UNIVERSITY OF SHEFFIELD

Improvements to Test Case Prioritisation considering Efficiency and Effectiveness on Real Faults

by

David Paterson

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the
Faculty of Engineering
Department of Computer Science

March 2020

Abstract

Despite the best efforts of programmers and component manufacturers, software does not always work perfectly. In order to guard against this, developers write test suites that execute parts of the code and compare the expected result with the actual result. Over time, test suites become expensive to run for every change, which has led to optimisation techniques such as test case prioritisation. Test case prioritisation reorders test cases within the test suite with the goal of revealing faults as soon as possible.

Test case prioritisation has received a lot of research that has indicated that prioritised test suites can reveal faults faster, but due to a lack of real fault repositories available for research, prior evaluations have often been conducted on artificial faults. This thesis aims to investigate whether the use of artificial faults represents a threat to the validity of previous studies, and proposes new strategies for test case prioritisation that increase the effectiveness of test case prioritisation on real faults.

This thesis conducts an empirical evaluation of existing test case prioritisation strategies on real and artificial faults, which establishes that artificial faults provide unreliable results for real faults. The study found that there are four occasions on which a strategy for test case prioritisation would be considered no better than the baseline when using one fault type, but would be considered a significant improvement over the baseline when using the other. Moreover, this evaluation reveals that existing test case prioritisation strategies perform poorly on real faults, with no strategies significantly outperforming the baseline.

Given the need to improve test case prioritisation strategies for real faults, this thesis proceeds to consider other techniques that have been shown to be effective on real faults. One such technique is defect prediction, a technique that provides estimates that a class contains a fault. This thesis proposes a test case prioritisation strategy, called G-Clef, that leverages defect prediction estimates to reorder test suites. While the evaluation of G-Clef indicates that it outperforms existing test case prioritisation strategies, the average predicted location of a faulty class is 13% of all classes in a system, which shows potential for improvement. Finally, this thesis conducts an investigative study as to whether sentiments expressed in commit messages could be used to improve the defect prediction element of G-Clef.

Throughout the course of this PhD, I have created a tool called Kanonizo, an open-source tool for performing test case prioritisation on Java programs. All of the experiments and strategies used in this thesis were implemented into Kanonizo.

Publications

Through the course of this thesis, the following pieces of research have been published at peer-reviewed venues:

[1]	David Paterson, Gregory M. Kapfhammer, Gordon Fraser and Phil McMinn — “Using Controlled Numbers of Real Faults and Mutants to Empirically Evaluate Coverage-Based Test Case Prioritization”. In *Internation Workshop on Automated Software Test (AST 2018)*
[2]	David Paterson, José Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser and Phil McMinn — “An Empirical Study on the Use of Defect Prediction for Test Case Prioritization”. In *International Conference on Software Testing, Verification and Validation (ICST 2019)*

Acknowledgements

Firstly, I would like to express the most sincere graditude to my supervisor, Prof. Phil McMinn. During the course of my studies, Phil has constantly supported, motivated and pushed me, as well as providing invaluable guidance and knowledge in both my research and my writing. I would also like to thank Dr. Gregory Kapfhammer, who has sacrificed a great deal of his own time in order to contribute to projects, yet never wavered in his enthusiasm. Quite simply, it would not have been possible to complete the achievements of this thesis without the help of Phil and Greg. I would also like to thank Prof. Gordon Fraser and Prof. Rui Abreu, who have provided valuable insight and experience when collaborating on projects.

Secondly, I would like to thank the members of the Verification and Testing groups at the University of Sheffield — while there are too many to mention individually, special thanks must go to Thomas White, José Campos and Abdullah Alsharif, who have always been happy to discuss ideas and provide support, regardless of the hour of the day.

Finally, a special mention must go to my family and my partner Laura, who do not have the faintest idea what I do, but have supported me unconditionally and remained patient throughout my studies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer software has become a major part of global industry, with an estimated \$439 billion being spent on enterprise software in 2019 and a steady year-on-year increase of around 9% [3]. Almost all businesses, whether they have 10 employees or 100,000 employees, can make use of computer software to make their jobs easier. Using software instead of paper-based alternatives is often quicker, easier, more reliable and more scalable.

The global demand for computer software has had a knock-on effect in the software development industry. In the United Kingdom alone, there are an estimated 338,000 programmers and software development professionals employed as of 2018, compared with 224,000 in 2011 [4].

Unfortunately, in practice, software does not always perform perfectly. This may be as a result of incorrect logic used in the code (i.e. a "defect"), a component malfunction (i.e. a "failure") or the result of a human using the software incorrectly (i.e. an "error"). While bugs are inevitable, companies aim to minimise the number of bugs that exist in software. In particular, software bugs can be remembered because of their financial cost (e.g. the Knight banking group \$440m error [5]), because of the disaster involved (e.g. the Ariane 5 Flight 501 [6]) or because they are simply funny (e.g. the Windows 98 crash during demonstration [7]). One of the most common ways to guard against bugs is to write automated *test cases*. Test cases should execute a small part of the program as a user would do, and ensure that the expected behaviour occurs. Software testers have to attempt to envisage every possible way in which the software will be used in order to cover every use case to ensure that there are no scenarios in which buggy behaviour occurs.

1.1 Regressions

Regressions are a specific type of software bug that occur when a change results in an *unintended side-effect* somewhere else in the code [8]. Consider a small example that involves a method `divide(int x, int y)` which returns the value of $\frac{x}{y}$, and a second method `invert(int x)`, which returns the value of $\frac{1}{x}$, and internally calls `divide(1,x)`. The `invert` method may assume that `divide` returns an integer value regardless of the input passed in, but a developer makes a change that states `if(y==0) return null`. This change in itself may not change any observable behaviour for the `divide` method, but may result in a *regression* when trying to use `invert` with the argument 0.

The existence of regressions causes a big problem in software testing. Since programmers cannot assume that the development process is linear, and that new bugs can appear in old parts of the code, it becomes necessary to re-execute all existing test cases whenever a change is made, in order to provide confidence that previous issues have not resurfaced. Whenever a regression is found, a developer will have to fix the code and add a new *regression test case* to show that the fix has dealt with the regression. This test case will then remain a part of the test suite indefinitely, such that if the regression re-occurs, the test case should flag it to a developer.

In the above example, a test would be added to the test suite that executes `invert(0)` and ensures that the correct behaviour is observed. This means that, in future, even if the behaviour of `divide` is changed, the `invert` method is still shown to be working correctly in this scenario.

Over time, regression test suites grow as software grows, for example in the case of Apache Geode, where the release build can run in excess of 18 hours [9]. In order to combat the growing cost of testing for regressions, a family of test optimisation techniques have been proposed in the literature [10]. The first technique is test case selection, which aims to identify a subset of relevant test cases to execute based on changed code or code that uses the changed code. The second technique is test suite minimisation, which aims to identify redundancies in the test suite, finding test cases that are very similar and removing them from the test suite, while aiming to retain all the fault-finding capability of the test suite. The final technique is test case prioritisation, which aims to reorder test suites in order to reveal faults as soon as possible, with the most “fault-revealing” test cases placed at the start of the test suite.

1.2 Limitations of the State of the Art for Test Case Prioritisation

Given the excessive time that it can take to test for regressions, a successful test case prioritisation strategy could be very beneficial. Previous research has frequently shown that re-ordering test suites can lead to savings either in the number of test cases (e.g. [10–12]) or in the time taken to detect faults (e.g. [13–15]).

Due to the difficulty in accumulating a large repository of real faults, researchers have been forced to use artificial faults when investigating test case prioritisation strategies [10, 11]. There are two main types of artificial fault: seeded faults are introduced manually by someone who has a reasonable understanding of how the program should work, while mutants are faults generated procedurally according to a ruleset (e.g. changing a “+” to a “-”). While seeded faults offer a more realistic experience, they are still hard to obtain in large numbers, resulting in a standard set being used for a large number of previous studies. Conversely, it is easy to generate a large number of mutant faults for any program using a tool such as Major [16]. However, these faults are very often simple, sometimes resulting in no change in behaviour to the program under test, which is referred to as an “equivalent mutant” [17].

While it is perfectly possible that strategies proposed in previous studies will translate perfectly well in practice, there is a clear gap in the knowledge about how real faults compare with artificial faults in evaluations of test case prioritisation strategies. If test case prioritisation strategies perform significantly better when reordering test suites for artificial faults than for real faults, then this thesis must also try to devise a new strategy that will be more effective for real faults.

Furthermore, it is common for businesses to utilise version control systems (VCS) and continuous integration (CI) as part of their development process. CI servers are constantly running full “builds” of a program as new changes are made, specifically ensuring that the new version compiles and passes all existing test cases. As a result, developers very rarely trigger the test suite to run, instead relying on a CI server to notify them if one of their changes has resulted in a test case failure.

As a result of this practice, it is also necessary for any test case prioritisation strategy to integrate with these systems so that there is no requirement for developers to manually execute test case prioritisation. For example, the Maven Surefire plugin[1] runs the test suite as part of the build and allows the specification of the order, according to a limited number of pre-set orderings (e.g. alphabetical), in which to run the test cases. This

[1] https://maven.apache.org/surefire/maven-surefire-plugin

particular function could be extended to use the result of test case prioritisation as the preferred order of running test cases.

1.3 Aims of this Thesis

The primary aim of this thesis is to develop test case prioritisation strategies that are more effective in practice. This thesis aims to empirically evaluate the differences between real and artificial faults when prioritising test cases, before proposing and evaluating new test case prioritisation strategies in the current state of the art to assess their effectiveness.

1. To empirically evaluate the comparative effectiveness of test case prioritisation strategies on real and artificial faults.
   While there have been a number of previous studies in test case prioritisation that have shown the technique to be effective, they have been forced to use artificial faults due to the difficulty associated with obtaining large numbers of real faults. This raises questions about the validity of the results, since it is not necessarily guaranteed that real faults behave in the same way as artificial faults. Chapter 3 of this thesis proposes an empirical evaluation comparing the effectiveness of existing test case prioritisation strategies on real and artificial faults.

2. To develop new strategies for test case prioritisation that will be more effective at prioritising test suites for real faults.
   Since Chapter 3 shows that test case prioritisation strategies are relatively ineffective for real faults, one of the key issues that must be addressed is improving test case prioritisation strategies for real faults. One technique that has been shown to be effective on real faults is defect prediction [18], which estimates the likelihood that files in a program are faulty. Chapter 4 proposes and evaluates a test case prioritisation strategy that uses defect prediction to reorder test cases. Finally, Chapter 5 investigates possible improvements to the defect prediction used in Chapter 4.

1.4 Organisation and Scientific Contributions of this Thesis

Given the prevalent problems faced in test case prioritisation, particularly with regards to the use of real faults in empirical evaluations, this thesis focuses on improvements to test case prioritisation, specifically how to increase the practical effectiveness of the technique. This thesis begins with a comprehensive literature survey in Chapter 2, including discussions about previous test case prioritisation strategies and where they have been successful. Where available, Chapter 2 discusses the use of real faults in previous empirical studies. Since Chapter 2 identifies a lack of research comparing the effectiveness of test case prioritisation strategies on real and artificial faults, Chapter 3 presents a large scale empirical evaluation of eight existing test case prioritisation strategies on up to 262 real faults and mutants. This research reveals key insights into the use of real faults in empirical studies, and questions the reliability of experiments that do not use real faults. In particular, Chapter 3 finds that existing test case prioritisation strategies perform poorly on real faults, in many cases barely outperforming random orderings. As a result, Chapter 4 proposes and evaluates a new test case prioritisation strategy based on the technique of defect prediction. The results of this research show a promising connection between defect prediction and test case prioritisation, but also indicate that further improvements can be made. One of the potential improvements is investigated in Chapter 5, which investigates whether sentiment in commit messages can be used to indicate whether the commit is faulty or not, with the intention of devising a test case prioritisation strategy that would utilise commit messages to determine how to reorder the test suite. Finally, Chapter 6 recaps how the aims and objectives of this thesis have been met, and suggests a number of ideas for future work to investigate.

Chapter 2: "Literature Review" — This chapter presents a comprehensive review of the previous studies in test case prioritisation. This begins with an overview of automated software testing and an introduction to the problems faced when automatically testing for regressions, before giving the formal definition of test case prioritisation [10]. Following this, I introduce many of the previous test case prioritisation strategies. This chapter identifies the shortcomings of previous studies in terms of the type of fault used, motivating the work in Chapter 3, and identifies defect prediction as a candidate solution for improving test case prioritisation on real faults, which is investigated further in Chapter 4 and Chapter 5.

Chapter 3: "Using Controlled Numbers of Real Faults and Mutants to Empirically Evaluate Coverage-Based Test Case Prioritisation" — One of the key problems identified in Chapter 2 is that, at the time of writing Chapter 3, there was no indication as to whether strategies that were previously evaluated on artificial faults would also be effective on real faults[2]. This is largely due to a lack of repositories containing real faults, and no studies that have demonstrate how real faults compare with artificial faults when evaluating test case prioritisation strategies.

[2]Luo et al. [19] have since conducted an empirical evaluation on this topic, but the research presented in Chapter 3 was conducted and published at AST before Luo et al. had published their findings.

Chapter 3 presents an empirical evaluation of existing test case prioritisation strategies on real faults and mutants. Previous studies using artificial faults have indicated that test case prioritisation strategies can result in faults being found using fewer test cases. This experiment questions these results using 262 subject programs from the DEFECTS4J dataset [20], a large repository of subject programs that contain real faults. Furthermore, DEFECTS4J contains a version for each subject that does not contain the real fault (a "fixed" version), allowing the introduction of artificial faults. This experiment evaluates eight test case prioritisation strategies categorised as either "coverage-based" or "history-based". This results in the following contributions.

Contribution 3.1: *A comparison of how coverage-based test case prioritisation strategies perform on real faults and mutants*

Contribution 3.2: *A comparison of how history-based test case prioritisation strategies perform on real faults and mutants*

Moreover, the primary metric for evaluating the effectiveness of test case prioritisation strategies, called "Average Percentage of Faults Detected" ($APFD$) and introduced in Section 2.7, includes the number of faults as part of the equation. Many previous studies have assumed that changes to $APFD$ are independent of the number of faults that is present in a program, and thus have contained very different numbers of faults in subject programs, ranging from 1 [21] to 500 [22]. In order to investigate whether the number of faults is an independent factor in changes to $APFD$, this chapter conducts an evaluation on programs that contain 1, 5 and 10 faults respectively, observing the impact that increasing the number of faults has on $APFD$, resulting in the following contribution.

Contribution 3.3: *A comparison of how the number of faults present in a program affects the performance of test case prioritisation strategies*

Some of the key findings from this research are that (a) it is important to use *real faults* when evaluating test case prioritisation strategies to ensure validity of results and (b) the performance of existing test case prioritisation strategies on real faults is notably poor, with most strategies barely outperforming random orderings.

Chapter 4: "An Empirical Study on the Use of Defect Prediction for Test Case Prioritisation" — Given the findings of Chapter 3, it is clear that improvements are needed to test case prioritisation when evaluating on real faults. One technique that has been shown to be effective at predicting the location of real faults is defect prediction [18, 23–25]. Defect prediction aims to predict the likelihood for all files in a repository that the file will contain a fault, leveraging software metrics [25] or version control history [26] to produce a numeric score representing the likelihood of a fault occurring for each file in a program. In Chapter 4, I propose a test case prioritisation strategy that leverages defect prediction scores.

The first contribution of Chapter 4 is a parameter tuning experiment to discover how effective a particular defect prediction implementation, named Schwa [26], can be. Chapter 3 demonstrated the importance of using real faults in evaluations of test case prioritisation strategies. Therefore, for this chapter, I use DEFECTS4J [20], a repository containing 395 real faults, to provide subject programs for this study. The first contribution of this chapter is a parameter tuning study to discover the best configuration of Schwa for the DEFECTS4J subject programs.

Contribution 4.1: *A parameter tuning study to determine the best parameters for defect prediction to find real faults in* DEFECTS4J

Following this, I propose a test case prioritisation strategy, called G-Clef, that leverages defect prediction scores in order to prioritise test cases. Firstly, the strategy ranks all classes in the system by their defect prediction score. Then, for all classes, it identifies the set of test cases that cover the class, and uses a "secondary objective" (e.g. coverage) to order this set of test cases before placing them in a prioritised suite. G-Clef is implemented into KANONIZO (see Appendix A), resulting in the following contribution.

Contribution 4.2: *An implementation of a new test case prioritisation strategy, G-Clef, that leverages defect prediction*

There are two available parameters for G-Clef. The first is the secondary objective, discussed in Section 4.4.1. A secondary objective receives a set of test cases as input and returns an ordering. I implemented four secondary objectives, two of which use code coverage to reorder test cases, one that uses a constraint solver to attempt to maximise coverage, and one that simply returns a random ordering.

The second parameter in G-Clef is the number of classes to consider as a "group". If the defect prediction is not perfect, it may be the case that the class that contains the fault is ranked outside the top 5 classes. In this case, G-Clef can consider "grouping" classes together, so instead of considering the test cases that cover one class, G-Clef could consider 1% of all classes. This chapter therefore conducts a second parameter tuning experiment to see which combination of secondary objective and class grouping performs best, resulting in the following contribution.

Contribution 4.3: *A parameter tuning study to determine the best parameters for G-Clef*

Finally, this chapter contributes an empirical evaluation of G-Clef using the same subjects and strategies that were used in Chapter 3, which led to the following contributions.

Contribution 4.4: *An evaluation of G-Clef against existing coverage-based strategies*

Contribution 4.5: *An evaluation of G-Clef against existing history-based strategies*

This chapter reveals that defect prediction can be a more effective strategy for test case prioritisation than existing coverage and history-based strategies. In the experiments G-Clef performed significantly better than six out of the eight strategies it was compared against, and was never significantly outperformed by any strategy. However, this research also demonstrated that there was even higher potential for defect prediction to improve test case prioritisation. In particular, despite Schwa being successful at predicting the location of real faults, the average position of the faulty class was 13% of the total classes for each subject. This indicates that improvements to defect prediction could yield significant improvements to test case prioritisation.

Chapter 5: "Using Sentiment in Commit Messages to Predict Whether a Class is Faulty — An Investigative Study and Implications for Test Case Prioritisation" — One of the key findings of Chapter 4 is that there is potential to increase the effectiveness of test case prioritisation through improvements to defect prediction. Furthermore, as previously mentioned, there is a need for test case prioritisation to integrate with VCS and/or CI systems in order to be useful in practice. In particular, every VCS involves commit messages that are written by developers to describe every change that they make. While commit messages are intended to be short and objective, it is possible that developers may express feelings against particularly badly written files. In this way, commit messages may act as a surrogate for defect prediction, indicating which files are most likely to be faulty through opinionated commit messages. Chapter 5 of this thesis conducts an evaluation of sentiment in commit messages, in particular aiming to discover whether faulty files can be identified by their commit history. Since Defects4J contains information about bug-fixing commits and provides a list of files that have been faulty through the repository history, Chapter 5 begins with an analysis of sentiment and subjectivity scores of over 17,000 commit messages extracted from Defects4J subject programs, resulting in the following contribution.

Contribution 5.1: *An evaluation of sentiment and subjectivity in* Defects4J *commit messages*

For each of the 395 faults in Defects4J, there is a file that contains the commit hash, which is a long, unique string of characters that represents a commit to version control. This allows the commits to be filtered by those that fix a bug, and those that do not.

Since it is useful to be able to predict from a commit message whether it fixes a bug, I compare the sentiment scores of bug-fix commits with non-bug-fix commits, resulting in the following contribution.

Contribution 5.2: *An evaluation of sentiment and subjectivity in bug-fix commits against non-bug-fix commits*

Additionally, for each of the faults in DEFECTS4J, there is a list of files that are changed as a part of the fix. From this, I can filter the commits to see whether sentiments change when developers are working on faulty files when compared to non-faulty files. If developers feel strongly negative about files that are faulty, or strongly positive about files that are not faulty, this could result in an effective defect prediction strategy that ranks classes based on the set of associated commit messages. This in turn could help to improve G-Clef with more accurate defect estimates. Therefore, Chapter 5 contributes an evaluation of sentiments present in commits to faulty files and non-faulty files to investigate whether a defect prediction strategy could leverage this information.

Contribution 5.3: *An evaluation of sentiment and subjectivity in faulty files against non faulty files*

While this research did not strongly support a test case prioritisation strategy based on the sentiment scores of commit messages, there are a number of promising improvements that could be made in the future. For example, a better sentiment analysis model that is specific to programming terms could improve the results.

Chapter 6: "Conclusions and Future Work" — The final chapter of this thesis summarises the work presented throughout. Additionally, this chapter describes a number of ideas for future work arising from this thesis, including further improvements to test case prioritisation.

Appendix A: "KANONIZO" — Throughout the course of this thesis I have developed an open-source test case prioritisation tool called KANONIZO (which means "arrange" in Greek). All of the strategies used in the Chapter 3 and Chapter 4 were implemented into KANONIZO. Appendix A gives details about the inner workings of KANONIZO and demonstrates the benefits it can provide to the research community.

Chapter 2

Literature Review

This chapter breaks down previous literature, starting with key definitions that will be used throughout this thesis. Following this, this chapter looks into software testing, before introducing test case prioritisation with a motivating example. This chapter then looks at the available metrics for evaluating test case prioritisation, before introducing the approaches that have been presented in previous literature. Next, this chapter looks at the type and number of faults that have been used in previous evaluations of test case prioritisation. Finally, this chapter introduces the related techniques that will be used in Chapter 4 and Chapter 5, firstly motivating the use of defect prediction in test case prioritisation and linking previous studies that have considered defect prediction for test case prioritisation, before introducing sentiment analysis and discussing how it can potentially be used to improve defect prediction.

2.1 Definitions

In software testing, there are a number of key terms that must be defined — in particular, each of the following terms will be used frequently throughout this thesis. While they are easily confused and sometimes used interchangeably, the formal definitions are provided below to distinguish between different words.

Bug

A *bug* in software is probably the most generic way of describing a program doing something that it shouldn't. Patton [27] defines a bug as occurring when one or more of the following conditions is met:

1. The software doesn't do something that the specification says it *should do*
2. The software does something that the specification says it *shouldn't do*
3. The software does something that the specification *doesn't mention*
4. The software doesn't do something that is not mentioned in the specification but *should be*
5. The software is difficult to understand, hard to use, slow, or in the eyes of a software tester, would be viewed by an end-user as "plain not right"

The IEEE Standard Glossary of Software Engineering Terminology [28] states that the term "bug" is synonymous with "error" and "fault", but in practice "bug" is an umbrella term for those terms ("error" and "fault"), each of which is subtly different.

Defect

A defect is an "imperfection" or "deficiency" in a work product where the product does not meet its requirements or specifications and needs to be repaired or replaced [29]. Every *fault* is a defect, but not every defect is a fault. A defect is not a fault if it is detected prior to executing the software (e.g. through static analysis or inspection).

Error

An error in software is a human action that causes an incorrect result [29].

Failure

A failure occurs in software when a product is no longer capable of performing a required function or an event in which a system does not perform a required function within specified limits [29]. A failure may be caused by a fault, and a fault may cause multiple failures.

Fault

A fault is the manifestation of an error in software [29].

Test Case

A test case (or test) is an activity in which a system or component is executed under specified conditions, with the results observed and an evaluation being made [28]. An example of a test case is given in Figure 2.2.

Test Suite

A test suite is an unordered list of test cases [8], or a group of *related* test cases for a particular feature in software [27].

Regression

A regression in software occurs when a developer makes a change to software that causes an unintended side effect in an unmodified part of the program [28]. A regression can also be the re-occurrence of a bug that was previously thought to be fixed [27].

2.2 Unit Testing

A unit test case is a (typically small) snippet of code that exercises some functionality of the program, and includes assertions about the program behaviour that compare the expected output of the program with the actual output. It is typically the role of a developer to assert what the "expected" output of a program is, since they have the best understanding of the code, and can provide examples of what should happen under different circumstances. An example of an implementation of a `substring` method, which starts with a string and returns a smaller chunk of the string using a "from" index (inclusive) as the start point and a "to" index (exclusive) as the end point (e.g. `substring("hello",1,3)` ⇒ `"el"`), is given in Figure 2.1. Note that this implementation contains several bugs that can be discovered through test cases. An example test case for this method given in Figure 2.2. This test starts by setting up some variables and preconditions that are required in order for the test to run (lines 3 and 4). The test then executes a single method of the program on line 5 with a single input ("Hello world"), storing the result in a new variable. Finally, line 6 contains an *assertion*, which checks whether the expected output of the program matches the actual output. This demonstrates the typical structure of test cases, which should be small and should only execute a small part of the program, in order to pinpoint any failures should they occur.

```
public String substring(String in, int from, int to){
  String substring = "";
  for(int i = 0; i < in.length(); i++){
    if (i < from || i >= to){
      continue;
    }
    substring += in.charAt(i)
  }
  return(substring);
}

```

FIGURE 2.1: An example method under test

```
@Test
public void testSubstring(){
  String s = "Hello world";
  String expected = "Hello";
  String actual = substring(s,0,5);
  assertEquals(expected, actual);
}
```

FIGURE 2.2: An example JUnit test

One of the main challenges for developers writing software test cases is distinguishing correct program behaviour from incorrect behaviour. In Figure 2.2, there is have an `expected` value, which is determined to be "Hello". Determining this as the "correct" response is known as the *test oracle problem* [30], and usually requires a human with understanding of the code under test. Furthermore, it is usually not feasible to test methods with every possible input and output. In Figure 2.2, a developer would need to write a separate test case for every possible string, with every combination of "from" and "to" indices. Therefore, a key part of the testing process is identifying places where a program could go wrong. If a developer was to write test cases for the method shown in Figure 2.1, the first thing they may notice is that there is no checking on the values of the "from" and "to" input variables. Assuming that a string will always have length $>= 0$, anyone who uses this method *should* pass in values for "from" and "to" that are also $>= 0$. Therefore, the developer may write a test case that calls `substring("test", -2, 2)`. If the method is implemented correctly, this should result in an exception stating that the input is not valid. However, in the implementation given in Figure 2.1, it will return `"te"`. Therefore, this test case would reveal a bug in the program. Intuitively, if a user of this code is trying to create a substring with a "from" index and a "to" index, it should also follow that the "from" index will be lower than the "to" index. Therefore, the developer may also write a test case that calls `substring("test", 4, 2)`. This case should also result in an exception, but in Figure 2.1, it returns an empty string.

Moreover, a developer may wish to test what happens if the "to" index exceeds the length of the input string (e.g. `substring("test", 2, 10)`). Note that none of these use cases are given in the method specification, making them fall under definition 4 of Patton [27], that the software "doesn't do something that is not mentioned in the specification but *should be*". Finally, according to definition 1 in Patton [27], a bug occurs when the software *doesn't* do something the specification says it *should do*. Therefore, a tester must show that, when given valid input, the method returns as expected. This leads to the test case shown in Figure 2.2. Note that it is the job of a developer to recognise scenarios in which code under test may return invalid output, and which inputs may trigger faulty behaviour.

2.2.1 Coverage

One of the most common quality assurance metrics used with test suites is the *coverage* of the suite. Coverage refers to the units of code in a piece of software that have are executed by test cases contained within the test suite. Since it is impossible to detect a fault without executing the line(s) on which the fault is manifested, there is an assumption that increasing coverage implies higher quality in a test suite. However, considering the earlier example in Figure 2.1, if the method is called `substring(null,0,0)`, this will reveal cause a `NullPointerException` to be thrown on line 3. If any non-null string is passed to this method, the exception does not occur, but line 3 will still be executed. Therefore, it is possible to execute a line of buggy code without revealing the bug.

Coverage can be measured at different levels within a program:

- Line/Statement Coverage - Each line of code written by a developer can be covered by test cases
- Branch/Decision Coverage - Every time a program meets a conditional statement (e.g. `if`), there are two branches the program can take — the `"true"` branch and the `"false"` branch
- Block coverage - Similar to branch/decision coverage, blocks of code are sections where the code should follow one execution path except in the case of an exception
- Function/Method coverage - Functions and Methods represent logical chunks of the program, and measuring the coverage of methods requires less work than individual lines/branches.
- Class Coverage - The simplest form of coverage is to see which classes are executed by test cases

A simple example of coverage can be seen in Figure 2.3. In this example, there is one test case that only executes one half of the decision on line 4. The coverage tool used in this instance highlights the decision in yellow to indicate that one of the branches has been followed, but not the other. Lines highlighted in green have been executed by the test case. One of the important things in Figure 2.3 is that while the test case has achieved 71.4% line coverage, there is only 50% branch/decision coverage. This issue leads to a discussion regarding which type of coverage is "best" to use in order to assure program quality [31].

```
1   public void methodWithOneBranch(int x) {
2       System.out.println("The value of x is " + x);
3       x = (int) Math.sqrt(x);
4       if (x < 100){
5           System.out.println("The value of sqrt(x) is < 100");
6       } else {
7           System.out.println("The value of sqrt(x) is >= 100");
8       }
9   }
10
```

FIGURE 2.3: Coverage of a simple method - lines that have been executed are in green, while lines that have not been executed are in red. Branches that have been partially executed are in yellow

2.3 Types of Testing

In unit testing, test cases aim to execute small chunks of code that should be self-contained and work without influence of any other code (i.e. units). Each unit in a program should be tested as a part of unit testing. Unit testing also requires knowledge of the implementation details, knowledge of the programming language used, and knowledge of the structure of the program in order to effectively perform. These attributes form what is generally known as *white-box testing*, in which the tester is fully aware of all the implementation details of the software. The alternative to this is *black-box testing*, which is a different type of testing involving no knowledge of the program structure or style, using the program as a user would in order to test a specification. An example of a black-box testing tool is QFTest[1], where testers can set up sequences of clicks, menu actions and keyboard actions to define a sequence of events in a way that a user might interact with a system. They can then make assertions about the state of the program, for example if a box should have appeared on screen, they can test for this. Testing software in this way creates more complex inter-dependencies between parts of the code,

[1] https://www.qfs.de/en/qf-test

which can help to identify when behaviour is not correct, and can also be very crucial in finding faults that would otherwise be detected by users. Patton [27] defines black-box testing as any form of testing looking at data, logic or states (i.e. anything that would be triggered and observed by an end-user), while white-box testing involves looking at computation errors, control flow errors or input/output errors (i.e. the sort of mistakes developers might make during implementation).

2.4 Test Suite Ordering

Depending on the programming language and the tools that are used, there may be several ways in which a test suite can be run. In old versions of JUnit (before JUnit 4), developers created "Suite" objects, in which developers would specify the classes that contain test cases, therefore creating a natural order. In Ant[2], developers specify patterns in an xml file that are used to recognise test cases, which internally uses the file system to find test classes. Crucially, test suites should be independent [32], meaning that there is no specific "default" ordering of a test suite.

2.5 Test Case Prioritisation

Test case prioritisation is a process that aims to reorder a test suite such that any regressions that occur during software evolution are detected as quickly as possible. The intention for test case prioritisation is to be run infrequently, and to re-use the prioritised ordering for a long period, in contrast to test case selection, which is re-run every time a new change is made.

Rothermel et al. [10] formally defines test case prioritisation in Definition 2.1. This definition identifies a prioritised test suite (T') from the superset containing all possible orderings of the original test suite (T) that maximises the value of a function f. Ideally, f would be a function that states exactly how good at detecting regressions a particular ordering is. However, in practice it is not possible to know whether a change has caused a regression until the entire test suite has been run. Therefore, f is an approximation of fault detection — for this reason, finding an appropriate f has been the subject of a lot of research in test case prioritisation to date. In Chapter 3, I evaluate how effective existing approximations for f are when using real faults, while in Chapters 4 and 5, I use defect prediction as a surrogate for the function f.

[2]https://ant.apache.org/

Definition 2.1. Test Case Prioritisation
Requirements:
T, a test suite
PT, the set of permutations of T
f, a function that produces a score for a permutation PT
Problem: Find $T^{'} \in PT$ such that

$$(\forall T^{''})(T^{''} \in PT)(T^{''} \neq T^{'})[f(T^{'}) \geq f(T^{''})]$$

Considering Definition 2.1, it is important to note that the concept of test case prioritisation is not reliant on a particular software version, nor is it aware of any modifications to the software.

2.6 Motivation

Although many different papers have given different interpretations as to why test case prioritisation is needed, all of them largely revolve around the same concept. Table 2.1 presents a program that has 10 known faults and five test cases (A-E). Since test case C detects seven of the 10 faults and test E detects the remaining three, all of the known faults in this program can be detected by placing these two test cases first. Although the remaining test cases do not reveal any known faults that are not revealed by test case C or E, the ordering of these test cases may still be important for detecting *future faults*. In practice, a test case prioritisation strategy would need to achieve this ordering without knowing how many faults are actually detected by the test cases, since it is not possible to know about the existence of faults before running the test suite. One

Test Case	Fault									
	1	2	3	4	5	6	7	8	9	10
A	X				X					
B	X				X	X	X			
C	X	X	X	X	X	X	X			
D					X					
E								X	X	X

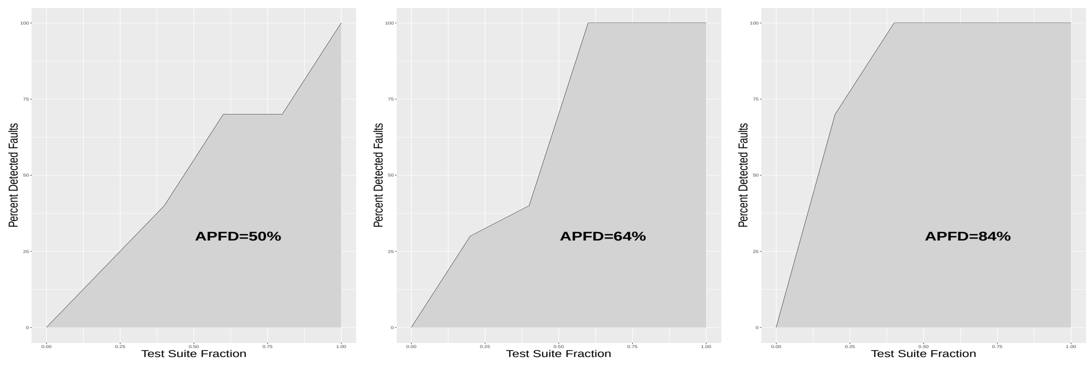
TABLE 2.1: An example of how test case prioritisation can result in faster fault detection, taken from [11]

of the assumptions that is commonly made in test case prioritisation is that each test case has equal cost [14]. If each test case has equal cost then detecting faults using fewer test cases also means detecting faults using less time. In practice, it is most likely that a company is more interested in saving time than running fewer test cases, so it is important to verify this assumption or ensure that test case prioritisation strategies also consider the *time taken* to find faults.

Another assumption that is made is that the time cost associated with actually prioritising test cases is negligible [33]. If a prioritised test suite detects faults 20 minutes faster than the original test suite, but the prioritisation took 30 minutes to run, then there is a still a negative overall impact of using test case prioritisation. Malishevsky et al. [34] proposed a model to calculate the total cost of prioritising test cases that incorporates the cost of analysis (source code analysis, analysis of changes, code coverage calculation), the cost of actually running the prioritisation strategy and the time saving achieved by the prioritised ordering. Elbaum et al. [35] conducted an experiment in which they considered several *cost-benefit threshold* values, which indicate by how much a compared strategy must outperform the other in order to justify its cost. If the threshold was set at 5% or higher, using random orderings were better than any of the other approaches studied. Do et al. [33] conducted a similar experiment using the models proposed in Malishevsky et al. to calculate the difference in total *delay* (total time taken to prioritise - time savings as a result of prioritisation) between random orderings and prioritised orderings. Furthermore, Do et al. [36] conducted a study in which the testing time was restricted to set levels (25%, 50%, 75%, 100%) to see how this affected the cost-benefit trade-off of running prioritisation. Their experiments concluded that some, but not all, approaches are beneficial in spite of their costs when the testing resources are constrained.

2.7 Evaluation Measures for Test Case Prioritisation

Given that the goal of test case prioritisation is to maximise the fault-detection capability of test suites in the earliest possible stage, there should be an effectiveness measure for a candidate solution that represents this early fault detection. Rothermel et al. [10] introduced the measure of Average Percentage Faults Detected ($APFD$), an area under curve score that represents how many faults have been detected at each point of the test suite execution. While it is not possible to determine the location of faults in production software without executing the entire test suite, in the presented empirical evaluations, fault information is available to researchers in order to evaluate their approaches. The equation for $APFD$ is given in Equation 2.1 [11], where m is the number of faults, n is the number of test cases, and TF_i is the index of the test case in the current suite that



(A) Test Ordering: A-B-C-D-E, $APFD = 50\%$ (B) Test Ordering: E-D-C-B-A, $APFD = 64\%$ (C) Test Ordering: C-E-B-A-D, $APFD = 84\%$

FIGURE 2.4: Examples of *APFD* for different test case orderings, replicated images from Rothermel et al. [10]

discovers fault i.

$$APFD = 1 - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{1}{2n} \tag{2.1}$$

Figure 2.4 represents the *APFD* of 3 possible orderings of Figure 2.1. In the original ordering, Figure 2.4a, the final fault is not detected until the very last test case is executed, so the *APFD* score is low. However, in Figure 2.4c, the faults are discovered much faster, with all 10 of the faults being discovered by the first two tests, in particular with the first test discovering seven faults, so the *APFD* is much higher. This principle is applied to a large number of test case prioritisation techniques to evaluate their effectiveness. It is worth noting that the value of *APFD* can never be “perfect” — in order to achieve an *APFD* score of 1, a test suite would need to identify all faults in software before any test cases are run. The limit of *APFD* tends towards 1 if test suites identify *all* faults by running a single test case.

2.7.1 Problems with the *APFD* Metric

APFD is not a perfect assessment of how effective a test case prioritisation technique is. *APFD* assumes that all possible faults in a system under test have been found. The impact of this is that in situations where not all faults are found, the actual impact of finding faults is lessened, since the ‘pool’ of possible *APFD* scores is lessened. Additionally, *APFD* does not punish test suites for failing to find faults at all. In Figure 2.1, if TF_i is undefined, it is assumed to have a value of 0, meaning faults that are not discovered by any test case have no negative impact on the score, only failing to contribute [37].

2.7.2 *NAPFD*

NAPFD, or Normalised *APFD*, is a metric presented by Qu et al. [37]. *NAPFD* aims to overcome one of the weaknesses of *APFD* described above, by calculating a value p, which represents the ratio of total faults discovered by the entire test suite. This equation was developed in the context of test suite reduction, in which the number of faults detected by an entire suite, or the number of tests contained in the test suite, is susceptible to change. By containing the p value, smaller test suites would be rewarded for detecting the same number of faults as a larger suite.

$$NAPFD = p - \frac{\sum_{i=1}^{m} TF_i}{n \times m} + \frac{p}{2n} \tag{2.2}$$

The equation in Equation 2.2 is exactly the same as Equation 2.1, with the exception of the p value. In practice, for test case prioritisation techniques that do not also reduce the size of the prioritised suite, it is expected that this equation will reduce to the standard *APFD* equation.

2.7.3 $APFD_c$

$APFD_c$ is another extension of *APFD*, with a focus on prioritising test cases in a manner that makes the prioritised suite cost effective. The advantage of this method is that a prioritised suite is not only produced based on fault detection, but also execution cost. If every fault has a severity attached f_i, and every test case has a cost attached t_i, then the formula given by Equation 2.3 becomes the calculation for an $APFD_c$ score for a test suite. This equation is provably reducible to Equation 2.1, as shown in Malishevsky et al. [14], in the case where every test cost is one and every fault severity is one. Most of the early techniques in test case prioritisation only focused on a single objective, whether that would be code coverage, mutant coverage or some other heuristic. By considering the combination of coverage and cost, the test case prioritisation problem becomes a *multi-objective* problem.

$$APFD_c = \frac{\sum_{i=1}^{m}(f_i \times (\sum_{j=TF_i}^{n} t_j - \frac{1}{2} t_{TF_i}))}{\sum_{j=1}^{n} t_j \times \sum_{i=1}^{m} f_i} \tag{2.3}$$

2.7.4 Failing to Find Faults

Walcott et al. [13] discussed an *APFD* metric that would punish test suites for failing to find faults in their study on Time Aware Test Suite Prioritisation. In their work, the authors discussed another multi-objective test case prioritisation approach, with an addition to the *APFD* metric to deal with cases in which faults are not found. By replacing TF_i with $|m|+1$ (where m is the number of test cases) when TF_i is undefined, it becomes possible to punish test suites for failing to find faults. However, this situation is unlikely to occur with most test case prioritisation strategies, since the whole suite would be executed after being reordered, and therefore any faults that were detected with the original ordering would still be detected by a prioritised ordering.

2.8 Coverage-Based Approaches

Each test case will exercise a series of statements and decisions that uniquely represent a likely use case of the software. By tracking the statements and/or decisions that are executed by individual test cases, coverage-based test case prioritisation approaches aim to order the test cases in such a way that will execute the largest amount of code in the smallest amount of time. Since coverage-based approaches can happen at several different levels, from now on, I will refer to statements, branches, decisions, blocks, methods etc. as *goals*.

2.8.1 Total Coverage

The simplest example of a coverage-based test case prioritisation approach is the total coverage strategy. The total coverage strategy involves repeatedly taking the test case with the highest number of goals covered, until there are no test cases left to select. This approach generally serves as a baseline in the literature after first being proposed by Rothermel et al. [10], since it is simple and easy to implement, and has been used in almost all studies that look at coverage-based test case prioritisation (e.g. [11, 12, 14, 22, 38]). Table 2.2 shows how the total strategy would order the test cases given in Table 2.1, and shows that the ordering that would be produced is sub-optimal, since test B only discovers faults already found by test C. This shows one of the clear limitations of the total coverage strategy.

Test Case	Number of Goals Covered
C	7
B	4
E	3
A	2
D	1

TABLE 2.2: Ordering of test cases from Table 2.1 according to greedy/total algorithm

2.8.2 Additional Coverage

In order to try and overcome the obvious flaws in the total coverage strategy, the additional coverage strategy was defined by Rothermel et al. [10]. Additional coverage builds on the concept of total coverage, but with the added concept of uncovered goals. At each stage, the additional coverage strategy selects the test case that covers the highest number of *uncovered goals*, then removes each goal covered by the selected test case from the set of uncovered goals. In contrast to the total coverage strategy, the additional coverage strategy *does* select the best possible ordering for the simple example shown in Table 2.1. As with the total coverage strategy, almost every test case prioritisation paper that looks at coverage-based test case prioritisation includes additional coverage, since it is also simple and easy to implement, and outperforms the total coverage strategy in almost all implementations.

Test Case	Number of New Goals Covered
C	7
E	3
D/A/B	0

TABLE 2.3: Ordering of test cases from Table 2.1 according to additional algorithm

2.8.3 K-Optimal Coverage

The K-Optimal Greedy Approach is an approach to solving prioritisation problems by selecting the k parts that, when combined, result in the largest number of goals covered [39]. This enhances the total coverage strategy by attempting to eliminate the case where multiple test cases have high coverage and a high overlap in goals covered. This strategy was adopted by Li et al. [12], with k=*2*, to create the 2-optimal algorithm. Starting with an empty test suite, this strategy repeatedly selects pairs of test cases that cover the highest number of goals between them, and adds them to the prioritised test suite.

As with the selection of single test cases, there are total and additional methods for implementing K-Optimal. In Li et al. [12], the additional coverage strategy was adopted.

One of the negative aspects of the 2-Optimal Greedy algorithm is the associated cost complexity. Using additional coverage requires re-calculation of the remaining coverable goals after the selection of every two test cases. To consider every possible pair of tests, including the re-adjusting of coverage information has complexity $O(mn^2)$ and this must be repeated n times, meaning the overall complexity of this algorithm is $O(mn^3)$. As the number of test cases increases, the complexity associated with calculating the best ordering increases exponentially. In their study, Li et al generated small and large test suites for their subject programs, with small test suites ranging from 8-155 tests, and large suites ranging from 228-4,350 tests. In the case of the larger test suites, the complexity of the 2-optimal algorithm should lead to incredibly long running times. However, the authors did not include the time taken to run the experiments in their evaluations, instead including them as a threat to validity.

2.8.4 Optimal-Coverage Approach

One of the key assumptions in coverage-based test case prioritisation techniques is that if coverage is maximised, then fault-detection capability of the test suite is maximised. This assumption is investigated by Hao et al. [21], who presented the coverage problem as an integer linear programming (ILP) problem. Integer Linear Programming takes a series of constraints and variables, allowing for the maximisation of any single variable by calculating the possible points that do not violate any constraints. In this case, the variables included are execution order $x_{ij}, 1 \leq i, j \leq n$ and statement coverage $y_{jk}, 1 \leq j \leq n, 1 \leq k \leq m$, while the constraints in place are that each test case can only appear once, and that the statement coverage must be in accordance with the best n test cases in T.

The authors conjectured that using optimal coverage would result in higher rates of fault detection than using additional coverage approaches. To test this, they collected eight C programs and 11 versions of two Java programs and prioritised them using both the optimal and additional coverage techniques, before comparing the results of each prioritisation technique. The effectiveness of each technique was measured by three factors: impact on APxC metric, impact on fault detection, and difference in time complexity for the different strategies. The results of this study show that, despite additional coverage never *outperforming* optimal coverage in terms of APxC, there are many cases where the optimal approach produces exactly the same value as the additional coverage strategy. This implies that additional coverage is often the best ordering of test cases in order to maximise coverage.

There are far more interesting results presented in regards to the second research question. When calculating the effectiveness of test case prioritisation using $APFD$, additional coverage significantly outperforms optimal coverage in a lot of the studied programs. The effect is not significant for the studied Java programs, although the authors conjecture a reason for this being that there are far fewer mutant groups and tests in the Java programs than in the C programs. In addition to these comparisons, there is an additional comparison with the *ideal* strategy. The *ideal* strategy (sometimes referred to as optimal) involves placing the fault-detecting test cases first. Obviously this strategy is not available in practice since faults are not known, but it is often used to place techniques between the worst case scenario and the best case scenario. When compared to *ideal* test suites, most of the prioritised suites have higher APxC values, but perform significantly worse in terms of fault detection. This result implies that pursuing coverage beyond the levels already achieved by the additional coverage strategy may not be worthwhile, since the increases in coverage do not guarantee any increase in $APFD$.

2.8.5 Coverage Granularity

Many studies investigating the effectiveness of coverage-based test case prioritisation have considered many different levels of coverage granularity (e.g. [10–13, 38, 40, 41]). Consider the method shown in Figure 2.5. Instrumentation can tell us several things about whether or not this method was covered by a certain test case.

- Method/function coverage — This coverage granularity tells us whether or not a test case entered the method shown — it does not care for the path taken by the test case once the method has been entered
- Statement/line coverage — This will give the user a list of lines executed by a test case that executes the method — in this case some subset of 2—8, based on which if statements were followed
- Block coverage — Code encased within brackets is considered to be a block. Once a block has been entered, under normal circumstances (e.g. no exceptions) it should continue to execute the code in that block until completion, so this should represent an improvement on raw line coverage
- Decision/Branch coverage — Since one of the key concepts in programming is representing decisions and choices, this type of coverage looks at the decisions that were made in a particular execution of a test case. For example, in Figure 2.5, there are decisions on lines two, four and nine, each of which have two execution branches, one where the if statement is true, and one where the if statement is false.

However, it is not a general rule that the finer granularity coverage levels are objectively better than coarser levels. As has already been mentioned, Rothermel et al. [10] discovered that branch coverage performed better than statement coverage. Di Nardo et al. [40] evaluated the differences between function, block, basic block and decision level

```java
public static float max(float a, float b) {
  if (a != a)
    return a;   // a is NaN
  if ((a == 0.0f) && (b == 0.0f) &&
      (Float.floatToRawIntBits(a) == negativeZeroFloatBits)) {
    // Raw conversion ok since NaN can't map to -0.0.
    return b;
  }
  return (a >= b) ? a : b;
}
```

FIGURE 2.5: An example Java Method from the Math class

coverage, discovering that under most circumstances, basic block coverage provided the largest benefit in fault detection. An additional observation from this study is that in a total-coverage situation, decision and function level techniques were roughly similar, whereas in the additional-coverage strategy, decision coverage performed much better than the function level counterparts. Additional coverage strategies also introduced far more variance in results, with total coverage having very small error bars compared to the additional versions. Each of these different granularities of coverage can represent distinct things relating to program execution, and as such it is important to investigate what impact each of these granularities has on test case prioritisation. Rothermel et al. [10] investigated the use of branch and statement coverage, specifically testing both coverage types using the additional and total coverage strategies. In their paper, the authors discovered that branch coverage outperformed statement coverage, although they give no specific justification for why this effect is observed. Elbaum et al. [38] proposed a research question to investigate the effect of finer granularity over coarse granularity, specifically comparing statement and function level coverage. Elbaum et al. [38] found that there was enough evidence in their data to suggest that statement level coverage was more effective than function level. The authors conjecture that functions can be completely variable in size, while statements are always statements. Covering a function that only executes three statements is given the same weighting as covering a function executing 100 lines.

2.8.6 Limitations of Coverage-Based Approaches

On the face of it, coverage-based test case prioritisation has several benefits. It is reasonably simple to calculate coverage at any granularity, and there are a large number of successful implementations of coverage-based test case prioritisation in previous studies, including Di Nardo et al. [40], Rothermel et al. [10], Elbaum et al. [11, 38], Hao et al. [41] and Li et al. [12]. However, there is one key assumption made in coverage-based test case prioritisation that may not always be true: executing more code leads to the detection of more faults. There are two reasons why this may not be true — the firstly, faults may exist anywhere in the code. Consider a method that has only five lines of code, one of which is faulty. If unit tests are written for this method, they will have very low coverage compared to other tests for much larger methods. A coverage-based approach would result in this fault being extremely low priority. Moreover, as shown in Section 2.2, it is possible for tests to execute lines of code without exposing a fault. Developers sometimes neglect testing and only care about higher numbers of coverage instead of thinking of potentially faulty scenarios.

There have been some studies investigating the correlation between code coverage and test suite effectiveness. Inozemtseva et al. [42] created large numbers of artificial faults (9,552-50,302) in five programs. In order to measure effectiveness, the authors took random samples of the original test suite of different sizes (i.e. test suites with 3 tests, 10 tests, 30 tests, 100 tests etc.), then created a normalised effectiveness measure which combined the number of faults covered to the number of faults detected. They then proposed three research questions, relating to size, coverage and effectiveness correlations. Their findings were that, while the test suites with higher coverage were more effective than ones with lower coverage, some of this effect can be attributed to simply having more test cases. When the test suite sizes were controlled, the correlation was much less noticeable. That is to say, when comparing all test suites of size X on program Y, the coverage had much less impact on the effectiveness of the test suite than when the test suites of all sizes were considered. This implies that coverage had less impact on effectiveness than the size of the test suite did.

Kochhar et al. [43] build on this work, applying the same principle to real faults to see if the same observations could be made. Kochhar et al obtained 67 and 92 bugs from HTTPClient and Rhino, two large software systems built in Java. These faults were obtained by looking through the bug-tracking systems of each software package respectively. The authors used Randoop[3] to generate test suites for each buggy version of code that was checked out from the repositories, and created test suites representing 0.2%, 0.5%, 1%, 5%, 10% and 100% of the original test suite size. The reason for creating suites of different sizes is to see what impact the size of a test suite has on its effectiveness. For each bug, only one test suite of each size was generated. In their experiments, the authors found that there was a moderate correlation between test suite coverage and effectiveness for HTTPClient, while the correlation was strong for Rhino. While these results are less definite than in Inozemtseva et al, they still provide a reason to doubt that covering code earlier will lead to faster fault detection.

One of the main problems with coverage-based test case prioritisation is the fundamental reliance on executing the entire test suite in order to produce a prioritised test suite.

[3]https://randoop.github.io/randoop/

Since coverage information for all test cases must be known, all source code must be instrumented. In their research, Tikir et al. state that executing instrumented code can be up to 20 times slower than executing non-instrumented code [44]. Considering one of the main reasons for attempting test case prioritisation is to lower the time taken to discover faults in software, having to run the entire test suite every time software changes is an unrealistic expectation.

A study by Lu et al [22] investigated the impact that software evolution has on prioritised test suites. In their work, the authors produced prioritised suites based on coverage information from various different versions of evolving software. The results of this study showed that software changes have a large impact on prioritised test suites, and more importantly that a prioritised test suite becomes much less effective as software develops. This represents an ongoing challenge in test case prioritisation research, which must improve the longevity of prioritised suites in order to be viable in industry.

2.9 Mutation-Based Approaches

Mutation is a technique designed to simulate real faults occurring in software [45]. “Mutants” are small, rule-based syntactic changes to the program. This section describes how mutants can be used to demonstrate the quality of a test suite, and how mutant coverage can be used by test case prioritisation strategies.

2.9.1 Mutation

The previous section reveals a flaw in using coverage as an indicator of test suite quality — simply covering code is that executing a fault is not the same as revealing a fault. There can be a series of pre-conditions and system state dependencies that mean a faulty line can go undetected even if it is executed by several tests. One of the ways of solving this problem is mutation testing. Figure 2.6 gives an example of a mutant that could be created in a program. By creating many such changes to software, and evaluating the test suite against each of these changes, it is possible to see how well the test suite is able to deal with changes to the code.

```
public void abs(int x){
  if (x < 0){
    return -x;
  }
  return x;
}
```

```
public void abs(int x){
  if (x > 0){
    return -x;
  }
  return x;
}
```

FIGURE 2.6: An example of a change that could be introduced by a mutation operator, "<" replaced with ">"

Mutants are said to be "alive" at the point at which they are created [45]. Mutants are "killed" if a test case that passed in the original version of a program now fails when executed on the mutated version [45]. High quality test suites should be able to kill a larger number of mutants, since it shows developers have considered a lot of potentially faulty scenarios.

One of the downsides to using mutation testing is the high computational cost [46]. Many mutants can be generated at a low cost, since the cost of analysing the source code is relatively low. However, in order to kill mutants in isolation, every test case must be executed with every mutant. As a result of this, the cost complexity of mutation testing is $O(mn)$, with m mutants and n test cases.

Another problem in the field of mutation testing is the equivalent mutant problem [17]. An equivalent mutant is a change in source code generated by an automated mutation tool that exhibits identical behaviour to the original program. These mutants are not killed by any test case since they are technically not faults. Yet, when a mutant is not killed by any existing test, it is extremely hard to know whether this represents an inadequacy in the test or an equivalent mutant. Figure 2.7 gives an example of an equivalent mutant, taken from Jia et al. [45].

```
for(int i = 0; i < 10; i++){
  // i is the same value
}
```

```
for(int i = 0; i != 10; i++){
  // i is the same value
}
```

FIGURE 2.7: An example of equivalent mutation from [45]

Mutation analysis was first used in test case prioritisation by Rothermel et al. [10], in a paper which introduced the concept of Fault-Exposing Potential, or FEP. Each test case is assigned a FEP score based on the ratio of mutants it kills compared to the number it executes. For this reason, FEP is sometimes referred to as a coverage-based approach [47]. In order to accurately represent the FEP of each test case, it is important that every test case is able to execute at least some mutants, which means a large number of mutants needs to be introduced into a program.

The calculation for Fault-Exposing Potential has changed a lot since its introduction by Rothermel et al. [10]. In the early stages, FEP would be calculated for test case t_k as $\sum_{i=1}^{|S(t_k)|} \frac{killed(s_i)}{total(s_i)}$, where $|S(t_k)|$ is the number of statements executed by test case k, $killed(s_i)$ is the number of mutants killed on line s_i and $total(s_i)$ is the total number of available mutants on line s_i. In work by Chen et al. [48], the authors, including two of the authors who had worked on Rothermel et al. [10] and Elbaum et al. [38], the definition of the FEP estimate that a test case would find mutant x was given by Equation 2.4. The Equation 2.4 calculates the FEP of a line of code x, that contains m mutants and is executed by k test cases. For each of the test cases that execute this line, n_i is the number of mutants on line x that are killed by test i. To calculate the FEP score of a test case is then calculated using the FEP scores of the lines that it covers. While the original calculations were focused around specifically test cases, the new approach is based around creating an FEP score for every executable statement in the program. This helps make the process more consistent between test cases executing the same statements.

$$FEP_x = \frac{\sum_{i=1}^{k} n_i}{m \times k} \tag{2.4}$$

In their experiments, Chen et al. [48] mention that their results are not intended to be necessarily applicable in real-world scenarios, since mutation analysis is expensive, only that they are more interested in demonstrating that test suites that have better FEP scores will also be better at detecting seeded faults. Crucially for test case prioritisation, this means that a strategy based on FEP will be likely to be prohibitively expensive to run frequently. Notably, in previous studies that have considered cost-benefit trade-offs for prioritisation strategies, FEP has been absent from these evaluations [14, 35, 49].

Generating large numbers of mutants is very easy to do, and there are many available tools that will generate large numbers as part of the compilation process and according to a specification [16, 50]. However, when there are more mutants and larger test suites, the problem becomes noticeably more complex. This effect is a result of having to run every combination of test cases and mutants together to determine the number of mutants killed by each test case. If the only interest is seeing the total number killed by the entire test suite, mutant m can be ignored once a test case t has killed it. However, if a 'kill map' (see Table 2.4) is needed, with every pairing of test case t and mutant m, such as in the case of FEP, you cannot ignore any mutants regardless of their killed status. As a result, the time complexity of mutation analysis is $O(mn)$. In Table 2.4, mutant one is killed by multiple test cases, while the other mutants are only killed by one test case. Kill maps become much more complex when there are more mutants and test cases, but fundamentally they record a test case/mutant tuple representing a single kill, and will record every test case that kills a mutant, not only the first test case.

Test Case	**Mutant**
1	1
1	5
1	27
2	1
2	6
⋮	⋮

Table 2.4: An example mutant kill map

Jia and Harman [45] discuss the problems associated with mutation testing, including the high cost of mutation analysis, and the oracle problem. The oracle problem refers to the idea that a test case is 'correct', meaning if it fails, it identifies a fault in the program. While this problem is not strictly related to the domain of mutation testing, and is applicable in all testing, test case failures in mutation analysis will result in killed mutants, while the mutant may actually have been exhibiting correct behaviour, if the test case is written badly. Considering the equivalent mutant problem introduced in Section 2.9.1, the inclusion of equivalent mutants in the FEP calculation leads to a lower estimate of the actual fault-exposing potential than the real value, since equivalent mutants are not killed by any test [48].

2.10 History-Based Approaches

While most of the code-based techniques described above have been shown to be effective, they are often memoryless and do not take into consideration that regression testing happens with every update to a program, rather than as a one-off process. In addition to this, historical fault information can actually help us to find the likely parts of the code that faults are going to appear in. Kim and Porter [51] suggested a test case prioritisation model based on historical data collected from previous test suite executions. In their work, Kim and Porter collected a set of previous test case execution data, which could have been fault data, coverage data, or any other information that defines the test case, and used a weighting value α to assign weight to the previous historical data.

Specifically, for version 0 of the test case, the value would simply be 0 if the test case passed, and 1 if the test case failed. Assuming a weighting α that determines the "decay" of information (favouring more recent results over older results), and a version k, the priority P of the test case is $\alpha P_k + (1-\alpha)P_{k-1}, 0 \leq \alpha \leq 1, k \geq 1$. Since all previous information would be included in the calculation of P_{k-1}, this means that incrementally all historical information would be included in the calculation of P_k.

Table 2.5 gives an example of some history-based information for a test case. This example considers five versions of the same program and the results of the test case T_k across all five versions — ✗ indicates that the test case failed, while ✓ indicates that the test case passed. Assuming an α value of 0.6, Equation 2.5 gives an example showing how the prioritisation for this test case would be calculated. This shows how older results are less important due to the decay over time, and also how test cases that frequently fail will be given high priority by this strategy.

Table 2.5: Table showing the history of a test case over five versions of software

Test Case	Version 4	Version 3	Version 2	Version 1	Version 0
T_k	✗	✓	✗	✓	✗

$$\begin{aligned}
&P_0 = 1\\
&P_1 = \alpha(0) + (1-\alpha)P_0 = 0 + 0.4(1) = 0.4\\
&P_2 = \alpha(1) + (1-\alpha)P_1 = 0.6 + 0.4(0.4) = 0.76\\
&P_3 = \alpha(0) + (1-\alpha)P_2 = 0 + 0.4(0.76) = 0.304\\
&P_4 = \alpha(1) + (1-\alpha)P_3 = 0.6 + 0.4(0.304) = 0.7216
\end{aligned} \tag{2.5}$$

Kim and Porter compared their history-based approach to some known regression test selection and minimisation techniques. In their experiments, test minimisation led to the smallest number of test cases having to be executed, but also provided the worst fault detection rate of all techniques. History-Based test case prioritisation provides an adequate compromise between the retest-all test selection strategy, which involves running every test case every time, and test suite minimisation, which had a negative impact on fault detection.

Huang et al. [52] proposed a strategy that incorporates varying fault severity into history-based test case prioritisation. Specifically, the paper introduces MCCTCP, a strategy that prioritises the test case(s) that have detected the *most severe* faults in the software history, using the $APFD_c$ metric. For example, in Java, a `NullPointerException` that causes a system crash may be considered more severe than a `AssertionFailedError`, which may just be an outdated test case.

Lin et al. [53] built upon this work and extended it to create their own version of history-based test case prioritisation, this time incorporating software version awareness. The authors argued that the maturity of a specific section of the code had a great impact on the likelihood that a fault occurred there, and as a result scaled down the impact of the previous results, meaning software that has been changed and tested many times becomes less likely to be highly prioritised, unless it has more recently revealed a fault.

In their experiments, Lin et al. found that their version-aware test case prioritisation technique outperformed the previous research by Kim and Porter [51] in six of the eight studied applications, indicating that using version-awareness aided the effectiveness of prioritisation.

Marijan et al. [54] developed a strategy called ROCKET, which gives a higher priority to test cases that have failed recently than those that have no failed for a long time. ROCKET uses a value ω, where $\omega = 0.7$ if the test case failed on its most recent execution, $\omega = 0.2$ if the test case failed on the execution before that, and $\omega = 0.1$ for any executions before that. Using this, the priority score for a test case is calculated by summing the ω scores over the entire history for the test case.

Elbaum et al. [55] proposed a classifier based on three conditions to prioritise test cases. There is a failure window (WF) that tracks the amount of time since a test case last failed, an execution window (WE) that tracks the amount of time since a test case was last executed, and an age variable (WN) that tracks how long the test case has existed. If a test case violates any of these criteria, it is assigned a priority score of 1, otherwise the score is 0. In JUnit, it is common that all test cases are executed every time the test suite is run, therefore it may be possible that WE is disregarded.

Cho et al. [56] described a statistical approach to prioritising test cases, based on the *expected* number of consecutive failures for a test case. The approach, named AFSAC, uses three values, calculated from the history of the test case. These are the maximum number of consecutive failures for a test case Fr_{max}, the minimum number of consecutive failures for a test case Fr_{min}, and the average number of consecutive failures for a test case Fr_{avg}. Following this, the current number of consecutive failures k is calculated for all test cases. If $k < Fr_{min}$, the test case is assigned a value α, if $Fr_{min} \leq k < Fr_{avg}$, the test case is assigned the value β, if $Fr_{avg} \leq k < Fr_{max}$, the test case is assigned the value γ, and if $k \geq Fr_{max}$, the test case is assigned the value δ. The paper does not give values for these weights, but states that $\alpha > \beta > \gamma > \delta > 0$.

Another approach to using history-based test case prioritisation was in Gupta et al. [57]. In their approach, Gupta et al. used the line coverage information of modified code, in conjunction with historical test case information. This approach is based off the idea that immediately changed code has the highest probability of being defective, so test cases that cover the changed code should have the highest priority, followed by the test cases that score highly using historical fault information, followed by the remaining cases. Notably, this strategy blurs the lines between test case selection and test case prioritisation, since it uses change information that is usually not considered in test case prioritisation.

2.11 Other Heuristic Approaches

Up until this point, there have been several categories of strategy (i.e. coverage based, mutation based, history based), and all strategies have fallen under one or more of these categories. However, other approaches have been presented in previous literature that do not fall under any of these categories.

2.11.1 Fault-Index (FI)

Fault-Index (FI) test case prioritisation is based around the idea that each line of code does not have the same probability of containing a fault. In Elbaum et al. [38], the authors state that there are many measurable attributes of code that can represent the likelihood that it contains a fault. In order to generate the FI score for a function, the authors use Principal Component Analysis to combine the numerous measurable code quality factors. They then compare this with the fault-index value for a baseline version of the software, in order to obtain an *absolute* value for fault-index, representing the likelihood of a fault occurring based on the complexity of the changes introduced in this version of the software. Since fault-index is representative of functions, it is possible to use total and additional approaches to it.

In their experiments, Elbaum et al. [38] used 14 different combinations of test case prioritisation techniques described in Table 2.6.

To combine FI and FEP, the authors first applied fault-index scores to all the functions, and then in the case of a tie, FEP was used to determine the ordering of the remaining cases. There is no mention of how many such ties occur in practice. In their results, the authors discovered that statement FEP worked best overall, and had to conclude that the inclusion of fault-proneness information had not significantly aided the fault detection

Technique	**Description**
Baseline	
Original	The order in which test cases are specified by developers
Random	Randomly arranged test cases
Coverage-Based	
Total Statement	Total coverage strategy at the statement-level granularity
Total Function	Total coverage strategy at the function-level granularity
Additional Statement	Additional coverage strategy at the statement-level granularity
Additional Function	Additional coverage strategy at the function-level granularity
FEP-Based	
FEP Statement Total	Ordered by the Fault-Exposing Potential score of each line
FEP Statement Additional	Ordered by the Fault-Exposing Potential score of each previously uncovered line
FEP Function Total	Ordered by the Fault-Exposing Potential score of each function
FEP Function Additional	Ordered by the Fault-Exposing Potential score of each previously uncovered function
Fault-Index	
Fault-Index Function Total	Ordered by the fault-index score of each function covered by a test case, summed up
Fault-Index Function Additional	Ordered by the fault-index score of each previously uncovered function covered by a test case
FI-FEP Based	
FI-FEP Function Total	Ordered by the combined FI/FEP score for each function covered by a test case
FI-FEP Function Additional	Ordered by the combined FI/FEP score for each previously uncovered function covered by a test case

TABLE 2.6: Description of techniques studied in Elbaum et al. [38]

process. Even comparing function FI to simple function total coverage produced no significant differences. Thus, the authors were forced to conclude that FI had not really worked. One conjectured reason for why FI had performed so poorly was that a lot of the faults that had been used in the experiments were simple, one line code changes, which led the FI score to be low.

2.11.2 Diversity-Based Approaches

One of the key concepts of achieving high coverage as early as possible in a prioritised test suite is to use a highly diverse set of test cases. Since a lot of developer-written test cases will have similar execution paths through the program, it is possible to achieve high coverage by creating clusters of test cases that cover similar parts of the program, then select a representative test case from each cluster additively.

In general, distribution-based test case prioritisation is based on creating a representation of a test case that can be compared with all other test cases. An example of this is used by Yoo et al. [58], where the representation of a test case would be a binary string, where each 1 or 0 represented a statement in the programs source code. While this can result in highly complex binary strings for large programs, it is very easy to compare strings using the Hamming distance. Once each test case has its representation, a clustering technique can be used to create k clusters of test cases, where each cluster represents the most similar test cases based on their representation. In both Yoo et al. [58] and Carlson et al. [59], an agglomerative clustering technique was applied, whereby n clusters of one test would start off, and at each iteration the most similar two test cases would be added to a new cluster, until there is k clusters.

Once the clusters have been formed, a prioritisation process takes place within the clusters, to find a better ordering for each test case that defines a cluster. In Carlson et al. [59], this process is done using a code coverage approach, a code complexity metric, fault history information or a combination of complexity and fault history. In addition to this process, Yoo et al. [58] performed an additional step of prioritising all clusters, based on a representative test case for each cluster, leaving them with an ordered set of ordered clusters. Test cases are then selected from each cluster iteratively, switching cluster after selecting a single test case, to produce a prioritised test suite.

In their experiments, Carlson et al. found that using a clustering approach to aid test case prioritisation proved effective on an industrial piece of software. In all pairwise comparisons of techniques (coverage vs clustered coverage, fault-based vs clustered fault-based etc), the clustered version outperformed their non-clustered counterpart in terms of Average Percentage Faults Detected of the prioritised suite. The authors also investigated the effect that a shorter running time had on the prioritised suites, finding that in all cases reducing the suite running time by 25 or 50% resulted in fewer faults being missed by the prioritised suites, implying that the suites were better suited in lower run times. When reducing by 75%, three out of the four prioritisation treatments were better on clustered versions than non-clustered versions.

Dickinson et al. proposed an approach to clustering test cases based around the observation that failed tests are often observed in small clusters [60]. Their method, entitled *failure pursuit*, repeatedly selects the k nearest neighbours of any failures found by running the test suite, spreading to the k nearest neighbours of any failures, until no more failures are found. This approach to clustering test cases exploits the idea that failing tests cases are often small groups of ideal candidates for clustering. In Dickinson et al. [60] and Leon et al. [61], failure pursuit performs reasonably well, justifying the authors' original concept. Leon and Podgurski authored a comprehensive review of clustering techniques, amongst other distribution-based approaches, comparing them with coverage-based approaches in a direct comparative empirical evaluation[61]. One of the key findings of this study was that distribution-based approaches and coverage-based approaches are good at detecting different types of faults, and as such they can be complementary. In their work, the authors investigated different combinations of clustering and coverage-based approaches, and found the best combination for their subjects was basic coverage + one-per-cluster sampling + pursuit + random.

2.11.3 Human and Expert Knowledge

The most likely people to know the most important test cases in a test suite are the people who wrote them. Developers have knowledge about the domain they work in, and the test cases that they write, including which ones cover important parts of the code, which ones cover error prone parts of the code, and which ones are simply there to boost code quality metrics. In an ideal world, test case prioritisation would be performed by developers simply deciding which tests they want to run first. In fact, developers perform test case prioritisation a lot in their every day work. Every time a developer runs a small subset of tests relating to a piece of code that they just changed, they are prioritising the best tests to ensure that their product still works in the shortest amount of time.

Unfortunately, for full suites of 2000 or more test cases, it is infeasible to ask developers to sift through each individual test case and rank it comparatively to every other test case. Tonella et al. proposed a machine learning algorithm called the Case-Based Rank (CBR) system, which combines automatic approximation of test case ranks (from other metrics such as coverage, fault-proneness etc.) and user knowledge into a machine learning model [62]. Given two test cases, the user is asked to judge which one should be given higher priority. They are not asked to give a justification for why they gave a test higher priority. There is also no assumption of any relationship between priority - that is to say it is possible that $a > b$, $b > c$, $c > a$. Given a test suite of size n, a user will be asked to make m comparisons between test cases to aid the prioritisation process. The number of comparisons required to perform CBR depends on the number of 'difficult' comparisons that have to made (e.g. where other metrics result in a tie between two test cases). In the case where users are being asked to provide judgements on pairwise comparisons, there is a trade off present between the number of comparisons and the quality of the produced model. Asking the users for more judgements means that the model will be better, but requires more manual effort. The results of a study into CBR indicated that there was potential for machine learning to become an important factor in test case prioritisation, however, there were also clear issues with scalability. Even for test suites under 100 tests in size, the number of comparisons asked for grew from 15 to over 700, with the results being presented on a logarithmic scale, clearly indicating this issue.

One attempt at reducing the amount of work done by 'experts' in test case prioritisation is Yoo et al. [58]. In their work, the authors combined the distribution-based test case prioritisation clustering technique with expert knowledge to produce a new approach. Using the Analytic Hierarchy Process, a matrix of test comparisons are built up, scaling from one to nine, depending on the users response to a comparison. In cases where no user response is available, coverage information is used to fill the matrix, but on a much less extreme scale, allowing user responses to dominate the matrix. In this instance, the clustering is performed to reduce the number of comparisons required from the user, making the technique more scalable to larger programs and larger test suites.

2.12 Metaheuristic Approaches

All of the previously described approaches are referred to in the literature as *heuristic* approaches. A heuristic approach involves there being a way of ranking each possible test case in a test suite such that the *next best* test case can always be selected and added to the prioritised suite. Heuristic approaches have been shown to be effective at producing test suites with higher rates of fault detection.

However, heuristic approaches rely on a fundamental assumption that each test case can be ranked in terms of some function measuring effectiveness. Heuristic approaches will often result in the same ordering being produced given multiple runs of the same

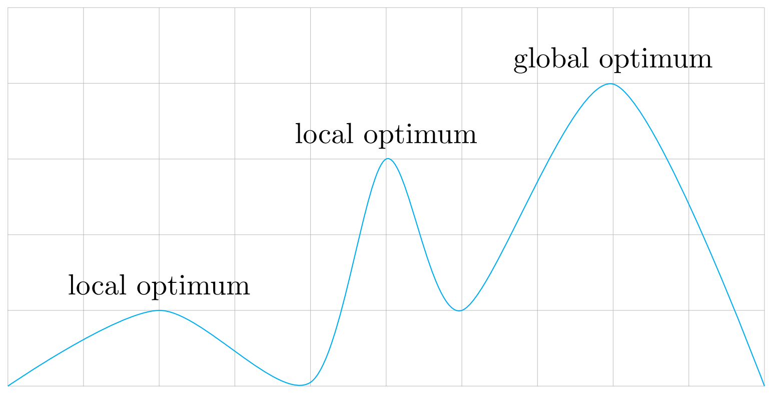


FIGURE 2.8: An example of a local optimum problem faced in a local search

algorithm, although there can be some randomness in the case of ties. Metaheuristic approaches aim to explore the search space (the space containing all possible solutions) of the problem efficiently using some starting point and repeatedly making changes before evaluating the current solution according to a fitness function [63]. A fitness function can be anything that takes a possible solution and returns a score approximating where the candidate solution fits in comparison to the global optimum. Changes made to candidate solutions in a metaheuristic search should contain no assumptions, and as such the only guide for a metaheuristic search should be the fitness function.

2.12.1 Local Search vs Global Search

Metaheuristic searches aim to make small changes to candidate solutions and improve the score returned by the fitness function. In complex problems such as test case prioritisation where there are $n!$ possible different orderings for n test cases, the search space of possible solutions is incredibly complex, with lots of points at which fitness improves and many where fitness deteriorates. A local search generally considers a single candidate solution and nearby solutions to increase fitness. In the case of test case prioritisation, a nearby solution to a candidate ordering would be one including one change (two test cases swap positions). The nearby neighbour with the best fitness becomes the new candidate solution, repeated until either an optimal solution is found or a search budget (a time limit or number of repetitions) has been used up. One of the downsides of a local search is that sometimes a *local optimum* is found, which represents the best possible solution in a set of nearby neighbours, without representing the best solution available in the full search space. An example of this is given in Figure 2.8, where two local optimum are visible that represent a sub-optimal solution in the search space.

A number of improvements have been made to local search algorithms in order to help them try and find the global optimum in a search space. One of the improvements involves using a population of candidate solutions rather than a single candidate solution. By having many starting points in a search algorithm, it is possible that more of the search space can be explored. Some of the population may reach locally optimal solutions, however it is also possible that other candidates will reach better a better part of the search space.

2.12.2 Genetic Algorithms

Genetic Algorithms (GAs) are an example of a population-based global metaheuristic search [64, 65]. Genetic Algorithms use a Darwinian theory of Evolution in order to find a candidate solution. In keeping with the biological name of this approach, candidate solutions are often referred to as *individuals* or *chromosomes*. Initially, several possible individuals are selected from the search space, and at each iteration, some individuals are chosen to be *evolved*. An evolution can take a number of forms, but fundamentally represents some change in the selected individual. After an individual has been evolved, if it is fitter than some existing candidates then it is added to the population. As a result, the fitness of the population should generally increase over time.

Two of the common evolutions that are used in GAs are mutation and crossover. A mutation in a genetic algorithm represents some change to an individual. In the case of test case prioritisation, where test suites are individuals in the population, this may mean switching the position of two test cases. Crossover means taking part of the solution from two different individuals (referred to as *parents*) and combining them to create a new *offspring* individual that is contains parts from both parents, while representing a new part of the search space.

Genetic Algorithms aim to mitigate the local optima problem in several ways. Firstly, by including many individuals in its population, and ensuring the initial individuals are highly diverse, the genetic algorithm should have a large number of points in the search space to work from. Additionally, if points in the search space are represented next to each other to represent 'nearby' solutions (i.e. one mutation away from each other), then using crossover should allow for more free movement around the search space. Crossover represents a large change from both the parent individuals selected, meaning that it can be a completely different point in the search space. Thirdly, the selection process for choosing individuals for crossover and mutation is not a purely random process. In many GA implementations, the selection process is biased to be more inclined to select fitter individuals for evolution, which should speed up the fitness improvement of the population. However, even a biased selection process still has the capability of selecting weaker individuals in the population, meaning that it is possible to escape local optima by using weaker individuals.

Algorithm 1 Genetic Algorithm

```
Require: Selection Function $s$, Mutation Rate $m$, Crossover Rate $c$
 1: Randomly generate or seed initial population $P$
 2: while Stopping Condition Not Reached do
 3:     Select Parents for Crossover according to $s$
 4:     Recombine parents according to $c$
 5:     Mutate offspring according to $m$
 6:     Evaluate fitness of offspring following changes
 7:     Select individuals for next population $P'$
 8:     $P \leftarrow P'$
 9: end while
10: return $p \in P$ such that $(\forall p' \in P)(p \neq p')[f(p) \geq f(p')]$
```

2.12.3 Crossover

One of the most important concepts in maintaining diversity in a genetic algorithm search is crossover. Crossover is the combining of two parent individuals to create two new offspring individuals that contain partial solutions from both parents. There are many ways in which crossover can be implemented, including single-point, two-point, partially matched crossover (PMX) and three-parent crossover.

Single-Point Crossover

Single-Point Crossover is a simple crossover technique that simply splits the parent solutions at an arbitrary point n, where $1 \leq n \leq k$, and takes the first n genes from parent one, and the remaining $k - n$ genes from parent one. In cases where there is not a pre-defined set of genes (for example a string searcher) may take partial solutions from both parents without having to worry about keeping any structure or defined set of genes. For example, consider "hello" and "world" as genes, with an intersection point of one, the two offspring can be "horld" and "wello" without having any impact on the potential usefulness of the solutions. However, in test case prioritisation and other sorting-based problems where a genetic algorithm is applied, it is necessary to keep a set of values (i.e. each test case should appear once and only once in a candidate solution). In this case, crossover works differently. The first n solutions are taken from parent one, and the remainder of the solution is chosen by iteratively selecting the remaining test cases from parent two in the order they appear.

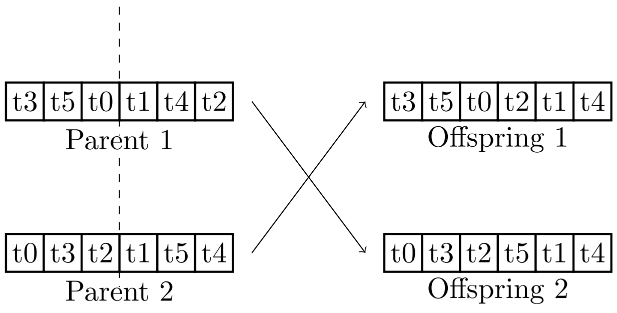


FIGURE 2.9: Single Point Crossover

In Figure 2.9, the intersection point is three. To create offspring one, the first three test cases are selected from parent one and copied over to the new individual (**t3 t5 t0**). Since each test case can appear once and only once, the remaining test cases are selected by iterating through the test cases in parent two, seeing if they exist in offspring one and adding them if not. Since **t0** and **t3** already exist in offspring one, the first selected test case is **t2**, then **t1**, ignoring **t5** and finally selecting **t4**

Two-Point Crossover

Two-Point Crossover is very similar to Single-Point Crossover, with the only difference being that two intersection points are chosen. Again, for some problems this is more conceptually simple than for others. Considering “hello” and “world”, with intersection points 1 and 4, the offspring become “horlo” and “welld”.

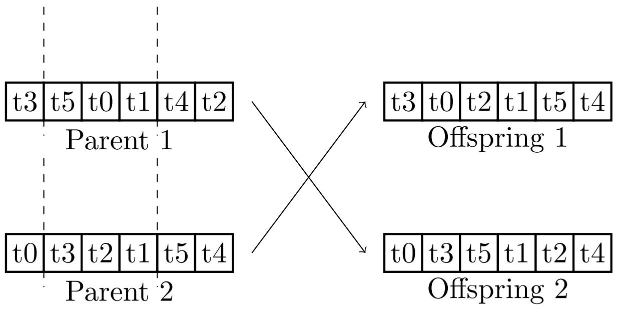


FIGURE 2.10: Two Point Crossover

In Figure 2.10, there are two intersection points (one and four). To create offspring one, the first element of parent one is automatically added (**t3**), since it is before the first intersection point. After this, the elements of parent two are considered. **t0** is added, **t3** is ignored since it is already in offspring one, then **t2** and **t1** are added. Finally, the remaining test cases are added from parent one again. The only two remaining cases are **t5** and **t4**, which are added in the order they appear in parent one.

Uniform Crossover

Uniform crossover works in a slightly different way from single and two-point crossover. Rather than taking a set of points at which to split the parent genes, uniform crossover takes a ratio of genes that should be included from each parent, and selects each gene in the offspring by applying the ratio. For example, if the ratio is 0.5, 50% of the genes from each parent should be included in the new offspring.

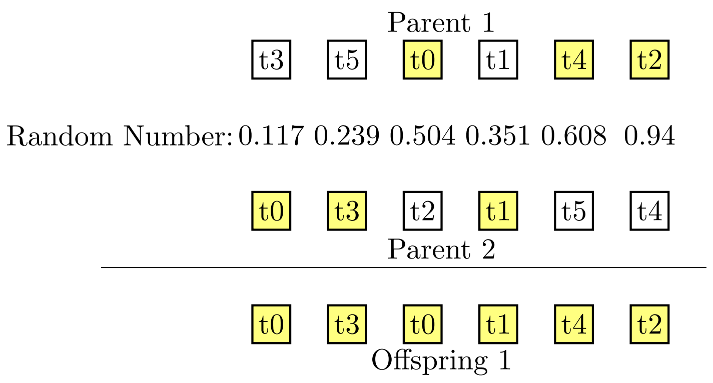


FIGURE 2.11: Uniform Crossover

In the example given above, the ratio is set as 50% from each parent, so random numbers that are generated below 0.5 will select the gene from parent one, while numbers above 0.5 will select from parent two. However, as with other crossover strategies, there is a clear issue when using a fixed set of genes. In the example shown, **t0** is selected as both the first and third gene of the resulting offspring, while **t5** is never selected. One of the options to deal with this is to select only genes that are not already selected for the new offspring. However, this would mean that if both genes are already in the candidate solution, there can be missed selections. Alternatively, it is possible to replace the duplicate genes with the unselected genes, either in place or at the end of the offspring (i.e. place **t5** in either gene three or gene six and shift the remaining test cases forward).

Three Parent Crossover

Traditionally, crossover takes two parent individuals and produces offspring individual(s), much like its biological equivalent. By taking attributes from both parents into account when creating the offspring individual, there is room for a lot of diversity in the produced offspring, allowing the search to avoid local optima. One attempt at creating additional diversity in the population is to use three parents for creating new offspring. This concept is described in Sivanandam et al. [66], and produces offspring by comparing the bits in parents to create the offspring.

Partially Matched Crossover (PMX)

Partially Matched Crossover (PMX) is a technique for tackling the travelling salesman problem described in Sivanandam et al. [66] and implemented in Yuan and Li [67], whereby each city should be visited once and only once. This makes it extremely relevant for test case prioritisation problems, since they are parallel to the travelling salesman problem. In PMX, the concern is with identifying *pairs* of test cases from two parent individuals that will be swapped around in the offspring gene. Similar to two point crossover, two intersection points are identified, n and m. With parents P_1 and P_2, $m - n$ pairs are identified $\langle P_{1i} P_{2i} \rangle, n \leq i \leq m$. For each identified pair, those genes swap places in the offspring solution. As a result, the offspring is similar to the parent, with the pairs of test cases swapped around.

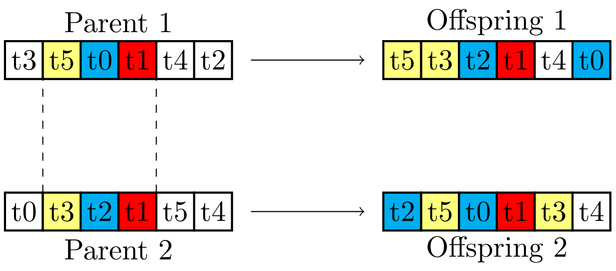


FIGURE 2.12: Partially Matched Crossover

In Figure 2.12, three pairs of test cases are identified, $\langle t5, t3 \rangle$, $\langle t0, t2 \rangle$ and $\langle t1, t1 \rangle$ (obviously t1 can be ignored, since it cannot swap with itself). In the offspring, the ordering from the corresponding parent will be taken (i.e. offspring one takes ordering from parent one), but with the pairs of test cases switched. This results in a new ordering that contains the exact same genes as the parent, but in a new ordering [66]. Crucially, for test case prioritisation, this is the only form of crossover that can be utilised, any crossover strategy must guarantee that the offspring have the same genes (test cases) as the parents, and the other crossover techniques will result in offspring that contain duplicate test cases and do not have all the required test cases.

2.12.4 Genetic Mutation

Genetic Mutation (not to be confused with program mutation described in Section 2.9.1) in metaheuristic algorithms refers to making some small, observable change to the state of one or more of the individuals contained in the population. Generally speaking, mutations are simple, non-guided changes that alter one or more properties. It is important for mutations not to be guided, since guiding mutations can lead to hitting a small part of the search space than general mutations. Also, if mutations are guided, there is an implication that there is a "good" way of changing an individual to get to the solution, in which case a metaheuristic is unsuitable.

A simple example of a mutation is to consider the string "hello". There are a number of mutations that can be applied to this string, including the addition of a letter, removal of a letter or changing of a single letter. These mutations are examples of how the nearby neighbours of an individual can be explored by a metaheuristic. In test case prioritisation, a mutation can *only* be the re-ordering of some test cases. If test cases can be added or removed, then the system becomes a metaheuristic for test suite minimization. It is expected that there will be a mutation rate property that controls how many mutations should, on average, be performed for every evolution in a metaheuristic search, and once it has been decided that a mutation will take place, each gene can be mutated with probability $1/n$, where n is the total number of genes, meaning there should on average be one mutation.

2.12.5 Properties required for Metaheuristic Search

Not every problem is amenable to a search-based approach. As described by Harman [64], there are a number of properties that a problem must have in order to be approximated using a search-based approach:

1. A representation of the problem that can be manipulated. In test case prioritisation terms, this means that there must be a way for us to represent a test suite that allows us to manipulate the ordering of the test cases.
2. A fitness function
3. A set of manipulation operators, such as crossover or mutation as described above.

2.12.6 Challenges with Metaheuristic Searches in test case prioritisation

One of the main problems in implementing a genetic algorithm in test case prioritisation is the difficulty in finding an effective *fitness function.* One field in which genetic algorithms have been successfully applied is test suite generation [68–70]. Test suite generation is the writing of test cases by a computer rather than by a human. Using a genetic algorithm to perform test suite generation usually involves a combination of coverage goals as a fitness function [68]. This works particularly well in test suite generation because as new test cases are added, or as new statements are added, the impact of these statements on the fitness can easily be calculated. Furthermore, the evaluation of how effective the generated test suite is usually measured in the same way as the fitness function. This situation is not possible in test case prioritisation. Firstly, since it cannot be known whether a test case reveals a fault or not, fitness functions cannot easily evaluate the fitness of an ordering of a test suite. In many previous studies, this has led to an assumption of coverage being a surrogate for faults [12]. This results in the APxC metric (Average Percentage of x Covered), where x can be a number of different coverage granularities. Moreover, in test case prioritisation, in order to measure effectiveness, the faults must be known. This means that increasing the fitness of a genetic algorithm will not necessarily result in an ordering that improves fault detection.

2.12.7 Basic GA

The simplest form of search-based technique is proposed for test case prioritisation in Li et al. [12]. This implementation of a genetic algorithm uses Average Percentage of Statements Covered (APSC) as a fitness function, since it is unrealistic in practice to know the value of $APFD$, and represents mutation as a single change to the test suite ordering, while a crossover between two test suites is easily achievable via the techniques discussed in Harman et al. [71]. In their experiments, Li et al. discovered that a GA could perform equally as well as coverage-based techniques such as additional coverage, where there was no significant difference between the additional coverage/genetic algorithm $APFD$ scores for small programs. However, one of the things this paper failed to analyse was the change in fault-detection capabilities of the test suites subjected to prioritisation via a genetic algorithm. No faults were analysed during this study, meaning that even though the GA achieved similar levels of coverage to the additional greedy algorithm, there is no indication of the impact this has on fault-detection.

One of the key points of this study into the effectiveness of genetic algorithms in the application of test case prioritisation is that there is scope for many different approaches to evaluating the fitness of a test suite. This paper encouraged further research into different configurations for genetic algorithms.

2.12.8 Weight-Based Genetic Algorithm

One way of modifying the genetic algorithm for test case prioritisation is to consider multiple objectives. While a single-objective genetic algorithm only focuses on one element of candidate solution fitness, such as Average Percentage of Statements Covered in Li et al. [12], Weight-Based Genetic Algorithms allow multiple fitness functions to all contribute to an overall fitness value for a candidate solution, with each fitness function assigned its own weight. Murata and Ishibuchi [72] suggested a genetic algorithm with random weights assigned to each fitness function. Since no fitness function should be objectively better than any other fitness function, using random weights should lead the solution towards a Pareto optimal score, rather than a score that satisfies all fitness functions without exceeding in any particular way.

The equation for calculating the Random Weight GA (RWGA) fitness is given in Equation 2.6, where $f_1, f_2, ..., f_n$ is a set of fitness functions and w_i is the weighting of fitness function i.

$$fitness = \sum_{i=1}^{n} f_i \times w_i \tag{2.6}$$

In a 2006 tutorial paper on multi-objective genetic algorithms, Konak et al. [73]. investigated the use of the RWGA. While the implementation of the RWGA is simple, and the transition from single objective to multiple objective can be easily achieved using RWGA, the candidate solutions can suffer when the Pareto front is non-convex. That is to say, if the trade off between the fitness functions is not a standard curve over all dimensions of the Pareto front, the RWGA can produce worse solutions.

In a 2016 study, Wang et al. [74] implemented a number of genetic algorithms for the purpose of prioritising test cases, including a RWGA implementation, and compared the results of RWGA with a Random Search on an industrial piece of software. The authors created 4 fitness functions, including Time Taken, Prioritisation Density, Test Resource Usage and Fault Detection Capability. It could be argued that using a Random Search as an industry standard is not necessarily representative of the actual state of the art in test case prioritisation, however, for industrial software, it is unlikely that any test case prioritisation measures are in place. Importantly, in comparison to the other algorithms investigated in the study, the RWGA outperformed every other algorithm studied.

2.12.9 NSGA-II

NSGA-II is a multiple-objective genetic algorithm based on the idea of Pareto dominance [75]. A Pareto representation of a problem involves having a dimension on a multi-dimensional graph for each objective that is covered. If any solution is beaten in all dimensions, it is considered to be Pareto-dominated. The NSGA-II algorithm outputs a set of non-dominated solutions using a ranking algorithm to determine which solution is best given the various dimensions. Wang et al. [74] implemented a variety of different genetic algorithms, including the NSGA-II, and showed that the NSGA-II could outperform a random search on an industrial piece of software.

Yoo and Harman [76] also implemented the NSGA-II algorithm in their work on test case selection. In this study, an extension to the NSGA-II was also suggested, called vNSGA-II, which kept sub-populations separate from each other in order to try and widen the Pareto frontier. Yoo and Harman used two objectives, coverage and cost, to form the Pareto landscape. From this, they determined that in multiple-objective settings, the additional greedy algorithm does not perform as well as multiple-objective alternatives, including when 3 objectives were used. In addition to this, the authors noted that the additional greedy algorithm formed a part of the Pareto frontier, indicating that the results from both the additional greedy algorithm and NSGA-II could be combined to form a better solution in terms of Pareto optimality.

2.12.10 Epistatic Genetic Algorithm

Yuan and Li [67] presented the idea of using epistasis in a genetic algorithm. Epistasis is a biological concept revolving around the impact of genes based on the values of other genes. For example, the gene that determines the colour of someone's hair is irrelevant if there is also a present gene that means the person is bald. Thus, some genes matter more than others.

In some cases, there will be an ordering of test cases that is a subset of the entire test suite that achieves the full coverage. This does not mean that other test cases are redundant, but once the maximum coverage has been achieved, there is no impact on coverage metrics such as APSC no matter what the order of the remaining test cases are. Given this, the authors present the idea of the Epistatic Test Case Segment (ETS), which is the permutation of all test cases starting with the first test case and ending with the test case that reaches the maximum coverage for that permutation.

As a result of using the ETS, the crossover function for an Epistatic GA changes. Normally, in single point crossover, the first n test cases are kept from parent 1, and the remaining $k - n$ test cases are selected in order from parent 2. In the case of epistasis, changing the order of two test cases that exist outside of the ETS will have no impact on the fitness. As the genetic algorithm evolves, the ETS should become shorter, and so the chances of selecting a point outside the ETS becomes higher. To change this, epistasis results in the keeping of the last $k - n$ test cases from parent 1, and filling the first n test cases from parent 2. This means that chances of changing the ETS are higher.

In their experiments, Yuan and Li found that coverage could be achieved faster using an epistatic genetic algorithm instead of a standard genetic algorithm. When compared with Partially Matched Crossover (PMX), the epistatic crossover required fewer iterations while still achieving a higher coverage value. There is, however, no indication from the available literature whether this was studied in relation to fault-detection capability, rather than simply achieving higher coverage at an earlier stage.

2.12.11 Memetic Algorithms and Hybridisation

One of the advantages of global searches over local searches is that they are less likely to encounter *local optimum* problems. A local optimum occurs when there are no neighbours to the current solution that provide an increase in fitness, but the best solution has still not been found. An example of a local optimum is given in Figure 2.8. Global searches attempt to overcome the local optimum problem by using a larger population of seeded individuals such that individual fitness can increase to find the global optimum, and by allowing crossover and mutation, which can move from one part of the fitness landscape to a completely different part within the same population

Memetic algorithms combine the global search of a genetic algorithm with a local search. Since a global search will quite often jump around the fitness landscape, the application of a local search is intended to refine individuals to guide the search to its optimal solution before it jumps off to another section of the landscape. Obviously, global searches cannot result in a lower fitness score, so sometimes this diversion to a separate part of the landscape can be beneficial. However, in cases where the optimal solution is on the same "hill" as the current candidate, sometimes a global search will divert away. Moscato and Cotta [77] give a description of a memetic algorithm.

Harman and McMinn [78] and Fraser et al. [79] implemented a memetic algorithm for test suite generation, showing that the technique was applicable to search-based testing problems. As with many techniques in regression testing, it appears that the use of memetic algorithms is also applicable in test case prioritisation. Nejad et al. [80] implemented a memetic algorithm for regression test case prioritisation. In their experiments, 4 different local searches were implemented to complement the genetic algorithm being used, a Hill Climbing algorithm, a Random Iterative Improvement Algorithm, a Stochastic Local search and a Simulated Annealing approach. Each of these local searches, when combined with the use of a genetic algorithm to form a memetic algorithm, provided an improvement in fitness at the same number of iterations when compared to a genetic algorithm. Specifically, it was noted that the simulated annealing approach was the highest increase in performance, which may be due to the fact that simulated annealing can accept a temporarily worse solution according to a probability, while hill-climbing can only accept better solutions, terminating if a better solution can not be found.

2.12.12 Hypervolume Genetic Algorithm

In previous studies on using metaheuristics for test case prioritisation, surrogates for fault detection rate have often been used to represent fitness. These types of coverage are Area Under Curve (AUC) metrics, and represent simple 2-dimensional fitness landscapes. By increasing the number of dimensions from 2 to n, it becomes possible to represent a much larger number of fitness functions, thus making the algorithm able to search for multiple objectives simultaneously, using a hypervolume to represent fitness. As with the NSGA-II algorithm, this algorithm relies on finding non-dominated solutions.

Di Nucci et al. [81] created a hypervolume based genetic algorithm (HGA), using 3 objectives to form the multi-dimensional fitness landscape. The objectives used were statement coverage, execution cost and previous fault detection information, while the implemented algorithms included the hypervolume-based GA and a cost-aware additional greedy algorithm developed by Yoo and Harman [76]. The authors compared the outcome of the $APFD_c$ metric from the additional greedy approach with the HGA, finding that for both 2 and 3 objective treatments, HGA outperformed the additional greedy approach for 4 out of 6 studied programs. In addition to this, the HGA average running time was significantly lower across all projects, indicating the potential for this technique to be adopted as a quicker alternative than heuristic approaches.

2.12.13 Diversity GA

Another approach to multi-objective test case prioritisation is presented in Panichella et al. [82], where the authors aim to improve a genetic algorithm by including diversity into the search process. The authors state that previous studies into the use of multi-objective GAs for test case selection may have been subject to a process called *genetic drift*, whereby all produced offspring are too similar to their parents and so the population cannot effectively evolve to aid fitness. This leads to a situation where local regions cannot be escaped.

In order to introduce diversity, the authors modified the NSGA-II genetic algorithm, adding in the concept of an evolution direction, which represents the general trend being observed by the current set of individuals, and an orthogonal population, which are individuals representing a change from the current evolution direction. According to an interval k, diversity would be injected by calculating the evolution direction and orthogonal population of the current solution, including individuals from both populations in the new solution.

In order to compare their technique with other multi-objective GAs, Panichella et al. [82] decided to use the vNSGA-II presented in Yoo and Harman [76], alongside an implementation of a multi-objective additional greedy search also presented in Yoo and Harman [76]. Their empirical results indicated that for two and three-objective problems, the proposed technique was able to outperform vNSGA-II.

2.13 Mutation Faults in Empirical Studies on Test Case Prioritisation

In order to assess the effectiveness of any given test case prioritisation technique, it is important to have programs that contain faults, in order to see whether these faults can be detected faster as a result of test case prioritisation. It is possible to obtain faults in one of three ways: the first is to obtain real source code or access to real bug repositories such as Atlassian JIRA[4]. This process results in the discovery of real software faults, which may aid in the validation of results, but is highly costly in analysing what is a fault and finding the changes in source code that can be used to patch the fault. In addition to this, real software repositories are not generally fond of their bugs being widely known, so this information is often private. Finally, this technique normally results in a relatively low number of faults being discovered. It is rare to find studies that use real faults because of the difficulty in finding the faults, and the ease of the alternatives. However, examples of studies on real faults are included in Kochhar et al. [43], Just et al. [83] and Di Nardo et al. [40].

The second method for obtaining faults is to manually seeding them into a program. This technique involves using people who know a reasonable amount about programming, and asking them to manually create bugs in positions where real developers may accidentally introduce bugs. While this technique normally results in 'good' faults, which again helps

[4]https://www.atlassian.com/software/jira

prove the validity of the results, it also usually results in a relatively low number of seeded faults, and in large systems it can require a great effort from the developers involved to understand enough about the code to manually seed good faults. Many previous studies have included some usage of the Software-Artifact Infrastructure Repository (SIR), including [10–12, 38]. While SIR does contain some examples of real faults, a lot of the studies focus on the use of seeded faults, since this provides many more faults to work with.

The third method of fault-generation is to use mutation. Mutation is a quick and easy way to introduce large numbers of simulated faults into a program. There are many examples of tools that can generate such mutants, for example Major [16] or PIT[5].

Mutation tools often introduce large numbers of faults into programs. For example, Major has eight possible mutation operators, and if all are defined, any time it is possible to introduce a mutant, it will. On a version of the JFreeChart[6], with all mutation operators available, Major generated over 70,000 mutant versions of the program, which contains 2,205 tests [20]. As discussed in Section 2.9, in order to know exactly which test cases kill which mutants, all tests must be executed on all mutants, resulting in over 154 million test case executions. This results in a very high computational cost to perform mutation analysis on such large programs with so many mutants.

Do and Rothermel [49] performed an empirical study on the use of mutation faults in test case prioritisation experiments, in which they looked to replicate previous studies using mutation faults rather than seeded faults, in order to see whether similar results could be observed from the use of mutation faults as was observed when using seeded faults. In their experiments, Do and Rothermel used 4 java programs, generating between eight and 2907 mutants. Mutants were removed if they resulted in no output from the test suite for the version they were created on. Mutants were also only generated in parts of the code that had changed in the current version, making the prioritisation somewhat version aware, although tests were being prioritised based on their coverage information rather than FEP or any mutation-based heuristic. The authors made the following observations:

- There is more spread of results when using mutation faults rather than seeded faults. The conjectured reason for this is that, with a higher number of faults available, there is a higher spread of available $APFD$ scores. The authors suggested that repeating experiments with more seeded faults would mean there was a similar landscape of $APFD$ scores.

[5] http://pitest.org
[6] https://github.com/jfree/jfreechart

- The program with the smallest number of seeded faults, *jtopas*, with just 8 faults, had the largest variance from the seeded fault experiments. This gives rise to doubt as to whether mutation is resulting in similar behaviour to real faults, since this experiment is what should be closest to using seeded faults, but actually it is the furthest away in terms of results.

- The authors detected that there was a relationship between the number of seeded faults and the results of prioritisation. The programs that were used for Experiment 2 had larger numbers of faults injected than the programs from Experiment 1, and provided more of a change from the results of using seeded faults.

Another study into the use of mutation faults in empirical studies was conducted by Just et al. [83]. While this study is not in the context of test case prioritisation, it is similar to Do and Rothermel in that it considers the difference between mutation faults and real faults. In their study, Just et al. used DEFECTS4J [84] to analyse whether or not mutation was a realistic representation of the kind of faults observed in real software. Firstly, Just et al. looked at whether real faults could be generated by using mutation tools, which resulted in observing that 73% of real faults could be coupled to changes made by mutation, with the most common mutation operators that matched real faults being conditional operator replacement (i.e. replacing "||" with "&&"), relational operator replacement (e.g. replacing "==" with ">=") and statement deletion mutants.

Secondly, Just et al. investigated the types of real faults that are not coupled to mutants. This question provides a valuable insight into the types of faults that can occur that are highly context-specific, or require expert knowledge in the target language in order to reproduce. Some of the examples presented in Just et al. [83] including changing `String::indexOf` for `String::lastIndexOf`, two methods that are semantically similar but not equivalent, and representative of a change that mutation could not introduce. For things like Strings and standard Java API it may be possible to know which methods can be semantically similar, but for user-created code it becomes much harder to recognize. In total, Just et al. [83] classified 95 faults that were not coupled with mutation techniques, indicating whether they could be fixed with an improvement to an existing mutation operator, required a new mutation operator, or were unrealistic scenarios for mutation to replicate.

Lastly, Just et al. considered the correlation between mutant detection and real fault detection. This work is similar to the work in Do et al. [49], although omitting the step of prioritising the test cases. In order to ascertain correlation between mutation score and fault-detection rate, the authors used automatically generated test suites, including at least one suite that would discover the fault, and at least one that would not. The scores of these test suites would then be compared to determine correlation. It was found that there is a positive correlation between mutation score and fault-detection rate, independent of the test suite coverage. The results of this study are significant since they show that test suites that have a higher rate of mutant detection (i.e. a higher mutation score) represent test suites that will also detect faults faster, in the context of real software faults.

2.14 Real Faults in Empirical Studies on Test Case Prioritisation

One of the key threats to construct validity across many studies in the field of test case prioritisation has been the lack of real faults used. Although there have been investigations comparing how real faults compare with mutation faults, including Do and Rothermel [49] and Just et al. [83], there have been very few studies that actually use real faults in the domain of test case prioritisation [19, 40]. One of the reasons behind this lack of empirical studies is the difficulty associated with obtaining real fault data. Most companies that produce software at the scale required for a large enough sample size do not have open-source repositories containing their code, since they run for profit. Even companies that do have open-source repositories often have no concrete identification of what is a bug. This requires researchers to analyse bug tracking repositories, identify the start and end-point for a bug, from being discovered to being fully fixed, associate the bug with a commit or series of commits.

Defects4J is a bug-repository containing five Java programs and over 300 real faults that have been encountered in software ranging from 22,000 to 96,000 SLOC, and containing from 2,205 to 7,927 test cases [84]. These bugs were mined from open source repositories such as JFreeChart[7] or Apache Math[8]. As discussed in Section 2.15, many studies have used a large number of faults to evaluate their techniques, which in itself is a threat to the validity of their results. Defects4J allows each individual bug to be checked out into its own working directory, with one real bug per working directory, to allow studies to be done involving that will be more representative of an actual use case in industry.

For the remaining chapters of this thesis, I use Defects4J as a source for subject programs. This may limit how generalisable the results of my research are, since there are many other datasets on which the proposed strategies could have been evaluated,

[7]`https://github.com/jfree/jfreechart`
[8]`https://github.com/apache/commons-math`

such as the PROMISE dataset[9], which has been frequently used as a source for subject programs in the field of defect prediction [85, 86]. Additionally, open-source datasets such as Eclipse[10] and Mozilla[11] provide a large number of real faults that could have been used to widen both the number of subjects and the complexity of the subjects used. One of the main motivations for choosing Defects4J is that it simplifies the experimental process to one a single source for subjects, which ensures that the process is consistent while still providing a range of subject programs. Additionally, Defects4J is an “all-in-one” package, including bugs that have been mined from each subject program using a documented bug-mining process. Defects4J also provides for each of its subjects a list of “trigger tests” (fault-revealing test cases), as well as including the Major mutation framework to allow the required artificial faults for Chapter 3. In order to collect the required information for additional subject programs, I would have needed to replicate each of these steps provided by Defects4J, with each step vulnerable to possible implementation errors.

Additionally, the Bugs.jar repository [87] provides a large dataset comprising over 1,000 real faults collected from eight open-source Java projects. Like Defects4J, this repository contains bugs that were mined using a documented process and, for each of the bugs included in the dataset, includes a patch that can be applied to represent the “fix”, and a log file containing the output from the tests that were collected at the time that the bug was present. This framework is a little less complete in terms of utilities than Defects4J, which provides command line tools to gather information in more useful ways, but is much more complete in terms of the range of bugs included. This repository only existed during the final period of my studies, and therefore the experiments had already been conducted using Defects4J and there was not enough time to repeat them using Bugs.jar.

Luo et al. [19] conducted an empirical evaluation using the Defects4J subjects, in which they compared the effectiveness of test case prioritisation strategies on real faults and mutants. In particular, this study aimed to evaluate the effectiveness of existing strategies on real faults, assess how representative mutants are of real faults in test case prioritisation studies, and investigate the properties of both fault types that can impact the performance of test case prioritisation. This study asks similar questions to the ones presented in Chapter 3 — notably, while the results obtained agree with the ones that were found in Chapter 3’s experiments, the published version of Chapter 3 was released before the work of Luo et al..

[9]http://promise.site.uottawa.ca/SERepository/datasets-page.html
[10]https://bugs.eclipse.org/
[11]https://bugzilla.mozilla.org/

2.15 Number of Faults used in Empirical Studies on Test Case Prioritisation

One of the often overlooked factors in almost all studies in test case prioritisation is the number of faults that are being evaluated. Since the most common use for test case prioritisation is in regression testing, it is not expected that a test suite will be finding multiple faults in a single run. As a result, experiments that are being conducted with higher numbers of faults may not be representative of the actual use cases for test case prioritisation. As an example, Elbaum et al. [38] generated 29 versions of each of their programs under study, with each version using a random number of independent faults from a *fault-base*, randomly choosing any number of bugs from one to the total number in the fault base. Hao et al. [41] used mutant groups in accordance with Do et al. [49] to create 20 different groups of five mutants. Malishevsky et al. [14] used seeded faults generated by students to insert between one and nine faults into each version of their subject program *emp-server*. Hao et al. [21] used a combination of seeded faults and mutation faults, again in accordance with Do et al. [49] to create between one and five faults for each program, originally generating five per program then removing any unqualified mutants (mutants that could not be detected by any test in the programs test suite). These papers are representative of the landscape of empirical studies in test case prioritisation.

The number of mutants used in an empirical study can have a large impact on the results. This is partially presented in Do et al. [49], where the authors conjecture that there is a relationship between the number of faults inserted and the performance of test case prioritisation techniques. This trend is particularly prevalent when using coverage-based techniques. Since coverage-based techniques are reliant on covering as much of the code as possible in the shortest amount of time, and the faults most susceptible to detection by coverage-based techniques exist in parts of the code exercised by high-coverage tests, using more faults increases the probability that at least one fault is exposed by chance. Detecting just one fault early in the execution can have a big impact on the *APFD* score.

Di Nardo et al. [40] contains an example of how the number of faults can also be used to present a much more convincing case for a presented technique. In their work, Di Nardo et al. used five versions of a program, containing a variable number of faults. In order not to bias their results, they multiplied the *APFD* score obtained by their prioritised test suite by the percentage of faults contained in that version. However, version one of their subject program contained 30 out of the total 37 faults being studied, meaning that version one contributed $\frac{30}{37} = 81\%$ of their overall results score. When combined with the fact that including more faults makes it more likely to achieve a good *APFD* score by chance, this makes the results of the study weaker.

2.16 Defect Prediction

Defect Prediction (or fault prediction [88]) is a technique aimed at estimating the likelihood of a particular file or function being faulty, typically using information from the source code and/or version control [18]. Predicting the location of faults in a program has the potential to be hugely beneficial for a test case prioritisation strategy, since test that cover the areas of the code that are *most likely* to be faulty can be placed sooner. Moreover, many of the previous studies on defect prediction have included real faults, and have shown the technique to be effective when using real faults. Therefore, in this section, I investigate defect prediction and its applicability as a test case prioritisation strategy. Chapter 4 of this thesis conducts an empirical evaluation of a test case prioritisation strategy that leverages defect prediction.

Graves et al. [23] studied a combination of nine aspects of software repositories that could be closely linked with defect occurrence. These included the number of lines of code, the complexity of the code, the number of past faults, the number of changes to files over the repository history, the average age of the code in a file, the organisation who developed the code, the number of developers who made changes, the connections between modules in the code, and finally developed a weighted time damp model that computes a modules fault potential by summing up previous changes, favouring more recent changes over older changes.

In their study, Graves et al. found that the number of changes to a file, and the number of times that a file has previously been faulty are good indicators of future faults. Notably, Graves et al. discovered that, once the number of changes was taken into account, models built around any measurable code quality metric (e.g. complexity, total lines of code) could not improve on the model.

Sliwerski et al. [89] proposed a method designed to identify both bug-introducing and bug-fixing commits. In order to identify a bug-fix, the approach, subsequently referred to as the “SZZ algorithm” after the three authors on the proposing paper, combines syntactic analysis of a commit message (e.g. looking for bug identifiers) and a semantic step using bug-tracking software information such as whether a bug identifier has been marked as fixed and whether the bug was assigned to someone who later committed code with the bug identifier. In order to identify a bug-introducing commit, the algorithm looks at all lines of code that were changed as part of a bug-fixing commit and identifies suspicious lines based on when bugs were reported. Using their algorithm, Sliwerski et al. conduct an experiment using two open-source projects: ECLIPSE and MOZILLA, comprising 180,000 commits. The authors conclude that bug-introducing commits are generally larger than bug-fixing commits, and amusingly conclude that in order to avoid bugs you shouldn't program on Fridays, since commits on this day accounted for the highest proportion of bug-introducing commits.

Similarly, Kim et al. [90] conducted a study in which they investigated fault prediction based on four hypotheses — if an entity was *changed* recently, *added* recently or *faulty* recently, it is likely that the entity will introduce faults soon. The fourth hypothesis is that if an entity was faulty recently, other "nearby" entities will *also* have higher chance of introducing faults. Kim et al. employed a cache-based system to store bug fixes and predict future bug fixes, achieving a prediction accuracy of between 73-95% at the file level, and between 46-72% at the function level.

Following this, Menzies et al. [24] proposed a prediction model based on 38 static code features such as McCabe's complexity [91] and the Halstead attributes [92]. Menzies et al. considered three learners from the WEKA toolkit [93] to identify the most discriminatory features and combinations of features. From their experiments, Menzies et al. were able to achieve a prediction accuracy of 71% and a false positive rate of 25%.

Zimmermann et al. [25] conducted an experiment on three versions of the Eclipse project, combining several code-based metrics at several granularities and calculating correlations between code features and fault existence, concluding that complex code is more likely to contain faults than less complex code.

Moser et al. [94] conducted a comparative study between the impact of code quality metrics and change metrics and their effectiveness on defect prediction. In their study, Moser et al. consider 31 code metrics, 18 change metrics, and three classifiers. From these experiments, the authors conclude that change metrics are a very positive indicator of defects, achieving an accuracy of $\tilde{7}5\%$, and a false positive rate of $< 30\%$.

Kim et al. [95] predicted defects on changes, rather than specifically on source files, providing a classifier with bug-introducing and clean commit messages in order to classify future commit messages as either bug-introducing or clean. In their experiments, the change-classification approach predicted bugs with between 64-92% accuracy depending on the project.

Hassan et al. [96] proposed two models for analysing code changes - the Basic Code Change (BCC) and Extended Code Change (ECC) models, which map previous changes to a complexity score and use these scores to predict the complexity of future changes involving specific files. From their experiments, Hassan et al. found that prior faults are a better indicator of future faults than simply modifications alone.

Rahman et al. [97] built upon the work of Kim et al., evaluating the “FixCache” approach described in [90], and proposing a naïve approach that simply ranked files by the number of times they had been involved with a closed bug. In their evaluation, they found the naïve approach to be roughly as effective as the previously proposed approach, such that in a blog post, Google Engineering Team referenced their usage of this information [98].

D’Ambros et al. [99] conducted an exhaustive evaluation of defect prediction techniques, taking these techniques and others and comparing them across a standard set of five applications. From this they conclude that the Moser et al. [94] approach is highly effective, and discover that the best performers overall are process, entropy of source code, and churn of source code metrics.

2.16.1 Machine Learning Approaches

In addition to these approaches, researchers have also employed machine-learning approaches to train defect prediction models. Wahono [85] conducted a systematic literature review of defect prediction approaches, in particular focusing on the methods and the datasets used in published studies. In this evaluation, Wahono includes 71 studies that almost exclusively use a model-based approach, with the most commonly adopted approaches being Naïve Bayes (used 14 times) and Decision Tree (used 11 times). A further study by Hall et al. [86] comprising 208 papers shows that Logistic Regression and Naïve Bayes are the two most popular modelling approaches, accounting for 56 and 33 papers respectively.

One such example is Okutan and Yildiz [100], who used a Bayesian network to predict the location of defects in 12 subject programs using eight code features to determine the likelihood that a class is faulty. The results of Okutan and Yildiz show that their approach is effective, and identifies the code features “Response For Class”, “Lines of Code” and “Lack of Code Quality” as the three most useful features for identifying defects. Conversely, the study shows that the “Number of Children” and “Depth of Inheritance Tree” are unreliable code features and should not be trusted to provide accurate predictions.

Ma et al. [101] proposed a technique for a cross-company defect prediction model, arguing that previous studies had used training data from a single company and therefore the results of the model were overfitted and would not generalise to other companies. In their experiments, Ma et al. train a defect prediction model using five subject programs and

17 code features and test their model on three versions of a different subject program. In their analysis, Ma et al. show that their proposed approach outperforms existing approaches when using cross-company subjects for training and testing.

He et al. [102] conducted an empirical study using a simplified set of metrics in order to evaluate whether a predictor built using a simple metric set can perform as well as those built using complex metric sets. The evaluation consists of 34 releases of 10 open-source projects and combinations of 20 different code features. In their work, He et al. discover that a model built using simplified metrics could perform nearly as well as one built using all metrics, as the simplified model was not significantly outperformed in terms of precision, recall or F-measure.

Bowes et al. [103] proposed a tool in order to aid the adoption of defect prediction by industry. The ELFF tool proposed in this paper is an IntelliJ plugin that visualises the results of defect prediction to developers by turning code that is likely to be defective yellow in the IDE. ELFF uses an implementation of the SZZ algorithm in order to collect historical defect information for a project through its version control system (SVN or Git), collects static code metrics and then performs defect prediction using one of a number of machine learning algorithms to calculate the probability that a class or a method is buggy.

Similarly, Borg et al. [104] provided an open implementation of the SZZ algorithm referred to as "SZZ Unleashed". This implementation collects 16 code features and includes an evaluation against the Jenkins dataset. The main contribution of this paper is the openly available implementation of the SZZ algorithm, in the hope that researchers will not need to re-invent the wheel and risk introducing mistakes in their own implementations.

There are two ways in which a defect prediction technique can report its dependent variable — a *categorical* classification will determine for a unit U in a code base whether it is "faulty" or "non-faulty" [86], whereas a *continuous* classification will often report the number of faults it predicts are contained within a unit of code [86]. In a categorical classification, it is easy to measure the precision (number of correctly reported faulty units vs total number of reported faulty units), recall (number of correctly reported faulty units vs total number of actual faulty units) and F-measure (combination of the previous two metrics). Classifiers that have a high *precision* will only report a few units as faulty, but the ones it does report will usually be correct reports, whereas classifiers with a high *recall* will report a lot of classes as faulty, identifying the actual faulty units but also misclassifying a number of non-faulty units as faulty. Such results are referred to as *false positives* — a defect prediction thought they were faulty but in reality they were not. Table 2.7 describes the four types of result that can be obtained from a categorical classification.

	Predicted faulty	Predicted non-faulty
Observed faulty	True Positive (TP)	False Negative (FN)
Observed non-faulty	False Positive (FP)	True Negative (TN)

Table 2.7: Confusion matrix

2.16.2 Parameter Tuning of Machine Learning Algorithms

One of the downsides of using machine learning approaches such as Naïve Bayes and Random Forest algorithms is that they have configurable parameters, the choice of which may have considerable impact on the performance of the classification accuracy. Tantithamthavorn et al. [105] conducted a study in which they employ four automatic parameter optimisation techniques to 26 classification techniques to observe the impact on classification accuracy. Previous work by Hall et al. [86] has shown that simply adopting the default parameters may lead to sub-optimal results. In their results, Tantithamthavorn et al. show that the performance of some classifiers can be boosted by up to 40% by optimising the parameters for the classifier. This impact is not consistent across all classifiers, with some benefitting more from the optimisation than others. This indicates that using the correct parameters is essential for both the reliability and the replicability [106] of studies on defect prediction.

2.16.3 Studies Linking Defect Prediction and Test Case Prioritisation

Finally, there have been a few studies that have linked defect prediction and test case prioritisation. Li et al. [107] considered 32 code features to identify sub-systems that were most likely to fail. Srikanth et al. [108] considered requirements reported by users as most likely to contain failures, while Wang et al. [109] proposed a test case prioritisation strategy based on software quality. Finally, Mirarab et al. [110] used software quality metrics in a Bayesian model that supported test case prioritisation. Notably, all of these studies use software quality metrics as a surrogate for fault existence, rather than looking at change metrics described by other papers.

2.17 Sentiment Analysis

In Chapter 4 of this thesis I present a test case prioritisation strategy that uses defect prediction information in order to identify classes that are likely to contain faults, and then test cases that cover those classes. In particular, I use the Schwa defect prediction tool [26], which mines Version Control Systems (VCS) repositories to identify information such as the number of changes a file has received, the number of unique authors who have made changes, and the number of fixes that are related to a file. The success of this strategy raises questions about other information that is relevant to VCS' that may also aid test case prioritisation. One such piece of information is a *commit message*. When a developer makes a change to a program and *commits* their work to a central repository, they must include a brief description about what has changed and why, such that other developers can understand the changes that have been made when they *pull* changes from the central repository to their local machines. If developers are particularly angry when they have made changes, it could be an indicator that they have worked on badly designed program files, which could also be an indicator of future bugs.

In Chapter 5 of this thesis, I conduct an evaluation of opinions in commit messages, in particular aiming to correlate strong opinions with faulty files or bug fixes. If there is a correlation between sentiment and faults (i.e. developers write more negatively about files that contain bugs), then a test case prioritisation strategy can be developed that leverages commit message information to reorder test cases.

When presented with a large body of text, such as a product review, the goal of sentiment analysis is to judge whether the person who wrote the review was feeling positive, negative, or neutral towards the product. For example, consider the following product review, taken from Liu and Zhang [111]:

```
I bought an iPhone a few days ago.  It was such a nice phone.  The
touch screen was really cool.  The voice quality was clear too.
However, my mother was mad with me as I did not tell her before I
bought it.  She also thought the phone was too expensive, and wanted
me to return it to the shop.
```

Looking at this review, it is clear to a human reader that the reviewer was happy with their purchase. However, it is impractical for humans to read every review written about a product to determine whether it was received well or not, particularly in cases where products may have thousands of reviews. Therefore, it is the task of sentiment analysis techniques to automatically infer sentiment through a variety of approaches.

Challenges

There are a number of key challenges in sentiment analysis that must be overcome in order for it to be effective. Firstly, analysers must consider negations, which invert the sentiment of surrounding words. For example, “good” is a positive sentiment while “not good” is a negative one. Recognising negators in bodies of text is very important as it could completely invert the polarity of the sentiment. However, since negations come in many forms (e.g. “un”, “not”, “n’t”), recognising them correctly is an issue.

Secondly, there is the issue of identifying the subject who holds a particular opinion. Using the earlier example, there are two people mentioned during the review — the reviewer and the reviewers mother. While the reviewer holds a positive sentiment, the reviewers mother holds a negative one. It is important when analysing text data to correctly understand the subject who holds a particular opinion in order to avoid confusion.

Furthermore, reviewers may speak sarcastically in reviews. Consider a review for an airline that states “I *REALLY* enjoyed the 6 hour delay”. This is a positive sentiment (enjoyed) accompanied by an intensifier, which is a word that increases the strength of the previous work (really), but it is referring to something negative (a delay). In this instance, a sentiment analysis strategy has to figure out that this is a sarcastic review and that the reviewer is speaking negatively.

Applications of Sentiment Analysis

The primary purpose of sentiment analysis is to help people to gather a picture of a general reaction when there are a large number of responses and analysing each one individually would be too large a manual effort — for example understanding discussion about a topic on Twitter [112, 113], understanding movie reviews where no specific rating is given [114, 115], and understanding the feeling towards political figures [116]. In short, the online profile of millions of people can be analysed in order to understand their emotions, what they feel strongly positive about and what they feel strongly negative about. There is possibly no more famous case than that of Cambridge Analytica, who harvested the profiles of 50 million Facebook users and used the information gained in order to influence people towards supporting campaigns it was working on, notably the campaign for Britain to leave the European Union and the Donald Trump election campaign [117–119].

Social networks, specifically Twitter[12] provide ideal candidates for sentiment analysis — posts on Twitter are usually publicly available unless the author has specifically protected their account, can be downloaded in large numbers using an API[13], are generally short in length ($\leq$ 280 characters for Twitter), and usually express opinions about current events. Go et al. [120] conducted a study in which they collected 1.6 million tweets using Twitter's API in order to train a 3 machine learning models to classify tweets as either positive or negative and categorise tweets by subject into one of seven categories (e.g. Product, Person, Movie etc.). Go et al. then evaluate their approach against a hand-annotated dataset of 216 tweets. The results of this study showed that models could be highly effective, achieving an accuracy of 82% in the best case. Pak and Paroubek [121] collected 300,000 texts from Twitter to train a Naïve Bayes classifier, before evaluating on the same hand-annotated dataset used by Go et al.. Bermingham and Smeaton [122] collected over 60 million posts using the Twitter API and manually classified a number of tweets into one of seven categories. In particular, this paper was aimed at discovering whether the shorter nature of "microblog" posts (such as those found on Twitter) could be more accurately classified than longer "blog" posts. In general, the sentiments of microblog posts could be more accurately identified than longer blog posts, and interestingly, the most discriminatory words for microblogs were much more generic than those in blog posts, for example "amazing" and "love" were highly discriminatory in microblogs whereas for blog posts the most discriminatory words were almost exclusively names, for example "reese witherspoon" and "heath ledger". Agarwal et al. [112] use a dataset of 11,875 manually annotated tweets to construct a classifier that achieved up to 75% accuracy — like Bermingham and Smeaton, the authors discovered that the most discriminatory words include terms like "love", "great" and "hate". Barbosa and Feng [123] use a dataset of roughly 200,000 tweets to evaluate their "TwitterSA" tool, discovering that it had a lower error rate than competitors.

Twitter is not the only place that sentiment analysis approaches have been applied to. Given the highly valuable information that can be gained from customers or consumers, the reviews market is also an ideal target for such techniques. Consider a company who release a new product and within a few days they have 10,000 reviews on a product review site. Even if the site offers a "star rating" system to allow reviewers to quantify their satisfaction, these may not always match up perfectly with the actual text of the review. A company can gain a huge amount of valuable information by understanding its customers — which individual features did people like and which ones did they not like [124]?

[12]`https://www.twitter.com`
[13]`https://developer.twitter.com/en/docs/tweets/search/overview`

Fang and Zhan [125] collected a corpus of 5.1 million product reviews from Amazon.com between divided into four major categories: beauty, book, electronic and home. From this data, they trained three separate models to identify sentiment in product reviews. They compared this with a machine-annotated dataset that used a simple classifier to identify the number of positive and negative words in a review. This approach was shown to be reasonably accurate, achieving an F-measure score of over 80% in the best case. Guzman and Maalej [124] conducted a finer grained study delving into the specific features that were either popular or unpopular. Using review data from the Apple App Store and the Google Play Store, the authors identify features using a strategy that identifies groups of two or more words that frequently occur together in the reviews (e.g. "user" and "interface"). The authors selected seven applications from a range of different categories across both app stores to total 32,210 reviews, including 2,800 manually annotated reviews. Following this, they trained and evaluated a model for feature sentiment recognition, achieving an average precision of 60% in the best case.

Kasper and Vela [126] proposed an approach to classify hotel reviews based on a model trained on 1,559 hotel reviews scraped from the internet, achieving up to 67% accuracy in the best case. Similarly, Elango [127] built models using 8,000 TripAdvisor reviews for hotels, achieving up to 79% accuracy in the best case.

Supervised Approaches

In supervised learning, a large amount of testing data must be collected that contains the full input data (e.g. text reviews) and an output label (e.g. positive, negative, neutral). In the domain of product reviews, reviews are often accompanied by a star rating for the product, between 1 and 5 stars. It is therefore possible collect a large corpus of training data by classifying 4/5 star reviews as positive, 3 star reviews as neutral, and 1/2 star reviews as negative. Once the training data is collected, a supervised learning classifier (e.g. Naïve Bayes) is used to generate a model that recognises the key "features" that identify positive and negative reviews. When confronted with new data, the model will use the same classification approach that was learned from the training data to classify the new review.

There have been a number of previous studies that have adopted supervised approaches to sentiment analysis [128]. Kang and Yoo [129] proposed an improved Naïve Bayes classifier that deals with the bias towards positive classification accuracy over negative classification accuracy, showing that this approach was more consistent between the two output classes. Hernández and Rodriguez [130] used a Bayesian Network to try and model the attitude of a reviewer using three different target variables — will to influence, subjectivity and polarity, showing an improvement in accuracy over existing sentiment analysis approaches. Chen and Tseng [131] use a Support-Vector Machine (SVM) with over 51 features extracted from each review, in addition to an "information quality" metric of their own definition to analyse 3000 reviews of digital cameras and mp3 players, achieving a high level of accuracy. Li and Li [132] also used an SVM to perform sentiment analysis on information collected from Twitter about three companies (Google, Microsoft and Sony) and three products (iPhone, iPad, Macbook), achieving comparable levels of accuracy with previous research. van de Camp and van den Bosch [133] conducted experiments on data collected from the Biographical Dictionary of Socialism and the Labour Movement in the Netherlands, using both an SVM and a Neural Network (NN) to classify relations between people in the data as either positive, negative or netural.

Unsupervised Approaches

Part of the problem that supervised learning approaches face is that they require a labelled set of data. For product reviews, it may be the case that there is a star rating to work with, but in many other cases there may be no indication about the sentiment except for in the text itself. This then requires significant manual effort to annotate each review as positive, negative or neutral, or requires the use of an unsupervised technique, which is designed to extract the opinion without the use of a pre-determined fixed model.

He and Zhou [134] proposed a weak supervised approach that started by labelling documents using a lexicon to recognise sentimental words, and then inferred a set of features based on the information gain from each word in the labelled documents. Their approach was shown to be more accurate than other state of the art weak supervised approaches.

Turney [135] proposed an unsupervised approach based on Pointwise Mutual Information (PMI), which identifies statistical dependence between words based on whether they co-occur. This information can be used to infer the presence of one word when another is observed. Turney utilised this information to define a semantic orientation (SO) of a phrase based on whether it had a higher PMI score for the word "excellent" (positive) or "poor" (negative). Read and Carroll [136] also used PMI alongside other weak-supervision approaches to classify articles from the Newswire site, comparing against supervised approaches.

Lexicon Approaches

One of the simplest approaches to sentiment analysis is to use a pre-defined lexicon of sentimental words. A lexicon will contain lots of examples of words that express sentiment, and an estimate for their polarity (i.e. an indication as to whether the word is positive, negative or neutral). For each word in a sentence, its polarity is extracted and adjusted if necessary (for example if a word is preceded by a negator), and then the average polarity for the sentence can be calculated. This approach has a major shortcoming in that it can not recognise context — for example the word "quiet" may be a positive word when referring to a car or a washing machine, but a negative word when referring to a speaker [111].

Hu and Liu [137] proposed a classifier that contained word orientations for each word in a sentence, where 1 was positive, 0 was neutral and -1 was negative. Furthermore, this approach identified specific features of the product in question (e.g. for a digital camera features include "picture"), and identify whether sentences contain sub-phrases that specific opinions about specific features. Kim and Hovy [138] also built a lexicon of sentimental words and considered the holder of an opinion when making decisions about the overall sentiment.

2.17.1 Sentiment Analysis on Version Control Messages

Although previous studies have been limited, there are a few examples of previous research that has calculated sentiment analysis in relation to version control systems (VCS) or continuous integrations (CI). Guzman et al. [139] conducted a study in which they analysed over 60,000 commit messages from 90 separate projects to determine whether sentiment was influenced by the programming language, time of day, day of week, geographical distribution or project approval. In most cases there was very little significance to their results, although it was shown that Java projects involve more negative sentiments than other programming langauges. Souza et al. [140] conducted an experiment that attempted to determine whether negative commit messages were more likely to result in continuous integration build failures, concluding that there is a slightly higher chance of a broken build following a negative commit message. Islam et al. [141] conducted a study that focused on bug-introducing and bug-fixing commits, aimed at determining whether such commits were generally more positive or negative. In their study, Islam et al. discovered that both bug-introducing and bug-fixing commits have significantly higher positive emotion scores than negative scores. Notably, none of the previous studies have considered comparing bug-fixing with non bug-fixing commits, or considered whether negative emotions are more likely to be associated with faulty files.

While it does not specifically concern commit messages, Binkley et al. [142] conducted a study in which they introduce the QALP (Quality Assessment in the large using Language Processing) score metric, which attempts to determine the likelihood that code contains bugs based on the cosine similarity between comments in a module and the code. This process involves creating a language-specific list of words (for example `while`, `strcpy` in C), and then splitting each document into two, one containing comments and the other containing code. In their experiments, Binkley et al. train a model using only three features: QALP, LoC and SLoC. This is done in order to avoid the possibility the that QALP feature would be ignored completely in favour of other metrics. Using the Mozilla dataset and a proprietary program, Binkley et al. discover that for Mozilla, the optimal model produced did not incorporate the QALP score in its calculate for defect prediction. However, for the proprietary program, the QALP score was included. This shows promise for using natural language techniques in order to predict defects in code, something this thesis will investigate further in Chapter 5.

More recently, Binkley et al. [143] conducted a study on the need for software-specific natural language techniques, in which they used a variety of both software-engineering and non-software-engineering datasets to determine the differences between software-specific models and non-software-specific models. The paper concludes that "further improvements in search related tasks within the SE domain will require moving beyond the black box application of IR techniques". This will also be further investigated in Chapter 5.

2.18 Conclusions

Methods for improving the performance of regression testing have received a lot of attention in software engineering research. There are many contributing factors to why so much research has been focussed in this area:

1. Regression testing is inherently an industrially relevant problem. Large software systems and large test suites take a long time to execute and detect regression faults, meaning that any performance improvement can positively impact businesses.
2. There are many aspects of regression testing that are interesting for researchers to explore, including automated test suite generation [69], test suite minimisation [76], test case selection [76] and test case prioritisation [10]
3. Regression testing, despite previous attempts to improve performance, is still slow. Fundamentally, it is necessary to have large regression test suites for large software

systems to ensure no regressions occur, but running large suites can be closer to days than simply a few minutes

Test case prioritisation is a technique for improving the performance of regression testing by generating an 'optimised' ordering in which to run the test cases that should allow for the highest number of faults to be detected as early as possible. It differs from test suite minimisation, which aims to reduce the cost of testing by removing unnecessary or unimportant test cases from the test pool, and from test case selection, which aims to use information about the current version of the program to select a subset of test cases to run. Test case prioritisation was introduced in 1999 by Rothermel et al. [10], who included basic coverage-based and mutation-based approaches to prioritising test suites. In the early stages of test case prioritisation research, generally coverage and mutation-based techniques were the commonly covered ground, alongside separate heuristics including Fault Index [11], History-Based [51] and, more recently, Expert Knowledge approaches [62]. Metaheuristics opened up a new field of possibilities for test case prioritisation as introduced by Li et al. [12]. From this, many studies have been conducted assessing various fitness functions, multiple objectives [74], epistasis [67] and hypervolumes [81]. Finally, the work of Burke et al. [144] has presented a new opportunity in test case prioritisation through the use of hyper-heuristics, whereby the specific heuristic used to evaluate a single test case prioritisation application is not fixed and can be flexible to take into account various conditions, allowing for a new area of exploration into heuristics for test case prioritisation.

2.18.1 Gaps in existing literature

Despite the considerable amount of research that has been conducted in test case prioritisation, there are still some gaps in the literature that can be considered as important threats to external validity.

1. Firstly, at the time of writing Chapter3, there was no research investigating the effectiveness of test case prioritisation on real faults[14]. Additionally, Di Nardo et al. [40] conducted a study using real faults, but included varying numbers of faults in their research and had a clearly biased metric towards systems with more faults. In related fields, [43] investigated the correlation between coverage and test suite effectiveness evaluating real faults. Until recently, it has been difficult to conduct research on real faults because large repository of real faults substantial enough to conduct empirical research on did not exist. With the introduction of DEFECTS4J

[14]Luo et al. [19] conduct an empirical evaluation on this topic, but it had not been conducted at the time of writing Chapter 3

[20], this has now become a possibility. In addition to containing large numbers of real faults, Defects4J comprises entirely of large software projects similar to those used in industry, including projects by Apache Commons. This addresses one of the key threats to validity presented in many previous studies, where it may not be true that results on the experimental subjects are applicable to real-world large applications.

In Chapter 3, I conduct a large-scale empirical evaluation of eight existing test case prioritisation strategies with up to 262 real faults. Importantly, from this evaluation, I conclude that evaluating test case prioritisation strategies on mutants may lead to invalid results, as well as discovering that existing test case prioritisation strategies perform poorly on real faults.

2. Despite promising research showing the effectiveness of defect prediction techniques on real faults, there has been no attempt to evaluate how well a test case prioritisation strategy based on defect prediction would work.

   In Chapter 4, I propose a new test case prioritisation strategy, named G-Clef, that uses defect prediction scores to rank classes by their likelihood of containing a fault. I evaluate G-Clef against existing test case prioritisation strategies on real faults, significantly outperforming six of the eight compared strategies.

3. Finally, there is an opportunity to better understand how version control commit messages correlate with the existence, introduction and fixing of real faults, since this could have implications in test case prioritisation.

   In Chapter 5, I conduct an evaluation of sentiment analysis in over 17,000 commit messages, with the intention of building a test case prioritisation strategy that would leverage sentiment scores. Despite the results not strongly supporting a test case prioritisation strategy, improvements to some of the tools could lead to a successful strategy in the future.

Chapter 3

Using Controlled Numbers of Real Faults and Mutants to Empirically Evaluate Coverage-Based Test Case Prioritisation

The content of this chapter is based on work undertaken during this PhD by the author, which has been published at the Workshop on Automation of Software Test 2018 [1]. The work presented in this chapter extends the published work with a much larger number of subjects and algorithms

3.1 Introduction

In the previous chapter, I explored the various approaches that have been proposed for prioritising test cases. While many of these have been shown to be effective, in many cases the evaluation took place on 'artificial' faults, which can be either mutant or seeded faults, under the assumption that artificial faults behave in the same way as real faults would in practice. Importantly, it may be the case that the research community is *overestimating* the effectiveness of prioritisation strategies as a direct result of the *type* of fault on which it was evaluated.

In addition to this, the primary metric for measuring the effectiveness of test prioritisation strategies is Average Percentage of Faults Detected ($APFD$). This metric is designed to handle programs that contain at least one fault, and is calculated using the area under a curve when plotting the percentage of faults detected against the percentage of test cases run. While other studies have shown that the *APFD* value increases when using some test prioritisation strategies, they have assumed that the *number of faults* is an independent factor in any changes to *APFD*. Furthermore, in continuous integration, where the software is built and tested for every developer change, it is likely that the number of faults introduced will be low, while previous studies have used up to 500 faults in a single subject program [22]. This chapter aims to address these issues through an experiment to test how the effectiveness of test prioritisation strategies varies between different fault types (i.e. real faults and mutants), and between controlled numbers of faults. These experiments use the DEFECTS4J [20] dataset, which is comprised of 395 real faults collected across six open source Java programs to obtain a large corpus of real faults. Since DEFECTS4J isolates each fault in to its own minimal patch file, it is possible to find out the patch(es) that are "compatible" with each other (that is to say, faults that can co-exist in the codebase without causing compilation problems). This allows for the creation of programs with multiple real defects at the same time. In addition to this, I used the Major mutation framework [16] to create large numbers of mutants which were sampled at random to create programs with controlled numbers of mutant faults. Once I had programs with the desired number of faults, I applied eight frequently used test prioritisation strategies to the programs to observe the impact on *APFD* values. Specifically, I aim to answer the following research questions:

RQ1: How does the effectiveness of coverage-based test case prioritisation compare between a single real fault and a single mutant?, **RQ2: How does the effectiveness of history-based test case prioritisation compare between a single real fault and a single mutant?** and **RQ3: How does the effectiveness of test case prioritisation compare between single faults and multiple faults?**.

By answering these three research questions, I can make important observations about previous experiments and establish guidelines for future experiments. Notably, similar research has subsequently been conducted — Luo et al. [19] also used DEFECTS4J subjects to evaluate the performance of test case prioritisation strategies on real faults, and investigate how representative mutants are of real faults in evaluations of test case prioritisation. While this study also compared *APFD* scores of test case prioritisation strategies evaluated on real faults and mutants, it does not compare strategies against the baseline to see whether the conclusions would differ as a result of the fault type. Luo et al. also investigate the specific mutation operators that are most representative of real faults. This additional research shows the importance of the research presented in this chapter.

In particular, the contributions of this chapter are as follows:

Contribution 3.1: *A comparison of how coverage-based test case prioritisation strategies perform real faults and mutants*

Contribution 3.2: *A comparison of how history-based test case prioritisation strategies perform real faults and mutants*

Contribution 3.3: *A comparison of how the number of faults present in a program affects the performance of test case prioritisation strategies*

The remainder of this chapter is structured as follows. In Section 3.2 I discuss how the experiment was formed, how the subjects were chosen, how I created programs with controlled numbers of both real faults and mutants, and the prioritisation strategies that were implemented into the KANONIZO tool. In Section 3.3 I discuss the results of the experiments. Section 3.4 discusses the implications of the results and detail some interesting examples that occurred during the experiment. Finally, this chapter is concluded by Section 3.5

3.2 Methodology

In order to investigate whether the effectiveness of test case prioritisation is different between real faults and mutants, I design an experiment through the following process.

3.2.1 Subject Programs

One of the challenges associated with evaluating test case prioritisation strategies with real faults is the lack of real faults that are readily available. In particular, no previous studies have tried to combine multiple real faults together. Therefore, I require a large repository of real faults, all of which must be isolated and available as patches, such that I can start with a fault-free codebase and incrementally add faults until the desired number is reached. In addition, any repository used for this experiment must have a full, developer-written test suite, containing at least one test case that reveals each isolated fault. The DEFECTS4J repository meets all of these requirements.

3.2.2 Creating programs with multiple real faults

Since one of the research questions relies on having programs that contain multiple faults simultaneously, I designed a method for combining real faults from DEFECTS4J.

```
diff --git a/source/org/jfree/data/time/Week.java b/source/org/jfree/data/time/Week.java
index 8228589..3cc4138 100644
--- a/source/org/jfree/data/time/Week.java
+++ b/source/org/jfree/data/time/Week.java
@@ -172,7 +172,7 @@ public class Week extends RegularTimePeriod implements Serializable {
      */
     public Week(Date time, TimeZone zone) {
         // defer argument checking...
-        this(time, zone, Locale.getDefault());
+        this(time, RegularTimePeriod.DEFAULT_TIME_ZONE, Locale.getDefault());
     }

     /**
```

FIGURE 3.1: The patch required to fix Chart v8, which includes API that is incompatible with previous versions

Algorithm 2 describes the automated process that I used to determine real fault patches that were compatible with one another. It is important to note that DEFECTS4J contains patch files which contain each of the faults present. For each subject version in DEFECTS4J, the script started by *checking out* the version into a clean directory. For all other patches except the one currently being investigated, the script attempted to apply the patch for the other fault. This process revealed some incompatible patches, for example patches that changed files that did not exist by this point in the repository. If the patch successfully applied, the script then attempted to compile the code with the new fault. This also revealed a few incompatible patches such as the one shown in Figure 3.1, which referred to a variable (`RegularTimePeriod.DEFAULT_TIME_ZONE`) that did not exist until a later version of the program. If the program could compile with both faults present, the patch number was added to the set of compatible patches. The directory was then reset before attempting any further patches to avoid additional dependencies. This process produced an output file similar to the one shown in Table 3.1, from which I randomly sampled the desired number of real faults where possible.

Project	Version	Compatible patches
Chart	1	2:3:4:6:8:9:11:13:16:17:23
Chart	2	3:4:6:8:9:11:13:16:17:23
Chart	3	4:6:8:9:11:13:16:17:23
Chart	4	6:8:9:11:13:16:17:23
⋮	⋮	⋮

TABLE 3.1: Example output from Algorithm 2 showing real faults that were compatible with each other

Algorithm 2 Process used to determine compatible real faults

```
Require: Set of Projects $P$, Set of versions for each project $V$
 1: for all $p \in P$ do
 2:     for all $v \in V$ do
 3:         compatible_patches $\leftarrow \emptyset$
 4:         checkout($p, v$)
 5:         for all $v' \in V \setminus v$ do
 6:             patch_file $\leftarrow$ get_patch_file($p$, $v'$)
 7:             apply patch
 8:             if patch successful then
 9:                 compile($p, v'$)
10:                 if compile successful then
11:                     compatible_patches $\leftarrow$ compatible_patches $\cup v'$
12:                 end if
13:             end if
14:             reset($p, v'$)
15:         end for
16:         write(compatible_patches)
17:     end for
18: end for
```

It is important to note that the history-based approaches used in this chapter's experiments could not be used on programs that contain multiple real faults. Table 3.2 demonstrates why this could result in a problem — if the version v_{n-2} in this table represents one of the real faults in the DEFECTS4J dataset, the test history analysis process described in Section 3.2.5 would record the test failure at version v_{n-2}. Any programs resulting from the process described in this section should be simulating a "current" version of the program, and therefore any prioritisation strategy should not know the outcome of any test on the current version before prioritising the tests. If the fault from version v_{n-2} is selected during the random sampling of the result of Algorithm 2, this results in the test case history for t_m containing information about a fault that is present in the *current version*. Any history-based approach would then utilise the fact that test t_m failed recently to give it a high priority. In reality, the approach should not know about the previous failure since it is expecting the fault to only be present in the current version. This presents an unfair advantage to a history-based approach, and as a result these experiments do not include multiple real faults when comparing history-based approaches.

Test Number	Result in Program Version					
	v_i	v_{i+1}	...	v_{n-2}	v_{n-1}	v_n
t_m	✓	✓	...	✗	✓	✓

Table 3.2: Collecting historical test data of a fault included in Defects4J

3.2.3 Creating programs with multiple mutant faults

In addition to real fault versions, this experiment required the generation of program versions that contained multiple mutant faults. Given a program p and a version v and a desired number of mutants n, Algorithm 3 describes the automated script that I created to create programs with multiple mutant faults. Consider a situation in which I am trying to generate a version of Chart-2 that contains five mutant faults. In the corresponding version containing real faults, Chart-2 contained the patches from Chart-6, Chart-9, Chart-13 and Chart-17. Firstly, I identified the class(es) for that version that contained the real faults and generated all possible mutants for that class.

Version	Class Name
2	`org.jfree.data.general.DatasetUtilities`
6	`org.jfree.chart.util.ShapeList`
9	`org.jfree.data.time.TimeSeries`
13	`org.jfree.chart.block.BorderArrangement`
17	`org.jfree.data.time.TimeSeries`

Secondly, I ran a "simple" mutation analysis across the generated mutants to identify those that were "killed" by at least one developer-written test case. This is crucial to the experimental setup, since if a mutant causes no observable change to a program and cause a test case that previously passed to fail, the mutant may be equivalent, and therefore useless in this experiment. After running this simple analysis, the script identifies mutants that were detected by at least one test case. For each of the faulty classes, the script then selected a killed mutant from that class at random.

Version	Class Name	Mutant Selected
2	`org.jfree.data.general.DatasetUtilities`	13
6	`org.jfree.chart.util.ShapeList`	16
9	`org.jfree.data.time.TimeSeries`	7
13	`org.jfree.chart.block.BorderArrangement`	35
17	`org.jfree.data.time.TimeSeries`	105

Finally, the script ran a "detailed" mutation analysis of this mutant, to produce a list of test cases that kill the mutant, which become the "trigger tests" (fault-revealing test cases) for that mutant.

Version	Class Name	Mutant Selected	Killing Test Cases
2	`org.jfree.data.general.DatasetUtilities`	13	...
6	`org.jfree.chart.util.ShapeList`	16	...
9	`org.jfree.data.time.TimeSeries`	7	...
13	`org.jfree.chart.block.BorderArrangement`	35	...
17	`org.jfree.data.time.TimeSeries`	105	...

I ran this script on the University of Sheffield High Performance Computing (HPC) Cluster [145], and allocated the maximum permitted runtime of 168 hours for mutation analysis to be run. Despite the extensive runtime and memory allocations, the Closure project could not run mutation analysis due to memory constraints, owing to the large number of mutants and test cases that need to be executed in order to conduct mutation analysis. Therefore, the Closure project was excluded from all experiments. In addition to this, the bug Chart-10 can not have mutants due to the fact that the class has no possible mutation targets [146], and the Mockito project has issues with mutating certain bugs [147].

Algorithm 3 Process followed to create programs with multiple mutant faults

```
Require: Program $p$, Version $v$, desired number of mutants $n$
 1: checkout($p, v$)
 2: target_classes ← get_faulty_classes($p, v$)
 3: mutate(target_classes)
 4: run_mutation_analysis()
 5: killing_tests ← $\emptyset$
 6: for all $c$ in target_classes do
 7:     killed_mutants ← get_killed_mutants($c$)
 8:     target_mutants ← sample(killed_mutants, 1)
 9:     for all $m$ in target_mutants do
10:         run_detailed_mutation_analysis($m$)
11:         killing_tests ← killing_tests $\cap$ get_killing_tests($m$)
12:     end for
13: end for
14: write_kill_map(killing_tests)
```

3.2.4 Test Case Prioritisation

In order to prioritise test suites, I created the KANONIZO tool [148]. KANONIZO, which means "arrange" in Greek, is an open-source tool for prioritising test cases in Java, into which I implemented the eight strategies described below. More details about Kanonizo can be found in Appendix A.

Coverage-based approaches

Firstly, I selected the four most common coverage-based strategies from previous literature, namely the greedy algorithm [10], additional greedy algorithm [10], genetic algorithm and random search. For the genetic algorithm, I use the $APLC$ fitness function proposed by Li et al. [12], which optimises the ordering of test cases for early line coverage. For further descriptions of these strategies, see Chapter 2.

History-based approaches

In addition to the coverage-based approaches described above, I also implemented four history-based approaches — ROCKET [54], MCCTCP [52], AFSAC [56] and Elbaum et al. [149]. For the AFSAC algorithm, the original proposal by Cho et al. does not give specific values for four configurable parameters with which to run the algorithm — α, β, γ and δ. These values are used to determine the priority of test cases based on which category they fall into (i.e. failed *more times* consecutively than ever before). The actual value of these four parameters will not impact the overall ordering of the test suite provided it follows the constraints given in the original proposal — $\alpha > \beta > \gamma > \delta > 0$. For the experiments below, I use values of $\alpha = 1$, $\beta = 0.7$, $\gamma = 0.4$ and $\delta = 0.1$. Section A.1.4.1 gives details on how these algorithms were tested in Kanonizo to ensure that they were faithful to the original proposals.

3.2.5 Test History Analysis

In order to run history-based test prioritisation strategies, it is necessary to collect information about previous executions of test cases. In particular, the strategies above may need to know how many times each test case has been executed, how many times it has failed in its lifetime, or how long each test execution takes. In order to collect this information, I wrote a script that uses version control information to iteratively check out a previous version of the software, starting at the most recent version and working back as far as the version control allows. For each version, the script compiles the code, runs the developer written test suite and record the test result, execution time and, if necessary, cause of failure. In some cases, previous versions of the program may not compile due to either mistakes in the commit, or due to a missing library that is no longer required by the more recent versions. In cases where compilation failures occurs, the script retries up to five preceeding versions. If none of these versions compile, the script terminates the history analysis at this point. In total, the script analysed over 150,000 commits from the Defects4J subjects.

One interesting facet of the Defects4J dataset is that each bug must have a developer-written test case that exposes it. However, some of these test cases only existed *after* the bug was discovered, and exist in Defects4J thanks to a manual process where a "special commit" is created for Defects4J purposes. When mapping Defects4J bugs back to the last known commit before the bug was fixed, sometimes the triggering test case is not present, meaning that the test case was written specifically with the knowledge that the bug exists, and for the sole purpose of preventing the bug returning as a *regression*. However, test case prioritisation requires the test case to be present in order to evaluate how effective the test case prioritisation technique is. For this reason, I exclude faults from the analysis where the trigger test was not present for at least one previous execution before the bug was fixed, leaving a total of 82 faults remaining for the analysis in **RQ2**. This avoids a potential bias in favour of certain history-based techniques which leverage information about the number of previous executions.

3.2.6 Evaluating the effectiveness of test case prioritisation

As discussed in Section 2.7, the most common metric for evaluating test case prioritisation strategies is $APFD$. The sooner *all* faults in a program are found, the *higher* the $APFD$ score will be for a test suite ordering. As a result, the aim of a test case prioritisation strategy is to maximise this value. Since $APFD$ is designed to handle multiple faults, it is an appropriate metric to use in this experiment. Since all test cases in will be retained in the prioritised test suite, the $NAPFD$ metric proposed by Qu et al. [37] would give the same result as $APFD$. Furthermore, in this experiment, over 98% of test cases execute in under one second, and all faults in Defects4J are assumed to have identical severity, meaning $APFD_c$ would also give the same score as $APFD$.

3.2.7 Statistical Analysis

The experiments presented in this chapter utilise three strategies that make random choices during their execution (genetic algorithm, random search and random baseline). Whenver such strategies are employed, there is a chance that positive results can occur as a result of stochastic decision making. In order to minimise this risk, I apply statistical testing in accordance with Arcuri et al. [150]. Firstly, the experiments concerning stochastic algorithms were repeated 30 times, in order to ensure an adequate sample size from which to draw conclusions. Secondly, I conduct two forms of statistical testing to the results of the experiments. The first statistical test is the Mann-Whitney U-Test. The Mann-Whitney U-Test, given two samples of data, calculates the likelihood that they originated from the same distribution, returning a *p-value* representing this probability. A *p-value* is < 0.05 estimates that there is a 5% chance that the samples could originate from the same distribution. Such a result referred to as a *significant* result. In the plots in Section 3.3, each boxplot is annotated with either a ✓ or a ✗, representing if the result displayed in the corresponding boxplot is significant or not.

While the Mann-Whitney U-Test is a good indicator of whether or not two samples are significantly different, it does not give an indication of *how* different the samples are. Therefore, in addition to the Mann-Whitney U-Test, I also apply the Vargha-Delaney $\hat{A}$ Test to measure the magnitude of the difference between the sample. Specifically, given two samples A and B, the Vargha-Delaney $\hat{A}$ Test calculates the percentage of occasions on which you would expect sample A to outperform sample B. Furthermore, Vargha and Delaney quanitifed and categorised the $\hat{A}$ value as **N**one ($|\hat{A} - 0.5| < 0.06$), **S**mall ($0.06 \leq |\hat{A} - 0.5| < 0.14$), **M**edium ($0.14 \leq |\hat{A} - 0.5| < 0.21$) or **L**arge ($|\hat{A} - 0.5| > 0.21$). In the boxplots in Section 3.3, the Vargha-Delaney $\hat{A}$ Test score is represented by an N/S/M/L at the top of the plot.

3.2.8 Threats to Validity

This study consider bugs taken from five large, open-source Java projects that are included as a part of DEFECTS4J. While these programs vary in their total lines of code, number of test cases and number of years under development, it may be possible that these results do not generalise to other programming languages or other subjects with different characteristics. I aim to address this threat by including as many subjects as possible, but future experiments could include subjects with different languages and different test suite styles. Additionally, since the experimental process makes some random choices on how to combine real faults, repeating these experiments with different combinations of faults may also lead to different results.

This paper makes use of eight prioritisation strategies, divided into coverage-based strategies [10] and history-based strategies [149]. While these strategies have been frequently used in previous studies [10], there are a number of other prioritisation strategies that have been proposed that are not considered, for example a number of genetic algorithms [12], clustering algorithms [59], and static code-based approaches [151]. Running these experiments with any of these algorithms may lead to different results. However, it is important to note that the purpose of this chapter is not to determine which test case prioritisation strategy is best, but to establish any relationship between different fault types when applying prioritisation strategies.

The coverage-based strategies used in this chapter primarily use line coverage as the primary input data for these strategies. However, there are other types of coverage that could lead to different results, for example branch coverage [37] or mutant coverage [49]. While the Kanonizo tool does provide support for other types of coverage, the purpose of these experiments was to identify differences between fault types rather than differences between coverage types, which can be found in previous research [12].

One of the fundamental assumptions involved in test case prioritisation is that test cases can be reordered freely without impact to the outcome of the test cases. This is referred to as the test independence assumption [32], and has been empirically shown to not always hold in practice. While the subjects in Defects4J use JUnit, which should guarantee that test cases are atomic and independent, there are some test cases in Defects4J that modify system state. However, Kanonizo collects coverage *before* test cases are re-ordered, and Defects4J provides the information about which test cases reveal failures. The outcome of the test case after prioritisation is not critical in order to evaluate effectiveness. Crucially, it should be noted that for JUnit tests the responsibility for ensuring test independence lies with the developer rather than with any test case prioritisation strategy, since it is difficult to know about test dependencies without extreme combinatorial testing [152].

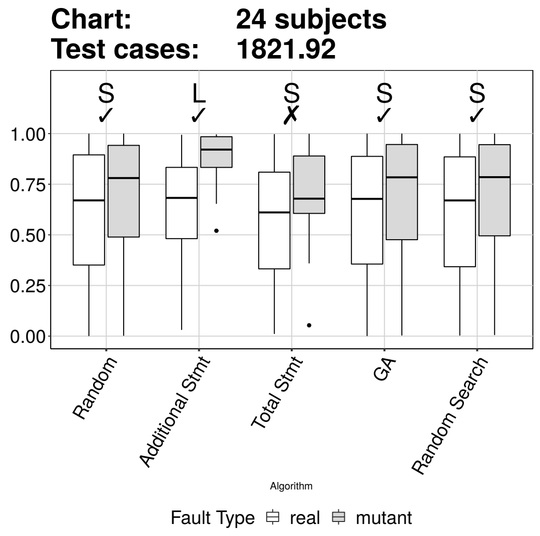
When working with mutation, there are a few important considerations. Firstly, it is important to consider the possibility of equivalent mutants, which are changes to a program that cause no semantic difference despite modifying syntax. In these experiments, I only include mutants that cause at least one of the developer written test cases to fail, ensuring that all mutants that are included are not equivalent to the original program. Additionally, Papadakis et al. [153] wrote about the dangers of using duplicate mutants or subsumed mutants and the effect that these types of mutants can have on mutation scores. In these experiments, I consider only a single mutant from each selected class, meaning that it is not possible that the experiments will contain duplicate mutants. Furthermore, while subsumed mutants can impact mutation score, for the purposes of these experiments a subsumed mutant is equally as important as any other fault, and as such no extra measures were taken to remove these mutants.

Finally, since I use my own tool Kanonizo and created all the experimental scripts myself for the purpose of these experiments, it is possible that defects in the tools may lead to invalid results. While developing Kanonizo I always developed a JUnit test suite that goes alongside the tool to ensure that the results obtained are correct. As discussed in Appendix A, Kanonizo is available as an open source tool [148], and all analysis scripts used are available [154] to allow for the external replication and verification of these results.

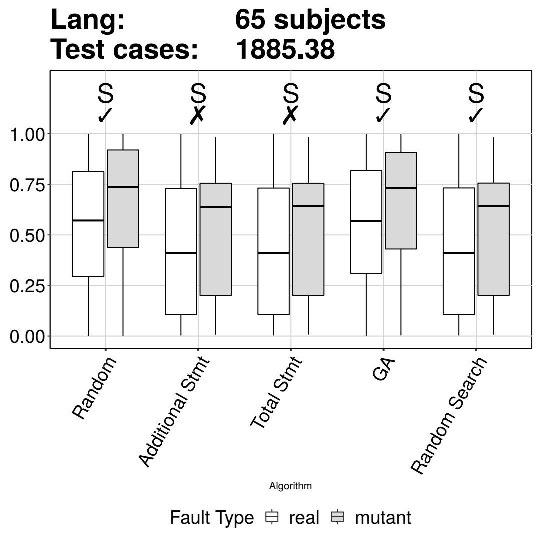
3.3 Results

3.3.1 RQ1: How does the effectiveness of coverage-based test case prioritisation compare between a single real fault and a single mutant?

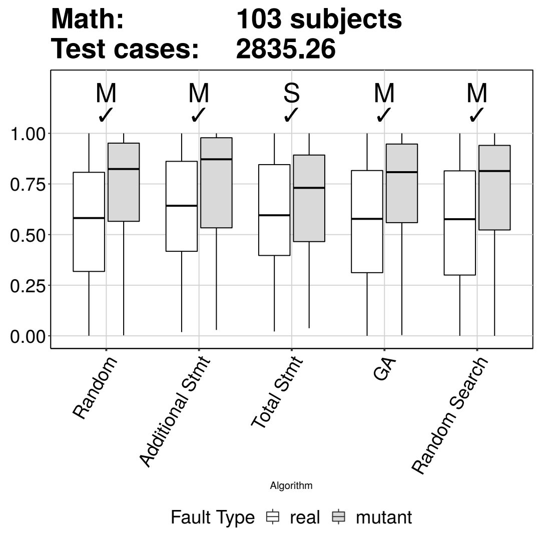
Figure 3.2 shows the *APFD* scores produced by running four coverage-based prioritisation strategies on a single real fault and a single mutant. The figure immediately reveals a clear difference in the effectiveness of test case prioritisation strategies between real faults and mutants, producing higher *APFD* scores for programs containing mutants for every project and every prioritisation strategy. In most cases, the difference is significant, with only 5 comparisons out of 25 that are not statistically significant according to the Mann-Whitney U-Test (Chart/Total Statement, Lang/Additional Statement, Lang/Total Statement, Mockito/Additional Statement, Time/Total Statement). Notably, the only cases where the result is not significant are those where experiments were not repeated 30 times to account for random choices, and therefore the fact that these samples have fewer data points may have contributed to lack of significance [150]. In each case, it is clear from looking at the boxplots that the *APFD* scores for mutants are higher, with even one case of a large $\hat{A}$ score (Mockito/Additional Statement) despite not being a significant result. This is supported by Table 3.3, which reports the mean number of test cases required in order for the two fault types to be exposed. In all cases, there are more test cases required in order to find a real fault than a mutant. The difference reported in Table 3.3 represents the raw difference represented as a total percentage of the test suite. For example, for the Chart project an average of 698 test cases are required in order to find a real fault, whereas 566 are required to find a mutant, leaving a difference of 132 test cases. When represented as a percentage of the 1823 test cases this project has on average, this results in with 7.24% more test cases. From this, it is clear conclude that mutants are easier to find than real faults, a phenomenon that is further discussed in Section 3.4. While this result indicates that researchers should be wary of exaggerating the effectiveness of test case prioritisation strategies when using mutants as subjects, this result alone does not explicitly imply that mutants are inappropriate subjects for such experiments. In Figure 3.2, the significance values were calculated across fault types. For example, for the Chart project with the Total Statement strategy, the samples provided to the Mann-Whitney U-Test were data from Chart/Total Statement/Real Faults and Chart/Total Statement/Mutants. By contrast, Table 3.4 considers statistical significance calculated



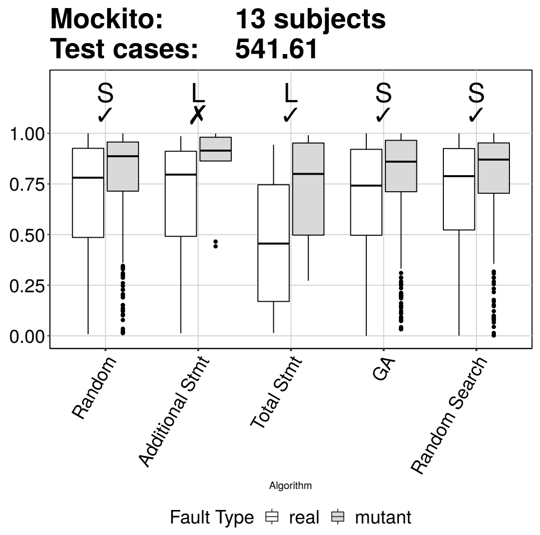
(A) Chart Project Statistics



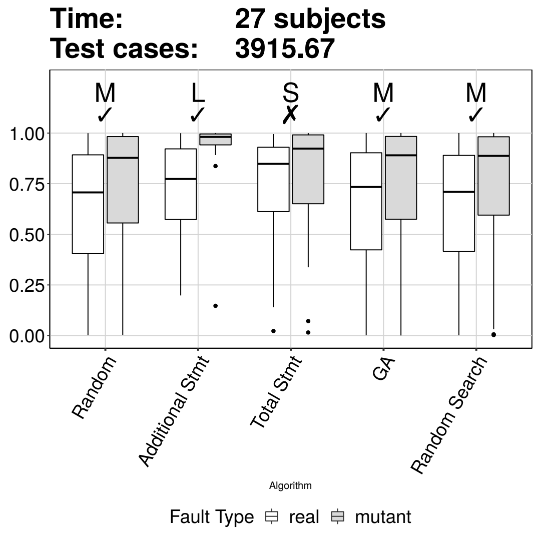
(B) Lang Project Statistics



(C) Math Project Statistics



(D) Mockito Project Statistics



(E) Time Project Statistics

FIGURE 3.2: *APFD* scores for programs containing one real fault (white) and one mutant (grey) prioritised with coverage-based strategies. A ✓ indicates a significant result according to the Mann-Whitney U-Test at a 5% confidence level, while a ✗ indicates the reverse. An annotation N/S/M/L indicates the $\hat{A}$ score, **N**one ($|\hat{A} - 0.5| < 0.06$), **S**mall ($0.06 \leq |\hat{A} - 0.5| < 0.14$), **M**edium ($0.14 \leq |\hat{A} - 0.5| < 0.21$) or **L**arge ($|\hat{A} - 0.5| > 0.21$).

Table 3.3: Mean number of test cases required to find faults, for real faults and mutants respectively

Project	Real	Mutant	# Test Cases	% Difference
Chart	698.38	566.31	1823.71	7.24
Lang	924.14	751.38	1885.38	9.16
Math	1216.60	768.58	2739.26	16.36
Mockito	460.08	292.63	1375.69	12.17
Time	1397.45	964.29	3915.67	11.06

as it would appear in a paper proposing a new technique. For example, for the Chart project with the Total Statement strategy, the samples provided to the Mann-Whitney U-Test are Chart/Total Statement/Real Faults and Chart/Random/Real Faults. In Table 3.4, an $\hat{A}$ score of > 0.5 indicates that, on average, the listed strategy should outperform a random ordering in terms of *APFD* score, while an $\hat{A}$ of < 0.5 indicates the reverse. From Table 3.4, it is clear that there are number of results that would change if mutants were used in place of real faults. For example, when evaluating the Additional Statement strategy on the Chart project, using real faults would lead to the conclusion that the strategy provides no real benefit over random ordering, while using mutants indicates a significant improvement over the baseline. Conversely, evaluating Total Statement on the Math project shows no real difference for real faults, whereas using mutants leads to the conclusion that this strategy is significantly worse than the baseline.

These results are consistent with Luo et al. [19], who demonstrate that for all strategies considered in their evaluation, the *APFD* score is higher for mutants than it is for real faults. While their study does not include a baseline to compare against, the mean *APFD* presented for each strategy are similar to the ones presented in this chapter. Furthermore, Luo et al. state, as this research question does, that there is no guarantee that results observed on mutant faults will correlate to similar levels of performance on real faults.

Table 3.4: Significance values of coverage-based strategies compared against their respective baseline

	Additional Stmt		Total Stmt		GA		Random Search	
	Real	Mutants	Real	Mutants	Real	Mutants	Real	Mutants
Chart	(N) 0.49	**(M) 0.69**	(N) 0.45	(N) 0.48	(N) 0.50	(N) 0.50	(N) 0.49	(N) 0.50
Lang	**(S) 0.39**	**(M) 0.34**	**(S) 0.39**	**(M) 0.34**	(N) 0.50	(N) 0.49	(S) 0.39	(M) 0.34
Math	(N) 0.55	(N) 0.55	(N) 0.53	**(S) 0.43**	(N) 0.50	(N) 0.49	(N) 0.50	(N) 0.48
Mockito	(N) 0.50	(S) 0.61	**(N) 0.46**	(S) 0.40	(N) 0.50	(N) 0.49	(N) 0.50	(N) 0.48
Time	(S) 0.58	**(L) 0.73**	(S) 0.59	(N) 0.55	(N) 0.51	(N) 0.50	(N) 0.50	(N) 0.50

RQ1: Using mutants to evaluate coverage-based test case prioritisation strategies leads to a significant increase in APFD, and may reverse the conclusions when compared to real faults.

3.3.2 RQ2: How does the effectiveness of history-based test case prioritisation compare between a single real fault and a single mutant?

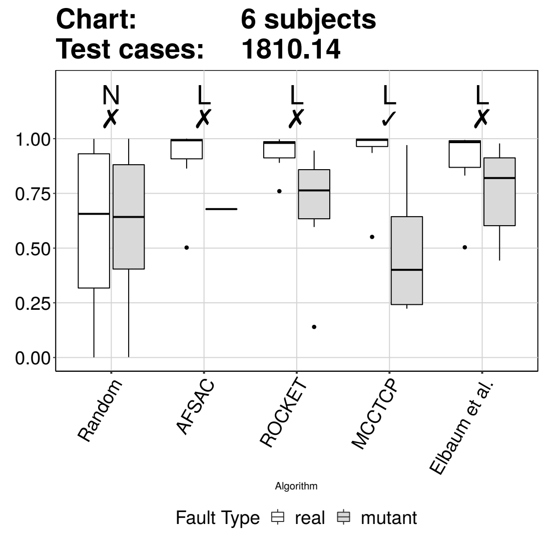
Figure 3.3 shows the *APFD* scores of programs containing real faults and mutants when prioritised using history-based strategies. It is important to note that, in accordance with Section 3.2.5, Figure 3.3 contains data from fewer programs than Figure 3.2 (60 faults instead of 232 — See Appendix B for details about the subjects included in these experiments). From Figure 3.3 it is apparent that there is a much more varied distribution of results. Whereas in Section 3.3.1 the results are generally consistent, with history-based strategies there are some cases where the *APFD* for real faults is higher than the *APFD* for mutants (e.g. Chart/ROCKET), as well as some cases where it is lower (e.g. Lang/MCCTCP). Notably, there are no cases where the *APFD* is significantly lower for real faults, and only one case where it is significantly higher (Chart/MCCTCP).

Table 3.5 shows the average number of commits, the percentage of commits in which the trigger test is present, and the percentage of occasions on which the trigger test has failed for each of the projects in Defects4J. Table 3.5 shows that, for the Chart project, the trigger tests have a high number of failures in their history. This means that the history-based strategies have a high chance of prioritising these test cases well, and therefore leads to the result shown in Figure 3.3, in which the Chart project has very high *APFD* scores for real faults. Conversely, for the Time project, despite the trigger tests being present in every single commit, there are no failures in the history of the trigger tests, resulting in the low *APFD* scores observed for this project.

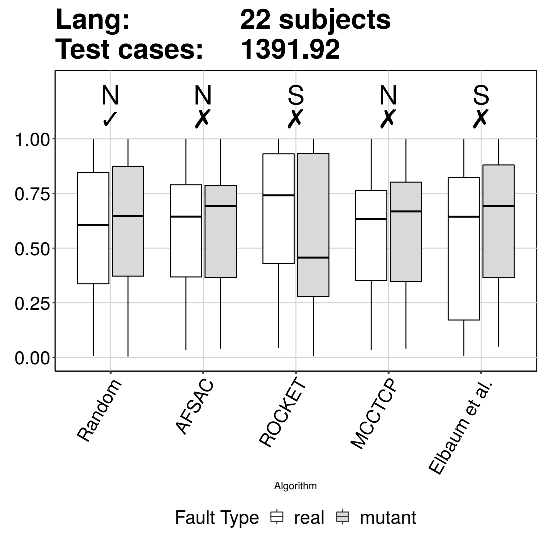
Table 3.6 shows the results of the Mann-Whitney U-Test and $\hat{A}$ scores for history-based strategies when compared against the baseline. As with Section 3.3.1, this shows

Table 3.5: Number of commits, percentage of commits in which the trigger test is present, and percentage of occasions on which the trigger test has failed in its history

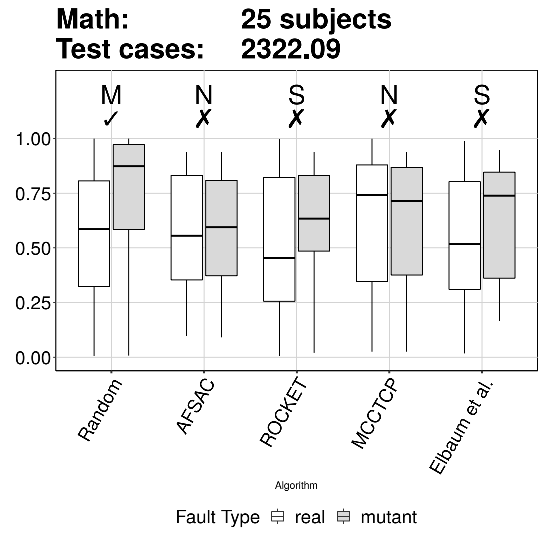
Project	# Commits	% Occurences	% Failures
Chart	24.33	72.78%	66.67%
Lang	159.33	87.16%	5.11%
Math	382.61	77.38%	5.56%
Mockito	105.33	65.20%	19.12%
Time	35.67	100.00%	0.00%



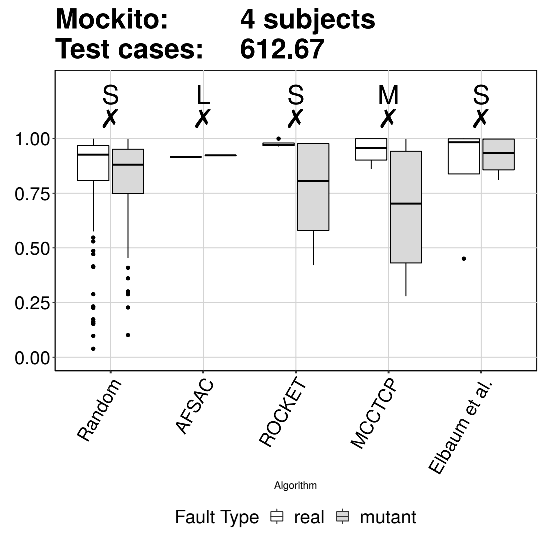
(A) Chart Project Statistics



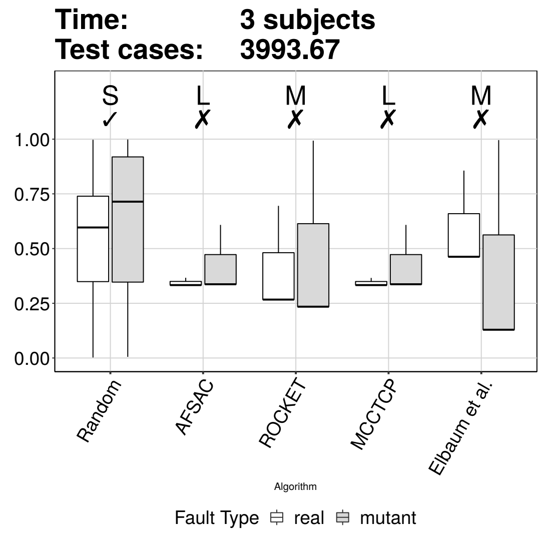
(B) Lang Project Statistics



(C) Math Project Statistics



(D) Mockito Project Statistics



(E) Time Project Statistics

FIGURE 3.3: *APFD* scores for programs containing one real fault (white) and one mutant (grey) prioritised with history-based strategies. A ✓ indicates a significant result according to the Mann-Whitney U-Test at a 5% confidence level, while a ✗ indicates the reverse. An annotation N/S/M/L indicates the $\hat{A}$ score, **N**one ($|\hat{A} - 0.5| < 0.06$), **S**mall ($0.06 \leq |\hat{A} - 0.5| < 0.14$), **M**edium ($0.14 \leq |\hat{A} - 0.5| < 0.21$) or **L**arge ($|\hat{A} - 0.5| > 0.21$).

Table 3.6: Significance values of history-based strategies compared against their respective baseline

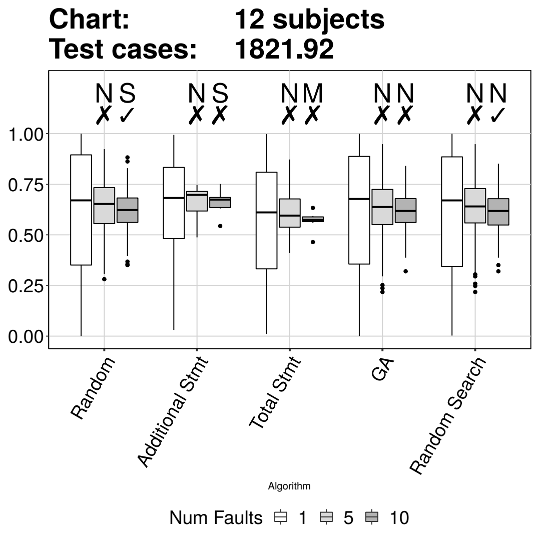
	AFSAC		ROCKET		MCCTCP		Elbaum et al.	
	Real	Mutant	Real	Mutant	Real	Mutant	Real	Mutant
Chart	**(L) 0.82**	(N) 0.56	(M) 0.71	(S) 0.56	**(L) 0.87**	(S) 0.38	(M) 0.68	(S) 0.62
Lang	(L) 0.73	(N) 0.52	(S) 0.59	(N) 0.45	(N) 0.50	(N) 0.51	(N) 0.48	(N) 0.53
Math	(S) 0.60	**(M) 0.29**	(N) 0.46	**(M) 0.31**	(N) 0.54	**(M) 0.32**	(N) 0.47	**(M) 0.33**
Mockito	(L) 0.75	(M) 0.64	**(S) 0.64**	(N) 0.48	(N) 0.46	(S) 0.43	(S) 0.59	(L) 0.72
Time	(L) 0.24	(L) 0.29	(M) 0.35	(N) 0.44	(L) 0.24	(L) 0.29	(N) 0.54	(S) 0.40

significant differences between real faults and mutants. For example, with MCCTCP on the Chart project, using real faults would lead to the conclusion that the strategy is highly effective, while using mutants would result in the opposite conclusion. For the Math project, every strategy has very similar results with no significantly better or worse results for real faults, whereas for mutants every strategy is significantly worse than the baseline. Notably, in Table 3.6, there are no cases where a project/strategy combination is significant for both fault types.

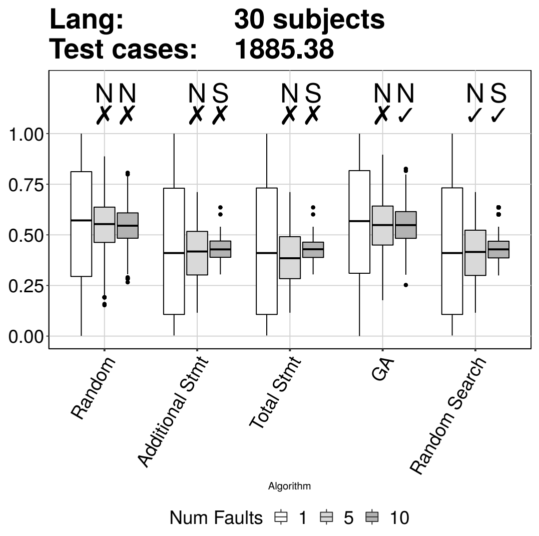
RQ2: History-based strategies are more effective than coverage-based strategies for some projects and less effective for others. As with Section 3.3.1, there are clear differences in the effectiveness of all strategies when using different types of faults.

3.3.3 RQ3: How does the effectiveness of test case prioritisation compare between single faults and multiple faults?

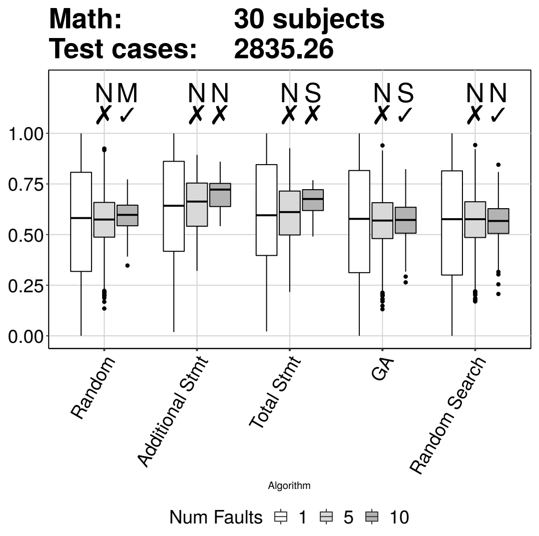
Figure 3.4 shows the *APFD* scores for programs containing different numbers of real faults. One of the immediately noticeable trends is that as the number of faults in a program increases, the variance in *APFD* scores decreases. This is because with only a single fault present, there are some situations in which test case prioritisation performs very well, and some where it performs poorly. As the number of faults increases, there is a higher probability that the variance exists within a single program, with some faults being found quickly and some faults requiring more test cases. This means that the *APFD* scores tend to normalise towards a single value, which causes the lower variance.



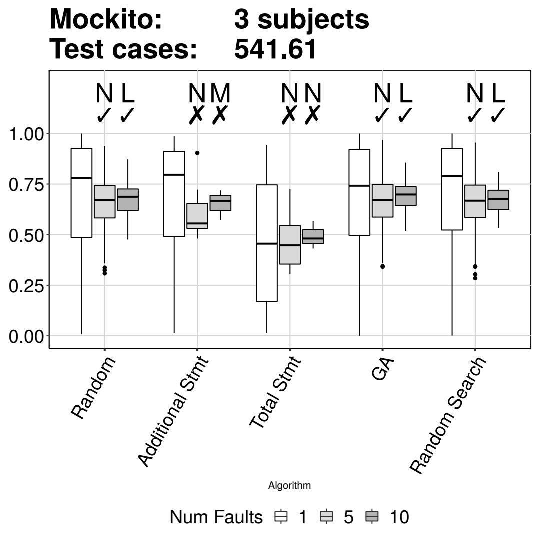
(A) Chart Project Statistics



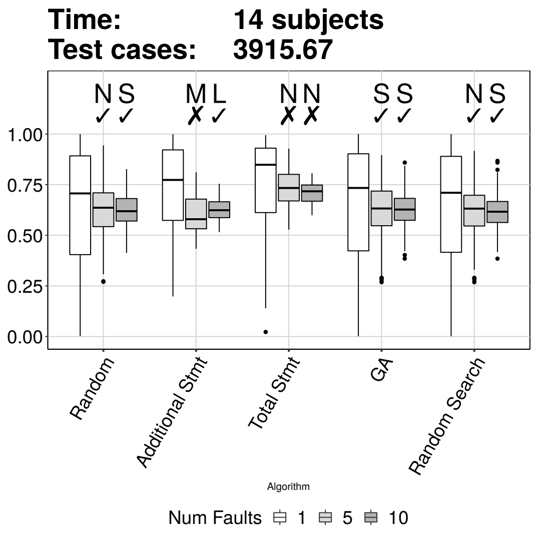
(B) Lang Project Statistics



(C) Math Project Statistics



(D) Mockito Project Statistics



(E) Time Project Statistics

FIGURE 3.4: *APFD* scores for programs containing single real faults (white) through to programs containing 10 real faults (grey). A ✓ indicates a significant result according to the Mann-Whitney U-Test at a 5% confidence level, while a ✗ indicates the reverse. An annotation N/S/M/L indicates the $\hat{A}$ score, **N**one ($|\hat{A} - 0.5| < 0.06$), **S**mall ($0.06 \leq |\hat{A} - 0.5| < 0.14$), **M**edium ($0.14 \leq |\hat{A} - 0.5| < 0.21$) or **L**arge ($|\hat{A} - 0.5| > 0.21$).

In addition to this, in most cases the median *APFD* score decreases as more faults are included (e.g. Chart/GA), although there are some cases where the median *APFD* increases (e.g. Math/Additional Statement). The reason for this is that the *APFD* metric requires detection of **all** faults in a program rather than just a small subset. In cases where there is only a single fault, the *APFD* scores can be heavily influenced by a single well-placed test case, however as the number of faults increases the probability of finding all faults quickly becomes lower.

Table 3.7 shows the significance values of the coverage-based test case prioritisation strategies when compared against their respective baselines. As shown in the table, as the number of faults in a program increases, the $\hat{A}$ scores and significance values become more extreme, with 13 significant values for programs with 10 faults, compared with only four significant values for programs with 1 fault. Furthermore, as the number of faults increases, the performance of test case prioritisation strategies tends to decrease when compared with the random baseline. When using only 1 fault, 13 out of 20 project/strategy combinations have an $\hat{A}$ score of ≥ 0.5, while when the programs contains 10 faults, only 8 out of 20 programs have an $\hat{A}$ score of ≥ 0.5.

RQ3: In addition to the type of fault, the number of faults present in a program can make a significant difference to the APFD scores — with more faults present in a program, there is a higher likelihood of obtaining a significant result

3.4 Discussion

One of the important results observed in Section 3.3 were that test case prioritisation performs very differently depending on the fault type that is included. Additionally, while previous literature has suggested that the genetic algorithm can provide benefits in test case prioritisation [12], there is little evidence of that based on these experiments — in most cases, the genetic algorithm had an identical performance to the baseline. This section investigates these results.

3.4.1 Real Faults vs Mutants in Test Prioritisation

While Section 3.3 reveals that there are clear differences between real faults and mutants when it comes to how effective test prioritisation is, these results do not develop an understanding concerning the syntactic and semantic differences between fault types that may be the root cause. Just et al. [83] conducted a detailed study in which they compared real faults and mutants, in particular investigating whether mutation operators could be used to generate the same real faults that occur in practice. These experiments show

Table 3.7: Significance values of different numbers of real faults, compared against their respective baselines

Project	1	5	10
		Additional Stmt	
Chart	(N) 0.49	(N) 0.55	(M) 0.66
Lang	**(S) 0.39**	**(L) 0.23**	**(L) 0.17**
Math	(N) 0.55	**(M) 0.65**	**(L) 0.84**
Mockito	(N) 0.50	**(N) 0.44**	(S) 0.40
Time	(S) 0.58	(S) 0.43	(N) 0.52
		Total Stmt	
Chart	(N) 0.45	(S) 0.39	**(M) 0.29**
Lang	**(S) 0.39**	**(L) 0.20**	**(L) 0.17**
Math	(N) 0.53	(N) 0.56	**(L) 0.77**
Mockito	**(N) 0.46**	**(S) 0.37**	**(L) 0.04**
Time	(S) 0.59	**(L) 0.73**	**(L) 0.82**
		GA	
Chart	(N) 0.50	(N) 0.48	(N) 0.49
Lang	(N) 0.50	(N) 0.49	(N) 0.50
Math	(N) 0.50	(N) 0.49	**(S) 0.43**
Mockito	(N) 0.50	(N) 0.51	(N) 0.54
Time	(N) 0.51	(N) 0.50	(N) 0.52
		Random Search	
Chart	(N) 0.49	(N) 0.49	(N) 0.47
Lang	**(S) 0.39**	**(L) 0.23**	**(L) 0.17**
Math	(N) 0.50	(N) 0.50	**(S) 0.42**
Mockito	(N) 0.50	(N) 0.50	(N) 0.47
Time	(N) 0.50	(N) 0.49	(N) 0.47

that for the 262 real faults considered, on average 2.23 lines of code were added (max. added lines 33), while on average 6.26 lines of code were removed (max. removed lines 49) in order to fix a real fault. When working with mutants, a maximum of one line of code can be added, and a maximum of one line of code can be removed. This shows the relative complexity of real faults when compared to mutants.

Furthermore, Figure 3.5 shows an example of a patch required to fix a real fault and a patch required to fix a mutant. This shows the complexity associated with fixing a real fault, since there is a huge amount of contextual knowledge required to understand that the `if` statement included is causing an issue and therefore must be removed. By contrast, the mutant is very clearly wrong from the first inspection, and would be highly unlikely to ever occur in practice. Luo et al. [19] also provide an example that shows how different fixing real faults and mutants are. In particular, the example in Luo et al. shows that *APFD* scores from strategies evaluated on real faults and mutants have low correlation when mutants do not properly reflect real faults occurring in a program.

```
diff --git a/source/org/jfree/chart/plot/CategoryPlot.java b/source/org/jfree/chart/plot/CategoryPlot.java
index 5d831f7..dc7d06b 100644
--- a/source/org/jfree/chart/plot/CategoryPlot.java
+++ b/source/org/jfree/chart/plot/CategoryPlot.java
@@ -2163,9 +2163,6 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
             markers = (ArrayList) this.backgroundDomainMarkers.get(new Integer(
                     index));
         }
-        if (markers == null) {
-            return false;
-        }
         boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
@@ -2448,9 +2445,6 @@ public class CategoryPlot extends Plot implements ValueAxisPlot,
             markers = (ArrayList) this.backgroundRangeMarkers.get(new Integer(
                     index));
         }
-        if (markers == null) {
-            return false;
-        }
         boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
diff --git a/source/org/jfree/chart/plot/XYPlot.java b/source/org/jfree/chart/plot/XYPlot.java
index 243f94b..50cf416 100644
--- a/source/org/jfree/chart/plot/XYPlot.java
+++ b/source/org/jfree/chart/plot/XYPlot.java
@@ -2290,9 +2290,6 @@ public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
             markers = (ArrayList) this.backgroundDomainMarkers.get(new Integer(
                     index));
         }
-        if (markers == null) {
-            return false;
-        }
         boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
@@ -2529,9 +2526,6 @@ public class XYPlot extends Plot implements ValueAxisPlot, Zoomable,
             markers = (ArrayList) this.backgroundRangeMarkers.get(new Integer(
                     index));
         }
-        if (markers == null) {
-            return false;
-        }
         boolean removed = markers.remove(marker);
         if (removed && notify) {
             fireChangeEvent();
```

(A) Patch required to fix real fault `Chart-14`

```
diff --git a/src/main/java/org/joda/time/Days.java b/src/main/java/org/joda/time/Days.java
index 116cc7d..0007de3 100644
--- a/src/main/java/org/joda/time/Days.java
+++ b/src/main/java/org/joda/time/Days.java
@@ -46,7 +46,7 @@ public final class Days extends BaseSingleFieldPeriod {
     /** Constant representing one day. */
     public static final Days ONE = new Days(1);
     /** Constant representing two days. */
-    public static final Days TWO = new Days(2);
+    public static final Days TWO = new Days(0);
     /** Constant representing three days. */
     public static final Days THREE = new Days(3);
     /** Constant representing four days. */
```

(B) Patch required to fix mutant fault `Time-18`

Figure 3.5: Patches required to fix different fault types

This is also reflected in the number of test cases that detect the faults. Since real faults are highly specific and targeted, there are often very few test cases that reveal them. On average, 2.37 test cases detect the real faults, compared to 10.9 test cases that detect mutants. This in turn contributes to the inflated *APFD* scores that were observed for mutants in Section 3.3.

3.4.2 The Results with the Genetic Algorithm

One of the more prominent trends in Section 3.3 was that the Random Search and Genetic Algorithm strategies had very poor performance for both real faults and mutants, in both cases very rarely improving upon random orderings. Given that the genetic algorithm is initialised with a population of random orderings and can only make changes that improve the overall fitness, it stands to reason that the minimum performance of the genetic algorithm should match the baseline, and there should be cases where the orderings produced are much better than the baseline.

One possible reason for this failure is the choice of fitness function for the genetic algorithm. Li et al. [12] use a fitness function called Average Percentage of Lines Covered, which aims to order tests in such a way that covers the most lines as early as possible. One of the downsides of this fitness function is that it is incredibly expensive to calculate, particularly for large systems with lots of test cases. For every coverable line of code, the *APLC* function must identify the index of the *first test case* in the prioritised suite to execute that line, which can result in millions of checks. Practically, this makes it unlikely that a user would achieve a high number of generations during a reasonable runtime (e.g. 60 seconds). These experiments achieved 10,000 generations, and yet were still not able to achieve good test suite orderings. As discovered by Hao et al. [21], in most cases the best possible solution for maximising coverage is produced by the additional statement strategy, which produces near-optimal or optimal levels of coverage. Therefore, the genetic algorithm with the *APLC* fitness function is never likely to outperform additional statement coverage regardless of how much time or how many iterations are allocated in its budget.

3.5 Conclusions

In this chapter I have investigated the effect of using different fault types and different numbers of faults when attempting to evaluate the effectiveness of test case prioritisation strategies. Through an experimental analysis, I have discovered that using mutant faults can lead to inflated *APFD* scores, and may in some cases result in opposite conclusions when compared to real faults, that is to say using real faults indicates that strategy A is significantly more effective than strategy B, while evaluating on mutants would result in strategy B being more effective. This is partly due to the fact that real faults are harder to detect than mutants, which in turn causes fewer tests to reveal real faults.

I have also discovered that the inclusion of many faults may cause differences when compared to single faults. While it is not the purpose of this chapter to determine the more likely scenario in practice, it is important to note that when programs contain more faults, the resulting $APFD$ scores may suffer due to only finding a subset of faults. Furthermore, as the number of faults in a program increases, there is a higher likelihood of obtaining a significant result.

Finally, this chapter has shown that existing test case prioritisation strategies are not very effective at prioritising test suites to detect real faults. When using real faults, there were only six combinations of project/strategy (40 combinations total) that resulted in statistically significant results, and of those, there were three examples where random orderings were significantly better than orderings produced by one of the strategies. This clearly shows the need for an improved test case prioritisation strategy for real faults.

Chapter 4

An Empirical Study on the Use of Defect Prediction for Test Case Prioritisation

The content of this chapter is based on work undertaken during this PhD by the author, which has been published at the International Conference on Software Testing, Verification and Validation 2019 [2].

4.1 Introduction

In Chapter 3 I investigated the effects of different fault types on the effectiveness of various test case prioritisation strategies. Some of the most positive results for detecting real faults were history-based techniques — in particular, the JFreeChart project from DEFECTS4J. However, there were also a number of instances where history-based techniques performed poorly, due to *trigger tests* having no historical failures.

In particular, one of the important findings of the previous chapter was that it is important to evaluate on real faults, since using artificial faults for evaluation may lead to incorrect conclusions. One technique that has been shown to be effective at finding real faults is defect prediction [18, 23–25]. Defect prediction estimates the likelihood that file within a software system is buggy, and leverages software metrics or version control information to produce these values.

In this chapter, I present a test case prioritisation strategy, called G-Clef, that uses defect prediction data to reorder a test suite such that the classes that have the highest chance of being buggy are covered first. I present a large parameter tuning experiment to investigate how to maximise the performance of my defect prediction tool, and an empirical evaluation of the proposed strategy using real faults from the Defects4J dataset, and compare against the state-of-the-art coverage-based and history-based strategies. Specifically, I aim to answer **RQ1: Which configuration of G-Clef is the most effective?**, **RQ2: How does G-Clef compare to previously proposed coverage-based test case prioritisation strategies at prioritising manually-written test cases?** and **RQ3: How does G-Clef compare to previously proposed history-based test case prioritisation strategies at prioritising manually-written test cases?**.

The contributions of this chapter are as follows:

Contribution 4.1: *A parameter tuning study to determine the best parameters for defect prediction to find real faults in* Defects4J

Contribution 4.2: *An implementation of a new test case prioritisation strategy, G-Clef, that leverages defect prediction*

Contribution 4.3: *A parameter tuning study to determine the best parameters for G-Clef*

Contribution 4.4: *An evaluation of G-Clef against existing coverage-based strategies*

Contribution 4.5: *An evaluation of G-Clef against existing history-based strategies*

The remainder of this chapter is structured as follows. Section 4.2 introduces defect prediction, before Section 4.3 discusses the Schwa defect prediction tool. In Section 4.4 I introduce G-Clef and discuss challenges of using defect prediction data to prioritise test cases. Section 4.4.3 talks through the setup and execution of the experiments, before 4.5 presents the results of the empirical evaluation. Finally, Section 4.6 concludes the findings of this chapter.

4.2 Defect Prediction

As explored in Chapter 2, defect prediction is a widely studied area within the field of Mining Software Repositories (MSR). Defect prediction typically uses either software quality metrics (e.g. [24]) or software repository information (e.g. [26]) to decide whether a class is buggy or not. There are two types of classification for this — the first is a binary classification per-file as to whether the file is faulty or not. This approach is typical of machine learning models and particularly of models that use static code features, since these models typically analyse the static features associated with each file and return a "yes" or "no" answer determining whether it believes the file is faulty or not (e.g. ELFF [103]). The second approach is to return a numeric score representing the probability that a given file is faulty for every file in a software project.

4.3 Schwa

Given a Git[1] repository of a Java project, the Schwa tool[2] [26] extracts information from each commit, such as its message, author, timestamp, list of all modified files, and the changes performed (i.e., the diff). It performs a defect prediction computation based on three metrics that have been shown to be effective at predicting defects: 1) revisions [23] (how often a Java class has been changed), 2) fixes [25] (how often a Java class has been fixed), and 3) authors [99] (how often a Java class has been modified by more than one developer). Schwa is robust, readily available software that is not language specific, making it a suitable choice for many subjects.

Rather than considering each commit as equally likely to have resulted in an issue, Schwa uses a value "Time-Weighted Risk" (TWR) [26, 155], to estimate how reliable a Java class is:

$$TWR(C) = \frac{1}{1 + e^{-12\alpha + w}} \tag{4.1}$$

For each commit (C) in the repository, the value TWR(C) for the commit is calculated based on how recently the commit was added — the oldest commit in the repository has an α value of 0.0, while the most recent commit has an α of 1.0, with every commit in between having an α increased by a fixed interval based on the total number of commits. The value w is used to weight the importance of newer commits as opposed to older commits. Lewis et al. [155] suggested $w = 12$ as a good value to score the files of two Google projects by their bug-propensity. Rather than a fixed value, Schwa uses $w = 2 + ((1 - TR) \times 10)$, where TR represents the time-range of bug fix commits: TR values close to 0.0 indicate will give more favour to newer commits, whereas TR values close to 1.0 will allow older commits to have a slightly higher TWR. It is important to note that if $TR = 0.0$, then w is equal to 12, the original value suggested by Lewis et al. [155].

[1] `https://git-scm.com/`, accessed March 2020.
[2] `https://github.com/andrefreitas/schwa`, accessed March 2020.

Schwa estimates the likelihood that a Java class c contains a bug using Equation 4.2, in which each of the three factors (i.e., revisions, authors, and fixes) is calculated and modified by a weight, where the sum of all weights must be equal to 1.

$$\begin{aligned}\beta_c = \mathit{RevisionsWeight} \times \sum_{\mathcal{R}_c \in \mathcal{R}} TWR(\mathcal{R}_\rfloor) \\ + \mathit{AuthorsWeight} \times \sum_{\mathcal{A}_c \in \mathcal{A}} TWR(\mathcal{A}_\rfloor) \\ + \mathit{FixesWeight} \times \sum_{\mathcal{F}_c \in \mathcal{F}} TWR(\mathcal{F}_\rfloor)\end{aligned} \tag{4.2}$$

$\sum_{\mathcal{R}_c \in \mathcal{R}} TWR(\mathcal{R}_\rfloor)$ is the sum of all TWRs in which c has been modified. $\sum_{\mathcal{A}_c \in \mathcal{A}} TWR(\mathcal{A}_\rfloor)$ is the sum of all TWRs in which a new author has modified c. $\sum_{\mathcal{F}_c \in \mathcal{F}} TWR(\mathcal{F}_\rfloor)$ is the sum of all TWRs in which c has been involved in a fix operation. $\mathcal{R}$, $\mathcal{A}$, and $\mathcal{F}$ represent the revisions', authors', and fixes' timestamps in which c has been involved. The value β_c is normalised to $[0, 1]$ and estimates the defect probability of c, $\mathit{defect}_c = 1 - e^{-\beta_c}$. Intuitively, a Java class c with a higher defect_c value is less reliable (i.e., is more likely to contain a bug) than those classes with a low defect_c value.

4.4 G-Clef

Algorithm 4 illustrates the procedure of G-Clef, which integrates defect prediction into a test case prioritisation strategy. If G-Clef were to adopt a categorical classification approach, such as the one used by Arisholm et al. [156], it would simply take the test cases that execute code in the classes that the defect prediction approach has determined are faulty, place those test cases first in the prioritised suite, and then add the tests for "non-faulty" classes. In the case of a false negative (i.e. a class that is faulty but is determined by the defect predictor to be clean), the impact of this would be huge, since there is no natural ordering of the "non-faulty" classes. As an example, consider a program with 2,000 classes and 20,000 test cases. If a defect predictor suggests that five of the classes are faulty, and G-Clef adds the tests covering those five classes to the prioritised test suite, we are left with 1,995 classes and (e.g.) 19,000 test cases, with no way of deciding which of the 1,995 classes we should choose next. According to Hall et al. [86], most classification approaches for defect prediction have a recall score of around 55-75%, meaning that classes that contain faults are misclassified as non-faulty between 25 and 45 percent of the time. As a result, for G-Clef, Schwa is a suitable choice of defect predictor since it provides a natural ordering of classes, reducing the impact of a false negative.

Algorithm 4 G-Clef

```
Require: Classes Under Test $C = \langle c_1, c_2, ..., c_n \rangle$
    Test Suite $T = \{t_1, t_2, ..., t_m\}$
    Function to return $defect_c$ score for class $c_i$, $b(c_i)$
    Function to determine classes covered by test $t_j$, $s(t_j)$
    Secondary Objective Function $g$
    Group Size $G$

Ensure: Prioritised Test Suite $T'$
 1: $C \leftarrow \textsc{Sort}(C,\ b)$
 2: $T' \leftarrow \langle\rangle$
 3: while $C \neq \langle\rangle$ do
 4:     $A \leftarrow \textsc{sublist}(C, G)$ // Take first G elements from C
 5:     $C \leftarrow \textsc{remove}(A, C)$ // Remove all elements of A from C
 6:     $T'' \leftarrow \langle\rangle$
 7:     for all $c_i \in A$ do
 8:         $T'' \leftarrow T'' \cdot \langle t_j \in T | c_i \in s(t_j)\rangle$
 9:     end for
10:     $T' \leftarrow \textsc{Unique}(T' \cdot \textsc{Sort}(T'',\ g))$
11: end while
12: return $T'$
```

In order to prioritise test cases using Schwa, G-Clef first orders the classes in a program by the defect prediction score produced by Schwa (line 1). Until the list of classes is empty, G-Clef iteratively takes the next group of G classes (see Section 4.4.2 for details), and identifies the tuple of test cases $T'' \subseteq T$ that execute lines of code in each class. Since this process returns many test cases, G-Clef applies a secondary objective g (line 10), discussed in the following subsection, to order T'' using an alternative heuristic (e.g., *coverage*). Finally, G-Clef places the ordered test cases into the prioritised suite (T') (line 5). Since G-Clef starts with the class that is most likely to be faulty, and selects all tests that cover this class, better bug prediction will directly result in faster fault detection during test suite execution.

To illustrate how G-Clef works, consider a small example program with 3 classes — `ClassA` has 100 test cases and a $defect_c$ score of 0.8. `ClassB` has 30 test cases and a $defect_c$ score of 0.35, while `ClassC` has 1000 test cases and a $defect_c$ score of 0.1. G-Clef starts by selecting all the test cases for `ClassA`, since this is the most likely to contain a bug. Following this, the *secondary objective* decides how the 100 tests for `ClassA` should be ordered. A good secondary objective will place first the test case that detects the fault.

Now consider a bug report that incorrectly assigns `ClassC` a $defect_c$ score of 0.9. Since G-Clef takes all the tests for `ClassC` first, there are now 1000 test cases being executed before they detect a bug. To address this problem, G-Clef *groups* classes together based on their likelihood of containing a fault. In this instance, a group size G of 2 would include `ClassC` and `ClassA` in the first group of classes, meaning the secondary objective has the combined set of tests from `ClassC` and `ClassA` (i.e., 1100 tests) from which to choose. Note that this selection process only involves iteratively selecting the class that is *next most likely* to contain a fault, rather than applying a clustering approach to decide which classes are more similar to each other.

4.4.1 Secondary Objective

G-Clef utilises a secondary objective to determine the ordering of test cases given a set of tests that cover a target class. It is important to note that G-Clef could use a linear combination of primary and secondary objective using a weighting function to avoid the need for a two-phase sorting process (i.e. sorting classes by $defect_c$ score and then sorting by secondary objective). For example, could assume that the score for a test case t is $4 \times defect_c + secondary\_objective(t)$ — this would mean that the primary objective (i.e. defect prediction score) is still weighted much more heavily than the secondary objective, but that a high secondary objective score could outweigh poor $defect_c$ scores. In the case where multiple classes are covered by t, an average could be taken of the $defect_c$ scores of all classes covered by t. Since the focus of this chapter is primarily using defect prediction, I investigate the impact of sorting classes by the $defect_c$ score, using four secondary objectives to prioritise test cases once G-Clef has established a subset based on defect prediction: **greedy** (or total statement) orders test cases by the total number of statements covered, **additional greedy** (or additional statement) keeps a track of the combined set of lines covered by the prioritised suite, selecting the test that covers the most *previously uncovered* lines, **random** returns a purely random ordering for test cases, ensuring diversity of the prioritised test cases. Finally, similar to the work by Hao et al. [21] and Campos and Abreu [157], I apply a **constraint solver**, representing the lines of code as constraints that must be covered by one or more test cases and finding the minimal set of tests that satisfies all of the constraints, thereby covering all of the lines of code.

4.4.2 Grouping Classes

In addition to the secondary objective, G-Clef may also need to group classes together. If a bug prediction report incorrectly assigns a high $defect_c$ score to a class with many test cases, G-Clef may suffer as a result. In this thesis, I investigate four different settings for grouping classes, with the default behaviour of G-Clef being the use of a single class. In addition, I run experiments using 5%, 10%, and 25% of the total classes that exist in each subject program. To avoid bias for or against subject programs that contain more classes than others, I use a percentage of the classes in the chosen project.

4.4.3 Experimental Setup

The intention of G-Clef is to outperform existing test case prioritisation strategies when evaluated on real faults. Therefore, I designed an empirical evaluation of G-Clef to compare against existing coverage-based and history-based strategies.

4.4.4 Subject Programs

To automatically perform an experimental analysis, the selection of subject programs used in this empirical evaluation adhered to the following requirements: 1) the programs used should be developed in Java (as the test prioritisation tools used only support Java), 2) it must be possible to "roll-back" changes from the repository (i.e., obtain previous versions of the source code) in order to support the collection of test history data for the history-based strategies, and 3) it must be possible to detect faulty behaviour in the current version of the program using a test suite. One particular collection of subject programs that meets all of the aforementioned requirements is DEFECTS4J [83]. All DEFECTS4J projects were collected from version control systems, meaning that it is possible to identify, check-out, and execute tests on previous versions of the software using a version control tool such as Git. Finally, DEFECTS4J provides a developer-written test suite for each program in the repository, which includes at least one test that triggers the faulty behaviour of the current version of the software, which are referred to as the *trigger tests*.

4.4.5 Coverage Analysis

One of the challenges in using defect prediction to order test cases is that defect prediction estimates the likelihood of faulty behaviour in *source code*, whereas G-Clef is specifically working with test cases. Therefore, G-Clef must map test cases to source classes using code coverage. For each test case in the system, G-Clef must know the set of classes that are executed during the course of the test case. For this, I used GZoltar [158, 159]. One of the important features of GZoltar is that it executes the test cases using the same build tools that developers would be using (e.g., ANT, MAVEN, and GRADLE), meaning that the code coverage collected is as similar as is possible to a "natural" execution of test cases by a developer. Additionally, since GZoltar produces a serialised coverage file, experiments can use the same file across test case prioritisation strategies to better ensure consistency.

4.4.6 Test History Analysis

Since these experiments also utilise test-history, I follow the same procedure described in Section 3.2.5 to collect historical execution data for all test cases in this experiment.

4.4.7 Schwa

The default configuration of Schwa uses these weights: 0.25 for revisions and authors, 0.5 for fixes, and 0.4 for *TR*. As each software project is unique in terms of, for instance, repository history and development model, these weights may vary in suitability for different projects. For example, the "authors" metric is irrelevant if only a single developer contributed to a project. For RQ1, I performed a tuning study of Schwa's weights and the *TR* value. As Schwa's feature weights and *TR* value are in the range of 0.0 and 1.0, I chose all values in this range with interval 0.1. Although there are 13310 different combinations, the sum of all weights must be equal to 1, leaving 726 valid combinations.

To assess the effectiveness of each combination at ranking a class that is *buggy*, an automated process randomly selected 5 faults of each of Defects4J's [83] projects (a total of 30 faults), and executed Schwa on the repository history of those faults. While it may have been possible to achieve higher performance by tuning Schwa's parameters *per project* rather than using a global value, this would limit how generalisable the results would be, since anyone wishing to use G-Clef would have to conduct a parameter tuning experiment on their own project before running it. By using all projects, I am able to find the best parameters for all projects without being biased towards any individual project.

As Schwa returns a $defect_c$ value for each class of the software under test, I ranked all classes by this value and identified, for each combination, the ranking position of the known buggy class. The best combination of weights and *TR* would rank the known buggy class first, on the other hand, the worst combination of parameters would rank the *buggy* class last.

4.4.8 Test Case Prioritisation

As in Chapter 3, I use the KANONIZO tool to conduct test case prioritisation experiments. Since the tool already provides implementations for the four coverage-based and four history-based strategies used in these experiments, I only had to extend KANONIZO by adding G-Clef.

4.4.9 Evaluation Measurements

For RQ2 and RQ3, I compare the effectiveness G-Clef to that of the existing test prioritisation strategies. The most commonly used evaluation metric in this field is *APFD*. However, this chapter considers 395 program versions, each containing a single fault. This reduces *APFD* to the percentage of tests that were executed before the fault was detected. DEFECTS4J provides a list of the trigger tests that detect each fault. To compare the strategies in the experimental evaluation, I calculate the percentage of each prioritised test suite that was placed higher than the trigger test for the subject program. For example, if the trigger test is the 50^{th} test case out of 1000 test cases, the suite is scored as 5%. As a result, in the plots presented in the following section, a lower score is indicative of better performance, rather than a higher score.

4.4.10 Analysis Procedure

I analysed all of the data resulting from the experiments by following well-established guidelines [150]. For instance, all algorithms that make random choices were repeated 30 times. Additionally, I use the Mann-Whitney U-Test to compare two different data sets, obtaining a p-value representing the likelihood that this data was observed as a result of chance. For the Mann-Whitney U-Test, I adopt a 95% confidence interval, meaning $p < 0.05$ indicates that the result is statistically significant. In addition, I use the Vargha-Delaney $\hat{A}$ test to compare G-Clef with existing strategies. For this, $\hat{A}$ values closer to 0 indicate that G-Clef, on average, is expected to outperform the existing strategy, while a value closer to 1 indicates that the previous prioritisation strategy, on average, is expected to outperform G-Clef.

4.4.11 Threats To Validity

Despite the fact that this study uses a high number of real faults from six different Java programs, this chapter's results may not generalise to other programs with either different characteristics or types of test suites or faults. Although I evaluate prioritisation strategies on manually written test suites, it is conceivable that the use of different test suites could *improve* the results for some prioritisation strategies, while *degrading* the results for others — in such cases, the results and conclusions presented in this chapter would not be generally valid.

Even though there is no evidence to suggest that this would occur, future work should further study prioritisation effectiveness for different types of test suites (e.g., automatically generated test cases from tools such as EvoSuite [68] and Randoop [160]). Additionally, even though Defects4J's programs have fast tests for which prioritisation is less necessary, my experiments yield useful insights when, for instance, tests run in a continuous integration environment (e.g., [55, 161]).

Moreover, this chapter does not consider the runtime of test cases when evaluating prioritisation strategies. It is possible that with long running test cases, new orderings may actually be *slower* to detect faults, even if they require fewer tests. However, approximately 98% of tests ran in under one second, making it unlikely that this would occur in practice.

G-Clef prioritises tests from an entire test suite rather than using a test case selection approach to identify relevant test cases. While this is consistent with many previous approaches (e.g., [10, 11, 55]), it is conceivable that using subsets of test cases may lead to different results. Future work should also examine the effectiveness of a hybrid approach that selects subsets of test cases in conjunction with defect prediction.

Additionally, the random sample of 30 faults used to tune Schwa's parameters in RQ1 may not have resulted in the best overall parameters for this tool, and thus using different subjects may have resulted in different parameters. To mitigate this, bugs were chosen from each of the projects in Defects4J, thereby avoiding bias towards any particular project. Next, I selected the test case prioritisation strategies used in the experiments as a representative sample of previous history-based approaches. Since this evaluation is not exhaustive, it is possible that using other strategies may lead to different results. This is mitigated by using a range of strategies from the literature that require different input and process the test execution history in different ways. One of the considerations when running Schwa is the number of commits that it analyses when calculating prediction scores. If Schwa can analyse the entire repository history, while a history-based strategy only has a small number of commits available due to reasons discussed in Section 3.2.5, then it may give G-Clef an unfair advantage. Therefore, I also conducted experiments in which the number of commits available to Schwa was limited to the number used by the history-based strategies, observing no significant differences in the overall results.

The bug prediction described in Section 4.2 uses the commit history of a repository as a black box. It has been shown, for example, that modelling commit authors could improve the effectiveness of identifying which commits introduce a bug, thereby improving the effectiveness of bug-predictors [162]. With that said, this chapter's main goal is to evaluate how leveraging defect prediction in test case prioritisation could lead to faster regression detection — and not what is the best defect prediction approach for this particular problem. Similarly, since the only requirement for a secondary objective is that it provides a numeric value for sorting test cases, any of the history-based approaches used in this chapter could also be used as secondary objectives, which may have been more effective than the secondary objectives investigated. Finally, as described in Section 4.4.1, a linear combination could be used in conjunction with the secondary objective to remove the need to sort by both $defect_c$ score and the secondary objective. While the experiments in this chapter do use groups of classes to determine whether including more classes in the secondary objective affects performance, we do not investigate whether a linear combination would perform better or worse than keeping the two objectives separate. Future work will investigate further secondary objectives in order to find the best combination of defect prediction and secondary objective.

The Schwa [26] tool does not consider static code features from the software, instead opting to use repository information to model the classes that make up the software under test. It has been shown by other studies that using code features can give a strong indication of whether software is faulty or not — for example, Menzies [24] built a prediction model based on 38 static code features to obtain a high precision score, while the ELFF tool [103] uses static code features collected from the JHawk[3] tool including cyclomatic complexity and line counts to provide its prediction scores. Moreover, the identification of a bug-fixing commit by Schwa is done using a regular expression inspired by GitHub documentation [4], rather than using the SZZ algorithm [89] that many other defect predictors use [103, 104]. As a result, the use of Schwa may inhibit the results of this research where using another tool would have provided better results.

A final validity threat is potential defects in the tools used during experimentation (i.e., Kanonizo [1] and Schwa [26]). Used without error in prior experiments, both of these publicly available tools have been extensively tested. Moreover, all of the data presented in this chapter and the scripts needed to reproduce the experiments are available at `https://bitbucket.org/josecampos/history-based-test-prioritization-data/`.

[3] `http://virtualmachinery.com/jhawkprod.htm`

[4] `https://help.github.com/en/github/managing-your-work-on-github/linking-a-pull-request-to-an-issue`

4.5 Results

4.5.1 RQ1: Which configuration of G-Clef is most effective?

Since G-Clef relies on the Schwa tool, this research question first investigates how to get the best defect prediction estimates from Schwa, before considering how to configure the two available parameters in G-Clef to maximise its performance. This research question also considers a situation where, instead of Schwa, a defect prediction tool exists that perfectly predicts the location of the buggy class (i.e. the first class considered by G-Clef always contains a bug). This is used to compare the performance of the secondary objectives without other external factors.

RQ1.1: What are the best parameters for Schwa?

Table 4.1 reports the three best and the three worst of Schwa's configurations identified during tuning. For the 30 randomly selected faults, Schwa works best, on average, with a revision weight of 0.6, fixes weight of 0.1, authors weight of 0.3, and a *TR* value of 0.0, which lines up with Graves et al. [23] finding that recent changes have a higher impact on the likelihood of code being buggy. A *TR* value of 0.0 means that w, as given in Equation 4.1, is equal to 12, which is the same value suggested by Lewis et al. [155]. Notably, the fixes weight is low for the three best configurations, and high for the worst three. This indicates that previous failures have low influence when it comes to predicting future failures. Furthermore, the revisions weight is high for the three best configurations, and low for the worst three, indicating that the more times a file has been changed, the more likely it is to be associated with defects.

Table 4.1: Parameters of top and bottom three Schwa configurations. For each configuration I report the revision, fixes, and authors weights, *TR* value, average, standard deviation (σ), and confidence intervals (CI) using bootstrapping at 95% significance level of the ranking position of the known buggy class.

Revision Weight	Fixes Weight	Authors Weight	Time Range	Avg. Pos.	Std. Dev. σ	Conf. Inter. CI
			top 3			
0.6	0.1	0.3	0.0	46.53	49.12	[27.71, 63.97]
0.7	0.1	0.2	0.4	46.57	49.49	[29.00, 62.93]
0.6	0.1	0.3	0.4	46.73	49.26	[27.90, 63.33]
			bottom 3			
0.1	0.6	0.3	1.0	88.07	109.20	[43.82, 125.10]
0.1	0.7	0.2	1.0	90.73	112.25	[46.46, 127.09]
0.1	0.8	0.1	1.0	91.43	109.50	[52.14, 125.59]

Table 4.2: Relative ranking position of buggy classes reported by the best Schwa configuration. We report the average number of classes (ranking size), minimum, maximum, average, and standard deviation (σ) of the relative ranking position of a buggy class, and the average $defect_c$ value of a buggy class (*def*).

	Ranking	Relative Ranking Position				
Project	Size	*min*	*max*	*avg*	σ	*def*
Chart	1016	0.1%	56.0%	16.6%	19.9	0.39
Closure	1478	0.1%	90.4%	9.2%	15.6	0.89
Lang	344	0.3%	52.3%	12.8%	14.3	0.96
Math	1069	0.1%	94.0%	17.7%	21.3	0.91
Mockito	1018	0.1%	86.6%	10.6%	19.7	0.85
Time	585	0.1%	67.6%	12.6%	17.1	0.80
Overall	1046	0.1%	74.5%	13.0%	18.0	0.86

One of the interesting values in Table 4.1 is the standard deviation column, which shows for all 6 configurations that the standard deviation is higher than the mean, despite the fact that the "position" value must be > 0. Looking at the best configuration, of the 30 subjects, 18 subjects had a position below the mean position (i.e. position < 46.53). Of these 18 subjects, 14 had a position < 20, and 9 had a position < 10. Although there were only 12 subjects that had a position value higher than the mean position value, 6 of these subjects had a position > 100. Conversely, considering the worst configuration, 19 out of 30 subjects had a position better than the mean position, 11 subjects had a position < 20, and 8 had a position < 10. The biggest difference for the worst configuration was that when Schwa failed to identify the buggy class correctly, the resultant position was much worse. In the worst configuration, 11 subjects had their buggy class ranked > 100, with two subjects having a position > 200 and two more subjects having a position of over 300. These negative results bring down the performance of the worst configuration, despite still achieving positive results for the majority of the subjects.

With 99% confidence, according to the Anderson-Darling statistical test [163], the ranking position of each buggy class of any Schwa configuration is not normally distributed. While some configurations outperform others in terms of average number of classes required, there is no single configuration that outperforms *all* others. For the following experiments, I continue using the top configuration as the input for Schwa.

RQ1.1: For the 30 faults randomly selected from the Defects4J*'s dataset, Schwa works best with the following parameters: revision weight of* 0.6*, fixes weight of* 0.1*, authors weight of* 0.3*, and a TR value of* 0.0.

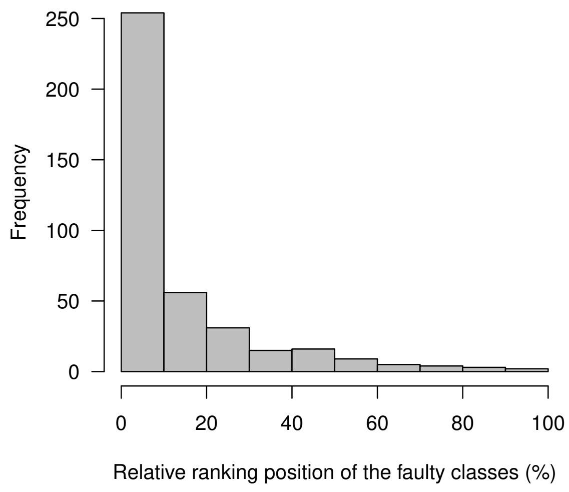


FIGURE 4.1: Relative ranking position of the buggy class.

RQ1.2: How effective is the best Schwa configuration at ranking a buggy class?

Given that the configuration determined above was calculated through a parameter tuning study of 30 bugs, it is possible to evaluate the configuration on the remaining 365 bugs that were not used in the parameter tuning experiment. Using the configuration found in Section 4.5.1, I calculate the ranking position, which is the index of the "true" buggy class in the ordering produced by Schwa. This value is then normalised across projects with different numbers of classes, which prevents bias against projects that have a larger number of classes. This value is referred to as the *relative ranking position*, which is calculated by dividing the ranking position by the total number of classes. Table 4.2 and Figure 4.1 report the relative ranking position of buggy classes. On average, the buggy classes of the Closure project appear in the first 9.2% of classes, with a total of 1478 classes, and a defect value of 0.89. As shown by Figure 4.1, a total of 267 bugs were correctly estimated within the top 10% of all classes in the subject programs. In fact, for 17 faults, Schwa ranks the buggy class as the most buggy one, and for 281 faults the relative ranking position of the buggy class is lower than the average value.

RQ1.2: Schwa ranks the buggy classes of all projects in the top 13.0%, with an average $defect_c$ *value of 0.86.*

RQ1.3: Assuming either an ideal or a non-ideal bug-prediction report, what are the best parameters for G-Clef?

As described in Section 4.4, G-Clef can be instantiated with different *secondary objective* functions and *grouping classes* values. To assess which combination of parameters works best (i.e., requires the execution of fewer test cases), I ran G-Clef on 365 faults with four different secondary objective functions (i.e., greedy, additional greedy, random, and constraints) and four grouping classes values (1, 5%, 10%, and 25%). As G-Clef relies on the outcome of a bug-prediction tool, I also defined two different scenarios to assess the influence of the underlying tool: 1) an *ideal* scenario in which a bug-prediction tool always places the buggy class first and 2) a *real* scenario when a bug-prediction tool ranks classes as previously described.

Table 4.3 reports the total number of test cases that must be executed in order to trigger the faulty behaviour of each real fault. For each configuration/project, Table 4.3 also reports the ranking position of each configuration at prioritising the fault revealing test case. For instance, if a configuration A ranks the trigger test in 3rd and configuration B ranks it in 16th, configuration A is ranked first and configuration B is second. In case of a tie, all configurations are ranked in the same position. As an example, for the Closure project the configuration requiring the execution of the fewest tests is constraints as a secondary objective and a grouping classes value of 1 (which ranked 5th, on average, among all configurations).

Overall,for the *real* bug-prediction scenario, G-Clef performs best with constraints as a secondary objective (5 out of 6 projects) and grouping classes value of 1 (3 out of 6 projects). On the other hand, for the *ideal* bug-prediction scenario, G-Clef works best with additional greedy as a secondary objective for 3 out of 6 projects (Chart, Lang, and Mockito).

The reason why the constraint solver performed relatively poorly with *ideal* bug prediction is that it also applies a minimisation to the test set. For example, if the buggy class is covered by 100 tests (including the trigger test), but the constraint solver finds a minimised set of 80 tests completely covers the class but does *not include* the trigger test, then G-Clef will not prioritise the trigger test until it covers another class. Future work could enhance the constraint solver secondary objective to ensure that all tests are used. Although G-Clef with constraints as a secondary objective only works best for 2 out of 6 projects (Closure and Time), overall it is ranked 2.4 (nearly the same as additional greedy). The overall configuration ranking, for both the *real* and *ideal* scenarios, is statistically significant according to the Friedman test.

RQ1.3: Assuming a perfect bug-prediction tool exists, G-Clef works best with constraints as a secondary objective and a grouping classes value of 1; for a real bug-prediction report additional greedy is the best secondary objective for G-Clef.

TABLE 4.3: Test case prioritisation results of G-Clef with different secondary objective functions and grouping classes values. The α column represents the *grouping classes* parameter (see Section 4.4.2 for more details), #t stands for the number of test cases that have to be executed in order to trigger the faulty behaviour, and R is the ranking position of a configuration. For the overall ranking position of each configuration the χ^2 and p-value of the Friedman test is also reported.

		Chart		Closure		Lang		Math		Mockito		Time		Overall	
Sec. Obj.	α	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R
Real bug-prediction data — $\chi^2 = 201.11$, $p - value < 0.0001$															
Greedy	1	626.8 (34.9%)	9.3	3404.2 (47.1%)	10.4	701.7 (37.3%)	8.3	1003.7 (35.1%)	8.7	528.5 (46.1%)	9.9	910.7 (23.2%)	7.8	1196.0 (38.1%)	9.3
Greedy	5%	691.4 (38.5%)	10.3	3436.3 (47.6%)	10.1	696.4 (37.0%)	7.7	927.6 (32.4%)	7.3	540.5 (47.1%)	10.1	954.6 (24.4%)	8.0	1207.8 (38.5%)	8.8
Greedy	10%	703.9 (39.2%)	9.6	3436.4 (47.6%)	10.1	670.9 (35.6%)	7.7	934.5 (32.7%)	7.7	540.5 (47.1%)	10.0	967.0 (24.7%)	8.5	1208.9 (38.5%)	8.9
Greedy	25%	790.6 (44.0%)	11.5	3436.0 (47.6%)	10.1	691.7 (36.7%)	8.2	948.7 (33.2%)	8.3	540.5 (47.1%)	10.1	932.1 (23.8%)	8.1	1223.3 (39.0%)	9.2
Add. Greedy	1	605.8 (33.7%)	7.7	2635.9 (36.5%)	7.6	711.8 (37.8%)	8.9	1032.0 (36.1%)	9.2	439.3 (38.3%)	6.8	987.1 (25.2%)	7.6	1068.6 (34.1%)	8.2
Add. Greedy	5%	559.1 (31.1%)	7.3	2783.2 (38.5%)	7.9	713.6 (37.9%)	8.4	998.7 (34.9%)	8.8	375.2 (32.7%)	6.5	1164.0 (29.7%)	9.4	1099.0 (35.0%)	8.2
Add. Greedy	10%	614.1 (34.2%)	7.9	2787.5 (38.6%)	8.6	748.2 (39.7%)	9.2	1024.0 (35.8%)	9.2	383.7 (33.5%)	7.0	1184.5 (30.2%)	10.5	1123.7 (35.8%)	8.8
Add. Greedy	25%	675.0 (37.6%)	10.0	2769.8 (38.4%)	9.5	825.0 (43.8%)	9.9	1070.8 (37.4%)	9.6	396.8 (34.6%)	7.9	1141.9 (29.1%)	10.4	1146.6 (36.5%)	9.5
Random	1	611.3 (34.1%)	8.2	2870.2 (39.7%)	9.1	712.4 (37.8%)	9.0	1016.7 (35.6%)	9.1	432.6 (37.7%)	7.2	1023.6 (26.1%)	8.8	1111.1 (35.4%)	8.8
Random	5%	576.4 (32.1%)	7.5	2806.9 (38.9%)	9.5	726.1 (38.5%)	8.8	986.5 (34.5%)	8.7	401.5 (35.0%)	7.6	1176.6 (30.0%)	9.8	1112.3 (35.4%)	8.9
Random	10%	553.5 (30.8%)	7.1	2786.3 (38.6%)	9.5	725.9 (38.5%)	9.1	1027.1 (35.9%)	9.2	410.2 (35.8%)	7.2	1205.9 (30.8%)	10.2	1118.2 (35.6%)	9.1
Random	25%	594.5 (33.1%)	9.0	2734.5 (37.9%)	9.3	761.1 (40.4%)	9.8	1086.2 (38.0%)	9.9	410.8 (35.8%)	8.4	1211.6 (30.9%)	10.2	1133.1 (36.1%)	9.5
Constraints	1	701.0 (39.0%)	7.2	1691.3 (23.4%)	5.0	708.6 (37.6%)	7.6	875.9 (30.6%)	7.1	452.0 (39.4%)	8.4	773.8 (19.7%)	5.9	867.1 (27.6%)	6.5
Constraints	5%	640.0 (35.7%)	6.8	1822.9 (25.2%)	5.8	780.5 (41.4%)	7.6	935.5 (32.7%)	7.5	444.7 (38.8%)	9.2	785.7 (20.0%)	6.0	901.6 (28.7%)	6.9
Constraints	10%	640.5 (35.7%)	7.3	1810.1 (25.1%)	6.5	789.7 (41.9%)	7.5	938.0 (32.8%)	7.6	447.3 (39.0%)	9.7	761.9 (19.4%)	6.6	897.9 (28.6%)	7.3
Constraints	25%	686.1 (38.2%)	9.1	1790.4 (24.8%)	7.1	824.2 (43.8%)	8.3	945.6 (33.1%)	8.2	432.2 (37.7%)	9.8	781.7 (19.9%)	8.1	910.1 (29.0%)	8.0
Ideal bug-prediction data — $\chi^2 = 39.63$, $p - value < 0.0001$															
Greedy	1	51.9 (2.9%)	3.2	1394.8 (19.3%)	3.2	36.9 (2.0%)	2.5	81.4 (2.8%)	2.7	229.6 (20.0%)	2.8	422.4 (10.8%)	2.5	369.5 (11.8%)	2.9
Add. Greedy	1	15.7 (0.9%)	1.8	879.7 (12.2%)	2.3	28.4 (1.5%)	2.2	78.9 (2.8%)	2.4	181.2 (15.8%)	2.0	431.6 (11.0%)	2.9	269.2 (8.6%)	2.3
Random	1	25.1 (1.4%)	2.5	839.8 (11.6%)	2.5	29.6 (1.6%)	2.5	75.2 (2.6%)	2.3	167.8 (14.6%)	2.2	539.4 (13.8%)	2.6	279.5 (8.9%)	2.4
Constraints	1	314.8 (17.5%)	2.5	1154.1 (16.0%)	2.0	349.3 (18.5%)	2.8	439.0 (15.4%)	2.6	335.7 (29.3%)	3.0	377.0 (9.6%)	2.1	495.0 (15.8%)	2.4

TABLE 4.4: Test case prioritisation results of G-Clef and coverage-based strategies. For each prioritisation strategy I report the total number of test cases (#t) that have to be executed to trigger the faulty behaviour, and its ranking position when compared to the other strategies.

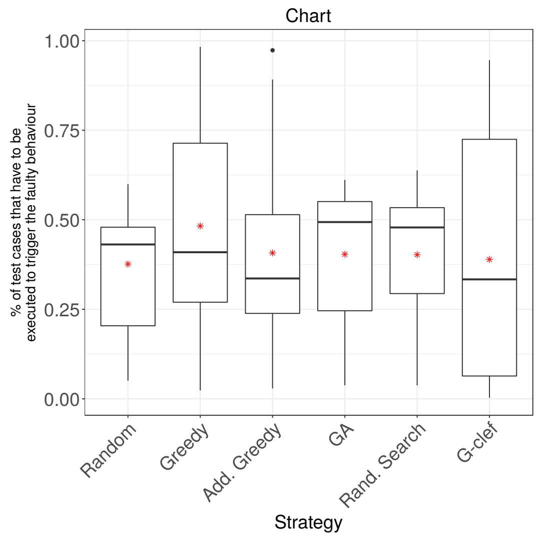
	Chart		Closure		Lang		Math		Mockito		Time		Overall	
Strategy	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R
$\chi^2 = 110.70,\ p - value < 0.0001$														
Greedy	859.1 (47.9%)	3.7	3439.4 (47.6%)	4.2	623.4 (33.1%)	2.6	909.5 (31.8%)	2.9	540.1 (47.1%)	4.1	970.0 (24.7%)	3.1	1223.6 (39.0%)	3.5
Add. Greedy	740.4 (41.2%)	3.6	1955.3 (27.1%)	2.6	939.8 (49.9%)	3.9	1046.2 (36.6%)	3.1	408.2 (35.6%)	3.1	953.1 (24.3%)	3.0	1007.2 (32.1%)	3.1
GA	719.4 (40.1%)	3.7	2817.6 (39.0%)	3.9	840.4 (44.6%)	3.9	1287.9 (45.0%)	4.1	423.9 (37.0%)	3.3	1385.3 (35.3%)	4.1	1245.8 (39.7%)	3.9
Random	674.6 (37.6%)	3.5	2811.0 (38.9%)	3.9	826.1 (43.8%)	3.6	1271.9 (44.5%)	4.1	425.5 (37.1%)	3.6	1410.2 (36.0%)	4.4	1236.6 (39.4%)	3.9
Rand. Search	717.7 (40.0%)	3.7	2828.7 (39.2%)	3.9	829.4 (44.0%)	3.6	1267.3 (44.3%)	4.0	422.1 (36.8%)	3.4	1400.6 (35.7%)	4.3	1244.3 (39.7%)	3.9
G-clef	701.0 (39.0%)	2.8	1691.3 (23.4%)	2.5	708.6 (37.6%)	3.3	875.9 (30.6%)	2.8	452.0 (39.4%)	3.5	773.8 (19.7%)	2.1	867.1 (27.6%)	2.8

4.5.2 RQ2: How does G-Clef compare to previously proposed coverage-based test case prioritisation strategies at prioritising manually-written test cases?

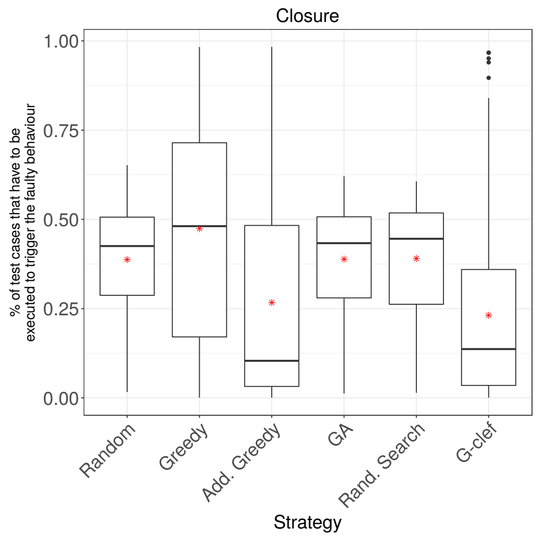
As stated in Section 4.4.8, KANONIZO has implementations for four coverage-based strategies that are commonly used in test case prioritisation evaluations, as well as a completely random ordering. Thus, for this research question, I compare G-Clef to these strategies. Since I use 30 subject programs for the tuning study in RQ1, those subject programs are eliminated from this RQ to avoid bias, leaving a total of 365 real faults. For each of these subjects, a script runs KANONIZO with each of the coverage-based strategies and the best configuration of G-Clef found in RQ1, and evaluates the prioritised test suite by the percentage of the test suite that is executed before the fault is found. Table 4.4 reports the average number of tests that are required to be executed before a fault is found across all projects and strategies, with the percentage of the test suite required reported in brackets. As shown by Figure 4.2, G-Clef often requires the fewest overall test cases in order to detect a fault (Closure, Math, Time). For the remaining three projects, in two cases (Chart and Lang) G-Clef was only beaten by a single other strategy.

TABLE 4.5: G-Clef vs coverage-based strategies. The # column reports the number of bugs for which G-Clef performed better than X and the total number of bugs per project, $\hat{A}$ column reports the effect size of X vs. G-Clef (a value lower than 0.5 means X performed worse than G-Clef, and a value greater than 0.5 means G-Clef performed worse than X), and p column reports the p-value of the Mann-Whitney U-test. Statistically significantly results at 95% significance level are given in bold-face.

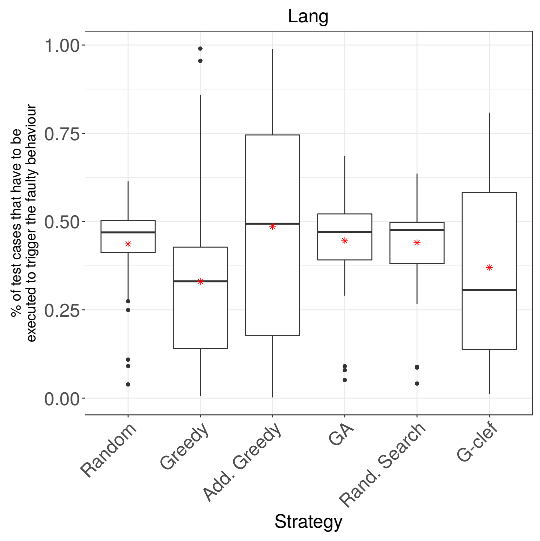
	Chart			Closure			Lang			Math			Mockito			Time			Overall		
Strategy	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p
Greedy	14 / 21	0.41	0.30	97 / 128	0.29	**0.00**	25 / 60	0.55	0.34	49 / 101	0.50	0.97	21 / 33	0.43	0.33	14 / 22	0.44	0.52	220 / 365	0.42	**0.00**
Add. Greedy	12 / 21	0.47	0.72	63 / 128	0.50	0.97	38 / 60	0.41	0.09	54 / 101	0.47	0.41	14 / 33	0.53	0.66	16 / 22	0.44	0.50	197 / 365	0.48	0.35
GA	14 / 21	0.44	0.48	96 / 128	0.25	**0.00**	33 / 60	0.41	0.10	73 / 101	0.28	**0.00**	13 / 33	0.48	0.79	18 / 22	0.23	**0.00**	247 / 365	0.31	**0.00**
Random	14 / 21	0.44	0.53	93 / 128	0.26	**0.00**	35 / 60	0.42	0.14	72 / 101	0.29	**0.00**	16 / 33	0.48	0.81	19 / 22	0.22	**0.00**	249 / 365	0.31	**0.00**
Rand. Search	14 / 21	0.44	0.48	98 / 128	0.26	**0.00**	32 / 60	0.42	0.14	73 / 101	0.28	**0.00**	17 / 33	0.48	0.80	18 / 22	0.24	**0.00**	252 / 365	0.31	**0.00**



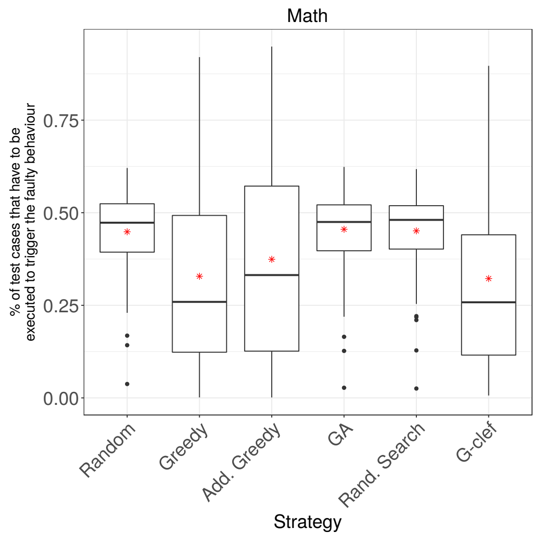
(A) Chart Project Statistics



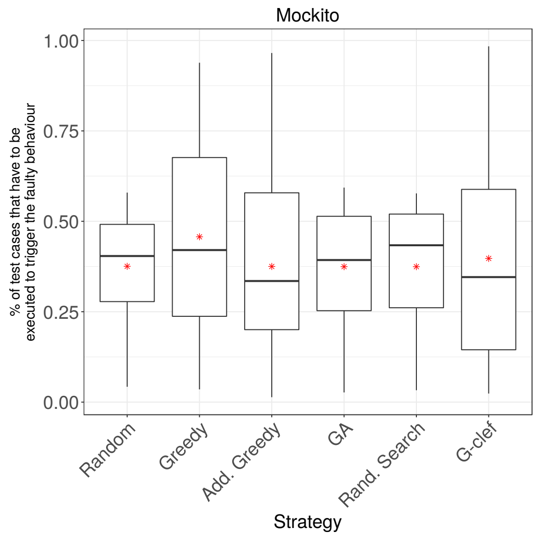
(B) Closure Project Statistics



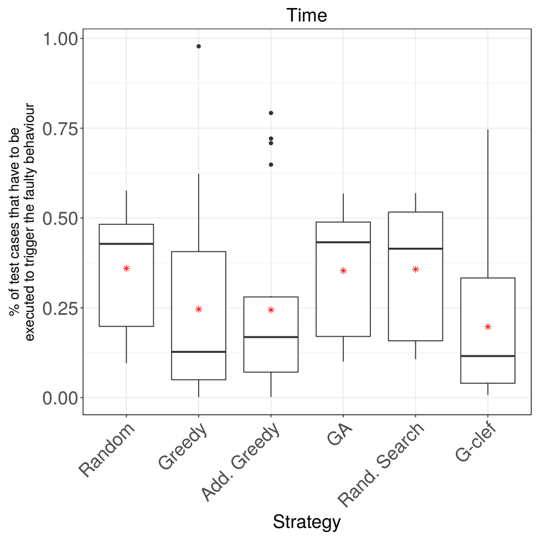
(C) Lang Project Statistics



(D) Math Project Statistics



(E) Mockito Project Statistics



(F) Time Project Statistics

$*$ represents the mean average % of test cases that have to be executed to trigger the faulty behaviour.

FIGURE 4.2: Test case prioritisation results of G-Clef and the coverage-based strategies.

Furthermore, as reported by Table 4.5, there are a number of cases in which G-Clef significantly outperformed other strategies, as reported by the Mann-Whitney U-Test. For the Closure project, G-Clef significantly outperformed all other strategies except additional greedy, while for both Math and Time, G-Clef significantly outperforms a further three strategies. Notably, there are only four combinations of project/strategy with an $\hat{A}$ score of > 0.5 (meaning on average the alternative approach is expected to outperform G-Clef). Overall, of the 1,825 combinations of subject/strategy included in this study, G-Clef performs best for 1,165, and significantly outperforms four of the five coverage-based strategies it was compared against.

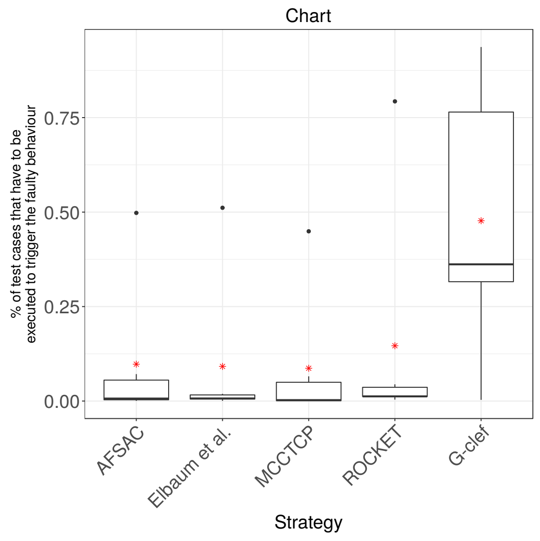
RQ2: G-Clef performs better than any other coverage-based strategy, statistically better than 4 out of 5 strategies.

4.5.3 RQ3: How does G-Clef compare to previously proposed history-based test case prioritisation strategies at prioritising manually-written test cases?

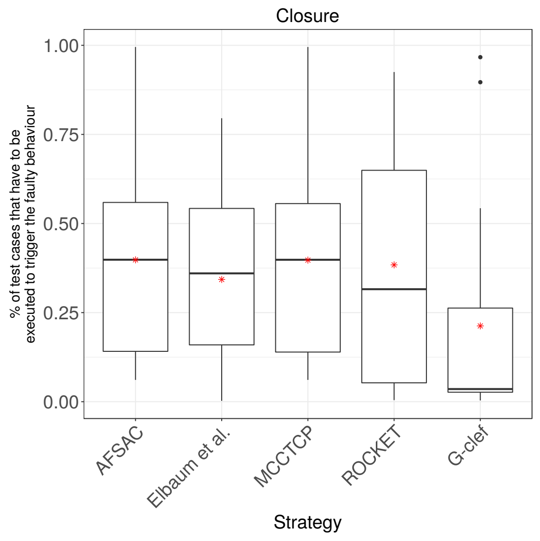
As with RQ2, RQ3 involves the execution of G-Clef compared against four history-based approaches described in Section 3.2.4. In the Defects4J dataset, there are a substantial number of subject programs for which the test case that detects a fault has no execution history. This may cause an unnecessary bias either in favor of or against certain history-based strategies, since some strategies rely on the number of prior executions and/or failures. In order to give a fair opportunity to all strategies, I only include bugs where the trigger test has at least one prior execution before the current version of the subject program. This means that this RQ considers 82 bug s. Table 4.6 reports the average number of tests that have to be executed before a fault is found across all projects and strategies. For four of the six projects (Closure, Lang, Math, Time), G-Clef had the lowest number of test cases required of any strategy. Additionally, as shown by Table 4.7, G-Clef was significantly better for five project/strategy combinations, and was only significantly outperformed once (Chart/MCCTCP [52]). For the Time project, while the Vargha-Delaney $\hat{A}$ effect size was 0.00 for three of the four competing approaches, due to only having three bugs for this project, I was unable to achieve a significant result for this project.

Table 4.6: Test case prioritisation results of G-Clef and history-based strategies. (Please refer to Table 4.4 for an explanation of each column.)

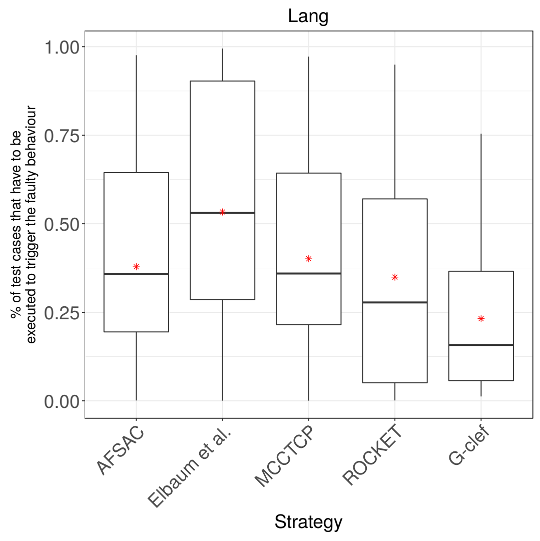
	Chart		Closure		Lang		Math		Mockito		Time		Overall	
Strategy	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R	#t	R
$\chi^2 = 15.87$, $p - value = 0.003$														
G-clef	854.7 (46.3%)	3.7	1576.0 (21.6%)	2.0	437.6 (24.0%)	2.2	931.0 (33.5%)	2.5	344.7 (26.6%)	3.8	439.7 (11.0%)	1.0	763.9 (24.1%)	2.5
ROCKET [54]	243.0 (13.2%)	3.8	2873.1 (39.4%)	3.3	628.3 (34.4%)	2.8	1270.3 (45.8%)	3.3	162.0 (12.5%)	3.0	2842.7 (71.2%)	3.7	1336.6 (42.1%)	3.2
Elbaum et al. [55]	151.2 (8.2%)	2.7	2452.2 (33.6%)	2.9	984.9 (54.0%)	3.7	1474.3 (53.1%)	3.4	392.3 (30.3%)	3.0	1521.3 (38.1%)	2.7	1162.7 (36.7%)	3.2
MCCTCP [52]	147.2 (8.0%)	1.9	2849.1 (39.1%)	3.3	734.9 (40.2%)	3.2	956.7 (34.5%)	2.5	169.3 (13.1%)	2.2	2619.7 (65.6%)	3.8	1246.1 (39.3%)	2.9
AFSAC [56]	165.7 (9.0%)	2.9	2854.6 (39.1%)	3.5	694.1 (38.0%)	3.1	980.2 (35.3%)	3.2	198.0 (15.3%)	3.0	2619.7 (65.6%)	3.8	1252.0 (39.5%)	3.2



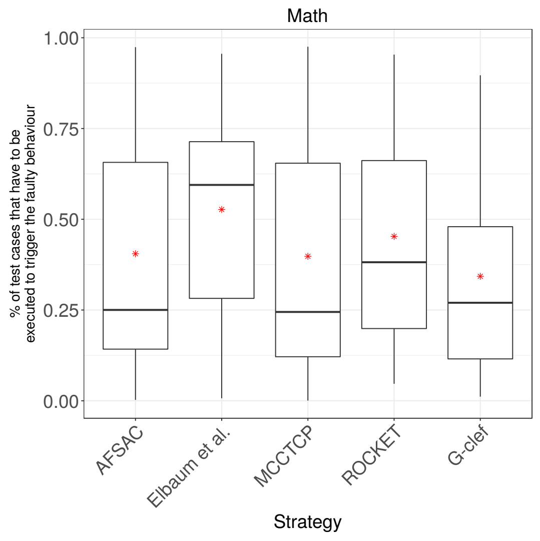
(A) Chart Project Statistics



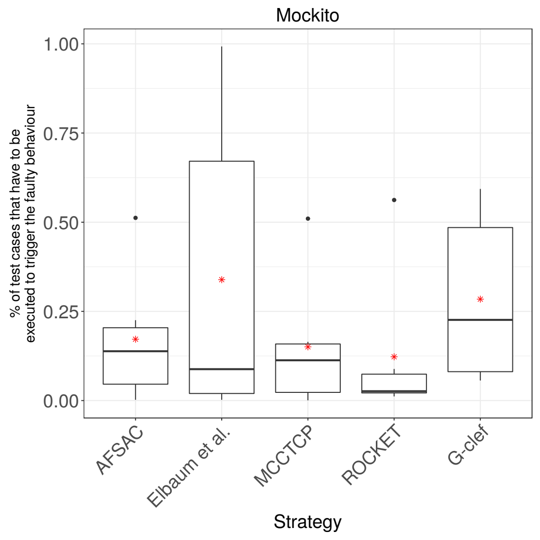
(B) Closure Project Statistics



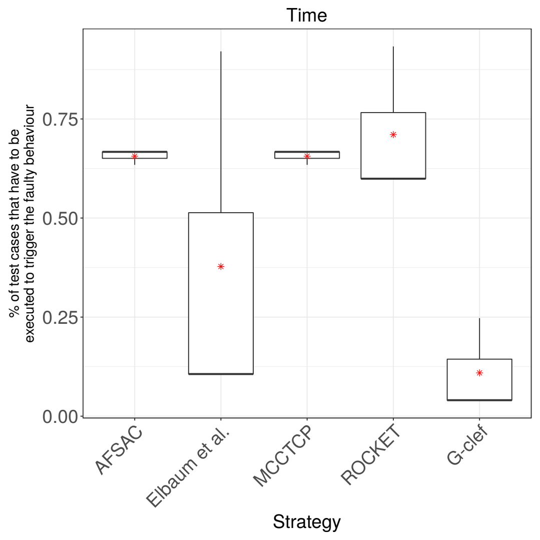
(C) Lang Project Statistics



(D) Math Project Statistics



(E) Mockito Project Statistics



(F) Time Project Statistics

$*$ represents the mean average % of test cases that have to be executed to trigger the faulty behaviour.

FIGURE 4.3: Test case prioritisation results of G-Clef and the history-based strategies.

Table 4.7: G-Clef vs history-based strategies. (Please refer to Table 4.5 for an explanation of each column.)

	Chart			Closure			Lang			Math			Mockito			Time			Overall		
Strategy	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p	#	$\hat{A}$	p
ROCKET [54]	2 / 6	0.76	0.15	15 / 20	0.34	0.08	12 / 21	0.43	0.42	15 / 26	0.39	0.16	2 / 6	0.78	0.13	3 / 3	0.00	0.08	49 / 82	0.42	0.09
Elbaum et al. [55]	2 / 6	0.78	0.13	13 / 20	0.36	0.15	15 / 21	0.27	**0.01**	19 / 26	0.32	**0.03**	3 / 6	0.56	0.81	3 / 3	0.22	0.38	55 / 82	0.38	**0.01**
MCCTCP [52]	2 / 6	0.86	**0.04**	16 / 20	0.25	**0.01**	16 / 21	0.32	**0.04**	15 / 26	0.46	0.63	1 / 6	0.67	0.38	3 / 3	0.00	0.08	53 / 82	0.39	**0.02**
AFSAC [56]	2 / 6	0.81	0.09	16 / 20	0.25	**0.01**	15 / 21	0.37	0.16	15 / 26	0.45	0.55	1 / 6	0.64	0.47	3 / 3	0.00	0.08	52 / 82	0.40	**0.02**

Yet, G-Clef overall achieved significantly better results than three of the four history-based strategies evaluated in this paper, and outperformed ROCKET for 49 out of the 82 bug s used in this evaluation.

Figure 4.3 contains a boxplot showing the percentage of test cases executed before the trigger test. One of the most noticeable results in Figure 4.3 is how effective history-based strategies were on the Chart project. On average, only 10.5% of the total test cases were required to find a fault, and for four of the six Chart subjects used, at least one of the history-based strategies was able to detect the fault in fewer than 10 test cases.

RQ3: G-Clef performs better than any other history-based strategy, statistically better than 3 out of 4 strategies.

4.6 Conclusions

This chapter presents a new strategy for prioritising test cases, called G-Clef that uses defect prediction for test case prioritisation. I present a large parameter tuning experiment in which I aim to maximise the performance of defect prediction on real faults from Defects4J, and a large empirical evaluation of G-Clef in which I compare against four coverage-based strategies and four history-based strategies. This chapter's results indicate that G-Clef is effective at prioritising test suites to find real faults, requiring the fewest test cases in most of the experiments, and never being significantly outperformed by any other strategy.

While the results of this paper show a promising link between defect prediction and test case prioritisation that can be leveraged, there is still a lot of room for improvement. For example, the defect prediction tool on average predicted the "true" buggy class as 13.0%. As defect prediction strategies improve, G-Clef will be able to find faults faster.

Chapter 5

Using Sentiment in Commit Messages to Predict Whether a Class is Faulty — An Investigative Study and Implications for Test Case Prioritisation

5.1 Introduction

The previous chapter identified that repository information, specifically that the number of changes, authors and fixes, can be useful determiners of future faults in software repositories. Using this information, I developed a test case prioritisation strategy G-Clef that utilised the defect prediction tool Schwa, and conducted an empirical evaluation of its effectiveness against a number of existing strategies. In particular, Chapter 4 shows a connection between defect prediction and test case prioritisation, and demonstrates that an improved approach to defect prediction could result in benefits for test case prioritisation.

In this chapter, I investigate whether commit messages to Version Control Systems (VCS) can be used to predict defects in faults. Similar to the work of Binkley et al. [142], who used the cosine similarity between comments in code and the code itself to predict whether the code was likely to contain faults, this chapter aims to identify whether developers write commit messages that are contain strongly positive or negative emotions, and to identify whether emotions within commit messages correlate with either bug-fixes or faults. If developers write more negatively about faulty files, then a prediction model could identify the files that are likely to contain faults through the set of commits associated with the file. There are a number of possibilities that could arise from this — in particular, in line with the rest of this thesis' work, a test case prioritisation strategy could be developed that would rank all files according to the sentiments associated with them, and then identify test cases that cover the most negative files.

This chapter aims to answer the following research questions: **RQ1: Do commits in** DEFECTS4J **contain sentiment or subjectivity?**, **RQ2: Can sentiment or subjectivity in commit messages be used to predict whether a commit fixes a bug?** and **RQ3: Can sentiment or subjectivity in commit messages be used to predict whether a file contains a fault?**

The contributions of this chapter are as follows:

Contribution 5.1: *An evaluation of sentiment and subjectivity in* DEFECTS4J *commit messages*

Contribution 5.2: *An evaluation of sentiment and subjectivity in bug-fix commits against non-bug-fix commits*

Contribution 5.3: *An evaluation of sentiment and subjectivity in faulty files against non faulty files*

The remainder of this chapter is organised as follows: Section 5.2 introduces the background of this chapter, specifically focusing on sentiment analysis and repository mining, which are both used in this study. Section 5.3 introduces the experiments conducted. Section 5.4 describes the results, before Section 5.5 presents a discussion about features of bug-fix commits and faulty files. Furthermore, Section 5.5 discusses whether commit messages could be used in test case prioritisation. Finally, Section 5.6 concludes this chapter.

5.2 Background

When working in large software teams, developers often make use of version control systems (VCS) in order to collaborate and ensure that changes to files are kept. VCS allow many developers to work on projects simultaneously, with developers "checking in" changes frequently. With every VCS check in, there is an associated message, which is supposed to contain an explanation from the developer about the changes that have been made, and sometimes justifications for why the changes have been made.

The intention of commit messages is to inform other collaborators on the project about what is different — at a glance, developers should know roughly which files have been changed and why. However, sometimes developers will use emotive language when describing the changes they have made. For example, in the Apache Commons Math project[1] there is a commit with the message `"Greatly improved multiplication speed for sparse matrices. Jira: MATH-248"`. This is encouraging for two reasons: firstly, it motivates the idea that sentiment may be present in commit messages ("greatly improved"), and secondly, this commit appears to be associated with a bug fix, as it references an issue number in the bug-tracking software JIRA[2]. Conversely, the Mockito project[3] contains a commit with the message `"remove this ugly mockito.iml that is a pain in the ass!!!! --HG--"`. This clearly shows that developers may write emotively about specific files that may have caused problems in the past.

5.2.1 Repository Analysis

The term *Mining Software Repositories* (MSR) refers to the process of extracting information from VCS, such as Git, incorporating changes to source code alongside metadata such as the commit author, commit date or commit message. There are many ways in which mining repositories can be useful, for example in studies how test code evolves with source code [164], how code smells change over time [165] or even locate real faults from software repositories, such as Defects4J [20].

5.2.2 Sentiment Analysis

Given a string containing a message that was entered alongside a commit, the goal of sentiment analysis is to identify the overall emotion associated with the string. Sentiment analysis has its origins in online review platforms, for example movie reviews from people who have seen them [166–168], or product reviews from people who have bought things from online retailers such as Amazon [125, 169–171]. For producers, being able to see what real customers are saying about them is useful, and being able to automatically determine the general impression of the entire customer base without having to manually read all reviews, is highly useful.

[1]https://commons.apache.org/proper/commons-math/
[2]https://www.atlassian.com/software/jira
[3]https://site.mockito.org/

The literature survey of this thesis reveals three main ways in which sentiment analysis is performed. The first is a supervised approach, whereby a large amount of data is manually tagged as one of two classes (i.e. positive/negative), and used as training data for a classification approach. The classifier may then produce rules that dictate what sort of phrases or words appear in both classes. When these rules are applied to new data, the classifier determines automatically which class the new data belongs to [172].

The second approached identified in previous literature is an unsupervised approach, which attempts to infer opinion words based on the words they are most closely associated with [135].

The third approach involves the creation of a large lexicon of "opinion" words, each of which is tagged with an approximation of how *polarising* the word is. For example, the word "great" is a strongly positive word, while "good" is only slightly positive. With the lexicon of words and their associated polarity scores, a sentiment analysis approach will sum up the polarity scores of all words contained in a piece of text [172].

Some of the crucial challenges associated with sentiment analysis are the use of *intensifiers* and *negators*. For example, if something is described as "really good", the sentiment is strong than simply being described as "good". This example shows the use of an *intensifier*, a word that increases or decreases the strength of the subsequent word. Conversely, *negators* reverse the opinion of the following word, for example "not good" has the opposite meaning to "good". Identifying the use of intensifiers and negators in sentences is crucial to the success of a sentiment analysis approach.

Sentiment analysis of commit messages has received some previous attention in research. Guzman et al. [139] conducted a study in which they compared the sentiment of commit messages to the programming language used, the day/time at which the commit was made, the location of developers and the project approval. Souza et al. [140] conducted a study that correlated sentiment of commit messages with Travis CI builds breaking, finding that *strongly negative* sentiments in commits are more likely to lead to broken builds. Islam et al. [141] conducted a study in which they considered bug-introducing and bug-fixing commits, aiming to discover if those specific commits are more likely to be positive or negative, and how bug-introducing and bug-fixing commits compare in terms of sentiment. Additionally, while it does not concern commit messages, Binkley et al. [142] conducted a study in which they used comments in code to predict the likelihood that files contained faults. This chapter conducts similar experiments using commit messages rather than code comments.

5.2.3 Using sentiment in commits to predict fault-fixing commits

In Chapter 4, I demonstrate the effectiveness of a test case prioritisation strategy based on defect prediction. Specifically, for each class, defect prediction gives an estimate that the class contains a fault. While the experiments in Chapter 4 were successful, they indicated further improvements were possible through a better defect prediction strategy.

As software evolves and developers make changes, they commit changes to a VCS with a message explaining what has changed and why, allowing other developers to understand when they “pull” the most recent changes. Over time, every file that makes up a program will have an associated set of messages. This effectively forms a “profile” for the class. Sliwerski et al. [89] have previously attempted to model bug-fixing commits with the SZZ algorithm — in their approach, they used regular expressions to pick out certain keywords in the commit message (e.g. “fixes” or “defects”), as well as recognising bug numbers that could correspond to a bug-tracking system like bugzilla[4] or JIRA[5]. For each recognised feature, Sliwerski et al. add one to a *syntactic confidence* that a commit fixes a bug. If this chapters experiments demonstrate that the use of sentimental or subjective language in commit messages can be used to aid the identification of commits as bug-fixing, then this could be used to extend the syntactic confidence in the SZZ algorithm [89].

If the set of commit messages for a class shows that developers often write negative messages, it could be a sign that the class is frustrating, tricky to understand, or badly designed. Since badly designed software can lead to more bugs [24], I design an experiment that aims to investigate whether negative commit messages can be used to predict classes that may be faulty. This could be used to integrate into the SZZ algorithm further syntactic confidence that a commit is bug-fixing. Notably, if commit messages can be used to predict faulty classes, G-Clef proposed in Chapter 4 can be extended to utilise this information.

5.3 Experimental Setup

This section describes the experimental design, including subject programs, analysis steps and the research questions this study answers.

[4]https://www.bugzilla.org/
[5]https://www.atlassian.com/software/jira

5.3.1 Subject Programs

This chapter has two main aims: identifying whether bug-fix commits are associated with stronger sentiment than non bug-fix commits, and identifying whether faulty files are associated with stronger sentiment than non-faulty files. In order to achieve this, I required subject programs with known faults, where it must be possible to identify the bug-fixing commit, and must be possible to identify which file(s) in a program are faulty. Defects4J provides the required information for this study. For each project, Defects4J contains a file that has, for each fault in the program, a row of data in the form `<bug_id>,<last_faulty_commit_hash>,<bug_fix_commit_hash>`. In addition, for each bug, Defects4J contains a file that lists the classes that were modified as part of the bug fix. By combining this information, there is a complete set of faulty files, and a set of commit hashes that were involved in bug-fixes.

5.3.2 Repository Analysis

In order to analyse subject programs in Defects4J, I required a way to extract all of the commit messages for each program. There are a number of tools available for this, for example `git log`, which comes pre-packaged with Git version control. In addition, I needed to know for each commit the set of files that was changed as part of the commit. This can also be achieved with pre-packaged Git tools, using `git show`.

However, there are also several libraries for analysing repositories in different languages (e.g. GitPython [173], JGit [174]). Specifically, the PyDriller library [175] is an easier way of analysing Git repositories in Python. PyDriller provides simple APIs for iterating through commits in a Git repository and for extracting information about individual commits, including the name of the developer, the set of files changed, the date of the commit and the message associated with the commit.

In Defects4J, five of the six projects use Git as a VCS by default. However, the JFreeChart project is built using SVN as its VCS. While I could have used another library to analyse the SVN repository, for consistency purposes I instead applied a patch[6] that converts the JFreeChart repository to Git using the Git-SVN tool [7].

Finally, since the Chart, Lang, Math and Time projects in Defects4J have been converted from SVN to Git at some point, there is some metadata added by the Git-SVN tool that allows mapping of Git commits back to its SVN origin. This is usually a line at the bottom of the commit of the form `git-svn-id: <url>`. Since this information adds

[6]Patch: https://github.com/jon-bell/defects4j/commit/c8b3d3792331bd989d512bda893e23b21b0aae6e.patch
[7]Git-SVN: https://git-scm.com/docs/git-svn

noise to the sentiment analysis process, these lines of commit messages were removed prior to running sentiment analysis.

5.3.3 Sentiment Analysis

This chapter's experiments look at how sentiments in commit messages are associated with either bug-fixes or faulty files. There are a large number of sentiment analysis packages available for developers, including NLTK[8], TextBlob[9], SentiStrength[10], SentiWordNet[11] and IBM's Watson Natural Language Understanding (formerly Alchemy)[12]. Jongeling et al. [176] conducted a study in which they used the some of above packages to determine whether consistent results could be achieved from each of them, collecting over 95,000 bug reports from four different sources and analysing each of them with all of the tools to see if they agree on the sentiment present in bug reports. From their analysis, the authors could not define a clear "best" tool since they did not compare the accuracy of the tools, but did note that the use of different tools may lead to different conclusions.

In order to analyse the sentiments associated with each commit, I used the TextBlob library [177]. TextBlob has been used in a number of studies [178, 179] and is built on top of NLTK, which has been used in many further studies [180–182].

TextBlob has a large corpus of "opinion" words combining both hand-tagged and inferred examples. All words contain a `polarity` p, a score $-1 \leq p \leq 1$ which indicates whether the word is strongly negative ($p = -1$) to strongly positive ($p = 1$), a `subjectivity` score s where $0 \leq s \leq 1$, which indicates whether the word is objective ($s = 0$) or subjective ($s = 1$), an `intensity` score i, where $0.5 \leq i \leq 2$ that indicates that the *following* word should have a modifier i applied to it (e.g. *very* good, *slightly* better). TextBlob also contains negations, by default these are "no", "not", "n't" and "never". These words change positive polarity for subsequent words to negative and vice-versa. Finally, TextBlob contains multiple definitions for the same word that can apply in different contexts, for example the word "ridiculous", which can be interpreted as either pitiful or humorous.

For every sentence, the TextBlob library tokenises the sentence, grouping together negators, intensifiers and opinion words together. For example, for the sentence "some really great sample text, a good example and a not bad negation", TextBlob will group

[8] https://www.nltk.org/
[9] https://textblob.readthedocs.io/en/dev/
[10] http://sentistrength.wlv.ac.uk/
[11] http://sentiwordnet.isti.cnr.it/
[12] https://www.ibm.com/cloud/blog/announcements/bye-bye-alchemyapi

the words "really"/"great", and "not"/"bad", since these words must be considered as a pair rather than individuals. TextBlob also identifies the word "good" as a word expressing sentiment, while the remainder of the sentence is discarded. Each of the tokens is then associated with a score, based on the combined polarity and subjectivity of the components within the token. In this example, "really great" has $p = 0.8$ and $s = 0.75$, "good" has $p = 0.7$ and $s = 0.6$, and "not bad" has $p = 0.35$ and $s = 0.\overline{6}$. The sentiment for the whole sentence is then calculated by taking the average of the polarity and subjectivity scores for each of the tokens, resulting in this sentence having a polarity of $0.61\overline{6}$ and a subjectivity of $0.67\overline{2}$.

5.3.4 Experimental Setup

This study aims to answer three research questions:

1. **RQ1:** Do commits in Defects4J contain sentiment or subjectivity?
   With this research question, I investigate the commits in Defects4J subject programs. Some other studies (e.g. Guzman et al. [139]) have considered the impact of different programming languages on the amount of sentiment present. This research question extends this previous research with new subjects.

2. **RQ2:** Can sentiment or subjectivity in commit messages be used to predict whether a commit fixes a bug?
   Defects4J provides a set of commit hashes that were involved in bug-fix commits. Using these commit hashes, it is possible to compare on a per-project basis whether commits that fix real faults have higher sentiment or subjectivity scores than those that do not. If commits with higher sentiment scores usually result in bug fixes, it may be possible to automatically classifying commits based on this information, which could aid the field of MSR. While Islam et al. [141] conducted a study of the sentiments of bug-introducing and bug-fixing commits, their focus was only on whether each type of commit was usually more *positive* or *negative*. Their study did not include a comparison against the remaining commits to see if there were any differences.

3. **RQ3:** Can sentiment or subjectivity in commit messages be used to predict whether a file contains a fault?
   With this research question, I consider the files that are part of the source code of a system under test. Every file in a system will have a minimum of one commit, the one in which it was introduced, and may be involved in a number of changes. This research question aims to discover whether there is a difference in how developers feel about files that are faulty compared to those that are not. This has potential

applications in the fields of defect prediction or test case prioritisation if, for example, developers often write negatively about files that contain faults than those that do not. For example, a defect prediction strategy may, for each file in the system, look at either the average or the most extreme commit messages associated with the file, and determine the likelihood of faulty behaviour based on these numbers.

5.3.5 Analysis Procedure

For each of the six projects in DEFECTS4J, there is have a series of commits. For each commit, there is a commit message, and a set of files changed. For this chapter, I created a script that used PyDriller to obtain all commit messages for a subject program, and used TextBlob to analyse the sentiment and subjectivity scores. The script also used DEFECTS4J to identify bug-fixing commits and faulty files, resulting in a file similar to the one in Table 5.1. These examples are selected for readability, and as such were selected due to having short message lengths. In additon, the `file_changed` field has been shortened to only include the final part of the path.

project	commit_hash	commit_msg	file_changed	sentiment_score	subjectivity_score	bug_fix_commit	is_faulty_class
Chart	a99d91	Tidy up.	/DefaultKeyedValuesTests.java	0.60	0.80	FALSE	FALSE
Math	b7d598	Raw type	/MathParseException.java	-0.23	0.46	FALSE	FALSE
Chart	b53ed3	Clean up.	/MovingAverage.java	0.37	0.70	FALSE	FALSE
Chart	b9d789	New test.	/RelativeDateFormatTests.java	0.14	0.45	FALSE	FALSE
Chart	6b1346	New test.	/LogFormatTests.java	0.14	0.45	FALSE	FALSE

TABLE 5.1: Example data collected in this study

For RQ1, I only consider the `sentiment_score` and `subjectivity_score` fields. For RQ2, I also consider the `bug_fix_commit` field, while for RQ3, I add the `is_faulty_class` field. RQ2 and RQ3 also require the use of statistical tests to determine whether the difference in sentiment and subjectivity are significant between bug-fix and non bug-fix commits, and between faulty and non-faulty files. For this, I use the Mann-Whitney U-Test, and the Vargha-Delaney $\hat{A}$ score is used to quantify the size of the difference.

5.3.6 Threats to Validity

There are a number of potential threats to the validity of this study. Firstly, this study only considers programs that are built in Java, which may not necessarily generalise to other programming languages. Indeed, Guzman et al. [139] looked at several programming languages to determine whether the language of a project had a significant impact on the sentiments attached, discovering that Java programs have more negative sentiments than other languages. However, this does not impact this study, which aims only to show the differences in sentiment between bug-fix and non bug-fix commits, or between faulty files and non-faulty files.

Secondly, all of the subject programs from DEFECTS4J are from professional organisations and are all mature programs (>= 5 years development). This may make it less likely that commits express strong sentiments, since organisations may have standards that must be observed for every commit to keep commit messages clean and objective. Future work will investigate this threat by repeating experiments with more subjects from GitHub.

This study assumes that the set of bug-fix commits reported by DEFECTS4J is complete, and therefore that all other commits in the repository do not fix faults. This assumption may not hold, since there are other commit messages in the dataset that refer to bug-trackers. DEFECTS4J includes a subset of the total faults present in a repository, and only includes those that can be clearly identified by at least one test case. To mitigate this, I also tested what happens if commits that contain either “JIRA” or “fix” and do not fix a bug are omitted, observing no significant differences in the results.

The TextBlob library uses a lexicon-based approach for sentiment analysis — this means that there is no training process required in order to produce estimations for sentiment and subjectivity. However, given the specific context of the dataset used (i.e. commit messages), an approach that includes a training step using manually annotated data, for example NLTK, may have yielded better results. Although time constraints meant that it was not possible to manually annotate a corpus of training data, future work will investigate whether this can result in a more positive classification process.

Finally, since this study makes use of external tools, it is possible that bugs in the external tools may result in errors in my evaluation. Despite being a relatively new tool, the PyDriller tool has received some attention from other researchers (e.g. [183–185]). Additionally, the TextBlob library has been used for sentiment analysis in research [186–188]. Since it is possible that I may have misused the libraries available, the scripts used to produce and analyse the data are also made publicly available, such that external researchers can reproduce and verify the results obtained in this study[13].

[13] https://github.com/djpaterson/commit_sentiment_analysis

Project	Positive	Negative	Neutral	Total
Chart	186	44	683	913
Closure	810	357	1720	2887
Lang	605	440	2533	3578
Math	1183	748	2948	4879
Mockito	617	243	2207	3067
Time	201	115	1369	1685

TABLE 5.2: Number of positive, negative and neutral commit messages per project

Word	Count	Average Sentiment
handy	12	0.38
tidy	39	0.38
great	22	0.31
nice	33	0.22
good	37	0.22
crash	35	0.06
cool	2	-0.13
missing	381	-0.15
stupid	5	-0.28
bad	51	-0.32
nasty	7	-0.62
terrible	1	-1

TABLE 5.3: Sentimental words found in commits

5.4 Results

5.4.1 RQ1: Do commits in DEFECTS4J contain sentiment or subjectivity?

Table 5.2 shows the number of positive, negative and neutral commit messages from the 17009 commits considered in this study. One of the apparent trends in this table is that in most cases, commit messages do not contain much, if any, sentiment. In total, 11460 (67.38%) commits returned a sentiment score of 0, indicating complete neutrality in these messages. This is not altogether surprising, since the purpose of including commit messages is to convey objectively the changes that have been made in the commit. However, there are clear exceptions to this. The Closure project has 810 positive commit messages out of 2887 total (28.06%), while Math has the highest percentage of negative commits with 748 (15.33%). In particular, commits for the Math frequently refer to `missing` tags, methods or comments, which results in largely negative commits, while commits for the Closure often refer to `more useful` or `better` approaches.

Another trend that is apparent in Table 5.2 is that it is more common for commit messages to be positive than negative, with every project containing a higher number of positive messages. Table 5.3 displays a number of words that were found in multiple positive and negative commit messages respectively. A large number of positive commit messages contained the word `tidy`, with a number of commits for each project that claimed to either tidy some documentation, comments, or methods. Furthermore, for some of the most negative commit messages, the sentiment analysis tool failed to pick up the fact that the word `fix` was present alongside a negative word such as `bad` or `stupid`. For example, commit hash `1aba91` in the Closure project contains the message `Fix a bug where we wouldn't warn about bad parameters` — despite being picked up as a negative sentiment, this should actually be positive as it is addressing an issue.

Despite the word "fix" being present in 1172 commit messages, the average sentiment for messages that contained the word "fix" was 0.04, showing that on its own, the word fix is not enough to indicate a positive or negative message. Moreover, while the word `cool` appears in two commits, it has a negative average sentiment score. This is because one of the commits in which the word is included contains the message `-ant doesn't work yet :( -refactored some names -rename : src/org/mockito/Matchers.java => src/org/mockito/CrazyMatchers.java rename : src/org/mockito/CoolMatchers.java => src/org/mockito/Matchers.java` (Mockito commit `da791d`). In this instance, the overwhelmingly negative sentiment of Ant not working, causes a deeply negative score, while the word `cool` isn't counted since it is part of a longer string.

Finally, Figure 5.1 shows the comparison of subjectivity scores with sentiment scores for the commit messages analysed. From Figure 5.1, it is clear that as commit messages become more subjective, there is a higher distribution of sentiment scores. This indicates that is difficult to compose commit messages that are objectively stating facts whilst not giving any opinions. Of the nine commits that contain positive sentiment without any subjectivity, the word `useful` appears in eight commits. For example, Time commit `fc46ba` contains the message `DayOfWeek not that useful`, which is stated as an objective fact. Conversely, of the seven commits that have negative sentiment without any subjectivity, the phrases `not useful` and `harder` appear two times each. In most cases however, it is not possible to achieve a high sentiment score without also having a high subjectivity score. By contrast, there are 193 cases where the subjectivity score is 1 while the sentiment score is 0. By inspecting these cases, the word `final` appears in 66 of those cases. This shows a contextual difference in how sentiment in general text may compare with sentiment in programming (as with `fix`), since the 'final' keyword in Java is applied to variables and methods and may, according to certain style guides, even be required [14].

RQ1: Most commit messages (67.38%) do not contain sentiment. The amount of sentiment in commits may depend on several project factors — the number of developers, the age of the project, the organisation in control of the project etc. Although many commits may contain negative sentiment, this may be a limitation of the sentiment analysis tool, since the word "fix" is not considered a positive one.

5.4.2 RQ2: Can sentiment or subjectivity in commit messages be used to predict whether a commit fixes a bug?

This section compares the sentiment and subjectivity scores of commits that fix bugs in Defects4J and commits that don't. Figures 5.2 and 5.3 show the sentiment and

[14]https://checkstyle.sourceforge.io/apidocs/com/puppycrawl/tools/checkstyle/checks/coding/FinalLocalVariableCheck.html

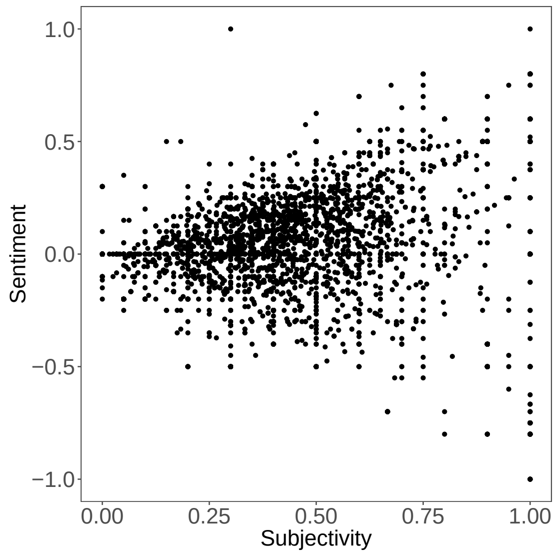


Figure 5.1: Subjectivity score of commit messages compared with sentiment score

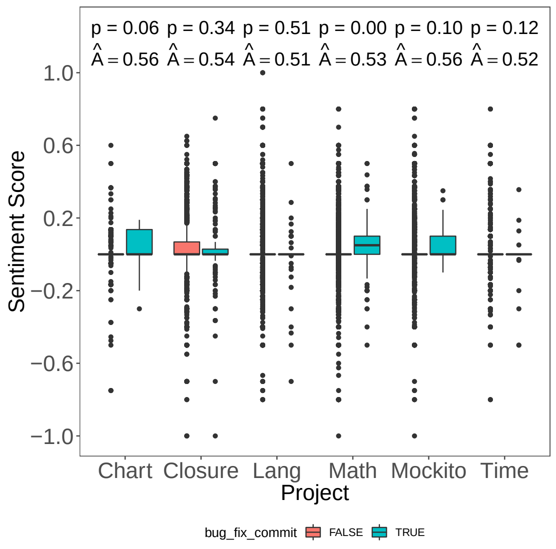


Figure 5.2: Sentiment scores of messages from commits that fix faults and commits that do not

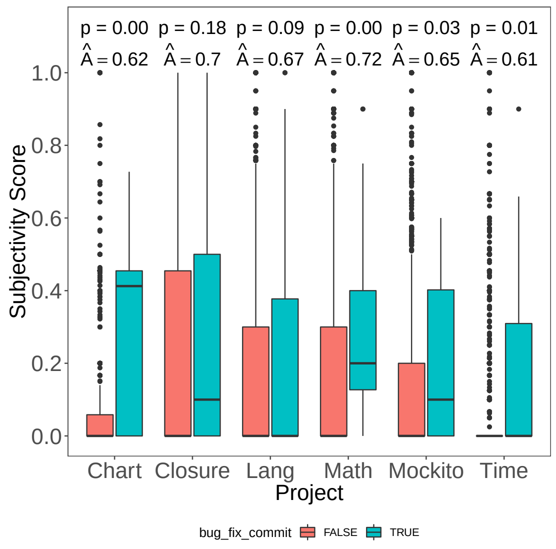


Figure 5.3: Subjectivity scores of messages from commits that fix faults and commits that do not

subjectivity scores respectively of each project analysed during this study. From this, it is clear that in most cases, the sentiment scores and subjectivity scores of commits that fix faults is higher than the scores of those that do not. The only exception to this is the sentiment score of the Closure project, which has a very slightly higher sentiment score for non-bug fix commits than for bug fix commits. Figures 5.2 and 5.3 also contain the results of the Mann-Whitney U-Test and Vargha-Delaney effect sizes for each project. This displays a large variance in the results. For example, in Figure 5.2, the Math project has a p value of 0.00, indicating that sentiment score may be a good indicator of whether a commit fixes a bug. However, the Lang shows the complete opposite, with

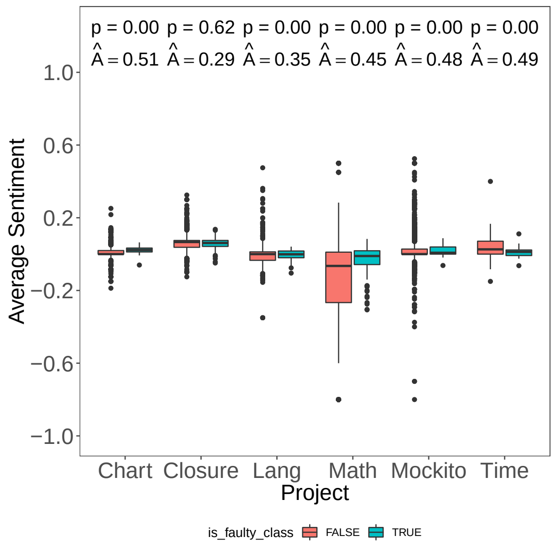


FIGURE 5.4: Sentiment scores of commits from files that contain faults and files that do not

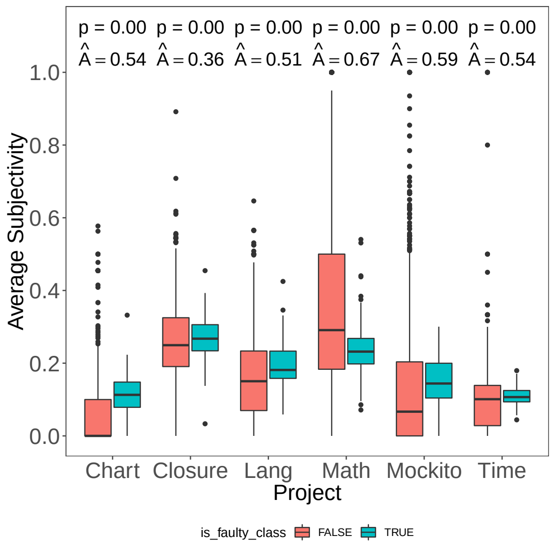


FIGURE 5.5: Subjectivity scores of commits from files that contain faults and files that do not

a p value of 0.51.

Figure 5.3 shows that, for four out of the six projects, the difference in sentiment scores for bug-fix commits and non bug-fix commits is significant. Furthermore, for the Math project, there is a high $\hat{A}$ score (0.72). This indicates that subjectivity, rather than sentiment specifically, is a better indicator of a commit containing a bug-fix. In both cases, the word `fixed` is the most discriminatory, appearing in 994 out of 16,579 non bug-fixing commits (5.99%), and in 67 out of 430 bug-fixing commits (15.58%). The word `fixed` adds a subjectivity score of 0.2 and a sentiment score of 0.1, resulting in higher subjectivity scores for bug fix commits that contain this word more frequently. Following this, the next most discriminatory word is `new`, which adds a subjectivity score of $0.\overline{45}$ for every occurrence. `new` appears in 25 out of 430 bug-fix commits (5.81%) and 587 out of 16,579 non bug-fix commits (3.54%). This also adds to the difference in subjectivity scores between bug-fix and non bug-fix commits.

RQ2: In a number of cases, the subjectivity and sentiment scores are higher for bug-fix commits than non bug-fix commits. Subjectivity score is a better determiner of whether a commit contains a bug than sentiment score, although neither are completely reliable and the results vary per project.

File	Positive	Negative	Neutral	Faulty
com.google.javascript.jscomp.TypeCheckTest	109	47	139	FALSE
org.mockito.Mockito	109	26	195	TRUE
com.google.javascript.jscomp.DefaultPassConfig	79	24	115	FALSE
com.google.javascript.jscomp.Compiler	75	22	91	TRUE
com.google.javascript.jscomp.NodeUtil	67	20	79	TRUE
org.mockito.exceptions.Reporter	63	12	104	TRUE
com.google.javascript.jscomp.TypedScopeCreator	59	16	57	TRUE
common_deplo	57	25	75	FALSE
org.apache.commons.lang.StringUtils	51	17	140	TRUE
com.google.javascript.jscomp.CompilerOptions	48	11	97	FALSE

TABLE 5.4: Number of Positive, Negative and Neutral changes per file

5.4.3 RQ3: Can sentiment or subjectivity in commit messages be used to predict whether a file contains a bug?

This research question leverages the information provided by DEFECTS4J about the files that have been included in fault-fixes. Specifically, for each of the bugs in the dataset, I identify the set of files that was changed in order to fix that fault. With this information, I separate the commits messages into those that contain a faulty file, and those that do not.

Figures 5.4 and 5.5 show the average sentiment and subjectivity scores respectively of files that are known to contain faults and those that are not. Figure 5.4 shows that in most cases, the average sentiment for all files is close to 0 regardless of the project. with high sentiment scores in both directions being rare. One reason for this is that sentiment scores can be both positive and negative, and therefore over the lifecycle of a file it will go through positive and negative changes. This is further indicated in Table 5.4, which shows the top 10 files containing positive changes, and indicates whether those files are known to be faulty or not. Table 5.4 shows that all files go through a number of positive, negative and neutral changes during their lifecycle, with positive changes usually slightly outweighing negative ones. While there is a high number of files in this table that are faulty, this is due in part to the fact that these files are also the most frequently changed within their own projects, making it more likely that at some point they will have contained a bug. While the results of the Mann-Whitney U-Test in Figure 5.4 show that there is a difference in sentiment scores for faulty classes vs non-faulty ones, the $\hat{A}$ scores indicate very little practical significance to these differences.

Furthermore, as shown by Figure 5.5, the difference in subjectivity score for faulty classes vs non-faulty classes is in most cases negligible, despite significant differences for every project. This indicates that the sentiment and subjectivity scores associated with a file are not a strong indicator of whether the file contains a fault.

RQ3: Sentiment and subjectivity scores are generally higher for files that are known to have contained real faults than those that have not. While these differences are statistically significant, the practical significance is negligible in most cases. Sentiment or subjectivity scores cannot be used to accurately predict whether a file is likely to contain a fault

5.5 Discussion

One of the notable results in Section 5.4 was that bug-fix commits are not easily identifiable from non-bug-fix commits. One of the reasons for this may be a shortcoming in the TextBlob lexicon. In total, the TextBlob lexicon contains 1528 words, of which 528 are classified as positive ($p > 0$), 620 are classified as negative ($p < 0$), and the remaining 380 are neutral ($p = 0$). This section looks at how a the language used in bug-fixing commits could be leveraged to produce a prediction model for whether a commit fixes a bug or not, and whether the same is true of faulty files.

5.5.1 Identifying bug-fix commits

In total, the experimental data collected for Section 5.4 contains 430 bug-fix commits, containing a total of 10,415 words (1,977 unique). Using a python script, I identified the most frequently used words that appear in these commits. Ignoring commons words like "the" and "by", the most common word that appears in bug-fix commits is "issue", which appears a total of 201 times. Following this, the word "fixes" appears 180 times, while other varieties "fix" and "fixed" and "fixing" appear 97, 73 and 11 times respectively. Unsurprisingly, "added" (169 occurrences), "changed" (161 occurences) and "deleted" (158 occurrences) are common words. The word "jira" appears in 59 bug fixes. This is likely due to the fact that when the JIRA bug-tracking system is used and an issue is created through a JIRA webpage, referencing the issue identifier within a commit causes the commit to appear on the bug reports webpage. Finally, the word "bug" occurs 30 times.

However, just because a word is used frequently does not identify that a commit fixes a bug. In addition to looking at bug-fix commits, I also analyse non-bug-fix commits to identify words that appear frequently. In total, there are 16,579 non-bug-fix commit messages in the dataset, comprising 223,829 words (12,614 unique). As with bug-fix commits, I ignore common words. In non-bug-fix commits, the most frequently used word is "added" with 3,982 occurrences, followed by "created" (2,694 occurrences), "java" (2,588 occurrences) and "changed" (2,389 occurrences). This shows that these words cannot be used to discriminate commit messages as either bug-fix or non-bug-fix, as they occur frequently in both types of commit.

The word “issue” appears 193 times in bug-fixing commits (44.88% of all words) compared to 546 times in non-bug-fix commits (3.29% of all words) — the fact that the word “issue” accounts for 44.88% of all words in bug-fixing commits is a strong indication that the word “issue” can be used to discriminate bug-fix commits against non-bug-fix ones. Despite the dataset containing far fewer examples of bug-fixing commits, the word “fixes” represents a higher proportion of words in bug-fixing commits (6.5% of all words) compared to non-bug-fix commits (1.36%), again indicating that the word “fixes” can be used to discriminate whether a commit fixes a bug or not. The presence of discriminatory words indicates that a supervised sentiment analysis approach could result in an effective model for classifying bug-fix commits. Interestingly, while “fixes” appears as one of the keywords adopted by Sliwerski et al. [89] in the SZZ algorithm, the word “issue” is not.

5.5.2 Identifying Faulty Files

In Section 5.4, one of the key aims is to identify whether “at-risk” files can be identified by commit messages. Specifically, if files that at risk of containing faults can be identified, then a test case prioritisation strategy can be built similar to G-Clef described in Chapter 4 that identifies test cases that cover the class.

In this study, I use a total of 298 faulty classes from Defects4J containing a total of 438 real faults. A total of 203 classes only contain one fault through the repository history. In Chapter 4, the parameter tuning study revealed that the best configuration of the Schwa tool gave a low weight to previous faults — this is reflected in this result, since most of these files only contain one fault and then remain fault free.

Table 5.5 shows the number of commits that are made to files, separated by whether the file contains a fault or not. This also reflects the findings from Chapter 4, which indicated that the number of revisions is important when predicting whether a class is faulty.

When comparing bug-fix and non-bug-fix commits, there were clear examples of discriminatory words that could be used to determine whether the commit was bug-fixing or not (“issue” makes up 44.88% of bug-fixing commits compared to 3.29% of non-bug-fixing ones). Commits to faulty files contain a total of 103,622 words (7,777 unique), while commits to non-faulty files contain a total of 177,647 words (11,092 unique). By conducting a similar analysis of commit messages to files that are faulty and files that are not, I

Table 5.5: Table showing the average number of commits to files that contain faults vs files that do not

Project	Non-Faulty File Commits	Faulty File Commits
Chart	4.23	12.04
Closure	7.13	45.52
Lang	11.26	39.00
Math	6.48	24.16
Mockito	5.03	45.18
Time	8.01	32.05

identify that discriminatory words are less common for identifying faults in files. The word "delta" appears 1,349 times in commits to faulty files (1.3%), and 1590 times in commits to non-faulty files (0.9%). Additionally, the word "changed" appears 1487 times in commits to faulty files (1.44%) and 1858 times in commits to non-faulty files (1.05%), a difference of 0.39%. This indicates that creating a linguistic model to predict faults in files would be very difficult and unlikely to have much success.

5.5.3 Test Case Prioritisation based on Sentiment Analysis

One of the key motivations for studying commit messages was that, if commit messages can be used as a proxy for defect prediction, it would be possible to create a test case prioritisation strategy that would rank files based on the likelihood of them being faulty, using the commit messages associated with each file to provide these estimates.

The possibility of having a test case prioritisation strategy that integrates effectively with a VCS is a very useful one, since it would at least partially address one of the barriers preventing test case prioritisation being used in industry.

However, given the results presented in Section 5.4 and the discussion presented in Section 5.5.2, it does not appear that a prioritisation strategy based on sentiment analysis would be very accurate, making it unlikely to be successful.

5.6 Conclusions

This chapter conducts an in-depth analysis of commit messages to version control systems, focused on positive or negative emotions associated with either bug fixes or faulty files, with the intention of developing a test case prioritisation strategy that uses sentiment analysis as a proxy for defect prediction.

When considering bug-fixing commits, these experiments indicate that there are significant differences in both sentiment and subjectivity scores between commits that fix faults and commits that do not. However, these differences do not result in high $\hat{A}$ scores, indicating that the practical significance is low. Further investigation into bug-fixing commits identifies key words that are particularly discriminatory for identifying bug-fixing commits (e.g. "issue", "fixes"), indicating that a linguistic model of commit messages could be successful at detecting bug-fixing commits.

When looking at files that have contained faults, there are significant differences in sentiment score for four out of five projects, and significant differences in subjectivity score for all five projects. However, since most of the $\hat{A}$ scores are close to 0.5, it is unlikely that faults can be accurately predicted from the commit messages associated with a file. Furthermore, while there are a number of discriminatory words for identifying bug-fixing commits, the same cannot be said of faulty files — words that appear frequently in commits to faulty files also appear frequently in commits to non-faulty files. This makes it unlikely that a linguistic model for predicting faults using commit messages would be successful, and in turn makes it unlikely that a test case prioritisation strategy based on sentiment analysis would be effective.

Chapter 6

Conclusions and Future Work

This section rounds out the thesis by recalling the aims and objectives laid out in the introduction, stating how these objectives have been met through the work presented in this thesis, and identifying avenues for future work.

6.1 Summary of Contributions Made by this Thesis

In Chapter 1 of this thesis, I outlined two main aims that I intended to achieve throughout the course of my studies.

1. To empirically evaluate the comparative effectiveness of test case prioritisation strategies on real and artificial faults.
2. To develop new strategies for test case prioritisation that will be more effective at prioritising test suites for real faults.

The following sections outline how these aims have been met through the research conducted throughout the duration of this thesis.

6.1.1 Empirically evaluating the effectiveness of test case prioritisation strategies on real and artificial faults

In Chapter 3, I conducted an investigation into the comparative effectiveness of existing test case prioritisation strategies on controlled numbers of real and artificial faults. Specifically, this chapter questions an assumption held in previous studies on test case prioritisation that if a new strategy results in an increase in *APFD* for artificial faults, that the same increase would be observed with real faults. In the evaluation, I used eight existing test case prioritisation strategies categorised as either coverage-based or history-based, and up to 262 real faults from DEFECTS4J. This led to the first contribution of my thesis:

Contribution 3.1: *A comparison of how coverage-based test case prioritisation strategies perform on real faults and mutants*

One of the most noticeable trends from this experiment is that the performance of test case prioritisation strategies is inconsistent when different fault types are used. That is to say, when using mutants, strategy A may outperform strategy B, while using real faults would indicate the reverse. Additionally, when using real faults, there were no significantly positive results for any coverage-based strategy. This result forms the second contribution of my thesis:

Contribution 3.2: *A comparison of how history-based test case prioritisation strategies perform on real faults and mutants*

When using history-based strategies in place of coverage-based ones, the results remain inconsistent between different fault types, further strengthening the idea that it is critical to use real faults when evaluating test case prioritisation strategies. However, there are some more positive results that indicate that software history can be a useful source of information for predicting future failures.

Finally, in Chapter 3, I conducted an evaluation investigating the effect that the number of faults in a program can have on the effectiveness of test case prioritisation. In previous research, the number of faults in a program has been assumed to be an independent variable, having no impact on the effectiveness of test case prioritisation strategies.

The results of the empirical evaluation indicate that the number of faults present in a program can significantly affect the spread of *APFD* scores. With fewer faults in a program, the *APFD* scores are more varied as some subjects achieve good *APFD* scores and others achieve poor ones. As the number of faults in a program increases, it becomes more likely that a single program will have some faults detected early and others detected late within the test suite, meaning that the extreme *APFD* scores are lost in favour of more average scores. Therefore, the third contribution of my thesis is listed below.

Contribution 3.3: *A comparison of how the number of faults present in a program affects the performance of test case prioritisation strategies*

Overall, Chapter 3 provided valuable insights into test case prioritisation evaluations, making it clear that future evaluations should use real faults to ensure reliable results.

Furthermore, Chapter 3 shows that state-of-the-art test case prioritisation strategies struggle when prioritising test suites for real faults, creating a need for improved strategies.

6.1.2 Developing new strategies that increase the effectiveness of test case prioritisation on real faults

Having established in Chapter 3 that test case prioritisation strategies struggle to prioritise test suites effectively for real faults, it was clear that a new test case prioritisation strategy must be developed. Chapter 4 proposes a new test case prioritisation strategy based on defect prediction, a technique that has been shown to be effective for real faults. Using the defect prediction tool Schwa, Chapter 4 begins by contributing a parameter tuning study to show how effective Schwa can be at detecting the real faults in DEFECTS4J.

Contribution 4.1: *A parameter tuning study to determine the best parameters for defect prediction to find real faults in* DEFECTS4J

Since the parameter tuning study revealed that Schwa can effectively predict the likelihood of classes being buggy, I developed G-Clef and implemented it into KANONIZO (Appendix A).

Contribution 4.2: *An implementation of a new test case prioritisation strategy, G-Clef, that leverages defect prediction*

G-Clef also has runtime parameters, which can take multiple values. Therefore, in order to determine the best configuration, this thesis contributes a parameter tuning study on 30 DEFECTS4J subjects.

Contribution 4.3: *A parameter tuning study to determine the best parameters for G-Clef*

Finally, I evaluated G-Clef on 395 real faults from DEFECTS4J and compared it with the eight existing test case prioritisation strategies used in Chapter 3 to determine whether G-Clef had made improvements over these strategies, resulting in the following contributions.

Contribution 4.4: *An evaluation of G-Clef against existing coverage-based strategies*

Contribution 4.5: *An evaluation of G-Clef against existing history-based strategies*

In the evaluation, I discover that G-Clef significantly outperforms six out of the eight strategies it was compared against, and was not significantly beaten by any existing strategy. However, while the results were positive, they also indicated a much greater potential for defect prediction that could yield further benefits for test case prioritisation if more accurate predictions can be made.

Therefore, in Chapter 5, I attempted to develop a defect prediction strategy that leverages sentiment analysis scores from commit messages. This is motivated by the idea that developers may write more opinionated commit messages when working with files that are badly written, and therefore have a higher chance of being buggy. If there is a correlation between sentiment of a commit message and files being faulty, this can be leveraged by test case prioritisation to create a strategy based around analysing commit messages, and specifically identifying files that are likely to be faulty through how developers write about them. Firstly, I analyse the Defects4J subject programs to see if there is sentiment or subjectivity in the commit messages, leading to the following contribution.

Contribution 5.1: *An evaluation of sentiment and subjectivity in* Defects4J *commit messages*

Despite there being a high number of commit messages that do not have any sentiment associated with them, this is due in part to using a lexicon-based sentiment analysis tool, which does not recognise "fix" as being a positive word in programming. Following this, I looked specifically at commits that *fix bugs* in Defects4J. This led to the following contribution.

Contribution 5.2: *An evaluation of sentiment and subjectivity in bug-fix commits against non-bug-fix commits*

This evaluation returned some interesting and surprising results. Firstly, commit messages contain much more subjectivity than sentiment, due partly to the high subjectivity/low sentiment score of words like "fixed", "final" and "new". Secondly, for four out of six Defects4J projects, the subjectivity score is significantly higher for bug-fix commits than non-bug-fix commits according to the Mann-Whitney U-Test, with $\hat{A}$ scores varying from 0.61-0.72. Finally, I considered files in Defects4J that have previously contained faults.

Contribution 5.3: *An evaluation of sentiment and subjectivity in faulty files against non faulty files*

Due to the potential implications in test case prioritisation, this contribution was important in determining whether a new test case prioritisation strategy should be developed that uses sentiment or subjectivity scores to determine the likelihood of a file being buggy. While the results do indicate that subjectivity scores are significantly different for all

DEFECTS4J projects, the $\hat{A}$ scores are lower than those reported for bug-fix vs non-bug-fix commits (0.46-0.67). While this indicates that a test case prioritisation strategy based on sentiment analysis may not be effective, there are a number of potential improvements that could be made to the tools that could, in the future, lead to a test case prioritisation strategy based on commit messages.

6.2 Future Work

There are a number of avenues for future work arising from the topics investigated throughout this thesis.

6.2.1 Improvements to test case prioritisation strategies

Despite the advances made in Chapter 4, there are still a number of potential improvements to test case prioritisation that could be effective for real faults.

Test case prioritisation for evolving software

As shown by Lu et al. [22], as software evolves, prioritised test suite orderings become much less effective in only a few software evolutions. Future work could investigate if any more recently developed strategies, such as G-Clef presented in Chapter 4 of this thesis, are more effective in evolving software, and if not, should focus on developing new strategies that produce orderings that are resistant to software evolution.

Test case prioritisation in Continuous Integration

One of the key barriers that prevents test case prioritisation becoming an industry practice is the lack of integration with industry-standard build tools such as Maven[1] or Gradle[2]. Liang et al. [189] conducted a study in which they prioritised builds in a continuous integration system by considering the likelihood that individual builds would fail. In most cases however, it is unlikely that a CI or build server will ever have a queue of enough builds to make this technique relevant. Future research should involve integrating test case prioritisation strategies with build and/or continuous integration systems to see whether the benefit of test case prioritisation outweighs the cost, and for how long prioritised test suites can remain useful before developers need to re-run prioritisation.

[1]https://maven.apache.org
[2]https://gradle.org

Hyper Heuristics

The majority of the work involved in metaheuristic approaches to test case prioritisation has revolved around finding a fitness function that accurately represents fault-detection capability. Li et al. [12] began using a simple APxC approach, while more recent approaches including Di Nucci et al. [81], Konak et al. [73] and Wang et al. [74] have adopted multiple objective approaches using more complex objectives than simple coverage. Throughout this thesis, the performance of the genetic algorithm in KANONIZO that uses APLC as a fitness function has been poor. The main reason for the poor performance, as discussed in Section 3.4, is that the fitness function used is both expensive to calculate and guides the search towards sub-optimal orderings. This raises the question "How do we find the *right* fitness function to use for test case prioritisation?". There may be no simple answer to this question, in fact there may be no single fitness function that is objectively the best fitness function for test case prioritisation, meaning that using any single heuristic or combination of heuristics can leave us with a sub-optimal set of results for any given problem. More generally, including other research fields within Computer Science, fitness functions tend to be designed to guide searches related to the problem they are created for. For example, a fitness function that is designed to optimise a test suite for test case prioritisation would be useless in the field of test suite generation, which is described in Wolpert and MacReady's "No Free Lunch Theorem" [190].

In 2003, Burke et al. [144] introduced the concept of *hyper-heuristics*. Hyper-heuristics can be thought of simply as applying a search for a fitness function by which to guide a search. This adds a layer of generality to any potential problem solver, since the fitness functions are no longer a part of the implementation design, instead being searched for, meaning different fitness functions will be discovered given the context of the problem and the search. Harman et al. [191] further discuss this concept in their investigation into a technique called "Dynamic Adaptive Search-Based Software Engineering". In their work, Harman et al. [191] describe the need for a *more holistic SBSE*, indicating that there are too many searches designed at solving just one problem when the focus should be on creating more general solutions.

Dynamic Adaptive SBSE

Modern software engineering has led many programs that are being developed by programmers to be highly flexible, customisable and extensible. This means, in simple terms, that all pieces of software come with lots of options that will determine how the program behaves at runtime. For example, in a genetic algorithm implementation, it makes sense to allow the mutation rate and crossover rate to be configurable, rather than hard coded, since different values for these parameters can have a large impact on the overall performance of the system.

Parameter tuning is a technique designed to choose the "best" values of runtime arguments in systems such as genetic algorithms in order to maximise performance. Since more complex systems have large numbers of "tuneable" parameters, it is infeasible to try running each possible combination of parameters. An example of this is given in Arcuri and Fraser[192], where running possible combinations of five parameters on 20 programs led to over *one million* experiments, requiring over two weeks of computational time on a high-performance computing cluster to complete. This can be seen as one simple explanation of the goals of Dynamic Adaptive SBSE [191], deploying a program that itself has some ability to control its own parameters based on the output it is observing, allowing a program to become "self-adaptive". The vision for Dynamic Adaptive SBSE stretches further than this, to programs that can adapt themselves to changing environments, taking multiple factors into account and deciding what changes should made in order to provide the best performance.

6.2.2 Improvements to Mutant Faults

In Chapter 3, one of the main discoveries is that test case prioritisation strategies are more effective for mutants than for real faults. When investigating this phenomenon, I discovered that one of the main reasons why test case prioritisation is more effective for mutants is that mutation operators generate unrealistic faults. Just et al. [83] conducted a study in which they investigated existing mutation operators to determine whether the real faults in Defects4J could be generated using any existing mutation operators. In their paper, Just et al. discover that a number of real faults could not be generated using existing mutation operators, and suggested improvements to mutation operators for example recognising language-specific methods that are semantically similar (e.g. `indexOf` and `lastIndexOf`). Furthermore, Luo et al. [19] investigate the impact of different mutation operators on *APFD* scores, showing that some mutation operators produce more realistic faults than others. If better mutation operators are developed, faults generated by mutation tools could become much more realistic, which could mean that evaluations of test case prioritisation strategies could use mutants in place of real faults without it affecting the results.

6.2.3 Improvements to Defect Prediction

In accordance with Chapters 4 and 5, future work should continue in the areas of defect prediction to continue to make improvements to test case prioritisation strategies. G-Clef, described in Chapter 4 is currently configured to use the Schwa defect prediction tool, but could easily be swapped out for better tools as new defect predictors are developed.

Improvements to Sentiment Analysis for Test Case Prioritisation

There are a number of avenues for future work arising from Chapter 5. Firstly, while my approach used a lexicon-based sentiment analysis strategy, this could be improved by creating a lexicon for sentiment analysis that incorporates programming terms with the correct context. For example, a `fix` should be considered a positive emotion, whereas the keyword `final` should be considered less subjective in this context. Additionally, other approaches to sentiment analysis, named supervised (e.g. Chen et al. [131]) or unsupervised methods (e.g. [136]). Secondly, there is clearly potential to use classifiers and machine learning techniques to learn from the corpus of commit messages in Defects4J, and to create a model to predict defective classes based on the text in the commit message.

6.3 Concluding Remarks

This thesis has aimed to improve the effectiveness of test case prioritisation strategies in practice. Firstly, through an empirical evaluation using real faults and mutants, this thesis demonstrates that using artificial faults can represent a threat to validity in evaluations of test case prioritisation strategies, since it may lead to a different conclusion than if real faults were used.

Secondly, this thesis proposes a new test case prioritisation strategy, G-Clef, that improves upon the state of the art by using defect prediction estimates to reorder test suites.

Finally, this thesis investigates whether the defect prediction used in G-Clef could be improved through the use of sentiment analysis, in particular investigating whether commit messages to version control could be used to predict whether a file contains a fault or not.

There are still a number of improvements that can be made to test case prioritisation to make the technique more effective. In particular, one of the areas of future work identified by this thesis is to find techniques that increase the longevity of prioritised suites. However, this thesis has advanced the state of the art in test case prioritisation by establishing important experimental guidelines and through the proposal and implementation of G-Clef.

Appendix A

Kanonizo

Over the course of my PhD, I have been developing the test case prioritisation tool Kanonizo, which prioritises test suites using a number of existing and newly developed strategies. The tool is open-source (`https://github.com/kanonizo/kanonizo`), written in Java (version 8) and requires 3 command like arguments in order to be run. `-s`, or `--source` represents the directory in the file system that contains all of the source code of the program under test. This is used, amongst other things, to calculate the lines of code covered when executing the test cases. `-t`, or `--tests` points to the folder containing all of the test cases that are to be executed. Test cases can be recognised in a number of ways — firstly, any test cases that are annotated with the `@Test` annotation are extracted using the annotation (excluding those with the `@Ignore` annotation). Parameterised test cases are recognised thanks to their `@RunWith(Parameterized.class)` annotation. Secondly, test cases written in JUnit 3 style, which must be public methods with no arguments, and must start with the word `test` are included. Finally, the JUnit 3 "suite" pattern, where there is a class containing all test suites, which themselves can be test suites, are recognised. This is to prevent any order violations when running the test cases that occur in certain Defects4J subjects when running tests alphabetically. Finally, the `-a` or `--algorithm` argument represents the strategy that will be used to reorder the test suite.

A.1 Kanonizo Process

Kanonizo begins by scanning the command line arguments, ensuring that each of the arguments is valid (i.e. source directory exists, strategy exists). After this, it identifies any required libraries, specified either through the `-l` command line option or by using

Maven[1], and adds these libraries to the classpath. Following this, it moves through the source directory recursively, identifying all files with the `.class` extension and loading them through an `Instrumenter`, so that coverage can be calculated from the test cases. Next, it identifies the test cases as described above, before delegating to the `Instrumenter` to collect the code coverage

A.1.1 Instrumentation and Coverage Collection

Many of the strategies implemented in KANONIZO come from previous literature (e.g. [10]), and therefore use code coverage to reorder the test suite, motivated by the idea that higher coverage implies higher fault detection. In order to collect code coverage, KANONIZO uses an `Instrumenter` at runtime to execute the test cases. The `Instrumenter` interface contains 18 methods that each relate to code coverage of specific test cases or classes under test. The default implementation of the `Instrumenter` interface is the `ScytheInstrumenter` class, which uses the Scythe[2] code coverage tool to collect line or branch coverage. Many of the strategies in KANONIZO require the coverage of individual test cases, and the `ScytheInstrumenter` implementation resets code coverage after each test case is executed. Assuming the methods described in the `Instrumenter` interface are overriden, any instrumenter can be used with KANONIZO to provide code coverage information to the strategies. If coverage is not required in order to make a strategy work, the `NullInstrumenter` class can be used, which disables the running of test cases at runtime to speed up the process.

A.1.2 Prioritising Test Cases

Once the required coverage information has been collected, the next step is to start the prioritisation strategy. There are two main categories of strategy in KANONIZO—firstly, it has `TestCasePrioritiser` strategies, which use information about individual test cases to *rank* test cases. These strategies typically pick a single test case at a time, and use the selected test cases to build up a suite one test at a time. Any class that implements the `TestCasePrioritiser` interface must implement the `selectTestCase` method, which takes a `List` of test cases as an argument and returns a single test case, and may optionally implement the `init` method, which takes a `List` of test cases as an argument and can be used to set an initial ordering or execute any steps that must come before prioritisation occurs.

[1]https://maven.apache.org
[2]https://github.com/thomasdeanwhite/scythe

The second category of strategies are `TestSuitePrioritiser`, which look holistically at the entire suite, and involve an `evolution` step to get from one test suite to the next. These strategies are defined by their fitness functions, which evaluate how "good" the ordering of the test suite is, and help to guide the prioritiser to a better solution by discarding orderings that are less "fit". In addition, these strategies typically need one or more stopping conditions, which determine when the algorithm has reached a good enough solution to report its results, either when a certain fitness value has been reached, or after a certain length of time, number of iterations etc. Any class that implements the `TestSuitePrioritiser` interface must implement the `generateInitialPopulation` method, which returns a `List` containing one or more orderings for the test suite, and the `evolve` method, which also returns a `List` or test suite orderings. They may also optionally add `EvolutionListener` objects, which notify subclasses when an evolution has finished.

By default, KANONIZO has implementations for seven `TestCasePrioritiser` strategies (Greedy [10], Additional Greedy [10], G-Clef [2], MCCTCP [52], ROCKET [54], AFSAC [56] and Elbaum et al. [55]) and two `TestSuitePrioritiser` strategies (Genetic Algorithm [12] and Random Search), but allows other implementations to be added.

A.1.3 Reporting

One of the important tasks in KANONIZO is reporting the results of the prioritisation. KANONIZO produces a number of output files that represent the ordering of the prioritised test suite, along with statistics about test runtimes, test outcomes, and other miscellaneous statistics. All output is written in CSV, and new writers can be added using the `CsvWriter` class and the `Framework::addWriter` method.

A.1.4 Configuration

One of the key facets about a number of strategies in KANONIZO is that they are require a number of parameters, which should be configurable from the command line (i.e. not hard-coded values). For example, the genetic algorithm has a population size, a time limit and an iteration limit. Importantly, KANONIZO does not assume these to be default values. Using the command line version, there are 74 configurable parameters by default that can all be changed using the command line. For example, to increase the time limit for the genetic algorithm, users can add `-Dmax_execution_time=100000` to allow a search budget of 100000ms. Using the GUI, selecting a strategy causes all its parameters to be displayed on the GUI to allow configuration before running.

A.1.4.1 Testing

In order to ensure that the implementation of the methods was faithful to the original papers, I developed a suite of unit tests alongside Kanonizo in order to simulate a number of scenarios.

For the Elbaum et al. [55] approach, any test that has either failed recently or only been added recently is given a higher priority. To test this, I created three mocked test cases using the Mockito[3] framework for Java. For each of these test cases I created a history where either the test had failed recently or only been added recently, asserting that after running the algorithm, this test case is at the beginning of the test suite.

ROCKET [54] considers how recently a test case failed as part of its priority. In order to test this I first constructed a scenario in which one test has failed recently, asserting that it is returned first by the algorithm. Additionally, I added a test case that demonstrates that a test that failed more recently is returned before a test that failed longer ago in its history. Finally, I considered a scenario in which two test cases failed at the same point in their history, and show that the original order is retained under these conditions.

For MCCTCP [52], the severity of the failure determines the priority of the test case. To test this, I constructed a scenario in which two test cases failed in the same execution, one with a `NullPointerException` and the other with an `AssertionFailedError`, showing that the test with the `NullPointerException` is given higher priority.

Finally, the AFSAC [56] algorithm has four main calculated values - fr_{min}, fr_{avg}, fr_{max} and the current number of consecutive failures. The priority of each test case depends on how the current number of consecutive failures compares with the three other calculated values. I wrote test cases to show what happens when the current number of consecutive failures falls into each of these categories.

A.1.5 Future Work

Kanonizo has a number of strategies implemented already, including history-based, coverage-based and metaheuristic. These can all be extended, overwritten, or configured to fine-tune the performance of each strategy given the subject. There are a number of other strategies that could be added to Kanonizo that would also improve performance. Furthermore, a plug-in for IDEs such as Eclipse[4] or IntelliJ[5] could allow for users to actually run the test cases in the order provided by Kanonizo. Furthermore, integration

[3] https://site.mockito.org/
[4] https://eclipse.org
[5] https://www.jetbrains.com/idea/

with build tools such as Maven[6] or Gradle[7] could allow persistent orderings of test cases that would be retained between runs and could allow prioritisation to become a part of the build process.

[6]https://maven.apache.org/
[7]https://gradle.org/

Appendix B

Subject Information

In Chapter 3, I conduct experiments in which I use a variety of coverage-based and history-based test case prioritisation strategies to prioritise test suites of programs containing either a single real fault or a single mutant. In Tables B.1-B.6, I summarise the faults from DEFECTS4J that were included and excluded from these experiments.

For Section 3.3.1, I consider coverage-based strategies evaluated on real faults and mutants. In order for a subject to be included in these evaluations, there must be a data point for *both* a real fault **and** a mutant for that subject. Therefore, Chart-10 and Chart-15 were excluded from this evaluation as we did not have a version of these subjects containing a single mutant. This means that, while Table B.1 shows 26 subjects for the Chart project, Figure 3.2 only shows 24 subjects. For some of the subjects, it was not possible to generate any mutants at all (e.g. Chart-10), while for others there were no mutants that were revealed by any of the developer-written test cases (e.g. Chart-15). In Tables B.1-B.6, these subjects are denoted as ✗ in the "1 mutant (coverage)" column. For some of the Mockito subjects, it was not possible to collect any coverage information due to the way Mockito manipulates the Java `ClassLoader` — these cases are marked with "NA" test cases and were not included in any experiments.

For Section 3.3.2, I consider history-based strategies which utilise the previous results of test case executions in order to determine how likely they are to fail again. For these experiments, I required that the fault-revealing test case had a least one previous execution prior to the current version of the software — this ensures that the history-based strategies were not given an unfair advantage due to a test case having no historical information to leverage. In Tables B.1-B.6, the columns "1 real (history)" and "1 mutant (history)" denote subjects that did meet (✓) and did not meet (✗) this requirement respectively. As with coverage-based strategies, there are some cases that did meet this requirement, but for which no mutants could be generated (e.g. Chart-10).

Table B.1: Chart Subjects

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Chart	1	2201	✓	✓	✗	✗
Chart	2	2199	✓	✓	✗	✗
Chart	3	2195	✓	✓	✗	✗
Chart	4	2187	✓	✓	✓	✓
Chart	5	2040	✓	✓	✗	✗
Chart	6	1894	✓	✓	✗	✗
Chart	7	1820	✓	✓	✓	✓
Chart	8	1820	✓	✓	✓	✓
Chart	9	1820	✓	✓	✓	✓
Chart	10	1812	✓	✗	✓	✗
Chart	11	1810	✓	✓	✗	✗
Chart	12	1806	✓	✓	✗	✗
Chart	13	1798	✓	✓	✗	✗
Chart	14	1794	✓	✓	✗	✗
Chart	15	1789	✓	✗	✗	✗
Chart	16	1787	✓	✓	✗	✗
Chart	17	1746	✓	✓	✗	✗
Chart	18	1744	✓	✓	✗	✗
Chart	19	1723	✓	✓	✗	✗
Chart	20	1649	✓	✓	✗	✗
Chart	21	1645	✓	✓	✗	✗
Chart	22	1644	✓	✓	✗	✗
Chart	23	1623	✓	✓	✓	✓
Chart	24	1620	✓	✓	✗	✗
Chart	25	1615	✓	✓	✗	✗
Chart	26	1589	✓	✓	✓	✓
Totals:			**26**	**24**	**7**	**6**

Table B.2: Lang Subjects

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Lang	1	2291	✓	✓	✗	✗
Lang	2	2287	✓	✓	✗	✗
Lang	3	2286	✓	✓	✗	✗
Lang	4	2285	✓	✓	✗	✗
Lang	5	2271	✓	✓	✗	✗
Lang	6	2263	✓	✓	✗	✗
Lang	7	2261	✓	✓	✓	✓
Lang	8	2205	✓	✓	✗	✗
Lang	9	2200	✓	✓	✗	✗
Lang	10	2198	✓	✓	✗	✗
Lang	11	2138	✓	✓	✗	✗
Lang	12	2137	✓	✓	✗	✗
Lang	13	2135	✓	✓	✗	✗
Lang	14	2073	✓	✓	✓	✓
Lang	15	2047	✓	✓	✓	✓
Lang	16	2046	✓	✓	✓	✓
Lang	17	1903	✓	✓	✗	✗
Lang	18	1902	✓	✓	✓	✓
Lang	19	1877	✓	✓	✗	✗
Lang	20	1876	✓	✓	✓	✓
Lang	21	1827	✓	✓	✓	✓
Lang	22	1825	✓	✓	✓	✓
Lang	23	1825	✓	✓	✗	✗
Lang	24	1822	✓	✓	✓	✓
Lang	25	1821	✓	✓	✗	✗
Lang	26	1790	✓	✓	✗	✗
Lang	27	1785	✓	✓	✓	✓
Lang	28	1763	✓	✓	✗	✗
Lang	29	1760	✓	✓	✓	✓
Lang	30	1733	✓	✓	✗	✗
Lang	31	1721	✓	✓	✗	✗
Lang	32	1670	✓	✓	✓	✓
Lang	33	1670	✓	✓	✓	✓
Lang	34	1670	✓	✓	✓	✓
Lang	35	1644	✓	✓	✗	✗
Lang	36	1628	✓	✓	✓	✓
Lang	37	1627	✓	✓	✗	✗
Lang	38	1624	✓	✓	✗	✗
Lang	39	1618	✓	✓	✓	✓
Lang	40	1643	✓	✓	✗	✗
Lang	41	1624	✓	✓	✓	✓
Lang	42	1872	✓	✓	✗	✗
Lang	43	1871	✓	✓	✗	✗
Lang	44	1848	✓	✓	✗	✗
Lang	45	1846	✓	✓	✓	✓
Lang	46	1798	✓	✓	✗	✗
Lang	47	2658	✓	✓	✗	✗
Lang	48	2593	✓	✓	✗	✗
Lang	49	2580	✓	✓	✓	✓
Lang	50	1785	✓	✓	✗	✗
Lang	51	1696	✓	✓	✓	✓
Lang	52	1696	✓	✓	✓	✓
Lang	53	1689	✓	✓	✗	✗
Lang	54	1681	✓	✓	✗	✗
Lang	55	1681	✓	✓	✗	✗
Lang	56	1662	✓	✓	✗	✗
Lang	57	1661	✓	✓	✓	✓
Lang	58	1660	✓	✓	✗	✗
Lang	59	1658	✓	✓	✗	✗
Lang	60	1655	✓	✓	✗	✗
Lang	61	1654	✓	✓	✗	✗
Lang	62	1652	✓	✓	✗	✗
Lang	63	1642	✓	✓	✗	✗
Lang	64	1637	✓	✓	✗	✗
Lang	65	1604	✓	✓	✗	✗
Totals:			**65**	**65**	**22**	**22**

Table B.3: Math Subjects

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Math	1	5064	✓	✓	✓	✓
Math	2	5037	✓	✓	✗	✗
Math	3	4901	✓	✓	✗	✗
Math	4	4885	✓	✓	✗	✗
Math	5	4820	✓	✓	✓	✓
Math	6	4813	✓	✓	✗	✗
Math	7	4804	✓	✓	✗	✗
Math	8	4721	✓	✓	✗	✗
Math	9	4697	✓	✓	✗	✗
Math	10	4453	✓	✓	✗	✗
Math	11	4431	✓	✓	✗	✗
Math	12	4417	✓	✓	✗	✗
Math	13	4405	✓	✓	✗	✗
Math	14	4404	✓	✓	✓	✓
Math	15	4134	✓	✓	✗	✗
Math	16	4132	✓	✓	✗	✗
Math	17	4047	✓	✗	✓	✗
Math	18	4042	✓	✓	✗	✗
Math	19	4036	✓	✓	✗	✗
Math	20	4035	✓	✓	✗	✗
Math	21	3997	✓	✓	✗	✗
Math	22	3989	✓	✓	✗	✗
Math	23	3967	✓	✓	✗	✗
Math	24	3966	✓	✓	✗	✗
Math	25	3932	✓	✓	✗	✗
Math	26	3853	✓	✓	✓	✓
Math	27	3853	✓	✓	✗	✗
Math	28	3852	✓	✓	✗	✗
Math	29	3690	✓	✓	✓	✓
Math	30	3646	✓	✓	✗	✗
Math	31	3531	✓	✓	✗	✗
Math	32	3522	✓	✓	✗	✗
Math	33	3504	✓	✓	✗	✗
Math	34	3488	✓	✓	✗	✗
Math	35	3479	✓	✓	✗	✗
Math	36	3469	✓	✓	✗	✗
Math	37	3498	✓	✓	✓	✓
Math	38	3208	✓	✗	✗	✗
Math	39	3207	✓	✓	✗	✗
Math	40	3147	✓	✓	✗	✗
Math	41	3142	✓	✓	✗	✗
Math	42	3121	✓	✓	✗	✗
Math	43	3101	✓	✓	✗	✗
Math	44	3067	✓	✓	✗	✗
Math	45	3022	✓	✓	✗	✗
Math	46	2945	✓	✓	✗	✗
Math	47	2945	✓	✓	✗	✗
Math	48	2943	✓	✓	✓	✓
Math	49	2902	✓	✓	✗	✗
Math	50	2900	✓	✓	✓	✓
Math	51	2889	✓	✓	✗	✗
Math	52	2866	✓	✓	✗	✗
Math	53	2473	✓	✓	✓	✓
Math	54	2368	✓	✓	✗	✗
Math	55	2349	✓	✓	✗	✗
Math	56	2348	✓	✓	✓	✓
Math	57	2331	✓	✓	✗	✗
Math	58	2302	✓	✓	✗	✗
Math	59	2235	✓	✗	✓	✗
Math	60	2218	✓	✓	✓	✓

Table B.4: Math Subjects (Cont).

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Math	61	2366	✓	✓	✓	✓
Math	62	2365	✓	✓	✓	✓
Math	63	2282	✓	✓	✓	✓
Math	64	2274	✓	✓	✓	✓
Math	65	2273	✓	✓	✓	✓
Math	66	2261	✓	✓	✗	✗
Math	67	2255	✓	✓	✓	✓
Math	68	2186	✓	✓	✓	✓
Math	69	2186	✓	✓	✗	✗
Math	70	2184	✓	✓	✗	✗
Math	71	2169	✓	✓	✗	✗
Math	72	2140	✓	✓	✓	✓
Math	73	2140	✓	✓	✓	✓
Math	74	2131	✓	✓	✓	✓
Math	75	2135	✓	✓	✓	✓
Math	76	2135	✓	✓	✓	✓
Math	77	2129	✓	✓	✓	✓
Math	78	2106	✓	✓	✗	✗
Math	79	2104	✓	✓	✗	✗
Math	80	2102	✓	✓	✗	✗
Math	81	2101	✓	✓	✗	✗
Math	82	2056	✓	✓	✗	✗
Math	83	2055	✓	✓	✗	✗
Math	84	2054	✓	✓	✗	✗
Math	85	1983	✓	✓	✗	✗
Math	86	1894	✓	✓	✗	✗
Math	87	1893	✓	✓	✗	✗
Math	88	1880	✓	✓	✗	✗
Math	89	1691	✓	✓	✗	✗
Math	90	1691	✓	✓	✗	✗
Math	91	1671	✓	✓	✗	✗
Math	92	1507	✓	✓	✗	✗
Math	93	1503	✓	✓	✗	✗
Math	94	1500	✓	✓	✗	✗
Math	95	1300	✓	✓	✗	✗
Math	96	1271	✓	✓	✗	✗
Math	97	1095	✓	✓	✗	✗
Math	98	1094	✓	✓	✗	✗
Math	99	1552	✓	✓	✗	✗
Math	100	1179	✓	✓	✗	✗
Math	101	1177	✓	✓	✗	✗
Math	102	1146	✓	✓	✗	✗
Math	103	1014	✓	✓	✗	✗
Math	104	1003	✓	✓	✓	✓
Math	105	887	✓	✓	✗	✗
Math	106	875	✓	✓	✗	✗
Totals:			**106**	**103**	**27**	**25**

Table B.5: Mockito Subjects

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Mockito	1	1388	✓	✓	✓	✓
Mockito	2	1397	✓	✓	✗	✗
Mockito	3	1388	✓	✓	✓	✓
Mockito	4	1388	✓	✓	✗	✗
Mockito	5	1377	✓	✓	✗	✗
Mockito	6	1367	✓	✗	✗	✗
Mockito	7	1366	✓	✓	✗	✗
Mockito	8	1365	✓	✓	✗	✗
Mockito	9	1362	✓	✓	✗	✗
Mockito	10	1341	✓	✓	✗	✗
Mockito	11	1330	✓	✗	✗	✗
Mockito	12	NA	✗	✗	✗	✗
Mockito	13	NA	✗	✗	✗	✗
Mockito	14	NA	✗	✗	✗	✗
Mockito	15	NA	✗	✗	✗	✗
Mockito	16	NA	✗	✗	✗	✗
Mockito	17	NA	✗	✗	✗	✗
Mockito	18	1387	✓	✓	✗	✗
Mockito	19	1387	✓	✓	✗	✗
Mockito	20	1379	✓	✓	✓	✓
Mockito	21	1359	✓	✓	✓	✓
Mockito	22	NA	✗	✗	✗	✗
Mockito	23	NA	✗	✗	✗	✗
Mockito	24	NA	✗	✗	✗	✗
Mockito	25	NA	✗	✗	✗	✗
Mockito	26	NA	✗	✗	✗	✗
Mockito	27	NA	✗	✗	✗	✗
Mockito	28	NA	✗	✗	✗	✗
Mockito	29	NA	✗	✗	✗	✗
Mockito	30	NA	✗	✗	✗	✗
Mockito	31	NA	✗	✗	✗	✗
Mockito	32	NA	✗	✗	✗	✗
Mockito	33	NA	✗	✗	✗	✗
Mockito	34	NA	✗	✗	✗	✗
Mockito	35	NA	✗	✗	✗	✗
Mockito	36	NA	✗	✗	✗	✗
Mockito	37	NA	✗	✗	✗	✗
Mockito	38	NA	✗	✗	✗	✗
Totals:			**15**	**13**	**4**	**4**

Table B.6: Time Subjects

Project	Bug Id	# Test Cases	1 real (coverage)	1 mutant (coverage)	1 real (history)	1 mutant (history)
Time	1	4041	✓	✓	✓	✓
Time	2	4041	✓	✓	✗	✗
Time	3	4038	✓	✓	✗	✗
Time	4	4014	✓	✓	✗	✗
Time	5	4013	✓	✓	✗	✗
Time	6	3998	✓	✓	✗	✗
Time	7	3980	✓	✓	✗	✗
Time	8	3970	✓	✓	✓	✓
Time	9	3970	✓	✓	✓	✓
Time	10	3954	✓	✓	✗	✗
Time	11	3949	✓	✓	✗	✗
Time	12	3936	✓	✓	✗	✗
Time	13	3916	✓	✓	✗	✗
Time	14	3906	✓	✓	✗	✗
Time	15	3894	✓	✓	✗	✗
Time	16	3893	✓	✓	✗	✗
Time	17	3883	✓	✓	✗	✗
Time	18	3873	✓	✓	✗	✗
Time	19	3871	✓	✓	✗	✗
Time	20	3868	✓	✓	✗	✗
Time	21	3866	✓	✓	✗	✗
Time	22	3830	✓	✓	✗	✗
Time	23	3828	✓	✓	✗	✗
Time	24	3826	✓	✓	✗	✗
Time	25	3810	✓	✓	✗	✗
Time	26	3806	✓	✓	✗	✗
Time	27	3749	✓	✓	✗	✗
Totals:			**27**	**27**	**3**	**3**

Bibliography

[1] David Paterson, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. Using controlled numbers of real faults and mutants to empirically evaluate coverage-based test case prioritization. In *International Workshop on Automated Software Test (AST 2018)*, pages 57–63, 2018.

[2] David Paterson, José Campos, Rui Abreu, Gregory M. Kapfhammer, Gordon Fraser, and Phil McMinn. An empirical study on the use of defect prediction for test case prioritization. In *International Conference on Software Testing, Verification and Validation (ICST 2019)*, pages 346–357, 2019.

[3] Gartner says global it spending to grow 3.2 percent in 2019. URL https://www.gartner.com/en/newsroom/press-releases/2018-10-17-gartner-says-global-it-spending-to-grow-3-2-percent-in-2019.

[4] Total numbers of programmers and software development professionals in the united kingdom (uk) from 2011 to 2018 (in 1,000s). URL https://www.statista.com/statistics/318818/numbers-of-programmers-and-software-development-professionals-in-the-uk/.

[5] The rise and fall of knight capital — buy high, sell low. rinse and repeat., . URL https://medium.com/@bishr_tabbaa/the-rise-and-fall-of-knight-capital-buy-high-sell-low-rinse-and-repeat-ae17fae780f6.

[6] Crash and burn — a short story of ariane 5 flight 501, . URL https://medium.com/@bishr_tabbaa/crash-and-burn-a-short-story-of-ariane-5-flight-501-3a3c50e0e284.

[7] Windows 98 crashed live on stage with bill gates, . URL https://www.theregister.co.uk/2018/04/20/windows_98_comdex_bsod_video/.

[8] Gregory M. Kapfhammer. Regression testing. In *The Encyclopedia of Software Engineering*. 2010.

[9] Apache Geode Nightly Test Report. Apache Geode nightly test report, 2018. URL https://builds.apache.org/job/Geode-release/103/.

[10] G. Rothermel, R. H. Untch, Chengyun Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *International Conference on Software Maintenance (ICSM 1999)*, pages 179–188, 1999.

[11] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *Transactions on Software Engineering (TSE 2002)*, 28(2):159–182, 2002.

[12] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *Transactions on Software Engineering (TSE 2007)*, 33(4):225–237, 2007.

[13] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis (ISSTA 2006)*, pages 1–12, 2006.

[14] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel, and Sebastian Elbaum. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, Department of Computer Science and Engineering, University of Nebraska, Lincoln, Nebraska, USA, 2006.

[15] Sara Alspaugh, Kristen R. Walcott, Michael Belanich, Gregory M. Kapfhammer, and Mary Lou Soffa. Efficient time-aware prioritization with knapsack solvers. In *International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 13–18, 2007.

[16] René Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *International Symposium on Software Testing and Analysis (ISSTA 2014)*, pages 433–436, 2014.

[17] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation Conference (GECCO 2004)*, pages 1338–1349, 2004.

[18] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Predicting the location and number of faults in large software systems. *Transactions on Software Engineering (TSE 2005)*, 31(4):340–355, 2005.

[19] Qi Luo, Kevin Moran, Denys Poshyvanyk, and Massimiliano Di Penta. Assessing test case prioritization on real faults and mutants. In *International Conference on Software Maintenance and Evolution (ICSME 2018)*, pages 240–251, 2018.

[20] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis (ISSTA 2014)*, pages 437–440, 2014.

[21] D. Hao, L. Zhang, L. Zang, Y. Wang, X. Wu, and T. Xie. To be optimal or not in test-case prioritization. *Transactions on Software Engineering (TSE 2016)*, 42 (5):490–505, 2016.

[22] Y. Li, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *International Conference on Software Engineering (ICSE 2016)*, pages 535–546, 2016.

[23] Todd L. Graves, Alan F. Karr, J. S. Marron, and Harvey Siy. Predicting fault incidence using software change history. *Transactions on Software Engineering (TSE 2000)*, 26(7):653–661, 2000.

[24] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *Transactions on Software Engineering (TSE 2007)*, 33(1): 2–13, 2007.

[25] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *International Workshop on Predictor Models in Software Engineering (PROMISE 2007)*, pages 9–9, 2007.

[26] André Freitas. Software Repository Mining Analytics to Estimate Software Component Reliability. Master's thesis, Faculdade de Engenharia da Universidade do Porto, 2015.

[27] R. Patton. *Software Testing.* 2006.

[28] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, 1990.

[29] Ieee standard classification for software anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, 2010.

[30] Earl Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *Transactions on Software Engineering (TSE 2015)*, 41(5), 2015.

[31] Lingming Zhang, Dan Hao, Lu Zhang, Gregg Rothermel, and Hong Mei. Bridging the gap between the total and additional test-case prioritization strategies. In

International Conference on Software Engineering (ICSE 2013), pages 192–201, 2013.

[32] Sai Zhang, Darioush Jalali, Jochen Wuttke, Kıvanç Muşlu, Wing Lam, Michael D. Ernst, and David Notkin. Empirically revisiting the test independence assumption. In *International Symposium on Software Testing and Analysis (ISSTA 2014)*, pages 385–396, 2014.

[33] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing JUnit test cases: An empirical assessment and cost-benefits analysis. *Empirical Software Engineering*, 11(1):33–70, 2006.

[34] A. G. Malishevsky, G. Rothermel, and S. Elbaum. Modeling the cost-benefits tradeoffs for regression testing techniques. In *International Conference on Software Maintenance (ICSM 2002)*, pages 204–213, 2002.

[35] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Journal*, 12(3):185–210, 2004.

[36] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. *Transactions on Software Engineering (TSE 2010)*, 36(5):593–617, 2010.

[37] X. Qu, M. B. Cohen, and K. M. Woolf. Combinatorial interaction regression testing: A study of test case generation and prioritization. In *International Conference on Software Maintenance (ICSM 2007)*, pages 255–264, 2007.

[38] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *International Symposium on Software Testing and Analysis (ISSTA 2000)*, pages 929–948, 2000.

[39] Shen Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44(10), 1965.

[40] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. Coverage-based regression test case selection, minimization and prioritization: a case study on an industrial system. *Journal of Software Testing, Verification and Reliability (JSTVR 2015)*, 25(4):371–396, 2015.

[41] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *Transactions on Software Engineering and Methodology*, 24(2):1–31, 2014.

[42] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *International Conference on Software Engineering (ICSE 2014)*, pages 435–445, 2014.

[43] P. S. Kochhar, F. Thung, and D. Lo. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER 2015)*, pages 560–564, 2015.

[44] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. In *International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 86–96, 2002.

[45] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering (TSE 2011)*, 37(5):649–678, 2011.

[46] W.Eric Wong and Aditya P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.

[47] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability (JSTVR 2012)*, 22(2):67–120, 2012.

[48] Wei Chen, Roland H. Untch, Gregg Rothermel, Sebastian Elbaum, and Jeffery von Ronne. Can fault-exposure-potential estimates improve the fault detection abilities of test suites? *Journal of Software Testing, Verification and Reliability (JSTVR 2002)*, 12(4), 2002.

[49] Hyunsook Do and Gregg Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Transactions on Software Engineering (TSE 2006)*, 32(9):733–752, 2006.

[50] M. Delahaye and L. du Bousquet. A comparison of mutation analysis tools for java. In *International Conference on Quality Software (QSIC 2019)*, pages 187–195, 2013.

[51] Jung-Min Kim and Adam Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *International Conference on Software Engineering (ICSE 2002)*, pages 119–129, 2002.

[52] Yu-Chi Huang, Kuan-Li Peng, and Chin-Yu Huang. A history-based cost-cognizant test case prioritization technique in regression testing. *Journal of Systems and Software*, 85(3):626–637, 2012.

[53] Chu-Ti Lin, Cheng-Ding Chen, Chang-Shi Tsai, and Gregory M. Kapfhammer. History-based test case prioritization with software version awareness. In *International Conference on Engineering of Complex Computer Systems*, pages 171–172, 2013.

[54] Dusica Marijan, Arnaud Gotlieb, and Sagar Sen. Test case prioritization for continuous regression testing: An industrial case study. In *International Conference on Software Maintenance (ICSM 2013)*, pages 540–543, 2013.

[55] Sebastian Elbaum, Gregg Rothermel, and John Penix. Techniques for improving regression testing in continuous integration development environments. In *International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 235–245, 2014.

[56] Y. Cho, J. Kim, and E. Lee. History-based test case prioritization for failure information. In *Asia-Pacific Software Engineering Conference*, pages 385–388, 2016.

[57] Avinash Gupta, Nayneesh Mishra, Aprna Tripathi, Manu Vardhan, and Dharmender Singh Kushwaha. An improved history-based test prioritization technique technique using code coverage. In *International Conference on Communication and Computer Engineering*, pages 437–448, 2015.

[58] Shin Yoo, Mark Harman, Paolo Tonella, and Angelo Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 201–212, 2009.

[59] Ryan Carlson, Hyunsook Do, and Anne Denton. A clustering approach to improving test case prioritization: An industrial case study. In *International Conference on Software Maintenance (ICSME 2011)*, pages 382–391, 2011.

[60] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: The distribution of program failures in a profile space. In *European Software Engineering Conference (ESEC 2001)*, pages 246–255, 2001.

[61] David Leon and Andy Podgurski. A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In *International Symposium on Software Reliability Engineering (ISSRE 2003)*, pages 442–453, 2003.

[62] P. Tonella, P. Avesani, and A. Susi. Using the case-based ranking methodology for test case prioritization. In *International Conference on Software Maintenance (ICSM 2006)*, pages 123–133, 2006.

[63] Christian Blum and Andrea Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR 2003)*, 35(3):268–308, 2003.

[64] Mark Harman. Search based software engineering. In *Computational Science (ICCS 2006)*, pages 740–747, 2006.

[65] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. 1st edition, 1989.

[66] S. N. Sivanandam and S. N. Deepa. *Introduction to Genetic Algorithms*. 2007.

[67] Fang Yuan, Yi Bian, Zheng Li, and Ruilian Zhao. Epistatic genetic algorithm for test case prioritization. In *International Symposium on Search-Based Software Engineering (SSBSE 2015)*, pages 109–124, 2015.

[68] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *European Conference on Foundations of Software Engineering (FSE 2011)*, pages 416–419, 2011.

[69] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *Transactions on Software Engineering (TSE 2013)*, 39(2):276–291, 2013.

[70] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *International Symposium on Search-Based Software Engineering (SSBSE 2015)*, volume 9275, pages 93–108. 2015.

[71] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. *Search Based Software Engineering: Techniques, Taxonomy, Tutorial*, pages 1–59. 2012.

[72] Tadahiko Murata and Hisao Ishibuchi. Moga: multi-objective genetic algorithms. In *International Conference on Evolutionary Computation*, volume 1, pages 289–294, 1995.

[73] Abdullah Konak, David W. Coit, and Alice E. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering System Safety*, 9(9): 992–1007, 2006.

[74] X. Wang and H. Zeng. History-Based Dynamic Test Case Prioritization for Requirement Properties in Regression Testing. In *International Workshop on Continuous Software Evolution and Delivery (CSED 2016)*, pages 41–47, 2016.

[75] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *Transactions on Evolutionary Computation*, 6(2), 2002.

[76] Shin Yoo and Mark Harman. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis (ISSTA 2007)*, pages 140–150, 2007.

[77] Pablo Moscato and Carlos Cotta. *A Gentle Introduction to Memetic Algorithms.* 2003.

[78] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *Transactions on Software Engineering (TSE 2010)*, 36(2):226–247, 2010.

[79] Gordon Fraser, Phil McMinn, and Andrea Arcuri. Test suite generation with memetic algorithms. In *Genetic and Evolutionary Computation Conference (GECCO 2013)*, pages 1437–1444, 2013.

[80] F. M. Nejad, R. Akbari, and M. M. Dejam. Using memetic algorithms for test case prioritization in model based software testing. In *Conference on Swarm Intelligence and Evolutionary Computation (CSIEC 2016)*, pages 142–147, 2016.

[81] Dario Di Nucci, Annibale Panichella, Andy Zaidman, and Andrea De Lucia. Hypervolume-based search for test case prioritization. In *International Symposium on Search-Based Software Engineering (SSBSE 2015)*, pages 157–172, 2015.

[82] A. Panichella, R. Oliveto, M. D. Penta, and A. De Lucia. Improving multi-objective test case selection by injecting diversity in genetic algorithms. *Transactions on Software Engineering (TSE 2015)*, 41(4):358–383, 2015.

[83] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 654–665, 2014.

[84] René Just. https://github.com/rjust/defects4j.

[85] Romi Wahono. A systematic literature review of software defect prediction: Research trends, datasets, methods and frameworks. *Journal of Software Engineering*, pages 1–16, 2015.

[86] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering*, pages 1276–1304, 2012.

[87] Ripon K. Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R. Prasad. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In *Proceedings of*

the 15th International Conference on Mining Software Repositories, page 10–13, 2018.

[88] Cagatay Catal and Banu Diri. A systematic review of software fault prediction studies. *Expert Systems with Applications*, 36(4), 2009.

[89] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? page 1–5, 2005.

[90] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *International Conference on Software Engineering (ICSE 2007)*, pages 489–498, 2007.

[91] T. J. McCabe. A complexity measure. *Transactions on Software Engineering (TSE 1976)*, SE-2(4):308–320, 1976.

[92] Maurice H. Halstead. *Elements of Software Science (Operating and Programming Systems Series)*. 1977.

[93] Weka 3: Machine learning software in java. URL https://www.cs.waikato.ac.nz/ml/weka/.

[94] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *International Conference on Software Engineering (ICSE 2008)*, pages 181–190, 2008.

[95] Sunghun Kim, E. James Whitehead, Jr., and Yi Zhang. Classifying software changes: Clean or buggy? *Transactions on Software Engineering (TSE 2008)*, 34 (2):181–196, 2008.

[96] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *International Conference on Software Engineering (ICSE 2009)*, pages 78–88, 2009.

[97] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. Bugcache for inspections: Hit or miss? In *European Conference on Foundations of Software Engineering (FSE 2011)*, pages 322–331, 2011.

[98] Bug prediction at google. URL http://google-engtools.blogspot.co.uk/2011/12/.

[99] Marco D'Ambros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5), 2012.

[100] Ahmet Okutan and Olcay Yildiz. Software defect prediction using bayesian networks. pages 154—181, 2014.

[101] Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. Transfer learning for cross-company software defect prediction. *Information and Software Technology*, pages 248 – 256, 2012.

[102] Peng He, Bing Li, Xiao Liu, Jun Chen, and Yutao Ma. An empirical study on software defect prediction with a simplified metric set. *Information and Software Technology*, pages 170 – 190, 2015.

[103] D. Bowes, S. Counsell, T. Hall, J. Petric, and T. Shippey. Getting defect prediction into industrial practice: the elff tool. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 44–47, 2017.

[104] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. Szz unleashed: An open implementation of the szz algorithm - featuring example usage in a study of just-in-time bug prediction for the jenkins project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation*, page 7–12, 2019.

[105] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E. Hassan, and Kenichi Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, pages 683–711, 2018.

[106] Tim Menzies and Martin Shepperd. Special issue on repeatable results in software engineering prediction. *Empirical Software Engineering*, pages 1–17, 2012.

[107] Paul Luo Li, James Herbsleb, Mary Shaw, and Brian Robinson. Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *International Conference on Software Engineering (ICSE 2006)*, pages 413–422, 2006.

[108] H. Srikanth, L. Williams, and J. Osborne. System test case prioritization of new and regression test cases. In *International Symposium on Empirical Software Engineering*, pages 10 pp.–, 2005.

[109] Song Wang, Jaechang Nam, and Lin Tan. QTEP: Quality-aware test case prioritization. In *International Symposium on Foundations of Software Engineering (FSE 2017)*, pages 523–534, 2017.

[110] Siavash Mirarab and Ladan Tahvildari. A prioritization approach for software test cases based on Bayesian networks. In *International Conference on Fundamental Approaches to Software Engineering (FASE 2007)*, pages 276–290, 2007.

[111] Bing Liu and Lei Zhang. *A Survey of Opinion Mining and Sentiment Analysis*, pages 415–463. 2012.

[112] Apoorv Agarwal, Boyi Xie, Ilia Vovsha, Owen Rambow, and Rebecca Passonneau. Sentiment analysis of twitter data. In *Proceedings of the Workshop on Languages in Social Media*, page 30–38, 2011.

[113] Younggue Bae and Hongchul Lee. Sentiment analysis of twitter audiences: Measuring the positive or negative influence of popular twitterers. *Journal of the American Society for Information Science and Technology*, pages 2521–2535, 2012.

[114] Palak Baid, Apoorva Gupta, and Neelam Chaplot. Sentiment analysis of movie reviews using machine learning techniques. pages 45–49, 2017.

[115] Humera Shaziya. Text categorization of movie reviews for sentiment analysis. pages 11255–11262, 2018.

[116] Martin Haselmayer and Marcelo Jenny. Sentiment analysis of political communication: combining a dictionary approach with crowdcoding. *Quality and Quantity*, pages 2623–2646, 2017.

[117] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The guardian*, page 22, 2018.

[118] Linda Risso. Harvesting your soul? cambridge analytica and brexit. *Brexit Means Brexit*, pages 75–90, 2018.

[119] Roberto J Gonzalez. Hacking the citizenry?: Personality profiling,'big data'and the election of donald trump. *Anthropology Today*, pages 9–12, 2017.

[120] Alec Go, Richa Bhayani, and Lei Huang. Twitter sentiment classification using distant supervision. *Processing*, 150, 01 2009.

[121] Alexander Pak and Patrick Paroubek. Twitter as a corpus for sentiment analysis and opinion mining. In *LREc*, pages 1320–1326, 2010.

[122] Adam Bermingham and Alan Smeaton. Classifying sentiment in microblogs: Is brevity an advantage? *CIKM 2010 - 19th International Conference on Information and Knowledge Management*, 2010.

[123] Luciano Barbosa and Junlan Feng. Robust sentiment detection on twitter from biased and noisy data. In *Proceedings of the 23rd International Conference on Computational Linguistics: Posters*, page 36–44, 2010.

[124] Emitza Guzman and Walid Maalej. How do users like this feature? a fine grained sentiment analysis of app reviews. In *2014 IEEE 22nd international requirements engineering conference*, pages 153–162, 2014.

[125] Xing Fang and Justin Zhan. Sentiment analysis using product review data. *Journal of Big Data*, 2(1):1–14, 2015.

[126] Walter Kasper and Mihaela Vela. Sentiment analysis for hotel reviews. *Speech Technology*, pages 96–109, 2012.

[127] GN Vikram Elango. Sentiment analysis for hotel reviews, 2018.

[128] Walaa Medhat, Ahmed Hassan, and Hoda Korashy. Sentiment analysis algorithms and applications: A survey. *Ain Shams Engineering Journal*, 5(4):1093–1113, 2014.

[129] Hanhoon Kang, Seong Joon Yoo, and Dongil Han. Senti-lexicon and improved naïve bayes algorithms for sentiment analysis of restaurant reviews. *Expert Systems with Applications*, 39(5):6000–6010, 2012.

[130] Jonathan Ortigosa-Hernández, Juan Diego Rodríguez, Leandro Alzate, Manuel Lucania, Iñaki Inza, and Jose A. Lozano. Approaching sentiment analysis by using semi-supervised learning of multi-dimensional classifiers. *Neurocomputing*, 92:98–115, 2012.

[131] Chien Chin Chen and You-De Tseng. Quality evaluation of product reviews using an information quality framework. *Decision Support Systems*, 50(4), 2011. Enterprise Risk and Security Management: Data, Text and Web Mining.

[132] Yung-Ming Li and Tsung-Ying Li. Deriving market intelligence from microblogs. *Decision Support Systems*, 55(1):206–217, 2013.

[133] Marshall Van de Camp and Antal Van den Bosch. The socialist network. *Decision Support Systems*, 53:761–769, 2012.

[134] Yulan He and Deyu Zhou. Self-training from labeled features for sentiment analysis. *Information Processing & Management*, 47(4):606–616, 2011.

[135] Peter Turney. Thumbs up or thumbs down? semantic orientation applied to unsupervised classification of reviews. In *Association of Computational Linguistics (ACL 2002)*, pages 417–424, 2002.

[136] Jonathon Read and John Carroll. Weakly supervised techniques for domain-independent sentiment classification. In *International CIKM Workshop on Topic-sentiment Analysis for Mass Opinion*, pages 45–52, 2009.

[137] Minqing Hu and Bing Liu. Mining and summarizing customer reviews. In *International Conference on Knowledge Discovery and Data Mining*, pages 168–177, 2004.

[138] Soo-Min Kim and Eduard Hovy. Determining the sentiment of opinions. In *Conference on Computational Linguistics (COLING 2004)*, 2004.

[139] Emitza Guzman, David Azócar, and Yang Li. Sentiment analysis of commit comments in github: An empirical study. *International Conference on Mining Software Repositories (MSR 2014)*, pages 352–355, 2014.

[140] R. Souza and B. Silva. Sentiment analysis of travis ci builds. In *International Conference on Mining Software Repositories (MSR 2017)*, pages 459–462, 2017.

[141] Md Rakibul Islam and Minhaz Fahim Zibran. Analysis of software bug related commit messages. 2018.

[142] D. Binkley, H. Feild, D. Lawrie, and M. Pighin. Software fault prediction using language processing. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, pages 99–110, 2007.

[143] Dave Binkley, Dawn Lawrie, and Christopher Morrell. The need for software specific natural language techniques. *Empirical Software Engineering*, pages 1–28, 2017.

[144] Edmund Burke, Graham Kendall, Jim Newall, Emma Hart, Peter Ross, and Sonia Schulenburg. *Hyper-Heuristics: An Emerging Direction in Modern Search Technology*. 2003.

[145] Sheffield hpc facilities. URL https://www.sheffield.ac.uk/cics/research/hpc.

[146] Mutation fails for chart-10. URL https://github.com/rjust/defects4j/issues/50.

[147] Mutation analysis does not work for some mockito versions. URL https://github.com/rjust/defects4j/issues/198.

[148] Kanonizo, 2018. URL https://github.com/kanonizo/kanonizo.

[149] S. Elbaum, A. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *International Conference on Software Engineering (ICSE 2001)*, pages 329–338, 2001.

[150] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Journal of Software Testing, Verification and Reliability (JSTVR 2014)*, 24(3), 2014.

[151] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic models. *Journal of Empirical Software Engineering*, 19(1):182–212, 2014.

[152] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie. idflakies: A framework for detecting and partially classifying flaky tests. In *International Conference on Software Testing, Verification and Validation (ICST)*, pages 312–322, 2019.

[153] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *International Symposium on Software Testing and Analysis (ISSTA 2016)*, pages 354–365, 2016.

[154] Experimental data from this paper's evaluation, 2018. URL https://www.bitbucket.com/testprioritisation/ast2018_data.

[155] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *International Conference on Software Engineering (ICSE 2013)*, pages 372–381, 2013.

[156] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *The 18th IEEE International Symposium on Software Reliability (ISSRE '07)*, pages 215–224, 2007.

[157] Jose Campos and Rui Abreu. Encoding Test Requirements as Constraints for Test Suite Minimization. In *International Conference on Information Technology: New Generations*, pages 317–322, 2013.

[158] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. GZoltar: An Eclipse plug-in for testing and debugging. In *International Conference on Automated Software Engineering (ASE 2012)*, pages 378–381, 2012.

[159] GZoltar. GZoltar, 2012 (accessed March 2020).

[160] Carlos Pacheco and Michael D. Ernst. Randoop: Feedback-directed random testing for Java. In *International Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA 2007)*, pages 815–816, 2007.

[161] David Saff and Michael D. Ernst. An experimental evaluation of continuous testing during development. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 76–85, 2004.

[162] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *International Conference on Mining Software Repositories (MSR 2011)*, pages 153–162, 2011.

[163] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, 1952.

[164] Andy Zaidman, Bart Van Rompaey, Arie van Deursen, and Serge Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[165] R. Peters and A. Zaidman. Evaluating the lifespan of code smells using software repository mining. In *European Conference on Software Maintenance and Reengineering*, pages 411–416, 2012.

[166] Gilad Mishne, Natalie S Glance, et al. Predicting movie sales from blogger sentiment. In *AAAI spring symposium: computational approaches to analyzing weblogs*, pages 155–158, 2006.

[167] Michelle Annett and Grzegorz Kondrak. A comparison of sentiment analysis techniques: Polarizing movie blogs. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pages 25–35, 2008.

[168] Vivek Kumar Singh, Rajesh Piryani, Ashraf Uddin, and Pranav Waila. Sentiment analysis of movie reviews: A new feature-based heuristic for aspect-level sentiment classification. In *International Mutli-Conference on Automation, Computing, Communication, Control and Compressed Sensing (iMac4s 2013)*, pages 712–717, 2013.

[169] John Blitzer, Mark Dredze, and Fernando Pereira. Biographies, Bollywood, boom-boxes and blenders: Domain adaptation for sentiment classification. In *Association of Computational Linguistics (ACL 2007)*, page 440–447, 2007.

[170] Callen Rain. Sentiment analysis in amazon reviews using probabilistic machine learning. *Swarthmore College*, 2013.

[171] Aashutosh Bhatt, Ankit Patel, Harsh Chheda, and Kiran Gawande. Amazon review classification and sentiment analysis. *International Journal of Computer Science and Information Technologies*, 6(6), 2015.

[172] Bo Pang, Lillian Lee, et al. Opinion mining and sentiment analysis. *Foundations and Trends in Information Retrieval*, 2(1–2):1–135, 2008.

[173] Gitpython: a python library used to interact with git repositories.

[174] Jgit by the eclipse foundation. URL https://www.eclipse.org/jgit/.

[175] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. PyDriller: Python framework for mining software repositories. In *International Symposium on the Foundations of Software Engineering (FSE 2018)*, pages 908–911, 2018.

[176] Robbert Jongeling, Subhajit Datta, and Alexander Serebrenik. Choosing your weapons: On sentiment analysis tools for software engineering research. pages 531–535, 2015.

[177] Textblob: Simplified text processing. URL https://github.com/sloria/textblob.

[178] S. Ahuja and G. Dubey. Clustering and sentiment analysis on twitter data. In *2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*, pages 1–5, 2017.

[179] Shubhodip Saha, Jainath Yadav, and Prabhat Ranjan. Proposed approach for sarcasm detection in twitter, 2017.

[180] Ankur Goel, Jyoti Gautam, and Sitesh Kumar. Real time sentiment analysis of tweets using naive bayes. In *2016 2nd International Conference on Next Generation Computing Technologies (NGCT)*, pages 257–261, 2016.

[181] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. Security and emotion: sentiment analysis of security discussions on github. In *Proceedings of the 11th working conference on Mining Software Repositories (MSR)*, pages 348–351, 2014.

[182] Prateek Garg and Vineeta Guide Bassi. *Sentiment analysis of twitter data using NLTK in python.* PhD thesis, 2016.

[183] Matias Martinez and Martin Monperrus. Coming: a tool for mining change pattern instances from git commits. In *International Conference on Software Engineering: Tool Track (ICSE 2019)*, 2019.

[184] Chaitanya S. Lakkundi, Vartika Agrahari, and Sridhar Chimalakonda. GE852: A dataset of 852 game engines. *CoRR*, abs/1905.04482, 2019.

[185] Moein Owhadi-Kareshk and Sarah Nadi. Scalable software merging studies with merganser. In *International Conference on Mining Software Repositories (MSR 2019)*, pages 560–564, 2019.

[186] Jared B Hawkins, John S Brownstein, Gaurav Tuli, Tessa Runels, Katherine Broecker, Elaine O Nsoesie, David J McIver, Ronen Rozenblum, Adam Wright, Florence T Bourgeois, and Felix Greaves. Measuring patient-perceived quality of care in us hospitals using twitter. *BMJ Quality & Safety*, 25(6), 2016.

[187] Isidoros Perikos and Ioannis Hatzilygeroudis. Recognizing emotions in text using ensemble of classifiers. *Engineering Applications of Artificial Intelligence*, 51:191–201, 2016.

[188] Toufique Ahmed, Amiangshu Bosu, Anindya Iqbal, and Shahram Rahimi. Senticr: A customized sentiment analysis tool for code review interactions. In *International Conference on Automated Software Engineering (ASE 2017)*, pages 106–111, 2017.

[189] J. Liang, S. Elbaum, and G. Rothermel. Redefining prioritization: Continuous prioritization for continuous integration. In *International Conference on Software Engineering (ICSE 2018)*, pages 688–698, 2018.

[190] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[191] M. Harman, E. Burke, J. A. Clark, and X. Yao. Dynamic adaptive search based software engineering. In *International Symposium on Empirical Software Engineering and Measurement (ESEM 2012)*, pages 1–8, 2012.

[192] Andrea Arcuri and Gordon Fraser. *On Parameter Tuning in Search Based Software Engineering*, pages 33–47. 2011.