Selection of Software Tests and Mutants with Contextual Bandits

Patryk Kucmann

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
(Software Engineering)
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Autumn 2019
Selection of Software Tests and Mutants with Contextual Bandits

Patryk Kucmann
© 2019 Patryk Kucmann

Selection of Software Tests and Mutants with Contextual Bandits

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo
Abstract

This thesis addresses the problem of finding a robust test suite for software testing by the use of mutation testing and machine learning. The goal is to find out a test suite that is immune to most of the mutants that can occur. The focus is set on finding those tests which are more successful in detecting mutations in the software, simultaneously learn which mutants are least likely to be detected and thereby of the highest strength. Learning to select the most reliable tests allows saving time during future test iterations, compared to retest-all approach. The project is constructed in a two-player game setting, having the attacker, which selects mutants of a program, playing against the defender, which selects test cases to find the mutants; their objective is to win by the kill factor. Each player is implemented as a contextual bandit to learn the patterns for successful selection. In our experiments, we evaluate different parameters, including algorithms to balance exploration and exploitation. We find reliable parameters on a set of four Java programs; however, algorithms and size matter, and therefore, scaling them for each program is vital. The results show the capability to learn test and mutant selection, although the effect is parameter-dependent and not equally strong on all program sizes in our experiments. Our findings identify the ability to learn test selection from a game-play setting. We discuss the results and position them in the context of future work on learning to select software test cases efficiently.
Acknowledgments

First of all, I would like to thank my principal supervisor Helge Spieker for his time, engagement, patience, and general support that contributed to my growth in this field of study and producing a piece of scientific work.

Many thanks to my intern UiO supervisor Dag Langmyhr and co-supervisor Arnaud Gotlieb for all the assistance they gave me.

I would also like to thank the Simula Research Laboratory as well as the University of Oslo for giving me this opportunity.

On a personal note, I would like to thank my parents Irena and Zygmunt, and my brother Arek for their continued support and trust.
Contents

1 Introduction 1

2 Background 5
  2.1 Software Testing ........................................... 5
  2.2 Mutation Testing ........................................... 7
    2.2.1 Competent Programmer Hypothesis .................. 8
    2.2.2 Mutation Testing Process .............................. 8
    2.2.3 Challenges of Mutation Testing ....................... 12
  2.3 Machine Learning ........................................... 14
  2.4 Related Work .............................................. 18

3 Methodology 21
  3.1 The Concept of the Game .................................. 21
    3.1.1 The Idea .............................................. 21
    3.1.2 Structure .............................................. 22
    3.1.3 The Objective .......................................... 24
  3.2 Techniques .................................................. 25
    3.2.1 Contextual Bandits .................................... 25
    3.2.2 Feature Engineering ................................... 27
    3.2.3 Generation Tools ...................................... 28
    3.2.4 Execution Tools ....................................... 30
    3.2.5 Additional Tools ...................................... 31
    3.2.6 Bandit Algorithms ..................................... 31
  3.3 Implementation of the Game ............................... 32
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Major’s Mutation Operators</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Major’s Mutation Operators</td>
<td>11</td>
</tr>
<tr>
<td>4.1</td>
<td>Information for Experimental Programs</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Parameters for Different Configurations</td>
<td>45</td>
</tr>
<tr>
<td>4.3</td>
<td>Results for Program Font Info</td>
<td>47</td>
</tr>
<tr>
<td>4.4</td>
<td>Results for Program Inflection</td>
<td>57</td>
</tr>
<tr>
<td>4.5</td>
<td>Results for Program Range</td>
<td>65</td>
</tr>
<tr>
<td>4.6</td>
<td>Results for Program Hierarchy Property Parser Result</td>
<td>67</td>
</tr>
<tr>
<td>4.7</td>
<td>Final Parameters for Different Configurations</td>
<td>70</td>
</tr>
<tr>
<td>4.8</td>
<td>Final Results for Program Font Info</td>
<td>71</td>
</tr>
<tr>
<td>4.9</td>
<td>Final Results for Program Inflection</td>
<td>74</td>
</tr>
<tr>
<td>4.10</td>
<td>Final Results for Program Range</td>
<td>76</td>
</tr>
<tr>
<td>4.11</td>
<td>Final Results for Program Hierarchy Property Parser</td>
<td>78</td>
</tr>
<tr>
<td>4.12</td>
<td>Loaded and Fresh Bandits Parameters for Different Configurations</td>
<td>79</td>
</tr>
<tr>
<td>4.13</td>
<td>Loaded and Fresh Bandits Results for Program Font Info</td>
<td>80</td>
</tr>
<tr>
<td>4.14</td>
<td>Loaded and Fresh Bandits Results for Program Range</td>
<td>80</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Example of mutation operators ........................................ 12
3.1 Mutants Defender Class Diagram .................................... 23
3.2 Mutants Defender Sequence Diagram ................................. 33
3.3 Mutants Defender Sequence Diagram Round Play ................ 34
3.4 Mutants Defender Sequence Diagram Testing ...................... 34
4.1 Mutants Exploration Rate .............................................. 49
4.2 Tests Exploration Rate .................................................. 49
4.3 c1 Kill Ratio ............................................................... 50
4.4 c2 Kill Ratio ............................................................... 50
4.5 c3 Kill Ratio ............................................................... 50
4.6 c4 Kill Ratio ............................................................... 50
4.7 c5 Kill Ratio ............................................................... 51
4.8 c6 Kill Ratio ............................................................... 51
4.9 c7 Kill Ratio ............................................................... 51
4.10 c1 Mutants Pick Ratio ................................................... 52
4.11 c1 Mutants Score ......................................................... 52
4.12 c2 Mutants Pick Ratio ................................................... 52
4.13 c2 Mutants Score ......................................................... 52
4.14 c3 Mutants Pick Ratio ................................................... 52
4.15 c3 Mutants Score ......................................................... 52
4.16 c4 Mutants Pick Ratio ................................................... 53
4.17 c4 Mutants Score ......................................................... 53
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.49</td>
<td>c2 Mutants Pick Ratio</td>
<td>60</td>
</tr>
<tr>
<td>4.50</td>
<td>c2 Mutants Score</td>
<td>60</td>
</tr>
<tr>
<td>4.51</td>
<td>c3 Mutants Pick Ratio</td>
<td>61</td>
</tr>
<tr>
<td>4.52</td>
<td>c3 Mutants Score</td>
<td>61</td>
</tr>
<tr>
<td>4.53</td>
<td>c4 Mutants Pick Ratio</td>
<td>61</td>
</tr>
<tr>
<td>4.54</td>
<td>c4 Mutants Score</td>
<td>61</td>
</tr>
<tr>
<td>4.55</td>
<td>c5 Mutants Pick Ratio</td>
<td>61</td>
</tr>
<tr>
<td>4.56</td>
<td>c5 Mutants Score</td>
<td>61</td>
</tr>
<tr>
<td>4.57</td>
<td>c6 Mutants Pick Ratio</td>
<td>62</td>
</tr>
<tr>
<td>4.58</td>
<td>c6 Mutants Score</td>
<td>62</td>
</tr>
<tr>
<td>4.59</td>
<td>c7 Mutants Pick Ratio</td>
<td>62</td>
</tr>
<tr>
<td>4.60</td>
<td>c7 Mutants Score</td>
<td>62</td>
</tr>
<tr>
<td>4.61</td>
<td>c1 Tests Pick Ratio</td>
<td>62</td>
</tr>
<tr>
<td>4.62</td>
<td>c1 Tests Score</td>
<td>62</td>
</tr>
<tr>
<td>4.63</td>
<td>c2 Tests Pick Ratio</td>
<td>63</td>
</tr>
<tr>
<td>4.64</td>
<td>c2 Tests Score</td>
<td>63</td>
</tr>
<tr>
<td>4.65</td>
<td>c3 Tests Pick Ratio</td>
<td>63</td>
</tr>
<tr>
<td>4.66</td>
<td>c3 Tests Score</td>
<td>63</td>
</tr>
<tr>
<td>4.67</td>
<td>c4 Tests Pick Ratio</td>
<td>63</td>
</tr>
<tr>
<td>4.68</td>
<td>c4 Tests Score</td>
<td>63</td>
</tr>
<tr>
<td>4.69</td>
<td>c5 Tests Pick Ratio</td>
<td>64</td>
</tr>
<tr>
<td>4.70</td>
<td>c5 Tests Score</td>
<td>64</td>
</tr>
<tr>
<td>4.71</td>
<td>c6 Tests Pick Ratio</td>
<td>64</td>
</tr>
<tr>
<td>4.72</td>
<td>c6 Tests Score</td>
<td>64</td>
</tr>
<tr>
<td>4.73</td>
<td>c7 Tests Pick Ratio</td>
<td>64</td>
</tr>
<tr>
<td>4.74</td>
<td>c7 Tests Score</td>
<td>64</td>
</tr>
<tr>
<td>4.75</td>
<td>Mutants Exploration Rate</td>
<td>66</td>
</tr>
<tr>
<td>4.76</td>
<td>Tests Exploration Rate</td>
<td>66</td>
</tr>
<tr>
<td>4.77</td>
<td>Mutants Exploration Rate</td>
<td>67</td>
</tr>
<tr>
<td>4.78</td>
<td>Tests Exploration Rate</td>
<td>67</td>
</tr>
<tr>
<td>4.79</td>
<td>Mutants Exploration Rate</td>
<td>71</td>
</tr>
<tr>
<td>Table</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------</td>
<td>------</td>
</tr>
<tr>
<td>B.57</td>
<td>c1 Tests Pick Ratio</td>
<td>109</td>
</tr>
<tr>
<td>B.58</td>
<td>c1 Tests Score</td>
<td>109</td>
</tr>
<tr>
<td>B.59</td>
<td>c2 Tests Pick Ratio</td>
<td>109</td>
</tr>
<tr>
<td>B.60</td>
<td>c2 Tests Score</td>
<td>109</td>
</tr>
<tr>
<td>B.61</td>
<td>c3 Tests Pick Ratio</td>
<td>109</td>
</tr>
<tr>
<td>B.62</td>
<td>c3 Tests Score</td>
<td>109</td>
</tr>
<tr>
<td>B.63</td>
<td>c4 Tests Pick Ratio</td>
<td>109</td>
</tr>
<tr>
<td>B.64</td>
<td>c4 Tests Score</td>
<td>109</td>
</tr>
<tr>
<td>B.65</td>
<td>c5 Tests Pick Ratio</td>
<td>110</td>
</tr>
<tr>
<td>B.66</td>
<td>c5 Tests Score</td>
<td>110</td>
</tr>
<tr>
<td>B.67</td>
<td>c6 Tests Pick Ratio</td>
<td>110</td>
</tr>
<tr>
<td>B.68</td>
<td>c6 Tests Score</td>
<td>110</td>
</tr>
<tr>
<td>B.69</td>
<td>c7 Tests Pick Ratio</td>
<td>111</td>
</tr>
<tr>
<td>B.70</td>
<td>c7 Tests Score</td>
<td>111</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The goal of this thesis is to explore how machine learning can be applied to improve the efficiency of the mutation testing process. Software testing is the process of finding scenarios where the applications behave differently than they should. Mutation testing is testing the test suite’s quality by artificially introducing small, but realistic bugs (see Sec. 2.2.1). It is, however, a part of ongoing academic discussion, how realistic they are [14]. Software testing is especially vital if the product to be created is complex and rapidly changing. Without software testing, we end up with a high, if not an absolute risk for undesirable software usage. This is particularly valid in many cases where unexpected scenarios occur, that were not considered during development. Correctly conducted testing of a product will reduce the risk of decreasing critical situations, or even destroying a company’s or service provider’s reputation.

Researchers have conducted extensive studies on software failures and the importance of software testing presented in this report [34]. The result of this report is a reminder of how valuable and ubiquitous software is for us. They base it on reports of software failures and their consequences. Over 600 recorded software failures have been identified impacting 3.7 billion people worldwide, 1.7 trillion US dollars in assets, and 314 companies. This report does not cover all the data from all the world’s sources,
which should alert us even more. These numbers show how much there is to save, but not only money. Software continuously controls more and more devices. Even lives are at stake in case of a failure in a plane or other machines.

Mutation testing is a strategy to imitate real faults by introducing artificial ones with mutations. The goal is to prepare a test suite that can detect them and thus, mutation testing is helping with measuring the strength of a test suite.

The thesis addresses this problem in a two-player game setting, where the problem is trying to solve itself with two parties playing against it each other autonomously. A mutation testing game \cite{30} inspired this setting idea, where two people can play against each other as a defender and an attacker. One side is the defender, which has a role in beating the attacker by using tests. The attacker has the role in introducing faults into the software, which are unnoticed by the defender. He is an adversary, and the defender tries to detect him. His action is to attack the code and mutate it, causing an unwanted behaviour, whereas the defender is trying to cure the code by healing the code to its original state. The defender has access to test suite, and attacker to mutant set; both of them want to find smart ways to select from them and thus get the highest probability to win in a battle of tests and mutants. Playing it as humans, however, would require too much of resources; therefore, machine learning is a suitable replacement in favour of both resources and efficacy.

The focus lies on mutation testing area for the most part; treating it as an application and treat machine learning as a tool for improving it. Machine learning is an essential component, as it can elevate mutation testing on a new level likely exceeding what humans can do in an automated manner. The motivation to use ML could also lie in finding patterns that allow us to understand which mutants are more effective.

Looking from the future perspective, scenarios, where the work relevance is promising, can emerge. First and foremost insight to a better
understanding of mutation testing. In this project’s setting, we can analyze its behaviour, with the possibility of modifying some parameters for gathering more detailed data. Machine learning is supposed to speed up the process by a more effective selection in a short amount of time. This speedup is helpful if we are time-constrained and need quick solutions. Unsurprisingly, time is one of the biggest obstacles in the software developing community, so additional time savings are appreciated. Mutation testing itself is a promising candidate to be widely used in the future, but in combination with machine learning, we are to get an upgraded version which could motivate industrial applications in the long-term. If successful, can be used, e.g. in continuous integration with limited time budgets, but also early testing, it has the potential to become fruitful.

We conduct experiments that are mainly meant to help to discover the right parameters, exploration aspect, and selective distribution for how good or bad are the mutants and tests. Another essential part is using the features and algorithms; however, we had to minimize the experimental aspect of those, considering the thesis’ size constraint.

This thesis creates a foundation yielding a decent amount of information about both positive and negative facets of this work. We produce an excellent exploration and make our bandits specify the best couple of tests. Most of the distributions are hard to follow without introducing any specialized tools for that purpose. The selective match to the actual values for the agents is too low for reliable results in that matter and therefore lacks further research.

The thesis is structured as follows: In the next Chapter 2, background information is presented to fully understand the project, where mutation testing and machine learning are in focus. The point of background information is to put a light on the things we are dealing with, where fundamentals and its important facets are made manifest. To broaden the perspective, in Section 2.4, we present related work in this area of research. We give an in-depth presentation and analysis of this project in Chapters
Chapters 5 and 6 concludes the thesis with a discussion of directions for future research and a reflection on the results and findings of this research project.
Chapter 2

Background

This section introduces all the crucial facets of this project needed for a full understanding and easy following of material presented in the central part of this thesis. More specifically, next subsections will explain the fundamentals of testing, mutation testing, as well as machine learning concepts relevant for this thesis.

2.1 Software Testing

Software testing aims to ensure that a program executes as intended. Therefore it is crucial to look for unintended behaviour of a program under software testing. Dijkstra described the nature of software testing as: "Testing shows the presence, not the absence of bugs." [10]. There are many different approaches and techniques for how to test software, and they work differently on a different piece of code. If done correctly, a program can be made very resistant to undesired actions, and thus be reliable enough to bring profit and trust to potential stakeholders, while present buyers keep firm reliance on the products. Without software testing, the risk of losing trust is growing drastically.

The technical facet of software testing is abundant in terms like test suite, test case and others. A test suite is a collection of the test at one’s dis-
posal when testing something in software. We can have many test suites, but usually, we want to check the strength of one. Tests producing sound effects can be picked up from various test suites, and in the end, constitute of one robust tests suite with only well-selected tests for a given branch of software.

Definition 2.1.1 (Test Suite). A test suite $T$ consists of $N$ test cases $\{t_1, t_2, ..., t_N\}$

Having a set of conditions or variables to perform a test, and using them in order to determine whether a program or system satisfies given requirements, we are doing a so-called test case. Test cases can also help detect complications in the requirements when being under the process of developing them. One example of a test case is to validate login credentials for personnel. It is crucial in testing to have test cases, as they allow us to find weak and robust facets of a product in development.

Another important concept in software testing is testing verdict. A test verdict is showing the conformance to requirements of a test case; it can be a pass, fail or inconclusive. The ETSI institute defines different types of test verdicts [11]:

Definition 2.1.2 (Test Verdict). A statement of "pass", "fail" or "inconclusive", specified in an abstract interoperability test case, concerning either the end-to-end interoperability of two or more System Under Test (SUT) concerning that test case when being executed.

Definition 2.1.3 (Fail Verdict). A test verdict given when the observed test result either demonstrates non-interoperability of equipment under test with respect to the end-to-end functionality on which the test case is focused, or demonstrates non-conformance with respect to at least one of the conformance requirement(s) on which the test purpose(s) associated with the test case is (are) focused, or contains at least one invalid test event, with respect to the relevant specifications.

Definition 2.1.4 (Pass Verdict). A test verdict given when the observed test result gives evidence of all equipment under test interoperating for
the end-to-end functionality on which the test case to focus on, or conformance to the conformance requirement(s) on which the test purpose(s) of the test case is (are) focused, and when all the test events are valid with respect to the relevant specifications.

Having a system under test means to create or find desired test cases, which we can use to find conformance to our requirements and specifications based on our test suite. Different test suites can be used to test different parts of software, but the critical aspect is to eventually end up with an excellent report on how well is the potential product ready for a client or customer.

2.2 Mutation Testing

Mutation testing is not another way to test software, but instead, it evaluates the strength of a test suite (a set of tests). In mutation testing, a mutant is a minor change in a code piece. Usually, only one change is made per mutant; this is part of the assumption that in general developers are competent, but make minor syntactic errors, a so-called Competent Programmer Hypothesis [2]. This hypothesis and another one is a base for mutation testing. The latter is called the coupling effect. Coupling effect emerges faults that are cascaded or coupled from other simple faults. This effect can have drastic consequences; because it can cause unforeseen changes which may harm a sensitive and vital area. Both hypotheses put a light on a real danger; starting with a single bug, we can end up with a tree of errors. Thus picking such faults up in early phases is a wise decision. Higher-order mutants are mutants with more than one mutation, and they further support the coupling effect by revealing subtle and vital faults.
2.2.1  Competent Programmer Hypothesis

Competent programmer hypothesis states that small syntactic errors is the cause of most software faults introduced by experienced programmers. According to this hypothesis, programmers write programs to the near perfection, and the faults appearing can be fixed by a few keystrokes. Mutants simulate these faults and thus if we catch them, we also have high probability of catching the real errors.

Mutation testing gives us a way to assess the quality of a test suite. We can either commence to improve the quality of a test suite or discard and move to another test suite. Knowing the basic concept of mutation testing, we can move on to its process.

2.2.2  Mutation Testing Process

What is the process back mutation testing, and how can it be understood? The word mutation is taken as a metaphor from biology; a perfect example is when an organism is a subject for radiation at a dangerous level, mutating the DNA; as a result, we get a new mutated version of that organism. That organism remains the same individual, but different at the same time. Usually seen as an abomination, something the environment do not want to keep; just like a mutant in a program, it is not welcome as it causes unhappiness. Even if this example can be arguable, we only care about the point that a mutant equals something unusual from the standard or plan. Keeping that point in mind, we can now dive into the details.

In a scenario where we have a program and a test suite (a set of tests) for it, we use a tool for generating mutants for this program, for example, Major \[19\] or PIT\[1\]. It depends on which program is a wise choice regarding mutating in a given case, but we are interested in the idea right now. When the tool creates mutants based on its mutation operators, many mutated copies arise. Having mutants to our disposal, what we want to do

\[1\] PIT: http://pitest.org/
is to kill those mutants. Killing a mutant means that a test has produced a distinct result from that on the original code. These tests who catches those mutants increase the score for their test suite. There are two ways of killing, strongly and weakly, e.g. the last mentioned was a strong kill. Weakly is when the state of execution is different immediately after modified location, not waiting until finishing producing an output [2, Ch. 5]. If a test does not kill a mutant, a few possibilities can occur. We can end up with an equivalent mutant, which is just another way of writing the same intended task; not a defect, but remains an open challenge in mutation testing. The second option could be a not optimal test suite; therefore, mutation testing evaluates the strength of a test suite for the viable faults, which mutation testing currently evaluates. In other words, it checks if a test can catch possible bugs that could appear. The book [2] offer some more detailed definitions; here are the essential definitions for this study:

**Definition 2.2.1** (Mutation Operator). A rule that specifies syntactic variations of strings generated from a grammar.

**Definition 2.2.2** (Mutant). The result of one application of a mutation operator.

**Definition 2.2.3** (Strongly Killing Mutants). Given a mutant \( m \in M \) where \( M \) (a set of all possible mutants) for a program \( P \) and a test \( t \), \( t \) is said to strongly kill \( m \) if and only if the output of \( t \) on \( P \) is different from the output of \( t \) on \( m \).

**Definition 2.2.4** (Weakly Killing Mutants). Given a mutant \( m \in M \) (a set of all possible mutants) that modifies a location \( l \) in a program \( P \), and a test \( t \), \( t \) is said to weakly kill \( m \) if and only if the state of the execution of \( t \) on \( P \) is different from the state of the execution of \( t \) on \( m \) immediately after \( l \).

It is a good way to start with a simple example. Consider a Python code presented in Figure 2.1. Now the code prints "a" as five without entering the body of the if statement. It does so because condition \( a > b \) evaluates
to false. For equivalent mutant, we could, for example, mutate < to <=, which would be different, but give the same result in that case. What mutation testing is about is the act of modifying or mutating a certain action in the code. There are plenty of different possible mutations we can make in programming. A mutant is the created change. For example, we could mutate "the greater than" sign or replace b with a. There are some mutations more common and suited better for a majority of programs to be prioritized when nothing indicates otherwise. The Major framework used to create the mutation testing part for this thesis's program allow mutation operators depicted in Tables 2.1 and 2.2.

<table>
<thead>
<tr>
<th>Mutation Operators</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AOR (Arithmetic Operator Replacement)</strong></td>
<td>$a + b \mapsto a - b$</td>
</tr>
<tr>
<td><strong>AOR (Arithmetic Operator Replacement)</strong></td>
<td>$a + b \mapsto a - b$</td>
</tr>
<tr>
<td><strong>LOR (Logical Operator Replacement)</strong></td>
<td>$a \hat{\land} b \mapsto a \lor b$</td>
</tr>
<tr>
<td><strong>COR (Conditional Operator Replacement)</strong></td>
<td>$a</td>
</tr>
<tr>
<td><strong>ROR (Relational Operator Replacement)</strong></td>
<td>$a == b \mapsto a &gt;= b$</td>
</tr>
<tr>
<td><strong>SOR (Shift Operator Replacement)</strong></td>
<td>$a &gt;&gt; b \mapsto a &lt;&lt; b$</td>
</tr>
<tr>
<td><strong>ORU (Operator Replacement Unary)</strong></td>
<td>$-a \mapsto \neg a$</td>
</tr>
<tr>
<td><strong>EVR (Expression Value Replacement)</strong></td>
<td>return $a \mapsto \text{return } 0$</td>
</tr>
<tr>
<td>Replaces an expression (in an otherwise unmutated statement) with a default value.</td>
<td>int $a = b \mapsto \text{int } a = 0$</td>
</tr>
</tbody>
</table>

Table 2.1: Major’s Mutation Operators
### Mutation Operators

<table>
<thead>
<tr>
<th>Mutation operator</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LVR (Literal Value Replacement)</strong></td>
<td>Replaces a literal with a default value:</td>
</tr>
<tr>
<td>- A numerical literal is replaced</td>
<td>0 ➞ 1</td>
</tr>
<tr>
<td>with a positive number, a negative number, and zero.</td>
<td>1 ➞ -1</td>
</tr>
<tr>
<td>with a positive number, a negative number, and zero.</td>
<td>1 ➞ 0</td>
</tr>
<tr>
<td>- A boolean literal is replaced</td>
<td>true ➞ false</td>
</tr>
<tr>
<td>with its logical complement.</td>
<td>false ➞ true</td>
</tr>
<tr>
<td>- A String literal is replaced with the empty String.</td>
<td>&quot;Hello&quot; ➞ &quot;&quot;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>STD (Statement Deletion)</strong></th>
<th>Deletes (omits) a single statement:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- return statement</td>
<td>return a ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- break statement</td>
<td>break ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- continue statement</td>
<td>continue ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- Method call</td>
<td>foo(a,b) ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- Assignment</td>
<td>a = b ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- Pre/post increment</td>
<td>return ++a ➞ &lt;no-op&gt;</td>
</tr>
<tr>
<td>- Pre/post decrement</td>
<td>return -a ➞ &lt;no-op&gt;</td>
</tr>
</tbody>
</table>

Table 2.2: Major’s Mutation Operators

We do not mutate already mutated fragments; we only mutate the original chunk for it to make sense. The reason for that is cause we want to test the original code for possible defects, and mutating mutated fragment would mean testing something else. On the mutated code on the right, the sign ">" has been replaced by "<". The execution now flows through the body of the if statement and modifies a to be 20. In comparison to the
Figure 2.1: Example of mutation operators

original code, another value is printed, and if a test catches it, it kills the mutant, gaining a better score on the mutation testing evaluation.

2.2.3 Challenges of Mutation Testing

Challenges are not absent in mutation testing; different aspects need to be considered before using in the right way. We introduce the main challenges in mutation testing, describing what works better and worse in its application. We start by showing the consequences of making a mutant; then we move into different categories of mutants. We explain how problematic the time consumption is, and how we can speed it up, so to measure the tests with mutation testing. In the end, we see it concerning automation.

Mutation testing requires to copy the original code and mutate it; the problem lies in the fact that only one line is mutated, while we need to copy the whole executable code to decide if the outcome is different. It leads to more substantial use of resources for data transfer and storage, depending on the size and number of mutants. Unless there are better ways to handle this issue, that is how the original mutation testing works.

There are two main classes of software mutants. Weak mutants need to satisfy these two conditions: a test must reach the mutated statement and achieve different program states between the mutant and the original pro-
gram. Strong mutants need to satisfy those mentioned above weak mutant conditions as well as to make sure the incorrect program state propagates to the program’s output so that the test will test it. Some mutants are harmless; they are useless, so-called equivalent mutants. The reason why they are harmless is that they mutate the program, so it behaves the same, producing the same results and not affecting its performance noticeably. The division makes it challenging to categorize mutants, especially if we end up with a massive set of them. The work in this thesis connects to the problem, where we look for a spectrum of mutants based on a test suite we have to our disposition. The thesis is trying pinpoint how effective is each mutant and test when in a comparison between their big sets. An additional distinction can exist between random selection and machine learning selection. Such a look on this issue will help with spotting differences in resources and efficacy.

Another thing is consumption of time for mutation generation, and testing it to identify mutants as killed or equivalent. We must multiply the time by the number of mutants plus generation and do other additional tasks.

Another critical aspect is the challenge to make it run faster or produce a more reliable measure of the test suite strength; select mutation operators or reduce the time are the most researched ones; even for small programs the high amount of possible mutants can be problematic. More about it will be presented in Chapter 2.4.

The positive side of mutation testing is the scale to which we can measure our tests and making them better that way. It is a competitive method with high effectiveness; with researching modified version of mutation testing for example in [18] they state a high percentage of success in different categories they tested concluding a successful study about results.

The general mechanism is not very complicated, it can be highly automated and is independent of the problem domain; applicable to any program, as long as we have the tests and a way to generate mutants.
2.3 Machine Learning

This chapter covers machine learning concept, what it is, and how it is related to our project.

The Core of Machine Learning

By the standard definition, machine learning is a scientific study of algorithms and statistical models. It is carried out by computer systems to effectively perform a specific task without taking explicit instructions, all by relying on inference and patterns, as a faster, but simplified version of a human brain. In order to make predictions and decisions, it builds a mathematical model of sample data.

Machine learning is what the name implies, namely a machine or a program, which learns. Its name can often be confused by an individual. Some put artificial intelligence and machine learning on the same pedestal, but humans made many definitions for what AI is. We will stick to a definition that properly fits this thesis. The important thing is that we all agree that AI is aiming to simulate humans behaviour with a noticeable difference, to execute same tasks multitude times faster, or many many more tasks in the same amount of time. That is what drives us to research for it, we humans are curious and impatient, though the second is a more in-depth topic.

Since artificial intelligence can be more confusing and will probably be explained differently in the future, we will use the term machine learning from now on. Machine learning is the way of getting an understanding of how things work in a set environment, together with the possible actions it can take. Using a metaphor can be a good starting point to understand this concept. Imagine a baby starting to crawl and do other activities for the first time in its first years until it learns and masters them. A machine-learned program will do the same; try and fail as many times as needed until its programmed goal gets reached. Using more technical terms, it
is an idea of linking algorithms and statistics, learning from surrounding data it can access. Deep learning is a part of machine learning and a popular method to make it work. We have a massive set of numbers without knowing its arrangement; then using some complicated algorithms, we set those numbers. The main task for machine learning is, therefore, to collect data, train it and master it, so we end up with a well-learned machine. Today we usually build one for a single task, but there is potential in it we yet are going to see.

We can conclude the definition that machine learning does learn statistical patterns from data and derive behavioural policies rules from these patterns, and helps us do many tasks. Machine learning can lead to a speed where one entity colossally exceeds what human brain capacity can manage in the same interval of time, depending on calculation power and chosen method.

**Application of Machine Learning in Mutation Testing**

The primary purpose of the project is to combine machine learning with mutation testing. Researches have done previous work in this area, but only recently has it started to attract attention. This attraction results from the breakthrough achievements machine learning has made in other fields, like image recognition or face speech reproduction. Machine learning is already blooming and has the potential to change our future we will hardly recognize.

Aside from its all possible applications, we want to focus on how it can be useful in combination with mutation testing. Mutation testing can be utilized by machine learning in a few ways. One possible implementation is to apply it for mutant and mutant operator selection; not every operator in a set is necessary for a given operator set, and not every combination of mutant will produce strong mutants. When we generate mutants, it depends on the program for which mutants are best suited; equivalent or trivial mutants are not welcome as suggested in paper [18]: "Existing
mutation techniques produce vast numbers of equivalent, trivial, and redundant mutants." Mutants need to be selected to reduce the redundancy of having something with little or no value for its purpose. Context and programming language and other aspects are essential to consider during selection. Abiding by the rules helps to reduce cost significantly, both in time and resources. It is also possible to split selection/filtering into more detailed tasks for machine learning, which would allow learning more fine-grained control over the process, but at the same time increases its complexity and difficulty to learn a successful policy.

We mentioned the most common way to use it, but another way could be to find relations between mutants and tests that are capable of killing them. Finding these relations is also known as subsumption. A quote from [22] paper explains it in the following way: "Informally, one mutant subsumes another if tests that kill the first also kill the second. Computing subsumption relations is, not surprisingly, undecidable." For more details, refer to just mentioned paper. Knowing how mutants are related can help us to better design the tests for a particular purpose, by grouping them and using that fact, for example. Learning test hierarchies is about discovering the best tests to kill the mutants and prioritize them, and this is another way of combining machine learning with mutation testing.

**Reinforcement Learning**

A well known and recently very auspicious area in machine learning is reinforcement learning. Reinforcement learning fills its environment with agents, actions, states and rewards. An environment is any place within which the actions and states can take place. An action is any move an agent can perform, while a state is a moment before or after the action. It is working in a cycle; an agent starts at the beginning of existence, then the agent acts, based on the environment it receives a reward going into a new state. A reward is a way to tell the agent how it performed in order to achieve a goal, good or bad; because it is striving to achieve perfection
in the given algorithm. Consider, for example, a snake game; in this game, we can move left or right; these moves are the actions. Every state would be each new position of the snake’s body after the action was carried out. Every crash into the walls would result as unfavourable, or zero rewards like dying. We could add a positive reward for picking up the apples, but because they appear randomly, it may have zero effect. The goal of this game is to be as long as possible before dying. The learning progression can last for thousands of rounds before it learns to die with a high snake’s body length. Once it does, we have a ready agent to perform a task at a level of a high skilled human; however it depends on the algorithm and environment we have for how well it can perform; nonetheless, currently every year we witness new successes in this field.

A fascinating occurrence has happened in two famous old school games: chess and Go. Artificial intelligence defeated their world champions, and reinforcement learning used in machine learning is the perpetrator. This occurrence has caused both fear and excitement for how reinforcement learning can shape our future. This way, new champions can be created in many computer games or simulate the behaviour of walking humans, quadrupeds and practically anything else; even if the movement may not perfect, we are getting there \[17, 21\].

Recently there is a trend to use reinforcement learning to learn Atari games. It is because they are quite simple and small in size. The common thing for all these games is that we do not care much about which game it is. Instead, we focus on complex, high dimensional problems like manipulation or navigation. We write general systems that with a little modification can be moved to different environments and act there just as beautiful. The reason for that is that the model learns from scratch, trying to find out the rules by himself, and focusing only on how to get the score better. It acts with impaired behaviour at the beginning, just to beat the master after enough iterations of training. It operates on calculating the probabilities based on the right and wrong actions, leading to a score,
which eventually gets impressively high. Thanks to the way all this works, it can be easily applied to many other environments.

However, we are still far from making them successful in a more ideal represented environment. A popular game like Dark Souls, where many players are doing speedruns (beating a game as fast as possible) is a good example, where the model would have to include a vast amount of data in order to learn it properly. Therefore we are continuously researching for more rapid computation hardware and better algorithms, which can handle vast quantities of data as smoothly and efficiently as possible. Sutton and Barto give a further in-depth introduction into reinforcement learning in [32].

2.4 Related Work

Ongoing research is present in utilizing mutation testing and machine learning, both separate and combined. There is a tremendous interest in machine learning from many fields and the public, that can be applied in many fields of heavy usage eventually; on the other hand, mutation testing helps to improve the tests, so all information systems work more reliable, and is appreciated by those who implement and use them.

SJ Guillaume in [15] researched mutant selection using machine learning, where she proposes a way to reduce the cost of mutation testing employing data mining and machine learning algorithms to reduce the number of mutant operators run. K-fold cross-validation algorithm is used to perform machine learning on mutation-selection with each fold being a single program on training models. Besides, their study includes algorithms such as the percentage reduction technique, the percentage reduction by mutation operator technique and operator reduction technique. The research met some difficulties in its way to achieve success in the form of programs, not producing desired information or accurate results. However, it accomplishes its goal: "This research accomplishes the goal of de-
termining if machine learning techniques produce better mutation subsets than algorithms that have been widely used and studied before this paper was written: random selection and operator selection [15].

J. and B. Strug [31] researched using classification (machine learning methods) for cost reduction of applying mutation testing. They propose structural similarity. This research deals with reducing the number of mutants to be executed; they explain it in this way: "In contrast to other mutant reduction approaches, which are based on some programming language related knowledge, this approach limits the number of mutants to be executed in a dynamic way i.e. depending on the structure of the program for which the mutants were generated." [31]. The authors report reductions in the number of mutants, but also conclude that further research in a few directions is possible.

Research from a Chinese team [35] proposes predictive mutation testing to alleviate the efficiency concern, the first approach to predict mutation testing results without mutant execution, using the Random Forest algorithm. Their experimental results on 163 real-world Java projects demonstrate that PMT (predictive mutation testing) can predict mutant execution results accurately. Also, they state that: "The comparison with traditional testing methodologies also shows that PMT is able to predict mutation testing results accurately with small overhead, demonstrating a good trade off between efficiency and effectiveness of mutation testing." [35]. Predicting results without executing the tests is an indicator, but not sufficient for actual results.

In [20], the authors try to study if mutants are a valid substitute for real faults in software testing. This study found many significant correlations pointing to exciting conclusions. Their study confirms a generally existing correlation on mutants as efficient replacements for real faults in software testing research; however, it is valid only if test suite’s mutation score correlates with its real fault detection rate. Furthermore, they found some impressive results on the coupling effect between real faults and mutants
generated. For more details, refer to their paper.

Another interesting article from [18] gives insight into how a program’s context is essential if we want to optimize mutant selection. The paper of the authors covers in-depth study focusing on mutant operators using program context where they try to eliminate equivalent, redundant and trivial mutants. Their results show the relevance of including program context for making decisions on mutant utility, which is vital to consider the decision of machine learning algorithms and their input features.

As the last example, Google has published its study on the state of mutation testing within Google corporation in 2018 [29]. They present a diff-based approach to mutation analysis that drastically reduces the number of mutants by omitting lines of code without statement coverage and lines that are determined to be uninteresting. They use experts to evaluate and judge for better precision. This approach is intended for large companies like Google. As a result, they significantly decrease the computation cost and cognitive overhead of surfacing mutation analysis results.
Chapter 3

Methodology

This chapter covers the concept of this project and its implementation. We present the work on a detailed level, trying to explain our approach in a general way. Firstly, we go through the concept and explain everything needed to build the skeleton form. Secondly, we present the implementation on an abstract level.

3.1 The Concept of the Game

This section devotes to give a descriptive explanation of our problem and the way we have approached it.

3.1.1 The Idea

The idea for this project is inspired by the game "Code Defenders"[30], a mutation testing game, where two people play against each other as a defender and an attacker. In the original mutation testing game, the primary purpose is education, whereas this thesis aims in research. Competitiveness here is about one player mutating the code, and the other writing tests. In the end, the player who can outsmart the other wins.

We use this idea, but instead of real people, we use machine learning agents that take the role of the players. The original idea expects the players to write their mutations and tests. To be able to use machine learning, we add a simplification. We rely on machine learning purely for the selection of the generated tests and mutants. It is not deliberately made to make it simple, but we are interested in exclusively selection. We can support this motivation by considering the following cases: Generating test cases for specific changes or learning to modify the programs are much more challenging problems, as they could become a topic for a master or PhD thesis. We, however, are engaged in the game playing setting, and also have robust tools for mutant and test generation is available for us to utilize. It is a part which takes a significant amount of time and is worth researching in order to improve this aspect of acquiring a fast and reliable selection of mutants and tests.

3.1.2 Structure

We depict the project in a UML diagram showing the most prominent facets of the program made for trying to solve our problem. In Figure 3.1, we see classes in our program being responsible for the main tasks. Below an informative explanation is given for each of them.
First, there needs to be an environment. The Game (the environment) has a job to ensure a connection between the defender and the attacker. It is an area wherein these two players can play. The Game has to offer everything they need to keep their connection, excluding their inner hierarchy, the functionality reserved for the agents themselves. For example, the game provides managing the results from the last round play, keeping both agents informed about the current state. Plotting graphs with the results, saving, and other relevant data is also happening here. In the lower hierarchy, we need some actors to make calculations. We decided to follow our developed scheme presented in Figure 3.1.

The Game contains two agents, namely an attacker and a defender. The defender is an agent responsible for operations regarding the tests.
All tests and their data is under its control. Same for the attacker, which controls mutants and their operations. Each of them has no access to each other; instead, the game is handling the rounds and their results. This game supports only two players (agents). To achieve better comparison, the agents have access to random selection in addition to several algorithms.

On the lower hierarchy level, the testing suite and mutation set resides. These are collections of all available tests and mutants to the agents’ disposition. Here we can access the data of each test and mutant, and operations regarding them. There can only be 1 test suite and mutation set per game.

Next, there are subsets for both mutation set and test suite. Subsets work as a unit for a further selection process, as a technique where we do not confront everyone versus everyone per encounter in around. Subsets are inherent to their sets and give purpose only for one round. There can be many subsets, but we restricted it to a random subset plus a selected subset per round. They also keep information about each test and mutant in their reach. For more explanation, refer to Chapter 3.3.

Each test and mutant represents a class itself, and exist inside the sets. Subsets can likewise reference tests and mutants. A test or mutant keep information necessary for the calculations like the score of times it got killed.

The score is the last class, and its purpose is straightforward. It keeps track of the points and enables to modify it in order to keep track of the value of each mutant or test. In the end, it can indicate how valuable each test and mutant is. To see the importance we need to study the proportions in correlation with the whole or develop a threshold system for that particular issue. Our solution is projecting the scores on a plot.

### 3.1.3 The Objective

The project represents progress in research by utilizing known techniques in order to discover a new way of showing the strength of mutants and tests in mutation testing. The way we use to measure the progress to-
Towards the goal is to count the time we need to learn a bandit, together with measuring the deviation accuracy. In our project, we can calculate the time by multiplying the round time with iterations, while the deviation accuracy is measured by comparing the score and selection charts of mutants and tests distribution. After having explained the blocks used to build this project, in the next sections from this chapter, we will present used techniques and how we dealt with them. In the following chapter, we present and discuss the results.

3.2 Techniques

Below we present solutions implemented in the project as well as the challenges we met while finding them.

3.2.1 Contextual Bandits

Bandits refer to the way bandit machines work, which are usually found at casinos. Contextual adds the aspect of context to a situation. They can be comprehended as an extension of multi-armed bandits, or as reinforcement learning in a simplified version.

Bandits can be both multi-armed as well as one-armed. The action of a one-armed bandit, when pulled, represents a probability of getting a reward. The more bandits in a place, the more are we willing to explore (try new machines). After a while, we stay with the most promising one, an exploitation phase. A multi-armed bandit lacks the information about its environment (context), and thus in a case where it would have to pick between two categories like a fish and bird, it would learn about the payoffs for each choice, treating them in the same context. Using a contextual model, we can do decision optimization based on previous observation, and also every situation will get personalized; various decisions are made in separate contexts [23]. Having many alternative actions to our disposi-
tion, we can make a decision, based on an observed context. The decision’s outcome defines the reward.

A contextual bandit performs in iterations and consists of \( n \) arms \( \mathcal{A} \). Each iteration \( j \) provides a context \( x_j \) to the bandit. The bandit selects an arm and receives a reward from the action \( r_{j,a_j} \) based on previous rewards and exploration strategy. Only one arm can be selected and receive feedback. Later, the bandit’s strategy gets updated from the observation \((x_j, a_j, r_j,a_j)\). The goal of the contextual bandit is to maximize the cumulative reward across all iterations \( \sum_{j=1,2,3,...} r_{j,a_j} \). The challenging part of the contextual bandits’ design is to find a balance between exploration, i.e. evaluating the effect of seldom used actions, and exploitation, i.e. repeating previously successful actions.

Contextual bandits are related to the machine learning area of reinforcement learning [32]. Reinforcement learning in contrast to contextual bandits adds additional function in the form of multiple consecutive actions. These actions lead to sparse reward information. Reinforcement learning agents are designed to operate over multiple subsequent iterations, where a chosen action influences the context in the next iteration. We may need to go through many state-action cycles before seeing an actual reward. This approach is useful in, for example, chess, where we only know the reward after finishing the game.

Contextual bandits especially attract researchers’ attention in the field of statistics, being the base on many concepts from statistics.

Bandit algorithms have been successfully applied in a variety of domains, such as news article recommendation [24], advertisement selection [26, 33], statistical software testing [5], constraint optimization [4, 25], or real-time strategy games [27].

There are many different types of reinforcement learning, but eventually the decision fell on contextual bandits as it looks promising in the case of this thesis; additionally many libraries have been made that supports it conveniently, so we could focus on the project as a whole, rather
than putting too much focus only on one facet of the thesis.

3.2.2 Feature Engineering

Feature engineering exists to enable machine learning algorithms to operate correctly and in a meaningful way. All machine learning algorithms use input data in order to produce output data, and we feed the input data as features. In order to create features, we need to extract domain knowledge of the data[9]. In our testing context, we need knowledge about our tests to derive features from them. We provide features in form of a feature vector, i.e. an array of integer or floating-point values; each of these values describes a unique characteristic representing the whole array in an ordered way. Representation of this kind enables to discriminate combinations of different properties or characteristics. Just like we differentiate between gold and wood, an agent can use it to learn which combinations are preferable than others in a giving environment.

Another aspect of features is that they can have various representations. While in traditional programming, the code is the centre of programmers’ focus, in machine learning projects the representation is the core. Numerical and categorical are the two most common representations for the features.

The numerical representation is quite simple, as it does not require any additional efforts because these integers or float values can be multiplied by a numeric weight as they are.

The categorical is represented by categories which may be anything that suits in a given context. For example, we can have categories of cats and dogs and impute different races to the respective category. Having this categorical structure, we need to map these strings to their respective numerical values, always staying the same, having more categories than one requires to remain the same order of arranging them. Using this system alone may impose problematic constraints. For example, have something having more than one attributes, like colours, and being both white
and black. To remove these constraints, we can apply a binary vector for each categorical feature. Vector elements applying to the corresponding attribute, are set with one, while the rest are set to zero. This system is called one-hot encoding when a single value is one, and a multi-hot encoding when multiple values are one. We use one-hot encoding for categorical representation in our program. This solution, however, becomes inefficient when the total amount of elements grows to a significant number; in this case, we need to look for a different solution, for example, sparse representation. [7, 16]

One of the challenges that makes feature engineering laborious and expensive is the fact that finding out useful features can take much effort, and discovering the best properties may take an indefinite amount of time. Nevertheless, there is more than one way of solving the problem, utilizing the dependence on the model and the data. Scientists have introduced feature learning to alleviate that problem, where the system does the finding for us. However, it still does only work great for specific fields of work.

Some people argue that features are the most critical aspect of the machine learning process. If looking from a metaphorical perspective, the features act as keys, where only with the correct keys, the ultimate doors (output) will open. It often takes much time to find those satisfying features, and ascertain that the over or under-fitting does not occur.

To do feature engineering on a highly trustful level, one needs to use expert knowledge and be aware of the time it can take to come up with satisfying features finally. Considering all these difficulties, we need to pinpoint that fine-tuning the features would go outside of this thesis range, and we need to remember that features have a high impact on the results achieved.

3.2.3 Generation Tools

The very first and most essential elements the game needs are tests and mutants. For that purpose, we decided to go with generating programs
made strictly for those and operating with them.

The very first problem we stuck upon when searching and implementing was to satisfy the compatibility of programming languages and their versions. In our case, we went with Java as it is one of the most popular and researched areas of these topics. Gaining version compatibility led to tricky workarounds as a result of the programs not getting any updates for newer versions of Java. Installing both Java 7 and 8 was a necessity. Securing correct paths at different points made it necessary to change the paths in files as well as in our scripts.

**Mutant Generation**

Eventually, the decision landed on Major [19], accessible and scalable mutation testing for Java. As it states in their home page: "The Major mutation framework enables efficient mutation analysis of large software systems as well as fundamental research on mutation testing." We could use it to generate a proper amount of mutants, where at average it produced over 100 mutants for the programs we used for experimenting. The advantage we got from Major is its reconfigurability for our use case. Additionally, Major provides out-of-the-box information about the mutants and allows to export the generated mutant source code easily. The way generation for Major works is by reading a file, which can be freely modified with predetermined mutation operators. Major consecutively uses this file on the source file, trying to mutate it into many distinct mutated files. To see Major’s mutants performance see the experiments Chapter 4. The main problem we faced with Major is that it runs on an already old Java seven version.

Once we finished the less demanding complications, a few programs were still not operating as expected. One example is PIT, a highly considered program by us for generating the mutants and run them on the tests. We tried to use it, but without success. PIT did not work because it created

---

2PIT: http://pitest.org/
the mutants in byte code, but we needed them in the source code. Also, there was too little context information about them, which would make it more arduous to do feature engineering on the mutants.

Test Case Generation

As for the testing part, the testing suite requires a tool that generates the tests. EvoSuite (published in [13]) offered all the functionality we needed to accomplish that. Here is a short description to EvoSuite from their GitHub page: "EvoSuite automatically generates JUnit test suites for Java classes, targeting code coverage criteria such as branch coverage. It uses an evolutionary approach based on a genetic algorithm to derive test suites. To improve readability, the generated unit tests are minimized, and regression assertions that capture the current behavior of the tested classes are added to the tests. EvoSuite generates the tests by analyzing the source file and writing them all into a single file where each method is a test with a unique number. Both academia and industry are using EvoSuite, as it is an established tool [1,3,12].

EvoSuite is built and updated to Java 8 and Major is only running on Java 7. That hurdle means the need for both versions; however, tests generated by EvoSuite can still be used by Major on Java 7.

3.2.4 Execution Tools

The Major framework carries out the execution, which operates the tests on the mutants and produces different valuable information we can later use to produce tables and graphs. That means we solely use EvoSuite to generate our tests, and Major to do both generation and execution.

Major generates reports containing useful information about the context and results of its execution. From these files, we can obtain kill and coverage information showing the correlation between different tests and

\[3\] EvoSuite GIT: https://github.com/EvoSuite/evosuite
mutants. EvoSuite, on the other hand, does not produce any relevant information that is of use, other than pointers to which test number refers to which test in the files.

3.2.5 Additional Tools

To acquire information necessary to extract features for the tests it was required to use a tool capable of retrieving more detailed coverage data, which is not provided by EvoSuite.

JaCoCo provides a structured retrieval of testing code coverage data. The structural coverage report can be generated for each test by JaCoCo. We used that in order to provide better features as context information for the bandit agent.

3.2.6 Bandit Algorithms

We have used four different bandit algorithms in this project:

- Epsilon Greedy [32]
- Active Explorer [8]
- Softmax Explorer [8]

Epsilon Greedy

Epsilon greedy works by taking a random action with probability $p$, or an action with the highest estimated reward, where the probability is then $1 - p$. In the first turn, it either chooses the best arm or goes to the second turn and chooses between many arms with the same probability.

Footnote: JaCoCo: https://www.jacoco.org/jacoco/index.html
Active Explorer

Active explorer is based on selecting a proportion of actions, according to an active learning heuristic, based on a gradient. It only operates with differentiable functions, preferably smooth ones.

Softmax Explorer

Softmax explorer picks an action according to probabilities that are determined by a softmax transformation on scores from a decision function that predicts each class.

Adaptive Greedy

Adaptive greedy is looking for the highest estimated reward, except when the estimation falls below a certain threshold. If the case occurs, then it takes action either according to an active learning heuristic or at random.

3.3 Implementation of the Game

Having introduced the scheme of this project, we can discuss its implementation. In this section, we cover all the essential aspects of our implementation, assisting in understanding and potential reconstruction. Following subsections explain the executed process.

3.3.1 Structure

We use illustrations for more apparent perception. Figures 3.2, 3.3, and 3.4 show sequence diagrams which present the main process occurring during the program runs. They are all connected and show different depths of the process.
Figure 3.2: Mutants Defender Sequence Diagram
Figure 3.3: Mutants Defender Sequence Diagram Round Play

Figure 3.4: Mutants Defender Sequence Diagram Testing

34
Sequence diagram 3.2 is portraying the main body of the process. It starts with defender and attacker generating their sets of tests or mutants. After that, we enter into a loop where we cycle through a predetermined amount of round iterations to calculate the ultimate results. Depending on the number of iterations, the results may vary. After that loop, we exit and compare results, producing an outcome straightforwardly.

Second diagram 3.3 shows the insides of the round loop. Firstly we randomly create subsets of the main sets for both mutants and tests. This way, we preserve the fair play for all entities in regards to mutants and tests separately, leading to balanced matches for each round of the game. The fair play is preserved because, in the long run, randomness will ensure equal chances of being a candidate for selecting. Secondly, from that random subset, the agents pick up a final subset, which in contrast to the previous is now picked based on the chosen algorithm for each agent. Then it moves into testing, which leads us to the last sequence diagram 3.4.

In the final sequence diagram, we separately confront each test with each mutant. With received information on how the battle went, we update each mutant and test with a value. This matter is further explained in the next section.

### 3.3.2 Core Mechanism

In order to produce the intended results, the programs’ parts need to be placed into a mechanism. Firstly we have space where we create objects of defender and attacker, as well as everything else unrelated to these two objects’ methods. In this space, we have several main mechanisms operating the calculations, which are the focus of the rest of this section, and as a whole, will help to clarify the entire process.
**Parameters**

We use parameters to measure various facets of this project, trying to find suitable values for them. Therefore we do experiments to understand their influence and learn about how to set them. It aids us to see what causes a change, and in what degree it does it. We decide to go with a few main parameters for that propose.

**Iterations**  Iterations parameter tells the round loop how many times it should run. It is essential to not do too few in order to get satisfying results; however, too many could not yield significantly better outcome considering the run time. The wisest decision is to initially set it to a value within the acceptable time and computational power resources.

**Subset Size**  Subset sizes refer to the initial subsets we get from the big pool. A few percents may be a rational choice if we have lots of mutants or tests in total.

**Pick Limit Multiplayer**  The pick limit multiplayer in percentage is used to calculate the next maximum size of the final subset chosen by the preset algorithm.

**Winning Threshold for the Defender**  Winning threshold is a value in percentage for the defender. The tests must at least exceed the threshold with its kills, to win with the mutants in one round.

**Mode for an Agent**  This mechanism takes the whole round’s set results to decide the outcome of the agent. The mode is to decide if we want an agent to be a bandit or something else, for example, pure random selection.
The last item in the list refers to the existing algorithms of the agent; for example, a bandit can use epsilon greedy or active explorer algorithms (sec. 3.2.6).

In addition to these modifiers, we can choose the running program.

Generation

Before we can start, the agents need their sets of mutants or tests to operate. Acquiring tests happens through generating them. As the defender and attacker get created, they generate and read the mutants and tests using programs mentioned in Section 3.2.3. These are stored in their pre-made objects, previously seen in Figure 3.1.

Round System

The round system can be perceived as a core of the program, a loop that runs through a predetermined number of iterations, which preferably should be at least a few hundreds. In each iteration, the agents are gathering new information through running the tests on mutants based on the set parameters. Based on the reinforcement learning principles, the agents are learning their job over time, and the selection is expected to improve the longer we run the loop until there is not much left to learn. Ideally, we end up in a balance between underfitting and overfitting.

The are several essential processes in the round system. We explain the facets of the round system in the following paragraphs as the vital mechanisms that run this process.

Filtering

The tests that are about to be confronted with the mutants must cover them; otherwise, the collection of information will take more iterations, as not all of the subset’s elements can be utilized in every round. Subsets are created and filtered in order to satisfy the coverage of tests on the selected
mutants each time the creation takes place, twice in this case. First to pick a random matching subset, and then to use the agent’s algorithm. This approach is necessary to make the filtering work and achieve high efficiency by its application. In short, filtering secures coverage of the mutants by the tests for maximal performance.

**Selection**

Selection is the process of picking up mutants and tests by a previously chosen algorithm. The simplest way is to select them randomly, but such a handling is not going to bring any good results, as it is purely lack oriented; however it can be used as an indicator to compare results on average. An algorithm that learns over time is the sought solution. An algorithm capable of doing so can with time be able to pick only the desired entities. This scenario involves subprocesses that make it possible, and we explain them in the next paragraphs.

**Execution**

Having the selected subsets, we are ready to commence the execution. The program uses tests on the mutants and produces an outcome in the form of files, which contain the results of the latest operation. The program responsible for running the tests is operating on its own; it merely needs to be provided with input data from our side.

**Updating**

One of the most crucial parts in the round loop is to keep our agents updated with the freshly acquired information. It is one of the inevitable actions machine learning requires, as only with updated values the learning can occur. After the execution has introduced new values, we update them before the learning part can commence. Values that are updated are, for example, kill, survive count or score based on kill ratio.
Reward

The reward is a principal value impacting the score of a mutant or test. The score itself represents the strength of a unit, the higher, the stronger or more valuable it is. Later it can be used to differentiate the proportions between themselves. To calculate the reward, we use the ratio of killed mutants by tests from the latest execution. Below is a formula of the reward which is read as "k" the kill ratio, "m" the killed mutants and "s" the size of the subset.

**Definition 3.3.1 (Reward Formula).** \( k = \frac{m}{s} \)

Happens the kill ratio to be higher than the winning threshold, then the defender wins, and contrariwise for the attacker to win.

Learning

Learning is part that finally occurs when the execution produced all the necessary values for one round. With the values we have got, the learning needs to occur if we use a reinforcement learning strategy. Learning is a step that prepares for the next round, giving a higher chance of selecting the right element. This thesis’s solution is dependent on feature-based learning described below.

Features

The way learning occurs for this algorithm is by introducing features explained in [3.2.2]. The values used in order to build the features for further learning are independent for both agents. The attacker is using values taken from the mutants’ context like an operator or type of child it had. It creates an encoder of these categories, where later in every round we use the encoder to read the specifications for each mutant values. The defender, on the other hand, uses a coverage report to extract data. Moreover, it operates on two dynamic values, namely how many times it killed the
mutants and self was selected until the current moment. Based on these features, a prediction is calculated, and the selection can take place.

**Contextual Bandits**

This thesis supports reinforcement learning in the form of contextual bandits with action-depended features. It is capable of running different prediction algorithms like epsilon greedy (sec. 3.2.6), explore first, or softmax explorer (sec. 3.2.6) for instance.

**Externals**

Additionally, we do use plots and tables to present our outcome effectively.
Chapter 4

Experiments

In this chapter, we show the conducted experiments. From those experiments, we dissect the achieved results and discuss them. This chapter is abundant in diagrams and tables to show the outcome in an approachable and informative way.

4.1 Expectations

The expectations we want to achieve by running the experiments are crucial. There are many values we can generate from this program. Ultimately we want to see the comparison between random selection and well-chosen configurations. To do that, we test many configurations, and from those, we select the most promising. Additional tests on the well-selected configurations get compared to those run at random, and we should end up with a decent comparison.

There is more than only one aspect we want to discover. Differentiating bad from good mutants and tests gives us valuable information for which entities to likely skip, and which need more attention; the proportions on a chart should fill us with this information. Another aspect is the exploration of discovered mutants and test. The faster the mutants and tests get explored, the fairer the run is; furthermore it decreases the number of
iterations in a run needed for a qualitative exploration and exploitation. Specifically, we have a few main goals we want to achieve:

- Can we attain an effective exploration of both mutants and tests?
- How are the chosen parameters affecting the results?
- Is a reliable agent selection of mutants and tests possible in our setting?
- What are the strong and weak facets of this project worth remembering?

4.2 Results

We have conducted experiments on a few programs with different configurations to see the differences. We are mostly interested in finding out how different configurations differ and what do the cause. Our final goal is to identify the benefits of a reinforcement learning strategy over random selection. Following that, we present the salient results the experiments produced

4.2.1 Experimental Programs

Several programs have been picked to serve as an experimental object. In theory, they are supposed to produce a bit different results because of the code structure. Below we list the programs prepared for the experiments together with a short explanation. These programs are taken from open source projects and were originally used to evaluate the original game Code Defenders [30]. All programs consist of a single class, but considering how long it takes to run the experiments, it is quite ideally suited for this project. All of them are Java programs.

Table 4.1 outlines some information about the programs.
Table 4.1: Information for Experimental Programs

<table>
<thead>
<tr>
<th>Programs</th>
<th>Lines</th>
<th>Methods</th>
<th>Tests</th>
<th>Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Font Info</td>
<td>170</td>
<td>18</td>
<td>40</td>
<td>113</td>
</tr>
<tr>
<td>Inflection</td>
<td>195</td>
<td>12</td>
<td>17</td>
<td>84</td>
</tr>
<tr>
<td>Hierarchy Property Parser</td>
<td>639</td>
<td>31</td>
<td>63</td>
<td>248</td>
</tr>
<tr>
<td>Range</td>
<td>475</td>
<td>26</td>
<td>59</td>
<td>130</td>
</tr>
</tbody>
</table>

Font Info is a program doing font modifications. It is capable of generating and setting new font styles as well as read the info of a font in a text. Font Info is a simple code piece.

Inflection

Inflection’s implementation uses Rails. The objective is to handle singularization and pluralization of Rails strings. Directly speaking it is about changing the form of a word (typically the ending) to express a grammatical function or attribute like a person, mood, tense, case, number and gender. It operates with functions like matching strings, replacing words, singularize, pluralize, and checks if a word is uncountable.

Range

Range is operating on an immutable range of objects from a minimum to maximum point inclusive. Objects need to fulfil a condition in the form of being implementations of Comparable, or a Comparator needs to be supplied. It supports functions concerning range like calculating intersection, overlapping, stepping outside of the range, and some more.
Hierarchy Property Parser

Hierarchy Property Parser implements a parser to read properties having a hierarchical tree structure. It is a simple program, conceptually similar to XML DOM/SAX parser. The program provides interfaces which make it possible to build a parser tree capable of traversing the tree. It can check whether a given string has a hierarchy structure with the separators, or tokenize a given string based on a separator and put the tokens into an array of strings. It also supports traversing and handling the branches.

Together with Range, they are the most complex programs in our collection for the experiments.

4.2.2 Configuration Experiments

The first task is to discover a configuration that is fair and reliable in order to produce trustworthy output. It is easy to type random input values and get wrong output values. In order to make a decision, factors like kill ratio, exploring rate, pick ratio and score will be used to make an evaluation. Round time also helps in discovering a configuration which is less or more resource consuming. Another critical aspect of researching on is the distribution of the actual distribution of the score mutants and tests achieved separately. The selection from subsets ratio can conceptually work similarly to the score; therefore, it should give interesting comparative information. The crucial part is to find a threshold in harmony with the killing ratio to maintain appropriate learning behaviour for the machine learning agent. Wins should oscillate around 50% for that reason.

In this step, we run four distinct programs, and seven configurations were prepared to more precisely pinpoint the ideal parameters explained in Section 3.3.2. Below we present the results gained by running the programs on these configurations. The goal is to find a suitable configuration, which will serve as a base to do additional tests with more detailed configurations.
### Configurations Explained

<table>
<thead>
<tr>
<th>Config</th>
<th>Rounds</th>
<th>MSS</th>
<th>TSS</th>
<th>MPLM</th>
<th>WT</th>
<th>AM</th>
<th>DM</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>500</td>
<td>12</td>
<td>4</td>
<td>50.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c2</td>
<td>500</td>
<td>10</td>
<td>10</td>
<td>50.0</td>
<td>15.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c3</td>
<td>500</td>
<td>10</td>
<td>10</td>
<td>75.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c4</td>
<td>500</td>
<td>4</td>
<td>12</td>
<td>50.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c5</td>
<td>500</td>
<td>10</td>
<td>10</td>
<td>50.0</td>
<td>35.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c6</td>
<td>500</td>
<td>10</td>
<td>10</td>
<td>25.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c7</td>
<td>500</td>
<td>10</td>
<td>10</td>
<td>50.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
</tbody>
</table>


Table 4.2 displays the parameters we care about and their corresponding values. The focus, for now, is only set on all but last three values, as they stay the same for the task of finding out the most suitable configuration. To explain the table: Config c1 through c7 represents the used configuration; rounds are the iterations or the loop count; MMS stands for mutants subset size, representing how many mutants get picked to the very first subset at random from the main mutant’s set; TSS stands for tests subset size, and just as mutants, the tests are selected at random to form the tests subset from the main test suite; MPLM stands for model pick limit multiplier, which signifies how many units in per cent (mutants and tests) are selected by an agent, for example 50% means half as much as the MSS or TSS; WT stands for winning threshold, a value in per cent, which the tests needs to achieve by killing the mutants in one round to win a round; AM and DM stand for attacker mode and defender mode, a version of reinforcement learning to use; BA stands for bandit algorithm, e.g. epsilon greedy (sec. 3.2.6).
There are some aspects of the parameters worth mentioning. The Rounds is set to 500 for all the experiments performing in this thesis. For the sake of time management, it was set to 500, as it in most cases equals about one hour of running the program. A more significant value would make the scope too huge or would require a more powerful computational power to run it on; therefore, the decision landed on 500. Five hundred iterations are enough to produce impressive results as we present farther below. MMS and TSS are mostly set close to half the value of the average tests created. We based it on the tests, as the generation of mutants is about 5 to 10 times as big, and we were interested to see results of them being equal.

Two of the configurations has three times bigger or lower value in the sets. By setting it to those values, we wanted to see how the proportions differentiation affect the result. MPLM is a tricky parameter. Generally, the smaller the value, the more critical it is for the agent to select the correct mutant or test; however, the learning aspect will need more time to learn about the mutants and tests. For that reason, we test it in our configurations with a midpoint, a lower and an upper quarter. We initially set the winning threshold to 50. Preliminary experiments, however, revealed that to make it fair for both mutants and tests, and hence achieve reliable results, the value needs to be close to the average of killing ratio, i.e. how many mutants were killed per round at average. For more info about killing ratio, refer to the next paragraph. AM and DM in this project support only "scikit-learn". Scikit-learn is free software, offering machine learning library for the Python programming language [28]. In the desire of applying something else, one would have to implement additional functionality for it; the difficulty of implementing it would depend on the technology used. A few others were attempted but did not work with our intentions. Finally "scikit-learn" delivered the sought functionality, and we decided to stay with it focusing more on other aspects. Until we find the satisfying configuration, epsilon greedy (sec. 3.2.6) serves as BA. Then a few more promising and supported by "Scikit-learn" will be tested.
More configuration combinations could be created for more reliable experiments; however, it would result in way too many charts and work to be conducted for this thesis. However, further research possibilities are open for future work on this topic.

Now we are going to go through the programs one by one and analyze the results they produced in the form of tables, charts and graphs. Afterwards, we do the same, but for more detailed configurations. The discussion of the achieved results takes place after the analysis of the result in Section 4.3.

**Font Info Results**

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>8.31</td>
<td>50.06</td>
<td>45.52</td>
<td>2.89</td>
<td>92.0</td>
</tr>
<tr>
<td>c2</td>
<td>23.24</td>
<td>49.82</td>
<td>51.61</td>
<td>4.34</td>
<td>27.6</td>
</tr>
<tr>
<td>c3</td>
<td>27.12</td>
<td>79.98</td>
<td>88.81</td>
<td>6.10</td>
<td>62.2</td>
</tr>
<tr>
<td>c4</td>
<td>33.60</td>
<td>50.34</td>
<td>55.66</td>
<td>3.58</td>
<td>43.2</td>
</tr>
<tr>
<td>c5</td>
<td>24.44</td>
<td>49.87</td>
<td>52.93</td>
<td>4.33</td>
<td>64.2</td>
</tr>
<tr>
<td>c6</td>
<td>17.19</td>
<td>30.13</td>
<td>28.50</td>
<td>2.99</td>
<td>56.4</td>
</tr>
<tr>
<td>c7</td>
<td>23.60</td>
<td>49.96</td>
<td>53.35</td>
<td>4.38</td>
<td>66.2</td>
</tr>
</tbody>
</table>

Table 4.3: Results for Program Font Info | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

Table 4.3 summarizes all the configurations for the program "Font Info" with valuable output. The table’s first column represents value for each configuration. K ratio stands for killing ratio and is the ratio of how many mutants were killed by the tests per round at average in per cent. PM mean is the mean of mutants picked from a subset through all rounds in
percentage. PT mean is the mean of tests picked from a subset through all rounds in percentage. The value should be about the same as the multiplier at which the units are selected. R Time counts the average round time or iteration in second. The last column shows the percentage of the wins for the attacker. Respectively the defender’s wins are the remaining percentage of the total of 100.

From the table observe that kill ratio is ranging between 8.31% to 33.60%, showing that mutants have little difficulties in resisting the tests with these configurations. We can conclude from the table that the cause is not the pick ratio from the subsets. Depending on the set threshold, mutants and tests subset sizes, the win percentage ranges from 27.6% to 92.0%. It is a lot and especially the configurations with three times more significant disproportions in subset size prove this inference. While having fewer tests leans too much in favour of the attacker, having more tests than mutants results in more balance. Changing the multiplier appears to not change much in kill ratio, about 10%, but the time for each round is twice as big. This reasoning leads us to not bet on too high MPLM, as it only increases the time, while we can gain a similar result with less time. Config 4 seems the most suitable from the table’s analysis.

In addition to these values, there is still valuable information to retrieve from the runs; one of them is the exploration rate for both mutants and test. Graphs 4.1 and 4.2 shows how the exploring rate developed through the all 500 iterations.
For the mutants, all except one configuration take approximately the same time to get fully explored. They reach the top around round 50, while c4 reaches there little before 150. Even when it seems not that much on the graph, c4 loses to the rest in this context. The tests exploration graph yields the same behaviour, the configuration with low subset performs worst, not even reaching the top. C6 and c4 are the winners reaching the top before round 100. It is worth noticing that all of them reach half of the explored tests at the same rate. Then the bigger subset makes it more likely to reach the top first. Close distance from reaching the top, much luck is involved in the subset sizes are the same. The conclusion from the observation is not to set the size too low; however, too high will not change much; 10-15 seems the right choice.

Another interesting aspect to look at is the distribution for each round. The distribution helps to discover disparities that could occur.
Figure 4.3: c1 Kill Ratio

Figure 4.4: c2 Kill Ratio

Figure 4.5: c3 Kill Ratio

Figure 4.6: c4 Kill Ratio
As we can see on charts 4.7 through 4.9, the variance is clearly visible. C1 is mainly staying below 20%, while only three times it went up to about 50%. On the fourth configuration, it is either 0, 50% or 100%, with 50% as most common. Most of these range below 60%, confirming the mean values from the Table 4.3. We have confirmed that the distribution can vary a lot depending on the configurations; nevertheless, in the end, we are interested in the mean, unless we need this information.

The last salient facet of this analysis is the distribution of mutants and tests according to the score and pick ratio. This information is especially important, as, in the end, we want to discover useful and ineffective tests or mutants. Below we confront these two facets presenting them in chart form, where the correlation can be observed in Figures from 4.10 to 4.23.
Figure 4.10: c1 Mutants Pick Ratio
Figure 4.11: c1 Mutants Score

Figure 4.12: c2 Mutants Pick Ratio
Figure 4.13: c2 Mutants Score

Figure 4.14: c3 Mutants Pick Ratio
Figure 4.15: c3 Mutants Score
The distribution of mutants by pick ratio is not reflected in the distribution by score. Moreover, as it is quite visible in the score charts, which is good and bad, this is harder to evaluate from the pick ratio charts. Especially when the MPLM is set to a high value, all the mutants seem to be picked at random. The smaller MSS or MPLM is, the more significant the variance in pick ratio distribution. To improve the result, increasing the number of iterations, or giving more insight into the features might help in achieving better results. Working on configurations and score system could also unravel the disproportions.
Figure 4.26: c2 Tests Pick Ratio

Figure 4.27: c2 Tests Score

Figure 4.28: c3 Tests Pick Ratio

Figure 4.29: c3 Tests Score

Figure 4.30: c4 Tests Pick Ratio

Figure 4.31: c4 Tests Score
For the tests, we see similar behaviour; however, the most prominent tests in the tests score distribution are also reflected in the tests pick ratio distribution.

For both tests and mutants, we can see in the score distribution, that no matter the configuration, we finally end up discovering at least the best and worst units. The main problem is that these experiments are not enough to get a proper selection.

We repeat the same experiments for the remaining three programs. After that, the detailed experiments in Section 4.2.3 produce more specific information.

### Inflection Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>12.74</td>
<td>49.99</td>
<td>54.57</td>
<td>3.80</td>
<td>86.2</td>
</tr>
<tr>
<td>c2</td>
<td>10.12</td>
<td>49.98</td>
<td>65.92</td>
<td>5.68</td>
<td>59.8</td>
</tr>
<tr>
<td>c3</td>
<td>16.25</td>
<td>79.97</td>
<td>96.47</td>
<td>9.57</td>
<td>86.8</td>
</tr>
<tr>
<td>c4</td>
<td>13.80</td>
<td>50.18</td>
<td>71.35</td>
<td>4.45</td>
<td>74.0</td>
</tr>
<tr>
<td>c5</td>
<td>10.76</td>
<td>50.06</td>
<td>65.06</td>
<td>5.66</td>
<td>89.6</td>
</tr>
<tr>
<td>c6</td>
<td>6.86</td>
<td>30.02</td>
<td>39.15</td>
<td>3.35</td>
<td>80.6</td>
</tr>
<tr>
<td>c7</td>
<td>10.20</td>
<td>49.87</td>
<td>65.05</td>
<td>5.62</td>
<td>91.6</td>
</tr>
</tbody>
</table>

Table 4.4: Results for Program Inflection | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

In this case in Table 4.4 kill ratio stays on a low level ranging between 6.86% and 16.25%. PT mean is very high, while PM mean is almost the same as in the previous program. Round time takes a bit longer in this case; however, the pattern for each configuration is the same as in "Font Info". The attacker wins in all configurations at around 80%.
The exploration (fig. 4.38 and 4.39) for the mutants and tests looks almost exactly the same as for the previous program. We notice that there are only 17 tests, though, resulting in almost instant exploration for all configurations. The conclusion from this is that the number of tests and mutants available matters in terms of exploration: the bigger sets we have, the more configurations matter.
Killing ratio distributions as in the previous program "Font Info" yield
the same pattern, but with weaker intensity. The intensity can thus be stronger or weaker, depending on the program, with the pattern of a configuration. Therefore they are not included in the next programs unless relevant to explain a point. They are accessible in appendix B.

Figure 4.47: c1 Mutants Pick Ratio

Figure 4.48: c1 Mutants Score

Figure 4.49: c2 Mutants Pick Ratio

Figure 4.50: c2 Mutants Score
Figure 4.69: c5 Tests Pick Ratio

Figure 4.70: c5 Tests Score

Figure 4.71: c6 Tests Pick Ratio

Figure 4.72: c6 Tests Score

Figure 4.73: c7 Tests Pick Ratio

Figure 4.74: c7 Tests Score
From Figures 4.47 to 4.60 we can deduce the same behavior as in case of "Font Info" regarding the mutants. However, the distribution of the tests by pick ratio is raging at about the same level for all configurations. The cause is most likely the low number of generated tests for this program.

**Range Results**

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>13.08</td>
<td>50.11</td>
<td>46.51</td>
<td>6.28</td>
<td>82.8</td>
</tr>
<tr>
<td>c2</td>
<td>33.52</td>
<td>50.00</td>
<td>54.47</td>
<td>10.27</td>
<td>13.4</td>
</tr>
<tr>
<td>c3</td>
<td>39.10</td>
<td>79.91</td>
<td>89.17</td>
<td>17.00</td>
<td>33.4</td>
</tr>
<tr>
<td>c4</td>
<td>46.60</td>
<td>50.07</td>
<td>55.75</td>
<td>8.24</td>
<td>29.2</td>
</tr>
<tr>
<td>c5</td>
<td>32.72</td>
<td>49.83</td>
<td>55.27</td>
<td>10.40</td>
<td>48.0</td>
</tr>
<tr>
<td>c6</td>
<td>27.66</td>
<td>30.03</td>
<td>29.26</td>
<td>6.94</td>
<td>39.0</td>
</tr>
<tr>
<td>c7</td>
<td>32.12</td>
<td>49.84</td>
<td>56.54</td>
<td>10.36</td>
<td>49.2</td>
</tr>
</tbody>
</table>

Table 4.5: Results for program Range | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

For the range program, the Table 4.5 kill ratio ranges from 13.08% up to 46.60%. PT mean and PM mean are almost the same, meaning there are enough tests and mutants to reflect the MPLM. Round time is especially long for this program, and the pattern remains. The attacker wins only in the first configuration, which is interesting when comparing it to the previous programs. We are aiming to reach the value of about 50%, and c5 is very close to it.
The exploration (fig. 4.75 and 4.76) for the mutants and tests manifest same behaviour as in "Font Info". C1 takes some time before getting close to the rest, although they are all quickly explored for the first two-thirds of tests, while for the mutants only c4 lags.

More Figures for this program can be found in appendix B for those particularly interested.
Hierarchy Property Parser Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>11.81</td>
<td>49.78</td>
<td>42.08</td>
<td>3.36</td>
<td>85.6</td>
</tr>
<tr>
<td>c2</td>
<td>24.52</td>
<td>50.01</td>
<td>58.22</td>
<td>4.56</td>
<td>24.6</td>
</tr>
<tr>
<td>c3</td>
<td>27.55</td>
<td>80.12</td>
<td>88.09</td>
<td>6.31</td>
<td>60.8</td>
</tr>
<tr>
<td>c4</td>
<td>33.10</td>
<td>50.78</td>
<td>50.70</td>
<td>3.68</td>
<td>44.0</td>
</tr>
<tr>
<td>c5</td>
<td>23.76</td>
<td>49.96</td>
<td>58.09</td>
<td>4.60</td>
<td>64.2</td>
</tr>
<tr>
<td>c6</td>
<td>23.19</td>
<td>29.96</td>
<td>30.24</td>
<td>3.52</td>
<td>42.0</td>
</tr>
<tr>
<td>c7</td>
<td>24.24</td>
<td>50.03</td>
<td>57.22</td>
<td>4.62</td>
<td>65.4</td>
</tr>
</tbody>
</table>

Table 4.6: Results for Program Hierarchy Property Parser Result | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

For Hierarchy Property Parser, in 4.6 kill ratio ranges from 11.81% up to 33.10%, very similar to "Font Info". Nothing new for PT mean and PM mean. Round time looks fine. Four wins for the attacker and three for the defender.

Figure 4.77: Mutants Exploration Rate

Figure 4.78: Tests Exploration Rate
In Figure 4.78 for the tests nothing extraordinary happens, except slower time in reaching the top. Figure 4.77 showing exploration rate for the mutants is a bit different this time, as almost 250 mutants were generated for this program. From the figure, we can infer that the more mutants, the harder it will be to reach the explore all of them, although the explorations are still very fast for the first 80% mutants. The proportions of choosing a subset size need to be adjusted by the percentage of the entire set to maintain an efficient exploration.

More Figures for this program can be found in appendix B for those particularly interested.

The Decision for Final Base Configuration

After a thorough analysis, the best universal configuration needs to be chosen for further experiments. Exploration is essential, and to keep it diverse and fast, we base the sizes of the subsets on the complete set. The three times bigger or lower disproportions do not seem to have any useful effect. From the observations, the right choice seems to be to set subset sizes at 10% of the whole set for mutants, and 20-30% for the tests. The different sizes of the sets cause this choice. While 10% of 250 mutants equals 25 mutants, from 60 tests, 30% yields 18 tests. Therefore we need to balance the disproportions a little. The rounds are a subject to be increased for better accuracy, but this is a question of time resources. Too high MPLM is giving very high pick ratio, which is terrible, as more iterations are needed to show visible results; also, it consumes more round time. The low MPLM of 25% is producing rather promising results, and a value between 20-50% seems like an optimal choice, between too low and too high. WT should match the kill ratio in order to maintain fairness for both attacker and defender. Only the program “Inflection” has low kill ratio values, while the other range between 15% and 30%. It is difficult to pinpoint the exact cause of it, but if found, then it could be used to derive a percentage from it. For the experimenting programs, the prospect of 20%
being good is promising.

Therefore the final configurations for the next experiments are the following:

- Rounds: 500
- Mutants Subset Size (MSS): 10% of all mutants
- Tests Subset Size (TSS): 25% of all tests
- Model Pick Limit Multiplier (MPLM): 30%
- Winning Threshold (WT): 25%

4.2.3 In-Depth Experiments

This subsection provides us with more comprehensive experiments based on previously discovered most promising configuration. These configurations are summarized in Table 4.7. The only difference is that MSS and TSS are now percentages of the respective total set value. Another change is random mode and three additional bandit algorithms.
Table 4.7: Final Parameters for Different Configurations

<table>
<thead>
<tr>
<th>C</th>
<th>R</th>
<th>MSS</th>
<th>TSS</th>
<th>MPLM</th>
<th>WT</th>
<th>AM</th>
<th>DM</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>ActiveExplorer</td>
</tr>
<tr>
<td>c2</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>random</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c3</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c4</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>random</td>
<td>random</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c5</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>AdaptiveGreedy</td>
</tr>
<tr>
<td>c6</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>random</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c7</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>SoftmaxExplorer</td>
</tr>
</tbody>
</table>

We continued with the same programs, and below we present the results of these new experiments in a similar way as previously. Regarding the charts, the focus shifts to the prominent facets in a new selective approach, as a result of already presented rules and shape.
Font Info Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>46.93</td>
<td>27.56</td>
<td>55.44</td>
<td>3.78</td>
<td>18.0</td>
</tr>
<tr>
<td>c2</td>
<td>28.19</td>
<td>27.14</td>
<td>51.25</td>
<td>3.55</td>
<td>40.0</td>
</tr>
<tr>
<td>c3</td>
<td>23.79</td>
<td>27.41</td>
<td>51.16</td>
<td>3.89</td>
<td>44.2</td>
</tr>
<tr>
<td>c4</td>
<td>25.92</td>
<td>27.29</td>
<td>50.17</td>
<td>3.29</td>
<td>41.0</td>
</tr>
<tr>
<td>c5</td>
<td>25.39</td>
<td>27.26</td>
<td>49.11</td>
<td>3.83</td>
<td>42.4</td>
</tr>
<tr>
<td>c6</td>
<td>25.92</td>
<td>27.18</td>
<td>50.30</td>
<td>3.58</td>
<td>39.4</td>
</tr>
<tr>
<td>c7</td>
<td>28.06</td>
<td>27.35</td>
<td>50.14</td>
<td>3.84</td>
<td>36.6</td>
</tr>
</tbody>
</table>

Table 4.8: Final Results for Program Font Info | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

On the first sight, the results from Table 4.8 look satisfying. Kill Ratio is a bit off only for c1, which is the configuration with Active Explorer (sec. 3.2.6) as bandit algorithm, and it affects the wins. Attacker oscillates at about 40% of winning, and the round time is rather low for all.

Figure 4.79: Mutants Exploration Rate

Figure 4.80: Tests Exploration Rate
Figures 4.79 and 4.80 display effective exploration, especially for the mutants. Though it is not that bad exploration for the tests, we could increase the percentage for the tests’ subset sizes, to more likely reach the top before round 100.

The distribution of kill ratio is most intense at around 33%, and least at 100%, with c1 deviating a little bit from the rest, see Figures 4.81 and 4.82.

Comparing the pick ratio with the score value shows a correlation for tests with a high score value, and very low, while the rest is ranging between zero and about 90%. The mutants are ranging around 10 to 50%. An odd behaviour can be observed on configuration one, that is in opposition to the rest, it shows low scores in an area, and this correlates with the pick ratio, see 4.83 to 4.86.
Figure 4.83: c1 Mutants Pick Ratio

Figure 4.84: c1 Mutants Score

Figure 4.85: c2 Mutants Pick Ratio

Figure 4.86: c2 Mutants Score
## Inflection Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>12.46</td>
<td>37.51</td>
<td>96.84</td>
<td>3.66</td>
<td>70.4</td>
</tr>
<tr>
<td>c2</td>
<td>9.06</td>
<td>37.46</td>
<td>98.78</td>
<td>3.37</td>
<td>76.2</td>
</tr>
<tr>
<td>c3</td>
<td>9.06</td>
<td>37.43</td>
<td>99.35</td>
<td>3.72</td>
<td>77.0</td>
</tr>
<tr>
<td>c4</td>
<td>8.39</td>
<td>37.90</td>
<td>99.27</td>
<td>3.14</td>
<td>76.8</td>
</tr>
<tr>
<td>c5</td>
<td>8.93</td>
<td>37.54</td>
<td>99.16</td>
<td>3.67</td>
<td>76.2</td>
</tr>
<tr>
<td>c6</td>
<td>9.19</td>
<td>37.37</td>
<td>99.34</td>
<td>3.47</td>
<td>74.8</td>
</tr>
<tr>
<td>c7</td>
<td>8.99</td>
<td>37.50</td>
<td>99.60</td>
<td>3.67</td>
<td>76.6</td>
</tr>
</tbody>
</table>

Table 4.9: Final Results for Program Inflection | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

Results for the Inflection program in Table 4.9 this time are very extreme. A too-small program is probably the most reasonable reason for that. Kill ratio oscillates around 10%, which is over twice as much as the set threshold. Making the rest of the results less reliable. Figures 4.87 and 4.88 we confirm this by showing almost instant exploration for the tests and the low amount of them.
Again, the first configuration shows exceptional behaviour; its last mutants are getting more chance for selection, and this is the opposite of what the score is telling. See fig. 4.89 and 4.90. The rest looks rather even, and the tests are close to 100%.
Range Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>71.34</td>
<td>23.17</td>
<td>48.33</td>
<td>11.59</td>
<td>3.0</td>
</tr>
<tr>
<td>c2</td>
<td>43.06</td>
<td>23.09</td>
<td>42.65</td>
<td>9.89</td>
<td>19.6</td>
</tr>
<tr>
<td>c3</td>
<td>45.06</td>
<td>23.16</td>
<td>48.28</td>
<td>10.88</td>
<td>17.2</td>
</tr>
<tr>
<td>c4</td>
<td>41.79</td>
<td>23.12</td>
<td>41.82</td>
<td>9.56</td>
<td>17.6</td>
</tr>
<tr>
<td>c5</td>
<td>43.66</td>
<td>23.09</td>
<td>52.17</td>
<td>11.09</td>
<td>16.4</td>
</tr>
<tr>
<td>c6</td>
<td>41.46</td>
<td>23.07</td>
<td>49.54</td>
<td>10.66</td>
<td>21.6</td>
</tr>
<tr>
<td>c7</td>
<td>40.13</td>
<td>23.10</td>
<td>49.41</td>
<td>11.00</td>
<td>20.6</td>
</tr>
</tbody>
</table>

Table 4.10: Final Results for Program Range | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

For Range, the kill ratio oscillates around 40-45, except the c1 with active explorer (sec. 3.2.6) algorithm at 71.34. Attacker’s wins are low below 22%, and the round time is unusually high around 10-11 seconds. (4.10)

Explorations are satisfying without any abnormalities.

Figure 4.91: Mutants Exploration Rate
Figure 4.92: Tests Exploration Rate
Once again, the c1 is showing an exploitative manner, whereas the others are difficult to differentiate; the contrast can be seen in Figures 4.93 through 4.96.

Figure 4.93: c1 Mutants Pick Ratio

Figure 4.94: c7 Mutants Pick Ratio

Figure 4.95: c1 Tests Pick Ratio

Figure 4.96: c7 Tests Pick Ratio
## Hierarchy Property Parser Results

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>46.33</td>
<td>11.96</td>
<td>38.40</td>
<td>4.67</td>
<td>15.4</td>
</tr>
<tr>
<td>c2</td>
<td>25.79</td>
<td>12.01</td>
<td>31.42</td>
<td>3.87</td>
<td>40.6</td>
</tr>
<tr>
<td>c3</td>
<td>25.26</td>
<td>11.97</td>
<td>34.86</td>
<td>4.21</td>
<td>43.2</td>
</tr>
<tr>
<td>c4</td>
<td>26.26</td>
<td>11.97</td>
<td>30.46</td>
<td>3.64</td>
<td>39.8</td>
</tr>
<tr>
<td>c5</td>
<td>25.86</td>
<td>12.08</td>
<td>34.19</td>
<td>4.23</td>
<td>38.8</td>
</tr>
<tr>
<td>c6</td>
<td>26.92</td>
<td>12.05</td>
<td>36.88</td>
<td>4.10</td>
<td>39.6</td>
</tr>
<tr>
<td>c7</td>
<td>25.39</td>
<td>12.07</td>
<td>33.43</td>
<td>4.28</td>
<td>42.4</td>
</tr>
</tbody>
</table>

Table 4.11: Final Results for Program Hierarchy Property Parser | Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

All except c1 have around 26% kill ratio. Because of the high amount of mutants, PM Mean is at a 12% low. Round time looks normal, and the wins are of 40% except c1 being at 15.4%. (4.11)

Explorations are satisfying without any abnormalities.

![Figure 4.97: Mutants Exploration Rate](image1)

![Figure 4.98: Tests Exploration Rate](image2)
All the distribution charts are looking almost identical, yet c1 again yields abnormal outcome for the mutants. See Figure 4.99 where this is depicted, and 4.100 for score comparison.

![Figure 4.99: c1 Mutants Pick Ratio](image1)

![Figure 4.100: c1 Mutants Score](image2)

**Experiments on Loaded Bandits**

We run two additional experiments, where the bandits we loaded from one program to two others. The intention back it is to check in what degree 500 iterations of a learned bandit from another program impacts the results. For this case, we used Hierarchy Property Parser as a trained bandit and loaded it on the programs Font Info and Range.

<table>
<thead>
<tr>
<th>Config</th>
<th>Rounds</th>
<th>MSS</th>
<th>TSS</th>
<th>MPLM</th>
<th>WT</th>
<th>AM</th>
<th>DM</th>
<th>BA</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1 Fresh</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
<tr>
<td>c2 Loaded</td>
<td>500</td>
<td>10</td>
<td>25</td>
<td>30.0</td>
<td>25.0</td>
<td>scikit</td>
<td>scikit</td>
<td>EpsilonGreedy</td>
</tr>
</tbody>
</table>

Table 4.12: Loaded and Fresh Bandits Parameters for Different Configurations | C: Configuration, R: Rounds or iterations, MMS: Mutants subset size of total set in %, TSS: Tests subset size of total suite in %, MPLM: Model pick limit multiplier in %, WT: Winning threshold in %, AT: Attacker Mode, DM: Defender Mode, BA: Bandit Algorithm

79
Table 4.12 depicts the configurations information used for these experiments. Fresh, means normally starting, while loaded, means using bandit from the Hierarchy Property Parser.

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>23.79</td>
<td>27.41</td>
<td>51.16</td>
<td>3.89</td>
<td>44.2</td>
</tr>
<tr>
<td>c2</td>
<td>25.73</td>
<td>27.16</td>
<td>47.71</td>
<td>3.93</td>
<td>43.4</td>
</tr>
</tbody>
</table>

Table 4.13: Loaded and Fresh Bandits Results for Program Font Info

Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

<table>
<thead>
<tr>
<th>Config</th>
<th>K Ratio</th>
<th>PM Mean</th>
<th>PT Mean</th>
<th>R Time</th>
<th>Wins</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>45.06</td>
<td>23.16</td>
<td>48.28</td>
<td>10.88</td>
<td>17.2</td>
</tr>
<tr>
<td>c2</td>
<td>41.33</td>
<td>23.06</td>
<td>53.30</td>
<td>11.74</td>
<td>18.4</td>
</tr>
</tbody>
</table>

Table 4.14: Loaded and Fresh Bandits Results for program Range

Config: Configuration, K Ratio: avg. kill ratio in %, PM Mean: mean of mutants picked from a subset in %, PT Mean: mean of tests picked from a subset in %, R Time: avg. time per round in seconds, Wins: wins for attacker in %

From the standard information in Tables 4.13 and 4.14 we do not see any particular discrepancies. Exploration as well is not afflicted by this change.

The best way of comparing these experiments is to look at the distribution of mutants and tests concerning the pick ratio. The ones that better match to the score is supposed to be superior. Below are the charts illustrating it.
Figure 4.101: Font Info c1 Mutants
Pick Ratio

Figure 4.102: Font Info c2 Mutants
Pick Ratio

Figure 4.103: Font Info c1 Mutants
Score

Figure 4.104: Font Info c2 Mutants
Score
It is difficult to see at first sight, but after a more prolonged examination of the charts, we can see a slight improvement. C2 are matching its score in a little higher degree, especially for the tests. However this is still too little to be sure if it is not just a luck.
Figure 4.109: Range c1 Mutants Pick Ratio

Figure 4.110: Range c2 Mutants Pick Ratio

Figure 4.111: Range c1 Mutants Score

Figure 4.112: Range c2 Mutants Score
Again for Range, the results are the same, a slightly better selection, yet still too little for reliable results.

4.3 Discussion

First, we tried to find a universal configuration. With our initial parameters, there were some discrepancies in the exploration aspect, which lead to the conclusion that it can be improved. The goal was to match the kill ratio to the winning threshold value to produce a reliable outcome.
The charts illustrating pick ratio by bandits showed a low matching factor when compared with the score achieved. Independent of the configuration, we saw the same pattern in the scores, but not the pick ratio. The tests were yielding a bit better results, which means that most likely, they did have better features implemented. The average round time varied upon the used program.

Second, the comparison between epsilon greedy (Sec. 3.2.6) algorithm and the random variant was conducted to supply us with a measurement of the efficacy. This time using the best configuration based on initial experiments had to secure a reliable outcome. We managed to find the right way of adjusting it to achieve a effective and efficient exploration. For this purpose, however, we used a percentage based on calculated subset sizes. Perhaps finding an efficient way to do the same with other parameters would lead to further improvement in reliability. The round time has not changed to a significant extent for each program. The biggest surprise under these experiments was the discrepancy when the bandit algorithm was an active explorer (Sec. 3.2.6). For the mutants, it looked that it is the only one capable of learning, but reflecting the opposite. That means for some reason it picked more often the worse mutants and least the best. If we mirror the chart, we can get a quite accurate comparison with the score values. On all the remaining ones, it was difficult to see any noticeable differences.

Additionally, we run two new experiments on the saved bandit from one of the programs on two others. We wanted to see if a learned bandit on a different program would lead to improved results. The only noticeable difference between a fresh and loaded bandit could be seen on the pick ratio graphs. Not much, but slightly more accurate connection with the score chart could be observed on the loaded bandit run for both tests and mutants.

There is a strong dependency on mutants’ and tests’ features when we compare the results. Tests got higher matching factor most likely because
of the dynamic features indicating previously killed mutants and selections by the agent, whereas mutants only have static features taken from the program context. Dynamic versus static features is a tough decision, as we need our agents to identify the unique units based on their characteristics. Dynamic features seem to improve accuracy but may be misleading in the long term. Therefore it is a vital aspect to take into account. Expert knowledge in this field or thorough experimenting is necessary to achieve reliable and practical use.

Testing out mutants with dynamic features as well would be an attractive approach to see how it compares to the tests. While it is complex to calculate additional static features without static program analysis, there are some other dynamic features we could use; for example, we could provide the information of how many times a mutant survived or was killed by the tests. It would not only give us information about the defeats, but also the victories. Features like chosen times in the subset or if killed last time do not make much sense, as they are unrelated to the bandits’ manner. Randomness in the feature values must be avoided to escape misinterpretation and because the contextual bandits rely on structure and meaning within the feature vector. Splitting relevant from irrelevant features, and then picking the best ones either by experts or experimenting is the right strategy.

Finally, this leads us to the conclusion that the project still needs work to produce stronger results. Some of the information was very useful and accurate like round time, exploration graphs, or kill ratio. However, the selection aspect was the most valuable for this project, and it shows that it needs more time to be spent on this. The selective distribution of the agents did not match the actual score in a degree to yield a comprehensive, reliable bandit selection. Nevertheless, unlike for the mutants, for the tests, we were able to point on the most prominent tests on every adequate chart. Our method and the results from the experiments form a basis for efficient mutant and test selection, but it will require future work
to strengthen the approach and make it applicable for practical deployment. Nonetheless, we did test out the parameters comprehensively and presented proper ways of evaluating the experiments, which were able to pinpoint many various aspects.
Chapter 5

Future Work

5.1 Alternative Ideas

Different ideas have come while brainstorming for how to create this game. As mentioned previously, we focus on selection, but this is only one of many possible approaches with the work of improving mutation testing.

Generation is another exciting facet which might benefit from using machine learning. For example, we could look for interesting aspects of mutants that get good results and use that information in order to produce a bunch of strong mutants. Depending on the amount of training data and way to do it, we could eventually end up with an excellent generator for mutants or even tests.

Detecting qualities of different mutants or tests may give us valuable information. This information could be used to categorize these qualities, which may lead to high probable detection of bad and good parts we want to keep or get rid of in selecting. Generation could be designed to use that database of these categories and make products with or without some preselected qualities based on chosen parameters.
5.2 Potential Continuation Paths

There is an undeniable room for improvement and additions to this project. Three areas are particularly salient, the number of iterations, features and algorithms. Where the results failed to accomplish the goal, most is the distribution of pick ratio. We wish to end up with a bandit that is capable of being loaded on any program, and quickly produce a reliable distribution in the strength of given tests and mutants. However, this can be complicated and requires more researching; creating such a bandit on a fresh program is a priority though.

One possible continuation is to perfect the selection of an algorithm with the given features. The features should be more studied and tested with the configurations. As we saw, the configurations are not perfect for each program, these need more polishing as well, though the exploration achieved excellent results in this thesis. Even so, while three algorithms behaved in a similar way, one output something completely different. Therefore it is a crucial research facet.

We did use 500 iterations as to the time limitations and computation power. If increasing iterations significantly, there is a high chance of gaining different results. Moreover, the number of programs can also be increased to gain a broader scope, yet this can increase the cost of experimenting significantly.

Another way to continue this work is to change the algorithm from bandits to another form of reinforcement learning. Adding more ways to confirm the results or introduce additional parameters which might be necessary, but were not discovered.

Thereby the continuation can take more than one path. It is more likely to require significant resources in the form of people, computation power and time.
Chapter 6

Conclusion

In the end, only a specific part of the results meet our expectations. The idea, its methods and structure form a good foundation for possible future work.

The main challenges are to make it all compatible with experimenting with the intended approach and goals. For instance, it is not easy to find satisfying and working libraries or programs to be compatible with our program. Parameters, features and algorithms are the cores of the challenges, and while we manage to answer these problems to some degree, there is still much to research on it; especially the features and algorithms, as we only grasp the tip of an iceberg, because of the time constraint.

The experiments are carried out on four different programs of sizes where the time to do these experiments does not extend over the thesis’s limit. We can experiment with a decent chunk of possibilities; however, it shows itself to not be enough in order to achieve all of our goals. The goals we achieve, however, are to make the bandit discover the best couple of tests with high reliability and attaining an excellent solution for fast exploration. We present an evaluation using a broad scope of parameters creating a good overview of vital aspects. Our results reveal a not fully mature method that requires more necessary work in order to deploy it for practical use. It is still a solid piece of research being in an early state.
with a promising future.

The thesis ends with an outline for future work, that yields a good portion of information pinpointing at the problems needed to be tackled.
References


Appendix A

GitHub Repository

Inserting the code here would take too many pages, and including only the main could leave a wrong impression; therefore, below we put a link to the GitHub repository containing the essential files, excluding installation files like, e.g. Java 7 and 8. Randomly generated files are not included, as they are dynamic depending on the state and program, although this function can be inferred by scanning the code.

Mutant Defender Program for Executing Selection of Tests and Mutants Experiments using Contextual Bandits:

https://github.com/KucykPatryk/Master-Thesis-Mutant-Defender-Project
Appendix B

Extra Figures for the Configuration Experiments

B.1 Range

Figure B.1: c1 Kill Ratio

Figure B.2: c2 Kill Ratio
Figure B.3: c3 Kill Ratio

Figure B.4: c4 Kill Ratio

Figure B.5: c5 Kill Ratio

Figure B.6: c6 Kill Ratio

Figure B.7: c7 Kill Ratio
Figure B.14: c4 Mutants Pick Ratio
Figure B.15: c4 Mutants Score

Figure B.16: c5 Mutants Pick Ratio
Figure B.17: c5 Mutants Score

Figure B.18: c6 Mutants Pick Ratio
Figure B.19: c6 Mutants Score
Figure B.26: c3 Tests Pick Ratio

Figure B.27: c3 Tests Score

Figure B.28: c4 Tests Pick Ratio

Figure B.29: c4 Tests Score

Figure B.30: c5 Tests Pick Ratio

Figure B.31: c5 Tests Score
Figure B.32: c6 Tests Pick Ratio

Figure B.33: c6 Tests Score

Figure B.34: c7 Tests Pick Ratio

Figure B.35: c7 Tests Score
B.2 Hierarchy Property Parser

Figure B.36: c1 Kill Ratio

Figure B.37: c2 Kill Ratio

Figure B.38: c3 Kill Ratio

Figure B.39: c4 Kill Ratio
Figure B.40: c5 Kill Ratio

Figure B.41: c6 Kill Ratio

Figure B.42: c7 Kill Ratio

Figure B.43: c1 Mutants Pick Ratio

Figure B.44: c1 Mutants Score
Figure B.45: c2 Mutants Pick Ratio

Figure B.46: c2 Mutants Score

Figure B.47: c3 Mutants Pick Ratio

Figure B.48: c3 Mutants Score

Figure B.49: c4 Mutants Pick Ratio

Figure B.50: c4 Mutants Score
Figure B.69: c7 Tests Pick Ratio

Figure B.70: c7 Tests Score