the GA in achieving better branch coverage.

This chapter is organised as follows: Section 5.2 presents related concepts to the test reduction and their applications to different domains. Section 5.3 defines in detail the rules that are followed when applying the test reduction approach. Section 5.4 presents the research questions in which this chapter aims to address and the conducted experiments to answer these questions. Finally, Section 5.5 gives an overall discussion of the findings of this investigation.

5.2 BACKGROUND

The individual representation is an important aspect of Evolutionary Algorithms (EAs) that plays an important role in solving an optimisation problem. However, when an EA considers an individual representation of variable length, a problem known as *bloat* may occur [25]. Bloat is a phenomenon that denotes the rapid growth in the size of individuals when evolution progresses with no considerable effect on the fitness. This leads to several issues such as the consumption of resources that are available to run the EA (e.g., machine RAM), the slow search progress as longer individuals need longer time to evaluate and thus result in few generations, and in the domain of test generation, the difficulty in understanding the final test suite that needs to be manually evaluated.

Genetic Programming (GP) is a well known form of EAs that suffers from the bloat problem [95] where programs are evolved such that a program is represented as a syntactic tree. When evolving programs, they tend to have unnecessary subtrees (also known as *introns*) that increase the individual size with no contribution to the program fitness. Although the presence of introns can be beneficial (i.e., avoiding the destructive effect of the crossover operator), their removal is found to be promising in improving the GP performance [27]; introns are detected and removed during evolution.

In the domain of test case generation, the bloat problem has been investigated where different techniques are applied during the evolution to control bloat [61]. For example, the bloat is prevented by (i) giving a high rank to tests that are shorter when considering the rank selection and (ii) setting a maximum length of a test case where an upper limit of the number of statements cannot be exceeded, especially when inserting new statements by the mutation operator.

Avoiding lengthy test cases is not only important during the search, it is also essential that the final test suite should be as short as possible to make it more understandable for users. This is known as *test suite minimisation* that can be applied as removing each unnecessary statement in each test case one at a time and then re-executing the reduced test suite to ensure the coverage does not decrease [13]. This process is repeated until reaching the minimum length of the reduced test suite that achieve similar coverage to the non-reduced test suite. However, the test suite minimisation approach is not only applied to remove statements of test cases, it is also considered to eliminate duplicate test cases that have no effect on the test suite coverage. The latter case is referred to as *test suite reduction* that aims to reduce the size of a test suite while maintaining its effectiveness by selecting a minimal subset of test cases from the test suite that satisfies the testing goals that are satisfied by the original test suite [182]. For example, when branch coverage is the considered testing goal, the reduced test suite is the minimal subset of test cases that achieve similar branch coverage as to the non-reduced test suite. To achieve that, different test suite reduction techniques have been proposed [182] such as the greedy approach that iteratively selects a test case that satisfies as maximum unsatisfied goals as possible. Other studies also proposed different approaches, especially in the unit test generation, that result in minimised unit tests that are effective in detecting defects, for example, using the program slicing and delta debugging [98].

The previously mentioned approaches attempt to reduce the length of an individual and minimize the final test suite as a post-processing step, except the bloat control techniques that are applied within the generation process [61]. Our approach is similar to the test suite minimisation approach that is applied on the final test suite [13] except that only unnecessary statements are removed without removing unnecessary test cases and the removal occurs during the evolution. The reason why we focus on removing redundant statements from test cases during the evolution is to mainly investigate that effect of such reduction on the fitness landscape and the performance of diversity maintenance techniques.

5.3 TEST CASE REDUCTION RULES

In this section, we demonstrate how test case reduction is applied, and more specifically the rules of removing redundant statements (*reduction rules*) are explained. In general, the removal of a statement from a test case relies on the statement type, its dependency with the other statements in the test case, and its redundancy. As there are different possible types of statements in a test case, we define the reduction rules based on three generic statement types that are:

- **Constructor** statement that creates a new object such as the statement in Line 1 in Listing 5.3.
- **Assignment-by-value** statement that explicitly initialises a new value that is used as a test input. This includes primitive, array, and null statements. An example is the statement in Line 2 in Listing 5.3.
- **Assignment-by-return** statement that implicitly initialises a new value by, for example, calling a method that returns a value such as the statement in Line 3 in Listing 5.3. Note that the declared variable in such statements (e.g., `boolean0`) might be used as a test input by other statements, for that it is considered as an implicit initialisation.

```
1.  ArtClass3 artClass3_0 = new ArtClass3(); //Constructor
2.  String string0 = "9|Xo #{";     //Assignment-by-value
3.  int int0 = artClass3_0.zoo(string0); //Assignment-by-return
4.  boolean boolean0 = artClass3_0.foo(int0);
5.  int int1 = artClass3_0.zoo(string0);
6.  String string1 = "G/]@WL-N7pT4r.-9H4c";
7.  int int2 = artClass3_0.zoo(string1);
8.  int int3 = artClass3_0.zoo(string1);
9.  boolean boolean1 = artClass3_0.foo(int1);
10. int int4 = artClass3_0.zoo(string0);
11. boolean boolean2 = artClass3_0.foo(int3);
12. String string2 = null;
13. int int5 = artClass3_0.zoo(string2);
```

Listing 5.3: A test case with redundant statements to explain the reduction rules

Another important factor that must be considered before removing a statement is the dependency between this particular statement and the other statements in the test case. This is because removing a statement (e.g., initialising a new value as test input) that other statements depend on (e.g., calling a method with that particular test input) causes a failure in the test case. There are two types of dependency between the statements of a test case. A statement x depends on a statement y such that y either creates an object or initialises a test input used by x, for example, the statement in Line 3 in Listing 5.3 depends on the statements in Lines 1 and 2. In this case, the statement x (Line 3) is called *dependent* statement. In contrast, the statement y (Line 1 and 2) is called *dependee* statement as it is depended on by the statement x.

Besides the statement type and dependency, the statement redundancy is another factor to be considered to decide whether a statement is removed or kept. A statement is considered redundant if it performs similarly to other statements such as calling a method with one exact

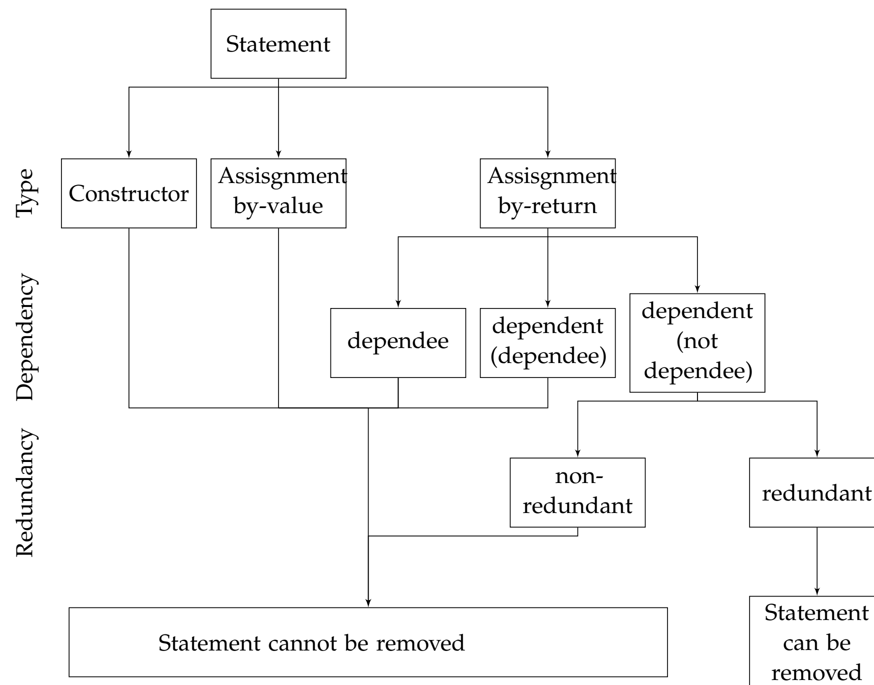


Figure 5.1: An overview diagram that demonstrates the reduction rules

test input. For example, both statements in Line 3 and 5 in Listing 5.3 test the `foo` method with the same test input `string0`. In this case, both statements are called redundant statements in the test case. However, the removal of a statement cannot only be based on the type, dependency, or redundancy (i.e., relying only on one of these three factors to remove the statement), but in fact, all the three factors must be considered to decide if the statement should be removed, which define the reduction rules as explained in Figure 5.1.

The removal of a statement in a test case follows five rules that rely on the statement type, dependency, and redundancy. For each statement, we first look at its type; if the statement is *Constructor* or *Assisgnment-by-value* then the statement is not removed as the created object or the initialised test input is possibly used by statements that are either inserted or changed during the mutation applied in later generations (Rule 1). In case the statement is *Assisgnment-by-return* then we look at its dependency where there are three possible cases; if the statement is *dependee* (i.e., depended on by other statements) then the statement is not removed since removing such a statement causes the failure of the test case (Rule 2). When the statement is *dependent*, there are two possible cases that are either (i) the statement is also *dependee* (e.g., statement in Line 3 in Listing 5.3) in which the statement is not removed (Rule 3) or (ii) the statement is not *dependee* such that no other statements depend on this one, and in this case, we look at its redundancy. If the statement is *non-redundant* then the statement is

not removed (Rule 4), otherwise the statement is removed since it is a duplicate of other statements (Rule 5).

To better understand the reduction rules, we apply the six rules on the statements of the test case shown in Listing 5.3. For simplicity, we refer to each statement by the line number such as statement 1 refers to the statement in line 1. Statement 1 cannot be removed according to the first rule. Statements 2, 6, and 12 cannot also be removed according to the first rule. However, the other statements follow the remaining rules where they all of *Assisgnment-by-return* type. In the case of statement 3, it is dependent and dependee as well since statement 4 considers its initialised input, which makes it difficult to remove this statement based on the third rule. The same rule is also followed with statement 8 as statement 11 depends on it, and thus cannot be removed. Moreover, statement 4, 11, and 13 are dependent but not dependee and when looking at their redundancy, they all are non-redundant; statement 4 is is the only statement that tests `foo` method with the input `int0`, statement 11 is non-redundant because no other statements test `foo` method with the input `int3`, and statement 13 is the only the statement that tests `zoo` method with the input `string2`. Therefore, and according to the fourth rule, these three statements are not removed. When looking at statement 9, one would expect that this statement should not be removed as it meets the requirements of the fourth rule. In practice, this statement can be removed as it is redundant since it test the `foo` method with the input `int1` (statement 5) that is exactly like the input `int0` (statement 3), which is already tested by statement 4. In this case, both statement 5 and 9 are removed as they conform to the fifth rule. Finally, statements 7 and 10 can be removed as they are redundant (i.e., statement 7 is a duplicate of statement 8 whereas statement 10 is a duplicate of statement 3) according to the fifth rule. Therefore, the reduced version of the test case is as follows:

```
ArtClass3 artClass3_0 = new ArtClass3();
String string0 = "9|Xo #{";
int int0 = artClass3_0.zoo(string0);
boolean boolean0 = artClass3_0.foo(int0);
String string1 = "G/]@WL-N7pT4r.-9H4c";
int int3 = artClass3_0.zoo(string1);
boolean boolean2 = artClass3_0.foo(int3);
String string2 = null;
int int5 = artClass3_0.zoo(string2);
```

Listing 5.4: The reduced version of the test case in Listing 5.3

Note that variables such as `int3`, `int5`, and `boolean2` are changed to `int1`, `int2`, and `boolean1`, respectively, according to the new order of the statements, but they are not changed in the reduced test case to make it easier for comparison.

5.4 EMPIRICAL STUDY

The aim of our study is to understand the impact of test case reduction on the individual size during the evolution and the performance of the GA, and whether reducing test cases avoids the presence of detrimental plateaus in the fitness landscape and the negative influence on length when increasing population diversity. Therefore, we design our study to answer the following research questions:

RQ 1: How does the test case reduction affect the fitness landscape properties?

RQ 2: How does the test case reduction affect the population diversity during evolution?

RQ 3: What is the effect of test case reduction on the performance of WSA and MOSA?

The test case reduction is expected to have a positive impact on the landscape structure as it is more likely to reduce the presence of plateaus in the landscape since removing redundant statements will possibly increase the chance of successful mutations (i.e., mutations that lead to changes in fitness values during the random walk). In terms of its impact on population diversity, we expect that applying the test case reduction approach possibly decreases the diversity, especially the genotype diversity as removing statements might increase the similarity between individuals. However, it is expected to alleviate the negative impact of increasing population diversity on the length as the reduction approach removes statements that possibly increase the individual length. Finally, the effect of test case reduction approach on the GA performance is anticipated to be a positive effect since removing redundant statements is more likely to enhance the mutation of tests that possibly improves the fitness during the evolution.

5.4.1 *Experimental Setup*

As this study is related to the studies that are presented in the last two chapters, we therefore consider a similar experimental setup to what is considered in those studies. We used EvoSuite to generate JUnit test suites for a given Java CUT and target coverage criterion using the Monotonic GA, which applies the WSA approach, and MOSA. Besides the default settings in EvoSuite [16], we consider branch coverage as target criterion, and no test archive when running the Monotonic GA. As a corpus of classes, we use the selection of 346 complex and non-trivial classes from the DynaMOSA study [127] where the complexity of classes ranges from 2 to 7939 branches.

5.4.1.1 *Experiment Procedure*

To answer the three research questions, we conducted similar experiments to those conducted in the last two chapters but with applying the test case reduction approach. More specifically, we first ran an experiment to understand the effect of test case reduction on the fitness landscape properties, similar to the experiment in Section 3.2.1.3 and Section 3.3.1, where the random walk is applied on each CUT and then the six fitness landscape measures are applied on the sequence of fitness values obtained by the landscape walks. The difference with this experiment is that the test case reduction is applied on each test case, and each test case of an individual test suite, of every step in the random walk.

Then, we ran another experiment to investigate whether the test case reduction approach affects the diversity level during the evolution, especially the individual length, when promoting population diversity, which is similar to the experiments in Section 4.2.3.2 and Section 4.3.3.1. However, this experiment only considers the adaptive fitness sharing based on fitness differences (AFS-fitness) as it seems the most promising technique in increasing diversity with the lowest negative effect on length. In this case, we run four versions of the GA; the default GA, the GA with the reduction approach, the GA with AFS-fitness, and the GA with AFS-fitness and reduction. The test case reduction approach is applied on each test case (i.e., including test cases of an individual test suite) after applying the mutation operator.

These two experiments involve running the default versions of the two algorithms (i.e., Monotonic GA using WSA and MOSA) in order to compare their performance against the random walk in the first experiment and the performance of the diversity technique in the second experiment when considering the test case reduction approach. However, running the two algorithms with the reduction approach is required to answer the third research question in which the performance of the default version of the two algorithms is compared to their performance when applying the reduction approach (i.e., whether the reduction approach affects the branch coverage).

For a fair comparison, we consider the same 331 classes that resulted from running the fitness landscape analysis experiment on the corpus of 346 classes with the first experiment. We also consider the same 311 classes that are used in the population diversity analysis experiment with the second experiment. This allows to compare the results of the previous experiments (without the reduction approach) to the results of the new experiments (with the reduction approach).

5.4.1.2 *Experiment Analysis*

In order to decide whether the reduction approach influences the landscape structure, we statistically compare the results of each landscape

measure obtained from the two versions of random walks, for example, we compare the result of AC measure when applied on the default random walk (i.e., AC_d) to its result when applied on the reduction-based random walk (i.e., AC_r). For that, we use the Vargha-Delaney's $\hat{A}_{12}$ effect size to evaluate whether a reduction-based landscape measure x (e.g., AC_r) indicates better improvement in the landscape structure than the default landscape measure y (e.g., AC_d); when $\hat{A}_{xy} > 0.5$ then the reduction approach is better in reducing plateaus, which is considered as an improvement in the landscape, whereas $\hat{A}_{xy} < 0.5$ indicates the default version is better than the reduction approach. Furthermore, we consider the Wilcoxon Mann-Whitney statistical test at a level of $\alpha = 0.05$ to determine if there is statically significant difference in the landscape structure with a high number of classes.

To observe the similarity in the diversity achieved by the default GA and each of the other three techniques (i.e., GA-reduction, AFS-default, and AFS-reduction), we use the Vargha-Delaney's $\hat{A}_{12}$ effect size measure. When $\hat{A}_{xy} > 0.5$ then the other techniques results in better diversity level than the default GA, and vice versa. This is also considered to evaluate the difference based on coverage and length. To analyse the effect of test reduction on the performance of a GA, we consider the branch coverage as the main measurement of a GA performance (i.e., whether the reduction approach increases/decreases the coverage). Therefore, we use both Vargha-Delaney's $\hat{A}_{12}$ effect size measure and Wilcoxon Mann-Whitney statistical test to statically validate whether there is a difference in the achieved coverage between the default GA and its reduction-based version.

5.4.1.3 *Threats to Validity*

To control threats of the stochastic behaviour of the considered techniques, i.e., Monotonic GA, MOSA, and a random walk, we repeated the experiment 30 times. In order to have a fair comparison with the results obtained in Chapter 3 and Chapter 4, we considered a similar corpus of 346 Java classes and similar search budget. Since the test case reduction approach is applied on each individual test after being mutated, the results of applying the test case reduction may vary when considering different mutation probabilities (i.e., insert, change, and remove test cases/statements probabilities) or even considering a different mutation operator. Moreover, the scope of this study is limited to apply the test case reduction only when a test case is mutated and therefore it is likely that the obtained results differ when applying the test case reduction on each offspring resulting from the crossover or any newly generated individuals during the evolution. To investigate the impact of test case reduction on population diversity, we considered the best technique in achieving high diversity with the minimum negative effect on coverage and length. However, consid-

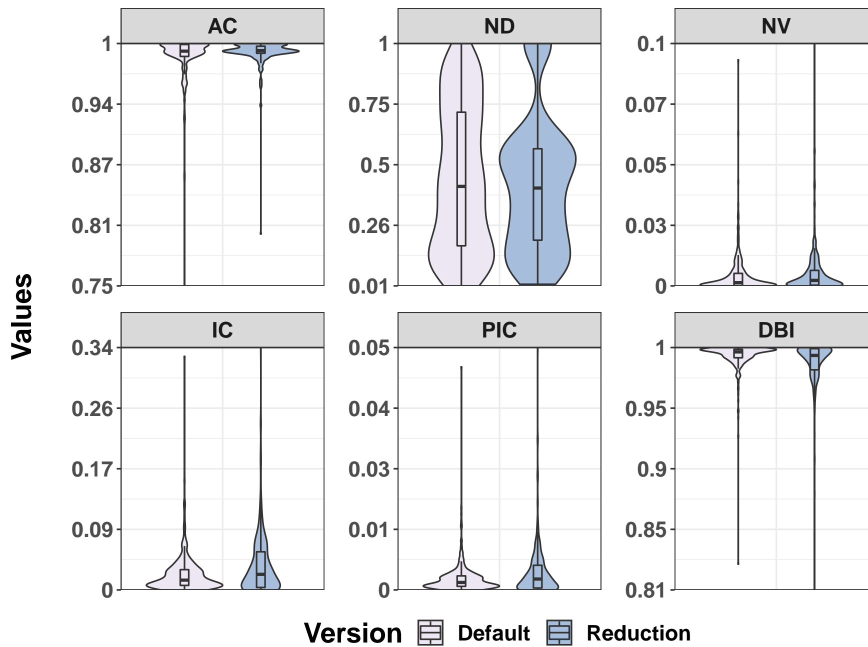


Figure 5.2: Comparison of the six fitness landscape measures with the WSA approach based on the default and reduction versions of the random walk.

ering other diversity maintenance techniques may result in slightly different outcomes.

5.4.2 *Experimental Results*

This section presents the results of the conducted experiments and discusses the answers to the three research questions.

5.4.2.1 *How does the test case reduction affect the fitness landscape properties?*

In order to understand the effect of test case reduction on the structure of the fitness landscape, we compare the results of the fitness landscape measures when applied on the two versions of the random walk; the default version (i.e., no reduction) and the reduction-based version. As we consider two different algorithms, this comparison is considered with each algorithm where Figure 5.2 shows the results with the WSA approach and Figure 5.3 shows the results with MOSA. In general, we observe that almost all the measures indicate that applying the reduction approach leads to a slight improvement in the structure of the landscape such that plateaus are smaller with the reduction version that with the default version.

In the case of the WSA approach, all the measures show that there is little decrease in the size of landscape plateaus except the AC measure. This is also confirmed in the results shown in Table 5.1 where the

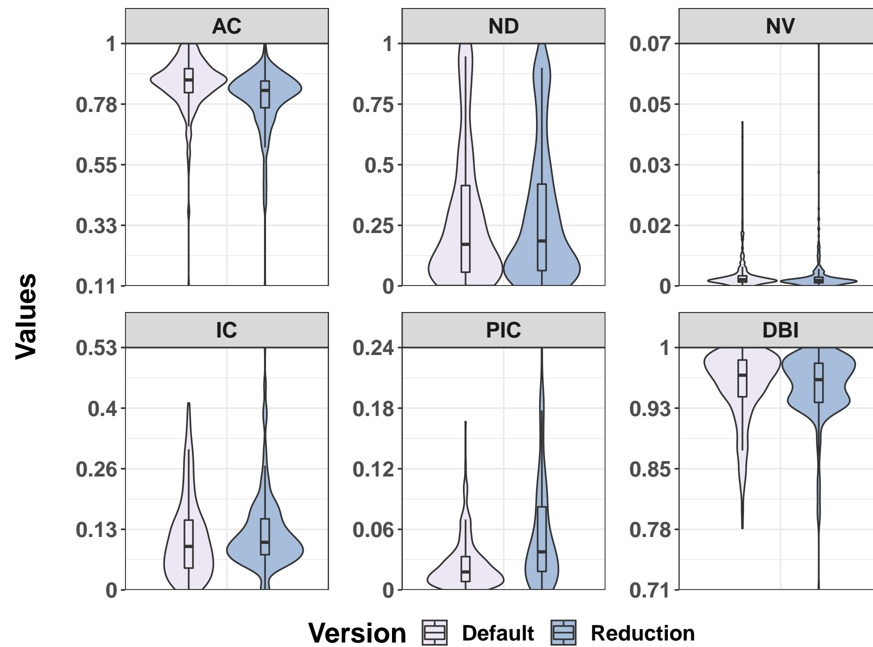


Figure 5.3: Comparison of the six fitness landscape measures with the MOSA based on the default and reduction versions of the random walk.

average effect size over the classes demonstrates that the reduction of unnecessary statements appears to slightly avoid the presence of plateaus and, in addition, increases the number of classes that show significantly better results than the default version. The AC measure indicates that the use of the reduction approach still results in values that are higher than 0.9 and thus does not make changes in the structure of the fitness landscape. In fact, it shows that the reduction approach removes those cases that indicate the existence of rugged areas, which probably leads to $\hat{A}_{12} = 0.48$ that is the default is somewhat better than the reduction version. In the case of neutrality measures, the ND measure shows that the number of neutral steps made at the start of the random walk with the reduction version (42%) is slightly lower than the default version (45%). Moreover, the reduction approach is able to slightly increase the number of neighbouring areas of individuals with equal fitness during the random walk as shown by the NV measure where the NV ≈ 3 with the default version and NV ≈ 6 with the reduction version.

For the information-based measures, the IC measure indicates that the use of reduction approach slightly increases the size of beneficial rugged areas and decreases flat areas in the landscape where IC with the reduction version (≈ 0.01) is a little higher than with the default version (≈ 0.04). This also can be seen with the PIC measure where the reduction approach can, to some extent, increase the modality of the landscape, and thus minimise plateaus as PIC is slightly increased with the reduction version. As a high value of DBI indicates a landscape

Table 5.1: Number of significant classes for the comparison of landscape measures with and without reduction when considering the two algorithms, and the average effect size $\hat{A}_{12}$.

Algorithm	Measure	$\hat{A}_{12}$	Number of sig. classes		
			Total	Default better	Reduction better
WSA	AC	0.48	63	40	23
WSA	ND	0.52	85	31	54
WSA	NV	0.53	71	15	56
WSA	IC	0.57	138	62	76
WSA	PIC	0.56	122	53	69
WSA	DBI	0.54	58	21	37
MOSA	AC	0.54	72	49	23
MOSA	ND	0.50	1	0	1
MOSA	NV	0.51	4	1	3
MOSA	IC	0.55	63	28	35
MOSA	PIC	0.58	116	38	78
MOSA	DBI	0.57	91	9	82

dominated by plateaus, the results of DBI with the reduction approach demonstrates that considering the test case reduction helps in slightly minimising the size of plateaus in the landscape since the reduction version results in lower DBI values than the default version. However, it should be noted that the three information-based measures suggest that the use of reduction approach result in cases that seem to point to the existence of rugged areas in the landscape, for example, there are few cases with the reduction version that result in higher PIC than the with the default version.

In the case of MOSA, all the measures indicate that the reduction approach positively affects the landscape structure in slightly decreasing the presence of plateaus except the ND measure that demonstrates no difference in the landscape properties between the default and reduction versions. The AC measure shows that the reduction approach leads to slightly few flat areas in the landscape than the default version with no effect on the cases that point to the existence of rugged areas in the landscape (i.e., those cases with $AC < 0.33$). The ND measure, however, indicates that the reduction approach does not affect the number of neutral steps made at the start of the random walk when compared to the default version, while the NV measure shows a trivial increase in the number of neighbouring areas of individuals with equal fitness during the random walk when considering the reduction approach where $NV \approx 5$ with the default version and

NV ≈ 7 with the reduction approach. In fact, the NV measure shows that the reduction approach results in cases with high NV values (i.e., $NV > 0.05$) that indicate more rugged areas in the landscape. The three information-based measures demonstrate that considering the reduction approach results in slightly few flat areas in the landscape, and in addition, more cases that seem to point to the existence of rugged areas in the landscape where the PIC measure shows the highest difference between the default and the reduction version.

> ***RQ1****: The test case reduction approach seems to affect the landscape properties by slightly decreasing plateaus and, to some extent, increasing ruggedness.*

5.4.2.2 *How does the test case reduction affect the population diversity during evolution?*

In this section, we look at how applying the test case reduction approach affects the population diversity level during the evolution. For that, we measure the diversity when the reduction approach is applied to each of the two algorithms, and compare it to the default version (i.e., no-reduction). This includes the achieved coverage throughout the evolution and also the average size of the individuals in the population.

Figure 5.4 shows the achieved coverage, length, and the three diversity measures throughout the evolution when considering the default and reduction versions of both the Monotonic GA using WSA and the adaptive fitness-based sharing (AFS). In terms of coverage, we clearly see that there is no difference between the four techniques as they achieve similar coverage during the search, which is confirmed by the average effect size shown in Table 5.2 where the difference between the coverage achieved by the default Monotonic GA and the coverage achieved by the other three techniques is trivial. In the case of the length, it is obvious that applying the reduction does not lead to a considerable decrease in the size of test suites during the evolution since the difference in the length between the default AFS and its reduction-based version is very small although the latter resulted in slightly lower length during the first five minutes of the search, and in addition, there is no high difference between length with the default Monotonic GA and its reduction-based version.

In terms of diversity, the three diversity measures demonstrate similar behaviour achieved by each of the four techniques (i.e., similar resulting diversity level with each diversity measure). Overall, the default AFS seems to achieve the highest diversity level during the evolution and its reduction-based version is slightly lower whereas the difference between the default Monotonic GA and its reduction-based version is trivial, which is confirmed by the average effect size shown in Table 5.2.

Table 5.2: The average effect size $\hat{A}_{12}$ computed for coverage, length, and the diversity measures with the default GA and default AFS/reduction-based GA/reduction-based AFS.

Algorithm	Technique	Cov.	Len.	Ent.	Phen.	Gen.
WSA	AFS (default)	0.49	0.42	0.63	0.62	0.58
WSA	AFS (reduction)	0.48	0.43	0.61	0.64	0.57
WSA	GA (reduction)	0.49	0.51	0.48	0.5	0.5
MOSA	AFS (default)	0.48	0.49	0.56	0.53	0.54
MOSA	AFS (reduction)	0.47	0.64	0.54	0.51	0.48
MOSA	GA (reduction)	0.47	0.65	0.49	0.47	0.43

Looking at the case of MOSA in Figure 5.5, we see that there is no considerable difference in the coverage as all the techniques reach similar branch coverage during the evolution except the reduction-based AFS that results in slightly lower coverage. Interestingly, the reduction approach has an obvious impact on the length as the reduction-based versions of MOSA and AFS result in smaller test suites than their default versions, which is confirmed by the effect size where the reduction versions are better in minimizing length. In the case of diversity, the entropy measure demonstrates that the four techniques achieve similar diversity level to the level achieved with the Monotonic GA except that there is no high difference between the diversity resulting from the two versions of AFS and the versions of MOSA; the default AFS is the highest followed by its reduction-based version and the default MOSA is slightly higher than its reduction-based version. The phenotype measure indicates nearly similar diversity level achieved by the four techniques to what is shown by the entropy measure except that default MOSA seems to achieve slightly better diversity than the reduction-based AFS. However, the genotype measure shows that the reduction versions of both MOSA and AFS lead to drop in the diversity level where the reduction-based MOSA results in the lowest diversity level whereas the reduction-based AFS is higher but still lower than the two default versions.

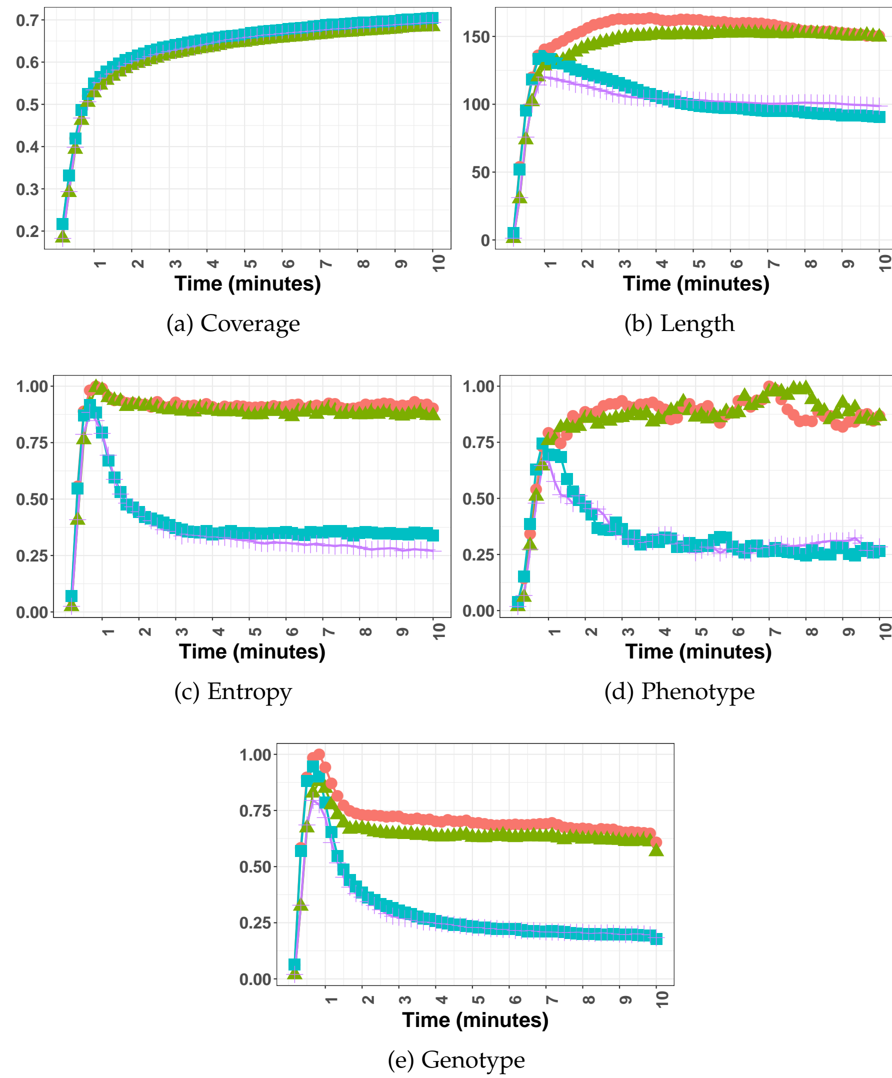
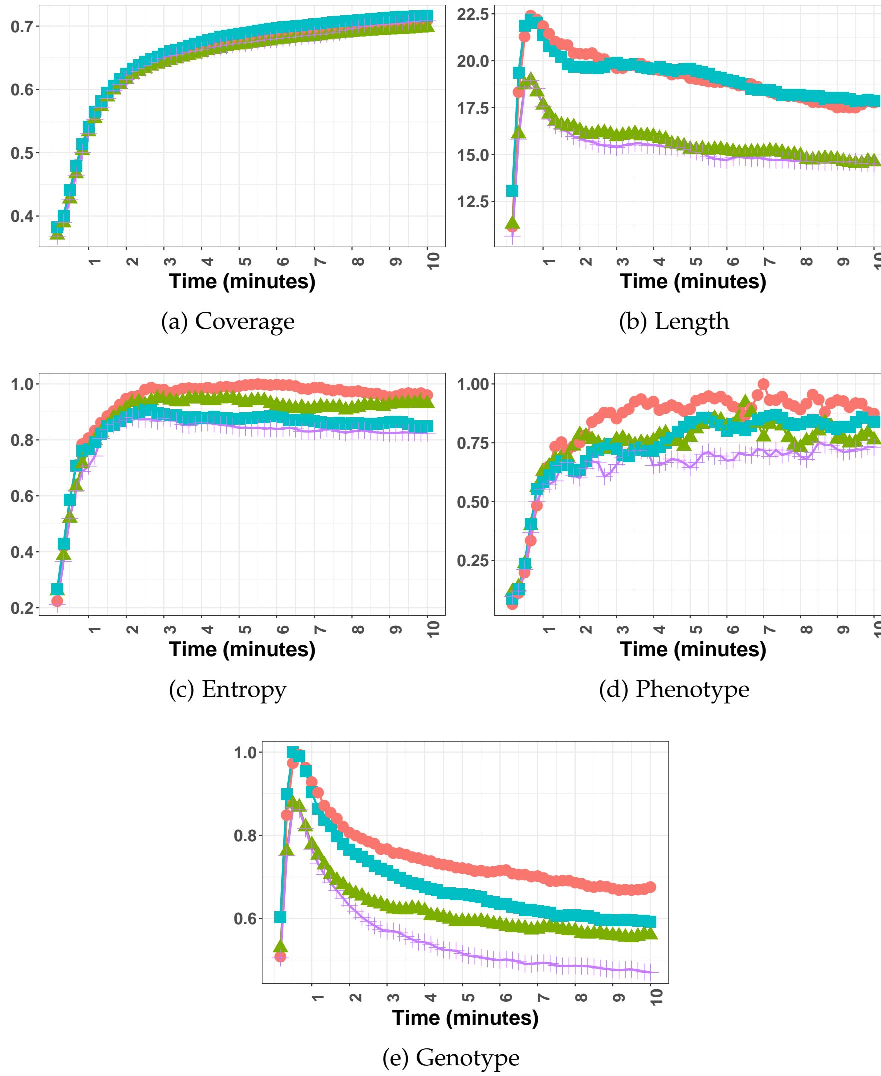


Figure 5.4: Coverage, length, and diversity measures over time with Monotonic GA using WSA and adaptive fitness-based sharing (AFS), and their reduction-based versions.



(a) Coverage (b) Length

(c) Entropy (d) Phenotype

(e) Genotype

AFS (default) **AFS (reduction)** **MOSA (default)** **MOSA (reduction)**

Figure 5.5: Coverage, length, and diversity measures over time with MOSA and adaptive fitness-based sharing (AFS), and their reduction-based versions.

RQ2*: The test case reduction approach leads to (i) no considerable effect on coverage, (ii) decrease in the length with MOSA but not with WSA, and (iii) a slight decrease in diversity except the genotype diversity with MOSA that shows a large decrease.*

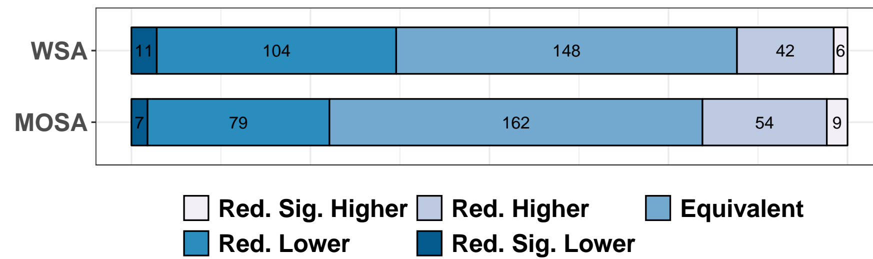


Figure 5.6: Comparison the performance of the Monotonic GA using WSA and MOSA based on the branch coverage ("Red. Sig. Higher" is the number of classes where reduction-based GA achieved significantly higher coverage than its default version, "Red. Higher" is the number of classes where reduction-based GA achieved higher coverage (but not significantly), "Equivalent" is the number of classes where both versions of the GA result in similar coverage, and both "Red. Sig. Lower" and "Red. Lower" represent the classes where the reduction-based GA results in lower coverage than the default.)

5.4.2.3 *What is the effect of test case reduction on the performance of WSA and MOSA?*

As the test case reduction is applied to the two algorithms, it is important to find out whether the reduction approach affects their performance, more specifically, by looking at the obtained branch coverage. Although we investigated the coverage in RQ2, we still need to perform an in-depth analysis of the achieved coverage on the class level. Figure 5.6 summarises the number of classes in which the reduction-based version of the two GAs achieve higher coverage than the default versions, and vice versa. We clearly see that there is a variety in the number of classes between the two algorithms where, for example, the number of classes for which the reduction-based version achieves higher coverage is not similar between the two algorithms. In general, there is a high number of classes for which the reduction-based GA results in a similar coverage to the default GA (i.e., Equivalent classes includes the classes that achieve full coverage). When comparing the two versions of the GA, we observe that the reduction-based GA achieves lower coverage with more classes (i.e., 115 classes with WSA and 86 classes with MOSA) than the default GA, but only a few classes for which the reduction-based GA result in significant decrease in the coverage (i.e., 11 classes with WSA and 7 classes with MOSA). In contrast, the number of classes for which the reduction-based GA achieves higher coverage is lower than the number of classes where the reduction results in lower coverage (i.e., 48 classes with WSA and 63 classes with MOSA where the significant increase in coverage is observed with only 6 classes in WSA and 9 classes in MOSA).

It should be noted that the difference in the achieved coverage between the two versions of the GA is mostly small with most of the classes. For example, the coverage in the case of `XmlElement` class

is 0.8 with the reduction-based Monotonic GA and 0.78 with the default version (i.e., $\alpha > 0.05$ and $\hat{A}_{12} = 0.52$). However, there are some classes where the difference in the achieved coverage is slightly large, and this is mostly the case with most of the classes that show significant increase/decrease in coverage. An example can be seen with the `AttributeModelComparator` class for which the coverage is 0.68 with the reduction-based MOSA and 0.54 with the default version (i.e., $\alpha < 0.05$ and $\hat{A}_{12} = 0.56$).

> ***RQ3***: *The test case reduction has no considerable impact on the branch coverage although there are few cases that show little influence on the coverage.*

5.5 DISCUSSION

The study of test case reduction with the two algorithms reveals that there is a difference in the effect of reducing test cases (i.e., removing redundant statements) when considering different individual representations. The results that are presented in the last three sections demonstrate that the effect of the reduction approach is more obvious with MOSA than with WSA. When analysing the impact of test case reduction on the landscape structure, we see that the reduction approach leads to a slight increase in the beneficial ruggedness and, to some extent, a decrease in the size of plateaus with MOSA more than with WSA; the increase in ruggedness is more obvious with MOSA as shown by the NV, IC, PIC, and DBI measures in Figure 5.3. A possible explanation is that an individual of a test case is more sensitive to the changes made by the reduction than an individual of a test suite (i.e., removing statements from test cases of an individual test suite does not show as large changes as when removing statements from an individual test case), which thus leads to possible changes in the fitness of an individual that affect the landscape structure.

This effect is also observed when analysing the impact of test reduction on the population diversity where applying the reduction approach with MOSA leads to a slight decrease in the diversity than with WSA, especially the genotype diversity that demonstrates a considerable drop in diversity (Figure 5.5e). Our conjecture of why the reduction approach decreases diversity with the test case representation is that removing statements, even if they are unnecessary, from an individual test case raises the possibility that individuals become more similar. In other words, the reduction approach possibly removes those statements that introduced further differences between the individuals, and this what makes the genotype diversity is highly affected.

When looking at the impact of test case reduction on the length, we also see that there is a remarkable decrease in the length of individual test cases more than the length of individual test suites. This confirms our finding that the test case representation shows more changes with

the reduction than the test suite representation. However, this difference between the two representations is not really observed when analysing the effect of reduction on the achieved branch coverage as the reduction has no considerable impact on the branch with both algorithms. This suggests that removing redundant statements is not guaranteed to enhance the performance of the algorithm in achieving higher coverage. One reason behind that is the low number of mutations that lead to successful reduced test cases. The reduced test cases are considered successful if applying the mutation on them results in test cases that improve the fitness, and thus increase the coverage.

5.6 SUMMARY

During the evolution, individuals tend to get bigger because of many statements that are added by the genetic operators. In our research, we observe that, during the random walk, the mutation operator inserts statements that might seem to be useless, and have no positive impact on the final fitness value. We also notice that diversity maintenance techniques lead to an increase in the length where many unnecessary statements are added to test cases, which cause such an increase in length. These extra statements are found to be just duplicates of other statements in a test case; both perform exactly (i.e., calling one method with the exact test input), and have no effect on the fitness.

As these statements seem to have a negative impact on the landscape structure and the individual length, we consider applying the test case reduction approach that works by removing these statements based on predefined rules. We explicitly examine the effect of this approach on the landscape properties (i.e., ruggedness and neutrality), the diversity level during the evolution, the individual length, and the achieved branch coverage. To achieve that, we compare the default techniques (i.e., without considering reduction) to their versions when applying the reduction. As a result, the reduction approach seems to (i) affect two properties of the landscape, especially with the test case representation, as there is a slight decrease in plateaus and increase in rugged areas (i.e., increase in the fitness changes), (ii) decrease the length with the test case representation and a slight decrease in diversity and more obviously the genotype diversity, and (iii) have no considerable impact on the achieved branch coverage by the GA.

6 CONCLUSION AND FUTURE WORK

This thesis focuses on the problem of the automated generation of object-oriented unit tests using search-based optimisation algorithms. Despite the success of these algorithms in generating unit tests that satisfy test goals (e.g., achieve high branch coverage), there are still cases where these algorithms fail to achieve the desired test goals. Therefore, this thesis aimed to investigate the reasons why search-based algorithms, more specifically GAs, cannot always generate potential unit tests that improve the branch coverage by investigating the following high-level research questions:

- Does the underlying structure of the search space affect the optimisation of unit tests? In particular, how do the features of the fitness landscape influence the generation of unit tests?
- How do the underlying properties of source code influence the fitness landscape features?
- What is the impact of population diversity on the generation of unit tests? and does improving diversity have a positive impact on the performance of GAs?
- How does the removal of unnecessary statements in unit test cases affect the fitness landscape features and the population diversity, and thus the performance of GAs?

6.1 SUMMARY OF CONTRIBUTIONS

This section summarises the answers to the previously mentioned questions, which present the contributions achieved in each chapter.

6.1.1 *Fitness Landscape Analysis*

In Chapter 3, we conducted an in-depth analysis of how unit test generation is influenced by the fitness landscape, and understand how the landscape properties relate to features of Java classes. We first investigated the features of the fitness landscape for the JUnit test generation problem using six fitness landscape measures applied to the series of fitness values obtained by the random walk. An empirical evaluation on 331 non-trivial Java classes reveals that fitness landscape is highly dominated by neutral areas, i.e., plateaus and the degree of neutrality increases with WSA approach (i.e., the test suite representation). When investigating the impact of landscape features on the

search performance, branches that have a large degree of neutrality in their landscape seem to be harder to cover, whereas branches that have a small degree of neutrality in their landscape seem to be easy to cover. Ruggedness is found to be beneficial to the search as it indicates the existence of gradients that make a branch easy to cover by GA, and possibly harder to cover by a random walk.

The analysis of how the underlying properties of source code influence the fitness landscape features indicates that the main causes for neutral fitness landscapes are (1) the accessibility of the methods that contain the branches where branches in private methods are difficult to cover, (2) the difficulty of satisfying preconditions on complex objects, and (3) the prevalence of boolean flags (i.e., boolean comparisons offering no guidance).

6.1.2 *Population Diversity Analysis*

Chapter 4 presented a study of the impact of population diversity on the generation of unit tests that mainly investigates whether the effectiveness of the GA is influenced by the diversity level during the evolution. We first measured the diversity in both algorithms (Monotonic GA and MOSA) to get an idea of whether both GAs are able to maintain a high level of diversity during the search. Measuring the diversity of generated unit tests based on entropy, genotypic, and phenotypic levels suggest that the default Monotonic GA is not as efficient as the default MOSA in maintaining higher diversity.

Then, we applied well-known diversity maintenance techniques to see whether they succeed at increasing the population diversity during evolution. As a result, applying these techniques on the two algorithms are found to be effective at promoting diversity throughout the evolution, which is more obvious with Monotonic GA as the increase in diversity caused by these techniques with MOSA is not as high as with Monotonic GA.

Investigating the effect of population diversity on the performance of GAs reveals that increasing diversity leads to (i) decrease the coverage although there are techniques that result in slightly similar coverage to the default GAs and (ii) a possible increase in the length. However, the negative effect of increasing diversity on coverage and length is reduced when considering the adaptive diversity approach, especially with the adaptive fitness-based sharing (AFS-fitness). As enhancing diversity results in adding more statements to an individual test, we investigated the type of statements that are added to see whether a specific statement type is favoured by the diversity techniques, and as a result, there is no dominant statement type that happens with enforced diversity.

6.1.3 *An Analysis of Test Case Reduction Approach*

In the last two chapters, we observed that test cases tend to have redundant statements (i.e., statements are just duplicates of other statements) that have no positive impact on the fitness of an individual. This is observed when applying the random walk where the mutation keeps adding unnecessary statements or even changing existing statements and also when applying a diversity maintenance technique where such redundant statements are added that lead to an increase in the length. Therefore, we asked a question of whether removing such statements affects both the landscape features (i.e., decrease the presence of plateaus) and population diversity level (i.e., reduce the negative effect on length) by applying the test case reduction approach with the two algorithms (WSA and MOSA).

To answer this question, we empirically evaluated the effect of this approach on the landscape features by applying the random walk with enabling the reduction approach and comparing it against the default random walk. As a result, the reduction approach seems to have an influence on the landscape features as there is a slight decrease in plateaus and increase in rugged areas, especially with the test case representation. We also evaluated the impact of the reduction approach on the population diversity by applying and comparing the diversity maintenance technique, more specifically the AFS-fitness, with the reduction approach to the default version of the diversity technique. The outcomes of this evaluation confirm that the reduction approach seems to decrease the length with the test case representation and slightly decrease the maintained diversity level that is more obvious the genotype diversity. However, evaluating the effect of the reduction approach on the performance of the two GAs indicates that the test reduction has no considerable impact on the achieved branch coverage by the GA.

6.2 FUTURE WORK

The research conducted in this thesis raises many questions that remain open and need to be explored in future research. In this section, we present these questions and suggest several ideas.

6.2.1 *Fitness Landscape Improvement*

In Chapter 3, we observed several causes for the detrimental neutral landscape such as the methods accessibility, the difficulty of satisfying the preconditions of methods, etc. Based on that, we recommend some potential ideas that possibly improve the fitness landscape. First, refining the fitness function that considers the inter-procedural distance information to overcome the issue of methods accessibility. Testability

transformations can also be used to remove the boolean flags that usually result in plateaus in the fitness landscape. As fitness is typically measured only directly on the class under test and not dependency classes, it is important to consider the code underlying the dependencies, such that there is guidance towards producing valid object configurations. Also, the search operators can be improved in order to increase chances of producing valid object configurations. Seeding also can be improved to ensure that valid object configurations are generated.

6.2.2 *Improved Versions of MOSA*

In this thesis, our investigation was mainly to investigate the search behaviour when considering Monotonic GA using WSA approach and MOSA. Recently, there have been further improvements introduced to MOSA that aim to improve its performance, and thus achieving better branch coverage. There are two extended versions of MOSA. First, DynaMOSA [127] extends MOSA with a dynamic selection of coverage targets based on their control dependency. Second, Parallel MOSA [19] that is a parallelised version of MOSA where a population is distributed among a number of semi-isolated sub-populations. Both versions were found to be effective in achieving higher branch coverage than default MOSA. Therefore, we recommend extending the experiments we conducted in this thesis to consider the two versions of MOSA that is by (i) investigating the impact of the landscape features on the test generation when considering each of the two versions, (ii) analysing the impact of population diversity and (iii) the test case reduction on the performance of each version.

6.2.3 *Coverage Criteria*

The investigation conducted in this thesis mainly targeted the branch coverage criterion. In literature, other coverage criteria are considered to guide the generation of unit tests [139] such as Weak Mutation, Output Coverage, Direct Branch Coverage, Exception Coverage, etc. These criteria are implemented as fitness functions to optimise the generation of test suites. Furthermore, these criteria can be combined, for example, by combining the branch coverage with the weak mutation that results in a combined fitness function. Therefore, we believe it is necessary to see how the fitness landscape changes for these fitness functions and understand the changing fitness landscape created by the combined fitness functions. Also, the effect of these fitness functions on the population diversity needs to be evaluated and see whether diversity, especially the phenotype and entropy diversity, changes when considering these functions.

6.2.4 *Fault Finding Effectiveness*

The focus of this thesis was on how the fitness landscape features, population diversity, and test case reduction affect the branch coverage. An important aspect of testing is to detect faults in the software under test, and ensuring that the generated tests can detect faults becomes a necessity, which can be achieved by considering the mutation analysis (Section 2.1.2.2). One possible idea is to extend our experiments to assess the potential impact on fault finding-capability through mutation analysis. That is to (i) investigate how the landscape features affect the achieved mutation score, (ii) see whether increasing diversity results in better mutation score, and (iii) understand the impact of test reduction on the mutation analysis.

6.2.5 *Alternative Diversity Techniques*

On search problems where the individual representation has a variable length, establishing distances between individuals for rewarding diversity most likely result in bloating problems. Therefore, it would be better to define novel and specialised diversity techniques that do not have such undesired side effects. For example, rewarding individuals that execute uncommon statement sequences/structures. Furthermore, applying adaptive fitness sharing is found to be beneficial to the search, but the question of when it does improve the search needs further investigation.

6.3 Overall Conclusion

Genetic algorithms (GAs) have been demonstrated to be effective in generating unit tests. Despite the success of GAs in generating tests that achieve high branch coverage, there are still cases where they fail to improve the coverage. Understanding why the GA search does not always find test inputs that cover branches is still an open research question that needs to be investigated. Therefore, this thesis aims to study the search behaviour when applying GAs to generate object-oriented unit tests by investigating the impact of (1) the underlying structure of the search space, (2) population diversity, and (3) controlling the bloat problem on the generation of unit tests. The findings of this investigation confirm that (1) the fitness landscape is mostly dominated by plateaus that are detrimental to the search, (2) promoting population diversity consistently leads to a negative impact on the coverage and length as it decreases coverage and increases length which is mitigated when diversity is promoted adaptively, and (3) removing redundant statements in test cases has a slightly positive impact on the landscape structure (i.e., reducing plateaus) and the population diversity (i.e., avoiding length increase) but has no impact

on the performance of GAs. These insights provide an explanation of why MOSA performs better than Monotonic GA using WSA approach as MOSA does not produce more plateaus in the landscape structure and it is more efficient in maintaining population diversity that does not increase length. This also supports the fact that evolving test cases is better than evolving test suites.

BIBLIOGRAPHY

[1] J-R Abrial, Matthew KO Lee, DS Neilson, PN Scharbach, and Ib Holm Sørensen. "The B-method." In: *International symposium of VDM Europe*. Springer. 1991, pp. 398–405.

[2] Salem F Adra and Peter J Fleming. "Diversity management in evolutionary many-objective optimization." In: *IEEE Trans. Evol. Comput.* 15.2 (2010), pp. 183–195.

[3] Nasser M Albunian. "Diversity in search-based unit test suite generation." In: *Proc. of SSBSE (2017)*. Springer. 2017, pp. 183–189.

[4] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. "Causes and Effects of Fitness Landscapes in Unit Test Generation." In: *GECCO 2020: Proceedings of the Genetic and Evolutionary Computation Conference*. ACM Digital Library. 2020.

[5] Nasser Albunian, Gordon Fraser, and Dirk Sudholt. "Measuring and Maintaining Population Diversity in Search-Based Unit Test Generation." In: *International Symposium on Search Based Software Engineering*. Springer. 2020, pp. 153–168.

[6] Aldeida Aleti, Irene Moser, and Lars Grunske. "Analysing the fitness landscape of search-based software testing problems." In: *Automated Software Engineering* 24.3 (2017), pp. 603–621.

[7] Domenico Amalfitano, Nicola Amatucci, Anna Rita Fasolino, and Porfirio Tramontana. "AGRippin: a novel search based testing technique for Android applications." In: *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. 2015, pp. 5–12.

[8] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[9] Saswat Anand, Alessandro Orso, and Mary Jean Harrold. "Type-dependence analysis and program transformation for symbolic execution." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2007, pp. 117–133.

[10] James H Andrews, Lionel C Briand, and Yvan Labiche. "Is mutation an appropriate tool for testing experiments?" In: *Proceedings of the 27th international conference on Software engineering*. 2005, pp. 402–411.

[11] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. "Using mutation analysis for assessing and comparing testing coverage criteria." In: *IEEE Transactions on Software Engineering* 32.8 (2006), pp. 608–624.

[12] Peter J Angeline and Kenneth E Kinnear. "Efficiently representing populations in genetic programming." In: (1996).

[13] Andrea Arcuri. "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage." In: *IEEE Transactions on Software Engineering* 38.3 (2011), pp. 497–519.

[14] Andrea Arcuri. "It really does matter how you normalize the branch distance in search-based software testing." In: *Software Testing, Verification and Reliability* 23.2 (2013), pp. 119–147.

[15] Andrea Arcuri and Lionel Briand. "Adaptive random testing: An illusion of effectiveness?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 265–275.

[16] Andrea Arcuri and Gordon Fraser. "On parameter tuning in search based software engineering." In: *International Symposium on Search Based Software Engineering*. Springer. 2011, pp. 33–47.

[17] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Formal analysis of the effectiveness and predictability of random testing." In: *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 219–230.

[18] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. "Random testing: Theoretical results and practical implications." In: *IEEE Transactions on Software Engineering* 38.2 (2011), pp. 258–277.

[19] Verena Bader, José Campos, and Gordon Fraser. "Parallel Many-Objective Search for Unit Tests." In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 37–47.

[20] André Baresel and Harmen Sthamer. "Evolutionary testing of flag conditions." In: *Genetic and Evolutionary Computation Conference*. Springer. 2003, pp. 2442–2454.

[21] André Baresel, Harmen Sthamer, and Michael Schmidt. "Fitness function design to improve evolutionary structural testing." In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. 2002, pp. 1329–1336.

[22] Luciano Baresi, Pier Luca Lanzi, and Matteo Miraz. "Testful: an evolutionary test approach for Java." In: *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE. 2010, pp. 185–194.

[23] Lionel Barnett. "Ruggedness and neutrality-the NKp family of fitness landscapes." In: *Artificial Life VI: Proceedings of the sixth international conference on Artificial life*. 1998, pp. 18–27.

[24] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. "The oracle problem in software testing: A survey." In: *IEEE transactions on software engineering* 41.5 (2014), pp. 507–525.

[25] Jeffrey K Bassett, Mark Coletti, and Kenneth A De Jong. "The relationship between evolvability and bloat." In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. 2009, pp. 1899–1900.

[26] Peter Boonstoppel, Cristian Cadar, and Dawson Engler. "RWset: Attacking path explosion in constraint-based test generation." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 351–366.

[27] Markus Brameier, Wolfgang Banzhaf, et al. *A comparison of genetic programming and neural networks in medical data analysis*. Secretary of the SFB 531, Univ., 1998.

[28] Paulo MS Bueno, Mario Jino, and W Eric Wong. "Diversity oriented test data generation using metaheuristic search techniques." In: *Information Sciences* 259 (2014), pp. 490–509.

[29] Edmund K Burke, Steven Gustafson, and Graham Kendall. "Diversity in genetic programming: An analysis of measures and correlation with fitness." In: *IEEE Transactions on Evolutionary Computation* 8.1 (2004), pp. 47–62.

[30] Edmund Burke, Steven Gustafson, and Graham Kendall. "A survey and analysis of diversity measures in genetic programming." In: *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2002, pp. 716–723.

[31] Jacob Burnim and Koushik Sen. "Heuristics for scalable dynamic test generation." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE. 2008, pp. 443–446.

[32] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[33] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Pasareanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. "Symbolic execution for software testing in practice: preliminary assessment." In: *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE. 2011, pp. 1066–1071.

[34] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An empirical evaluation of evolutionary algorithms for unit test suite generation." In: *Information and Software Technology* 104 (2018), pp. 207–235.

[35] Nachol Chaiyaratana, Theera Piroonratana, and Nuntapon Sangkawelert. "Effects of diversity control in single-objective and multi-objective genetic algorithms." In: *J. Heuristics* 13.1 (2007), pp. 1–34.

[36] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption." In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE. 2017, pp. 597–608.

[37] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. "Adaptive random testing: The art of test case diversity." In: *Journal of Systems and Software* 83.1 (2010), pp. 60–66.

[38] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. "ARTOO: adaptive random testing for object-oriented software." In: *Proceedings of the 30th international conference on Software engineering*. 2008, pp. 71–80.

[39] John Clarke, Mark Harman, R Hierons, B Jones, M Lumkin, K Rees, M Roper, and M Shepperd. "The application of metaheuristic search techniques to problems in software engineering." In: *SEMINAL (Software Engineering using Metaheuristic INnovative ALgorithms) technical report SEMINAL-TR-01-2000* (2000), p. 23.

[40] John Clarke, Jose Javier Dolado, Mark Harman, Rob Hierons, Bryan Jones, Mary Lumkin, Brian Mitchell, Spiros Mancoridis, Kearton Rees, Marc Roper, et al. "Reformulating software engineering as a search problem." In: *IEE Proceedings-software* 150.3 (2003), pp. 161–175.

[41] Lori A. Clarke. "A system to generate test data and symbolically execute programs." In: *IEEE Transactions on software engineering* 3 (1976), pp. 215–222.

[42] Edgar Covantes Osuna and Dirk Sudholt. "Empirical Analysis of Diversity-preserving Mechanisms on Example Landscapes for Multimodal Optimisation." In: *Proc. of PPSN 2018*. Springer, 2018, pp. 207–219.

[43] Edgar Covantes Osuna and Dirk Sudholt. "On the Runtime Analysis of the Clearing Diversity-Preserving Mechanism." In: *Evol. Comput.* 27 (3 2019), pp. 403–433.

[44] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. "Exploration and exploitation in evolutionary algorithms: A survey." In: *ACM computing surveys (CSUR)* 45.3 (2013), pp. 1–33.

[45] Christoph Csallner and Yannis Smaragdakis. "JCrasher: an automatic robustness tester for Java." In: *Software: Practice and Experience* 34.11 (2004), pp. 1025–1050.

[46] Edwin D De Jong, Richard A Watson, and Jordan B Pollack. "Reducing bloat and promoting diversity using multi-objective methods." In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2001, pp. 11–18.

[47] Leonardo De Moura and Nikolaj Bjørner. "Z3: An efficient SMT solver." In: *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 337–340.

[48] Kalyanmoy Deb and David E Goldberg. "An investigation of niche and species formation in genetic function optimization." In: *Proceedings of the third international conference on Genetic algorithms*. 1989, pp. 42–50.

[49] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II." In: *IEEE transactions on evolutionary computation* 6.2 (2002), pp. 182–197.

[50] Pedro A Diaz-Gomez and Dean F Hougen. "Empirical Study: Initial Population Diversity and Genetic Algorithm Performance." In: *Proc. of AIPR 2007*. 2007, pp. 334–341.

[51] Lydie Du Bousquet, Farid Ouabdesselam, J-L Richier, and Nicolas Zuanon. "Lutess: a specification-driven testing environment for synchronous software." In: *Proceedings of the 21st international conference on Software engineering*. 1999, pp. 267–276.

[52] Roger Duke, Gordon Rose, and Graeme Smith. "Object-Z: A specification language advocated for the description of standards." In: *Computer Standards & Interfaces* 17.5-6 (1995), pp. 511–533.

[53] Jon Edvardsson. "A survey on automatic test data generation." In: *Proceedings of the 2nd Conference on Computer Science and Engineering*. 1999, pp. 21–28.

[54] Anikó Ekárt and Sandor Z Németh. "Maintaining the diversity of genetic programs." In: *European Conference on Genetic Programming*. Springer. 2002, pp. 162–171.

[55] Mojtaba Behzad Fallahpour, Kamran Delfan Hemmati, and Ali Pourmohammad. "Optimization of a LNA using genetic algorithm." In: *Electrical and Electronic Engineering* 2.2 (2012), pp. 38–42.

[56] Robert Feldt, Richard Torkar, Tony Gorschek, and Wasif Afzal. "Searching for cognitively diverse tests: Towards universal test diversity metrics." In: *Software Testing Verification and Validation Workshop, 2008. ICSTW'08. IEEE International Conference on*. IEEE. 2008, pp. 178–186.

[57] Carlos M Fonseca and Peter J Fleming. "Multiobjective genetic algorithms." In: *IEE colloquium on genetic algorithms for control systems engineering*. Iet. 1993, pp. 6–1.

[58] Phyllis G Frankl and Oleg Iakounenko. "Further empirical studies of test effectiveness." In: *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*. 1998, pp. 153–162.

[59] Gordon Fraser and Andrea Arcuri. "Evosuite: automatic test suite generation for object-oriented software." In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 2011, pp. 416–419.

[60] Gordon Fraser and Andrea Arcuri. "Whole test suite generation." In: *IEEE Transactions on Software Engineering* 39.2 (2012), pp. 276–291.

[61] Gordon Fraser and Andrea Arcuri. "Handling test length bloat." In: *Software Testing, Verification and Reliability* 23.7 (2013), pp. 553–582.

[62] Gordon Fraser and Andrea Arcuri. "A large-scale evaluation of automated unit test generation using evosuite." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (2014), pp. 1–42.

[63] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. "Improving search-based test suite generation with dynamic symbolic execution." In: *2013 ieee 24th international symposium on software reliability engineering (issre)*. IEEE. 2013, pp. 360–369.

[64] Gregory Gay. "Challenges in using search-based test generation to identify real faults in mockito." In: *International Symposium on Search Based Software Engineering*. Springer. 2016, pp. 231–237.

[65] A. Ghosh and S. Tsutsui. *Advances in Evolutionary Computing: Theory and Applications*. Natural Computing Series. Springer Berlin Heidelberg, 2012. ISBN: 9783642189654.

[66] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. "Comparing non-adequate test suites using coverage criteria." In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. 2013, pp. 302–313.

[67] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. "Guidelines for coverage-based comparisons of non-adequate test suites." In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.4 (2015), pp. 1–33.

[68] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: directed automated random testing." In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[69] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. "Automated Whitebox Fuzz Testing." In: *NDSS*. Vol. 8. 2008, pp. 151–166.

[70] Rahul Gopinath, Carlos Jensen, and Alex Groce. "Code coverage for suite evaluation by developers." In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 72–82.

[71] John J Grefenstette et al. "Genetic algorithms for changing environments." In: *PPSN*. Vol. 2. 1992, pp. 137–144.

[72] Florian Gross, Gordon Fraser, and Andreas Zeller. "EXSYST: search-based GUI testing." In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 1423–1426.

[73] Brent Hailpern and Padmanabhan Santhanam. "Software debugging, testing, and verification." In: *IBM Systems Journal* 41.1 (2002), pp. 4–12.

[74] Margaret H Hamilton and William R Hackler. "Universal systems language: lessons learned from Apollo." In: *Computer* 41.12 (2008), pp. 34–43.

[75] Mark Harman and Bryan F Jones. "Search-based software engineering." In: *Information and software Technology* 43.14 (2001), pp. 833–839.

[76] Mark Harman and Phil McMinn. "A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation." In: *Proceedings of the 2007 international symposium on Software testing and analysis*. 2007, pp. 73–83.

[77] Mark Harman and Phil McMinn. "A theoretical and empirical study of search-based testing: Local, global, and hybrid search." In: *IEEE Transactions on Software Engineering* 36.2 (2009), pp. 226–247.

[78] Mark Harman, Lin Hu, Robert M Hierons, André Baresel, and Harmen Sthamer. "Improving Evolutionary Testing By Flag Removal." In: *GECCO*. 2002, pp. 1359–1366.

[79] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, André Baresel, and Marc Roper. "Testability transformation." In: *IEEE Transactions on Software Engineering* 30.1 (2004), pp. 3–16.

[80] Nguyen Thi Hien and Nguyen Xuan Hoai. "A brief overview of population diversity measures in genetic programming." In: *Proc. 3rd Asian-Pacific Workshop on Genetic Programming, Hanoi, Vietnam*. Citeseer. 2006, pp. 128–139.

[81] John H Holland. "Genetic algorithms and the optimal allocation of trials." In: *SIAM Journal on Computing* 2.2 (1973), pp. 88–105.

[82] John N Hooker. "Testing heuristics: We have it all wrong." In: *Journal of heuristics* 1.1 (1995), pp. 33–42.

[83] Jeffrey Horn and David E Goldberg. "Genetic algorithm difficulty and the modality of fitness landscapes." In: *Foundations of genetic algorithms*. Vol. 3. Elsevier, 1995, pp. 243–269.

[84] David Jackson. "Promoting phenotypic diversity in genetic programming." In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2010, pp. 472–481.

[85] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K Chang. "OCAT: object capture-based automated testing." In: *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 159–170.

[86] Yue Jia and Mark Harman. "An analysis and survey of the development of mutation testing." In: *IEEE transactions on software engineering* 37.5 (2011), pp. 649–678.

[87] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. "Are mutants a valid substitute for real faults in software testing?" In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 654–665.

[88] Fitsum M Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. "Orthogonal exploration of the search space in evolutionary test case generation." In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM. 2013, pp. 257–267.

[89] James C King. "Symbolic execution and program testing." In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[90] Kenneth E Kinnear. "Fitness landscapes and difficulty in genetic programming." In: *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*. IEEE. 1994, pp. 142–147.

[91] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. "Optimization by simulated annealing." In: *science* 220.4598 (1983), pp. 671–680.

[92] Joshua D Knowles, Richard A Watson, and David W Corne. "Reducing local optima in single-objective problems by multi-objectivization." In: *International conference on evolutionary multi-criterion optimization*. Springer. 2001, pp. 269–283.

[93] Abdullah Konak, David W Coit, and Alice E Smith. "Multi-objective optimization using genetic algorithms: A tutorial." In: *Reliability Engineering & System Safety* 91.9 (2006), pp. 992–1007.

[94] Bogdan Korel. "Automated software test data generation." In: *IEEE Transactions on software engineering* 16.8 (1990), pp. 870–879.

[95] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*. Vol. 1. MIT press, 1992.

[96] Mark Last, Menahem Friedman, and Abraham Kandel. "Using data mining for automated software testing." In: *International Journal of Software Engineering and Knowledge Engineering* 14.04 (2004), pp. 369–393.

[97] Marco Laumanns, Lothar Thiele, Kalyanmoy Deb, and Eckart Zitzler. "Combining convergence and diversity in evolutionary multiobjective optimization." In: *Evolutionary computation* 10.3 (2002), pp. 263–282.

[98] Andreas Leitner, Manuel Oriol, Andreas Zeller, Ilinca Ciupa, and Bertrand Meyer. "Efficient unit test case minimization." In: *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. 2007, pp. 417–420.

[99] Nancy G Leveson and Clark S Turner. "An investigation of the Therac-25 accidents." In: *Computer* 26.7 (1993), pp. 18–41.

[100] Mei-yi Li, Zi-xing Cai, and Guo-yun Sun. "An adaptive genetic algorithm with diversity-guided mutation and its global convergence property." In: *Journal of Central South University of Technology* 11.3 (2004), pp. 323–327.

[101] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. "Grt: Program-analysis-guided random testing (t)." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 212–223.

[102] Heikki Maaranen, Kaisa Miettinen, and Antti Penttinen. "On initial populations of a genetic algorithm for continuous optimization problems." In: *J. Global Optim.* 37.3 (2007), p. 405.

[103] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. "Evodroid: Segmented evolutionary testing of android apps." In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 599–609.

[104] Katherine M Malan and Andries P Engelbrecht. "A survey of techniques for characterising fitness landscapes and some possible ways forward." In: *Information Sciences* 241 (2013), pp. 148–163.

[105] LI Manolache and Derrick G. Kourie. "Software testing using model programs." In: *Software: Practice and Experience* 31.13 (2001), pp. 1211–1236.

[106] Ke Mao, Mark Harman, and Yue Jia. "Sapienz: Multi-objective automated testing for Android applications." In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 2016, pp. 94–105.

[107] Michael L Mauldin. "Maintaining Diversity in Genetic Search." In: *AAAI*. 1984, pp. 247–250.

[108] Brian Mc Ginley, John Maher, Colm O'Riordan, and Fearghal Morgan. "Maintaining healthy population diversity using adaptive crossover, mutation, and selection." In: *IEEE Transactions on Evolutionary Computation* 15.5 (2011), pp. 692–714.

[109] Phil McMinn. "Search-based software test data generation: a survey." In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156.

[110] Phil McMinn. "Search-based software testing: Past, present and future." In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 153–163.

[111] Nicholas Freitag McPhee and Nicholas J Hopper. "Analysis of genetic diversity through population history." In: *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*. Morgan Kaufmann Publishers Inc. 1999, pp. 1112–1120.

[112] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. "Automatic testing of object-oriented software." In: *International Conference on Current Trends in Theory and Practice of Computer Science*. Springer. 2007, pp. 114–129.

[113] Christoph C. Michael, Gary McGraw, and Michael A Schatz. "Generating software test data by evolution." In: *IEEE transactions on software engineering* 27.12 (2001), pp. 1085–1110.

[114] Frederic P Miller, Agnes F Vandome, and John McBrewster. *Manchester Small-Scale Experimental Machine: Von Neumann architecture, Computer, Victoria University of Manchester, Tom Kilburn, Williams tube, Manchester...(computing), Computer memory, Subtraction*. Alpha Press, 2009.

[115] James Miller, Marek Reformat, and Howard Zhang. "Automatic test data generation using genetic algorithm and program dependence graphs." In: *Information and Software Technology* 48.7 (2006), pp. 586–605.

[116] Webb Miller and David L. Spooner. "Automatic generation of floating-point test data." In: *IEEE Transactions on Software Engineering* 3 (1976), pp. 223–226.

[117] Irene Moser, Marius Gheorghita, and Aldeida Aleti. "Identifying features of fitness landscapes and relating them to problem difficulty." In: *Evolutionary computation* 25.3 (2017), pp. 407–437.

[118] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[119] U-M O'Reilly. "Using a distance metric on genetic programs to understand genetic operators." In: *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*. Vol. 5. IEEE. 1997, pp. 4092–4097.

[120] Carlos Oliveira, Aldeida Aleti, Lars Grunske, and Kate Smith-Miles. "Mapping the effectiveness of automated test suite generation techniques." In: *IEEE Transactions on Reliability* 67.3 (2018), pp. 771–785.

[121] Pietro S. Oliveto, Dirk Sudholt, and Christine Zarges. "On the Benefits and Risks of Using Fitness Sharing for Multimodal Optimisation." In: *Theor. Comput. Sci.* 773 (2019), pp. 53–70.

[122] Carlos Pacheco and Michael D Ernst. "Randoop: feedback-directed random testing for Java." In: *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 2007, pp. 815–816.

[123] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. "Feedback-directed random test generation." In: *29th International Conference on Software Engineering (ICSE'07)*. IEEE. 2007, pp. 75–84.

[124] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. "Automatic test case generation: what if test code quality matters?" In: *Proceedings of the Int. Symposium on Software Testing and Analysis*. ACM. 2016, pp. 130–141.

[125] Annibale Panichella, José Campos, and Gordon Fraser. "EvoSuite at the SBST 2020 Tool Competition." In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. 2020, pp. 549–552.

[126] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Reformulating branch coverage as a many-objective optimization problem." In: *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE. 2015, pp. 1–10.

[127] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets." In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 122–158.

[128] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "A large scale empirical comparison of state-of-the-art search-based test case generators." In: *Information and Software Technology* 104 (2018), pp. 236–256.

[129] Roy P Pargas, Mary Jean Harrold, and Robert R Peck. "Test-data generation using genetic algorithms." In: *Software testing, verification and reliability* 9.4 (1999), pp. 263–282.

[130] Karl Pearson. "The problem of the random walk." In: *Nature* 72.1867 (1905), pp. 342–342.

[131] Eric Pellerin, Luc Pigeon, and Sylvain Delisle. "Self-adaptive parameters in genetic algorithms." In: *Defense and Security*. International Society for Optics and Photonics. 2004, pp. 53–64.

[132] Alain Pétrowski. "A clearing procedure as a niching method for genetic algorithms." In: *Proc. of ICEC 1996*. IEEE. 1996, pp. 798–803.

[133] Erik Pitzer and Michael Affenzeller. "A comprehensive survey on fitness landscape analysis." In: *Recent advances in intelligent engineering systems*. Springer, 2012, pp. 161–191.

[134] Corina S Păsăreanu, Peter C Mehlitz, David H Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. "Combining unit-level symbolic execution and system-level concrete execution for testing NASA software." In: *Proceedings of the 2008 international symposium on Software testing and analysis*. 2008, pp. 15–26.

[135] Shahryar Rahnamayan, Hamid R Tizhoosh, and Magdy MA Salama. "Opposition-based differential evolution." In: *IEEE Transactions on Evolutionary computation* 12.1 (2008), pp. 64–79.

[136] Christian M Reidys and Peter F Stadler. "Neutrality in fitness landscapes." In: *Applied Mathematics and Computation* 117.2-3 (2001), pp. 321–350.

[137] Tea Robič and Bogdan Filipič. "Differential evolution for multi-objective optimization." In: *Proc. of EMO 2005*. Springer. 2005, pp. 520–533.

[138] José Miguel Rojas, Gordon Fraser, and Andrea Arcuri. "Seeding strategies in search-based unit test generation." In: *Software Testing, Verification and Reliability* 26.5 (2016), pp. 366–401.

[139] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. "Combining multiple coverage criteria in search-based unit test generation." In: *International Symposium on Search Based Software Engineering*. Springer. 2015, pp. 93–108.

[140] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. "A detailed investigation of the effectiveness of whole test suite generation." In: *Empirical Software Engineering* 22.2 (2017), pp. 852–893.

[141] Simon Ronald. "Duplicate genotypes in a genetic algorithm." In: *Proc. of CEC/WCCI 1998*. IEEE. 1998, pp. 793–798.

[142] Justinian P Rosca. "Entropy-driven adaptive representation." In: *Proceedings of the workshop on genetic programming: From theory to real-world applications*. Vol. 9. Citeseer. 1995, pp. 23–32.

[143] Christos Saltapidas and Ramin Maghsood. "Financial Risk The fall of Knight Capital Group." In: (2018).

[144] Bruno Sareni and Laurent Krahenbuhl. "Fitness sharing and niching methods revisited." In: *IEEE Trans. Evol. Comput.* 2.3 (1998), pp. 97–106.

[145] J David Schaffer. "Multiple objective optimization with vector evaluated genetic algorithms." In: *Proceedings of the first international conference on genetic algorithms and their applications, 1985*. Lawrence Erlbaum Associates. Inc., Publishers. 1985.

[146] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, and Siti Zaiton Mohd-Hashim. "A comparative study on automated software test oracle methods." In: *2009 Fourth International Conference on Software Engineering Advances*. IEEE. 2009, pp. 140–145.

[147] Ali Shahbazi. "Diversity-Based Automated Test Case Generation." PhD thesis. University of Alberta, 2015.

[148] Ali Shahbazi and James Miller. "Black-Box String Test Case Generation through a Multi-Objective Optimization." In: *IEEE Transactions on Software Engineering* 42.4 (2016), pp. 361–378.

[149] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)." In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 201–211.

[150] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. "Random or genetic algorithm search for object-oriented test suite generation?" In: *Proceedings of the 2015 annual conference on genetic and evolutionary computation*. 2015, pp. 1367–1374.

[151] Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser, Phil McMinn, Leonardo Mariani, and Andrea Arcuri. "Random or evolutionary search for object-oriented test suite generation?" In: *Software Testing, Verification and Reliability* 28.4 (2018), e1660.

[152] HISASHI Shimodaira. "A diversity-control-oriented genetic algorithm (DCGA): development and initial experimental results." In: *Trans. Inf. Process. Soc. Jpn.(Japan)* 40.6 (1999), pp. 2708–2716.

[153] Hisashi Shimodaira. "A diversity-control-oriented genetic algorithm (DCGA): Performance improvement by the reinitialization of the population." In: *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*. Morgan Kaufmann Publishers Inc. 2001, pp. 576–583.

[154] Ofer M. Shir. "Niching in Evolutionary Algorithms." In: *Handbook of Natural Computing*. Springer, 2012, pp. 1035–1070.

[155] Giovanni Squillero and Alberto Tonda. "Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization." In: *Inf. Sci.* 329 (2016), pp. 782–799.

[156] Mandavilli Srinivas and Lalit M Patnaik. "Adaptive probabilities of crossover and mutation in genetic algorithms." In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.4 (1994), pp. 656–667.

[157] Nidamarthi Srinivas and Kalyanmoy Deb. "Muiltiobjective optimization using nondominated sorting in genetic algorithms." In: *Evolutionary computation* 2.3 (1994), pp. 221–248.

[158] Matt Staats and Corina Păsăreanu. "Parallel symbolic execution for structural test generation." In: *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 183–194.

[159] Dirk Sudholt. "The benefits of population diversity in evolutionary algorithms: a survey of rigorous runtime analyses." In: *Theory of Evolutionary Computation*. Springer, 2020, pp. 359–404.

[160] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. "MSeqGen: Object-Oriented Unit-Test Generation via Mining Source Code." In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 2009, pp. 193–202.

[161] Jeff Tian. *Software quality engineering: testing, quality assurance, and quantifiable improvement*. John Wiley & Sons, 2005.

[162] Nikolai Tillmann and Wolfram Schulte. "Parameterized unit tests." In: *ACM SIGSOFT Software Engineering Notes* 30.5 (2005), pp. 253–262.

[163] Andrea Toffolo and Ernesto Benini. "Genetic diversity as an objective in multi-objective evolutionary algorithms." In: *Evolutionary computation* 11.2 (2003), pp. 151–167.

[164] Paolo Tonella. "Evolutionary testing of classes." In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pp. 119–128.

[165] Vedat Toğan and Ayşe T Daloğlu. "An improved genetic algorithm with initial population strategy and self-adaptive member grouping." In: *Comput. Struct.* 86.11-12 (2008), pp. 1204–1218.

[166] Shigeyoshi Tsutsui, Yoshiji Fujimoto, and Ashish Ghosh. "Forking genetic algorithms: GAs with search space division schemes." In: *Evolutionary computation* 5.1 (1997), pp. 61–80.

[167] John W Tukey. "The teaching of concrete mathematics." In: *The American Mathematical Monthly* 65.1 (1958), pp. 1–9.

[168] Rasmus K Ursem. "Diversity-guided evolutionary algorithms." In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2002, pp. 462–471.

[169] András Vargha and Harold D Delaney. "A critique and improvement of the CL common language effect size statistics of McGraw and Wong." In: *Journal of Educational and Behavioral Statistics* 25.2 (2000), pp. 101–132.

[170] JA Vasconcelos, Jaime Arturo Ramirez, RHC Takahashi, and RR Saldanha. "Improvements in genetic algorithms." In: *IEEE Transactions on magnetics* 37.5 (2001), pp. 3414–3417.

[171] Vesselin K Vassilev, Terence C Fogarty, and Julian F Miller. "Information characteristics and the structure of landscapes." In: *Evolutionary computation* 8.1 (2000), pp. 31–60.

[172] Thomas Vogel, Chinh Tran, and Lars Grunske. "Does Diversity Improve the Test Suite Generation for Mobile Applications?" In: *International Symposium on Search Based Software Engineering*. Springer. 2019, pp. 58–74.

[173] Thomas Vogel, Chinh Tran, and Lars Grunske. "A comprehensive empirical evaluation of generating test suites for mobile applications with diversity." In: *Information and Software Technology* 24.4 (2020), pp. 106–436.

[174] Joachim Wegener, André Baresel, and Harmen Sthamer. "Evolutionary test environment for automatic structural testing." In: *Information and software technology* 43.14 (2001), pp. 841–854.

[175] Edward Weinberger. "Correlated and uncorrelated fitness landscapes and how to tell the difference." In: *Biological cybernetics* 63.5 (1990), pp. 325–336.

[176] Darrell Whitley and Timothy Starkweather. "Genitor II: A distributed genetic algorithm." In: *Journal of Experimental & Theoretical Artificial Intelligence* 2.3 (1990), pp. 189–214.

[177] Abrham Workineh and Abdollah Homaifar. "Fitness proportionate niching: Maintaining diversity in a rugged fitness landscape." In: *Proceedings of the International Conference on Genetic and Evolutionary Methods (GEM)*. The Steering Committee of The World Congress in Computer Science, Computer ... 2012, p. 1.

[178] Sewall Wright. "The roles of mutation, inbreeding, crossbreeding, and selection in evolution." In: *Proceedings of the Sixth Annual Congress of Genetics*. Vol. 1. 1932, pp. 356–366.

[179] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan De Halleux. "Precise identification of problems for structural test generation." In: *Proceedings of the 33rd International Conference on Software Engineering*. 2011, pp. 611–620.

[180] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. "Symstra: A framework for generating object-oriented unit tests using symbolic execution." In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2005, pp. 365–381.

[181] Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. "Fitness-guided path exploration in dynamic symbolic execution." In: *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*. IEEE. 2009, pp. 359–368.

[182] Lingming Zhang, Darko Marinov, Lu Zhang, and Sarfraz Khurshid. "An empirical study of junit test-suite reduction." In: *2011 IEEE 22nd International Symposium on Software Reliability Engineering*. IEEE. 2011, pp. 170–179.

[183] Hong Zhu, Patrick AV Hall, and John HR May. "Software unit test coverage and adequacy." In: *Acm computing surveys (csur)* 29.4 (1997), pp. 366–427.

[184] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. "Comparison of multiobjective evolutionary algorithms: Empirical results." In: *Evolutionary computation* 8.2 (2000), pp. 173–195.

[185] Eckart Zitzler and Simon Künzli. "Indicator-based selection in multiobjective search." In: *International conference on parallel problem solving from nature*. Springer. 2004, pp. 832–842.