Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq
Student number: 01503338

Supervisors: Prof. dr. Bruno Volckaert, Prof. dr. ir. Filip De Turck
Counsellors: Jasper Vaneessen, Dwight Kerkhove

Master's dissertation submitted in order to obtain the academic degree of
Master of Science in de informatica

Academic year 2019-2020

Acknowledgements

Completing this thesis would not have been possible without the help and support of many people, some of which I want to thank personally.

First of all, I want to thank prof. dr. Bruno Volckaert and prof. dr. ir. Filip De Turck for allowing me to propose this subject and for their prompt and clear responses to every question I have asked. I especially want to thank you for permitting me to insert a two-week hiatus during the Easter break, so I could help out on the UGent Dodona project.

Secondly, I want to express my gratitude towards my counsellors Jasper Vaneessen and Dwight Kerkhove, for steering me into researching this topic, as well as their guidance, availability, and willingness to review every intermediary version of this thesis.

Furthermore, I want to thank my parents, my brother Stijn and my family for convincing me and giving me the possibility to study at the university, to support me throughout my entire academic career and to provide me with the opportunity to pursue my childhood dreams.

Last, but surely not least, I want to thank my amazing friends, a few of them in particular. My best friend Robbe, for always being there when I need him even when I least expect it. For both supporting my wildest dreams while protecting me against my often unrealistic ideas and ambition to excel. Helena for never leaving my side, for always making me laugh when I don't want to, and most importantly to remind me that I should relax from time to time. Jana for my daily dose of laughter, fun and for her inexhaustible positivity. Tobiah for the endless design discussions and for outperforming me in almost every school project, to encourage me to continuously raise the bar and to never give up. Finally, I want to thank Femke, Doortje and Freija for answering my mathematical questions, regularly asking about my thesis progression and motivating me to persevere throughout my entire education.

Thank you.

Pieter – Ghent, 2020

Summary

Summary in English will come here.

Samenvatting

Nederlandse samenvatting komt hier.

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**This abstract is very abstract.**

Keywords—**words, will, appear, here, soon**

I. INTRODUCTIE

Things will appear here. [1]

REFERENCES

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, "Beyond the waterfall : software development at microsoft," 02 1995.

Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

Supervisor(s): Prof. dr. B. Volckaert, Prof. dr. ir. F. De Turck, J. Vaneessen, D Kerkhove

Abstract—**Dit abstract is super abstract.**

Trefwoorden—**woorden, komen, hier**

I. INTRODUCTIE

Dingen komen hier. [1]

REFERENTIES

[1] Michael Cusumano, Akindutire Michael, and Stanley Smith, "Beyond the waterfall : software development at microsoft," 02 1995.

Lay summary

Lay summary will come here.

Contents

Terms

Acronyms

CI Continuous Integration. 2, 15, 19

IEEE The Institute of Electrical and Electronics Engineers. 4

SDLC Software Development Life Cycle. 4

TCP Test Case Prioritisation. 2, 19, 22, 23

TCS Test Case Selection. 2, 19, 21, 22

TDD Test-driven development. 14

TSM Test Suite Minimisation. 2, 19, 20, 22, 23

VCS Version Control System. 2

Glossary

black-box test a test case that was constructed without any knowledge of the function(s) under test. 7

MapReduce a programming paradigm that allows large amounts of data to be processed in a distributed manner. 45

REST Representational State Transfer is an architectural design pattern used by modern web applications. This design pattern encourages standardised communication using existing HTTP methods. 32

test suite the collection of all test cases in an application. 6

white-box test a test case that was constructed after fully inspecting the function(s) under test. 6

Chapter 1

Introduction

Given the complexity and rapid pace at which software is being built today, it is inevitable that sooner or later, bugs will emerge. These bugs can either be introduced by a malfunctioning new feature, or by breaking existing functionality (*a regression*). In order to detect bugs in an application before its users do, we require an adequate *testing infrastructure*.

This testing infrastructure consists of multiple *test cases*, collectively referred to as the *test suite* of the application. The quality of a test suite can be assessed in multiple ways. The first and most commonly used method is to measure which fraction of the source code is tested by at least one test case, a ratio which is indicated as the *coverage* of the application. Another possibility is to apply transformations to the source code and validate whether or not this results in a failed test case, a process indicated as *mutation testing*.

Ideally, this testing process should be automated and performed after every change to the source code. This process is generally very time-consuming, and as such has led to the creation of various automation frameworks and tools, collectively called Continuous Integration (CI). Common examples of CI practices are automatically running the test suite and estimating the code coverage after every pushed change to the Version Control System (VCS).

However, applying these practices and maintaining a qualitative test comes at a cost. Every addition or modification to the source code must be followed by at least one test case to validate its correctness. As a result of the speed at which the source code tends to grow, the test suite suffers from severe scalability issues. While it is desirable and ideally required to execute every single test case in the test suite, there are examples known to literature where this is not possible since this incurs an increasing delay in the development process, which in turn results in economic loss.

We can take three approaches to resolve this issue and reduce the time waiting for the test results: Test Suite Minimisation (TSM), Test Case Selection (TCS) and Test Case Prioritisation (TCP). The main subject of this thesis will be to implement a framework for TCP.

The structure of this thesis is as follows. The next chapter will introduce essential concepts used in modern software engineering. Chapter 3 will elaborate more on the three mentioned approaches and present accompanying algorithms. The implementation details of the new framework will be discussed in chapter 4. Afterwards, chapter 5 will evaluate the performance of this framework and provide insights into the characteristics of a typical test suite. More specifically, this chapter will investigate the probability of (repeated) test failure and the average duration of a test run. Finally, chapter 6 will present additional ideas and improvements to the framework.

Chapter 2

Software Engineering

The Institute of Electrical and Electronics Engineers (IEEE) defines the practice of Software Engineering as the "Application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is, the application of engineering to software" [21, p. 421]. The word "systematic" in this definition emphasises the need for a structured process, depicting guidelines and models that describe how we should develop software in the most efficient way possible. Such a process does exist under the name of the Software Development Life Cycle (SDLC) [21, p. 420]. If a developer prefers not to abide by any model and act as they deem correct without following any guidelines, we employ the term *Cowboy coding* [23, p. 34].

2.1   Software Development Life Cycle

An implementation of the SDLC typically consists of several phases and a model that describes the transition from every phase to another. Depending on the nature of the software, we can either omit some or add more phases. The five phases below were compiled from multiple sources [15, 20] and describe a generic approach to which most software projects adhere.

1. **Requirements phase:** In the first phase of the development process, the developers acquaint themselves with the project and compile a list of the desired functionalities [20]. Subsequently, the developers can decide on the financial details, the required hardware specifications as well as which external software libraries will need to be acquired.

2. **Design phase:** After the developer has gained sufficient knowledge about the project requirements, they can use this information to construct an architectural design of the application. This design consists of multiple documents, such as user stories and UML-diagrams. A user story describes which actions can be performed by which users, whereas a UML-diagram specifies the technical interaction between the individual components.

3. **Implementation phase:** In the third phase, the developers will write code according to the specifications defined in the architectural designs.

4. **Testing phase:** The fourth phase is the most critical. This phase will require the developers and quality assurance managers to test the implementation of the application thoroughly. The goal of this phase is to identify potential bugs before the application is made available to other users.

5. **Operational phase:** The final phase marks the completion of the project, after which the developers can integrate it into the existing business environment of their customer.

After we have identified the phases, we must define the transition from one phase into another phase using a model. Multiple models exist in the literature [15], with each model having its advantages and disadvantages. This thesis will consider the traditional model, which is still widely used as of today. The base of this model is the Waterfall model by Benington [6]. Similar to a real waterfall, this model executes every phase in cascading order. However, this imposes several restrictions. The most prevalent issue is the inability to revise a design decision when performing the actual implementation. To mitigate this problem, Royce has proposed an improved version of the Waterfall model [32], which does allow a phase to transition back to any preceding phase. Figure 2.1 illustrates this updated model.
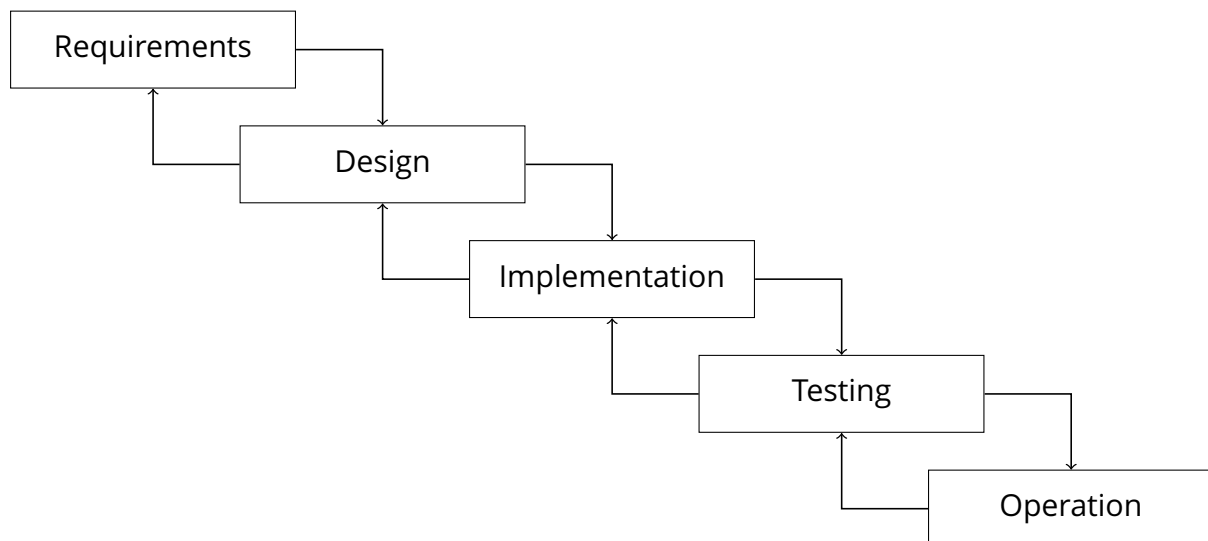
Figure 2.1: Improved Waterfall model by Royce

The focus of this thesis will be on the implementation and testing phases, as these are the most time-consuming phases of the entire process. The modification that Royce has applied to the Waterfall model is particularly useful when applied to these two phases in the context of *software regressions* [29]. We employ the term "regression" when a feature that was once working as intended is suddenly malfunctioning. The culprit of this problem can be a change in the code, but this behaviour can also have

an external cause, such as a change in the system clock due to daylight saving time. Sometimes, a regression can even be the result of a change to another, seemingly unrelated part of the application code [19].

2.1.1   Taxonomy of test cases

Software regressions and other functional bugs can ultimately incur disastrous effects, such as severe financial loss or permanent damage to the reputation of the software company. The most famous example in history is without any doubt the explosion of the Ariane 5-rocket, which was caused by an integer overflow [24]. In order to reduce the risk of bugs, we should be able to detect malfunctioning components as soon as possible to warden the application against potential failures before they occur. Consequently, we must consider the testing phase as the most critical phase of the entire development process and therefore include sufficient test cases in the application. The collection of all the test cases in an application is referred to as the *test suite*. We can distinguish many different types of test cases. This thesis will consider three categories in particular.

Unit tests

This is the most basic test type. The purpose of a unit test is to verify the behaviour of an individual component [36]. As a result, the scope of a unit test is limited to a small and isolated piece of code, e.g. one function. Implementation-wise, a unit test is typically an example of a white-box test [19, p. 12]. The term white-box indicates that the creator of the test case can manually inspect the code before constructing the test. As such, they can identify necessary edge values or corner cases. Common examples of these edge values include zero, negative numbers, empty arrays or array boundaries that might cause an overflow. Once the developer has identified the edge cases, they can construct the unit test by repeatedly calling the function under test, each time with a different (edge) argument value, and afterwards verifying its behaviour and result. These verifications are referred to as *assertions*. Example 2.1 contains a unit test written in Java using the popular JUnit test framework.

```java
public class ExampleUnitTest {
  static int square(int base) { return base * base; }

  @Test
  void testSquare() {
    Assert.assertEquals(25, square(5));
    Assert.assertEquals(4, square(−2));
    Assert.assertEquals(0, square(0));
```

```
 9    }
10  }
```

Example 2.1: Java unit test in JUnit.

Integration tests

The second category involves a more advanced type of tests. An integration test validates the interaction between two or more individual components [36]. Ideally, accompanying unit tests should exist that test these components as well. As opposed to the previous unit tests, a developer will usually implement an integration test as a black-box test [19, p. 6]. A black-box test differs from the earlier white-box tests in the fact that the implementation details of the code under test are irrelevant for the construction of the test. Since a black-box test does not require any details about the code, we can, in fact, construct the integration tests before we implement the actual feature itself. A typical example of an integration test is the communication between the front-end and the back-end side of a web application. Another example is illustrated in Example 2.2.

```java
 1  public class ExampleIntegrationTest {
 2    @Test
 3    public void testOrderPizza() {
 4      Session session = UserSystem.login("JohnDoe", "password");
 5      Assert.assertNotNull(session);
 6
 7      Pizza order = new Pizza("pepperoni");
 8      Assert.assertTrue(OrderSystem.order(session, order));
 9    }
10  }
```

Example 2.2: Java integration test in JUnit.

Regression tests

After a developer has detected a regression in the application, they will add a regression test [21, p. 372] to the test suite. This regression test must replicate the exact conditions and sequence of actions that have triggered the failure. The goal of this test is to prevent similar failures to occur in the future if the same conditions would reapply. An example is provided in Example 2.3.

```java
public class ExampleRegressionTest {
  /**
   * Regression: comments cannot be removed if a user changes
       their first
   * name.
   */
  @Test
  public void testRegression439() {
    Comment comment = CommentSystem.findById(1);
    Assert.assertNotNull(comment);

    // Change the first name.
    Assert.assertEquals("Bert", comment.user.firstName);
    comment.user.setFirstName("Matthew");
    Assert.assertEquals("Matthew", comment.user.firstName);

    // Try to remove the comment.
    comment.remove();
    Assert.assertTrue(comment.removed);
  }
}
```

Example 2.3: Java regression test in JUnit.

2.2   Test Suite Assessment

2.2.1   Coverage

The most frequently used metric to measure the quantity and thoroughness of a test suite is the *code coverage* or *test coverage* [21, p. 467]. The test coverage indicates which fraction of the application code is hit by at least one test case in the test suite. Internally, a coverage framework calculates the coverage ratio by augmenting every application statement using binary instrumentation. A hook is inserted before and after every statement to detect which statements are executed by the test cases. Many different criteria exist to interpret these results and thus to express the fraction of covered code [28]. The two most commonly used criteria are *statement coverage* and *branch coverage*.

Statement coverage   Statement coverage expresses the fraction of statements that are executed by any test case in the test suite, over the total amount of statements in the code [19]. Similarly, we can calculate the *line coverage* as the fraction of covered code lines. Since one statement can span multiple lines and one line may also contain more than one statement, both of these criteria are intrinsically related. Statement coverage is heavily criticised in literature [28, p. 37] since it is possible to achieve a statement coverage percentage of $100\,\%$ on code of which we can prove it is malfunctioning. Consider example 2.4. If a test case would call the example-function with arguments $\{a = 1, b = 2\}$, then the test case will pass, and every statement will be covered, resulting in a statement coverage of $100\,\%$. However, suppose we would call the function with arguments $\{a = 0, b = 0\}$. The first argument matches the first condition of the branch but will trigger a `division-by-zero` error, even though the previous combination of arguments reported a complete statement coverage. This simple example was already sufficient to illustrate that statement coverage is not trustworthy. However, statement coverage may still prove useful for other purposes, such as detecting unreachable code which we may safely remove.

```
1  int example(int a, int b) {
2  = 0 || b != 0) {
3    a / b;
4
5  }
```

Example 2.4: Irrelevant statement coverage in C.

Branch coverage   requires that the test cases traverse every branch of a conditional statement at least once [28, p. 37]. For an `if`-statement, we require two test cases, one for every possible outcome of the condition (`true` or `false`). For a loop-statement, we require at least two test cases as well. One test case should never execute the loop, and the other test case should execute every iteration. Optionally, we can add additional test cases for specific iterations. Observe that, while this criterion is stronger than statement coverage, it will still not detect the bug in Example 2.4. In order to mitigate this, we can use *multiple-condition coverage* [28, p. 40]. This criterion requires that for every conditional expression, every possible combination of subexpressions is evaluated at least once. If we apply this requirement to Example 2.4, the `if`-statement will only be covered if we test the following four cases.

- $a = 0, b = 0$

- $a = 0, b \neq 0$

- $a \neq 0, b = 0$

- $a \neq 0, b \neq 0$

It should be self-evident that achieving and maintaining a coverage percentage of 100% at all times is critical. However, this does not necessarily imply that every line of code, every statement or every branch must be explicitly covered [9]. Some parts of the code might be irrelevant or untestable, such as wrapper or delegation methods that only call a library function. All major programming languages have frameworks and libraries that enable the collection of coverage information during test execution, and each of these frameworks allows the exclusion parts of the code from the final coverage calculation. As of today, the most popular options are JaCoCo[1] for Java, coverage.py[2] for Python and simplecov[3] for Ruby. These frameworks report in-depth statistics on the covered code and indicate which parts require more extensive testing, as illustrated in Figure 2.3.

2.2.2 Mutation testing

The previous section has explained how we can identify which parts of the code require additional test cases. However, we cannot yet measure the quality and resilience of the test suite nor its ability to detect future failures. To accomplish this, we can employ *mutation testing*. This technique creates several *mutants* of the application under test. A mutant is a syntactically different instance of the source code. We can create a mutant by applying one or more *mutation operators* to the original source code. These mutation operators attempt to simulate typical mistakes that developers tend to make, such as the introduction of off-by-one errors, removal of statements and the replacement of logical connectors [31]. The *mutation order* refers to the amount of mutation operators that have been applied consecutively to an instance of the code. This order is traditionally rather low, as a result of the *Competent Programmer Hypothesis*, which states that programmers develop programs which are near-correct [22].

Creating and evaluating   the mutant versions of the code is a computationally expensive process which typically requires human intervention. As a result, very few software developers have managed to employ this technique in practice. Figure 2.2 illustrates the process of applying mutation testing. The first step is to consider the original program P and a set of test cases TS, to which we apply mutation operators to construct a broad set of mutants P'. Next, we evaluate every test case $t \in TS$ on the original program P to determine the correct behaviour. Note that this step assumes

---

[1] https://www.jacoco.org/jacoco/
[2] https://github.com/nedbat/coveragepy
[3] https://github.com/colszowka/simplecov

that if the original source code passes the test cases, it is correct. This assumption will only be valid if the test suite contains sufficient thorough unit tests. If at least one of these test cases fails, we have found a bug which we must first resolve before continuing with the mutation analysis. When P successfully passes every test case, we evaluate every test case for each of the mutants. A mutant p' is said to be "killed" if its outcome is different from P for at least one test case. Otherwise, we refer to the mutant as "surviving". After we have finished executing the test cases on every mutant, we analyse the set of surviving mutants. Every mutant that managed to survive implies a change in the source code that did not trigger any failure in the test cases. As a result, we need to introduce subsequent test cases until we have killed every mutant. However, it is also possible that the surviving mutants are functionally equivalent to P and are, therefore, correct. Since the detection of program equivalence is an impossible problem, we need to verify this manually [22, 31]. Finally, note that although we can use mutation testing to estimate the adequacy of the test suite, it is not flawless, as several mutation operators can cancel each other out [30].

After every mutant has either been killed or marked equivalent to the original problem, we can calculate the *mutation score* of the test suite using eq. (2.1). In an adequate test suite, this score is equal to 1, which indicates that the test suite was able to detect every mutant instantly.

$$\text{Mutant Score} = \frac{\text{killed mutants}}{\text{non-equivalent mutants}} \tag{2.1}$$
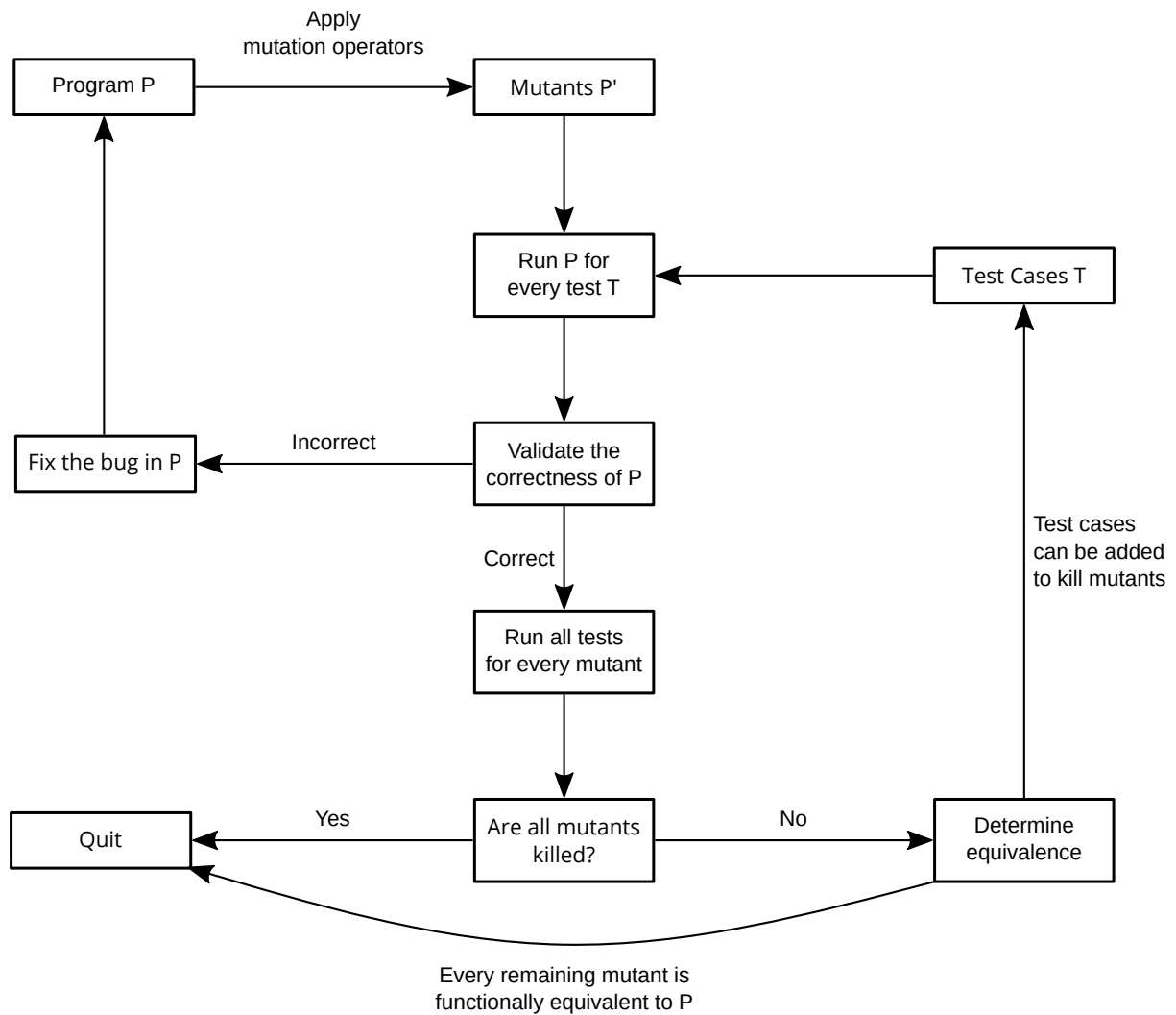
Figure 2.2: Process of Mutation Testing (based on [31]).

io.github.thepieterdc.http.impl

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
HttpClientImpl		59%		14%	7	14	18	40	2	9	0	1
HttpResponseImpl		55%		n/a	9	15	10	22	9	15	0	1
Total	88 of 211	58%	6 of 7	14%	16	29	28	62	11	24	0	2

(a) JaCoCo coverage report of https://github.com/thepieterdc/dodona-api-java

Coverage report: 75%

Module ↓	statements	missing	excluded	coverage
awesome/__init__.py	4	1	0	75%

```
1  def smile():
2      return ":)"
3
4  def frown():
5      return ":("
```

| **Total** | **4** | **1** | **0** | **75%** |

(b) coverage.py report of https://github.com/codecov/example-python

Helpers (88.41% covered at 22.84 hits/line)

12 files in total. 716 relevant lines. 633 lines covered and 83 lines missed

File	% covered	Lines	Relevant Lines	Lines covered	Lines missed	Avg. Hits / Line
app/helpers/standard_form_builder.rb	100.0 %	5	3	3	0	11.0
app/helpers/renderers/feedback_code_renderer.rb	100.0 %	25	16	16	0	5.4
app/helpers/institutions_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/api_tokens_controller_helper.rb	100.0 %	2	1	1	0	1.0
app/helpers/renderers/pythia_renderer.rb	93.94 %	290	165	155	10	3.6
app/helpers/renderers/feedback_table_renderer.rb	90.59 %	349	202	183	19	16.8
app/helpers/exercise_helper.rb	90.16 %	125	61	55	6	3.5
app/helpers/courses_helper.rb	86.67 %	36	15	13	2	28.4
app/helpers/repository_helper.rb	85.71 %	11	7	6	1	2.6
app/helpers/application_helper.rb	85.59 %	220	111	95	16	62.6
app/helpers/users_helper.rb	84.62 %	20	13	11	2	1.4
app/helpers/renderers/lcs_html_differ.rb	77.69 %	236	121	94	27	38.2
Showing 1 to 12 of 12 entries						

(c) simplecov report of https://github.com/dodona-edu/dodona

Figure 2.3: Statistics from Code coverage tools.

2.3   Agile Software Development

2.3.1   Agile Manifesto

Since the late 1990s, developers have tried to reduce the time occupied by the imple-
mentation and testing phases. As a result, several software pioneers have proposed
new implementations of the SDLC, which were later collectively referred to as the *Ag-
ile development methodologies*. This term was coined during a meeting of seventeen
prominent software developers, in which they have defined the following four funda-
mental values of Agile development in the *Agile Manifesto* [4].

1. *Individuals and interactions* over processes and tools.

2. *Working software* over comprehensive documentation.

3. *Customer collaboration* over contract negotiation.

4. *Responding to change* over following a plan.

According to the authors, we should interpret these values as follows: "While there
is value in the items on the right, we value the items on the left more" [4]. When we
examine these values more closely, we can observe that they all share a common phi-
losophy, which is that software engineering should be a fast process in which commu-
nication and a short feedback loop is critical to avoid missteps. Since 2001, a variety of
different programming models have arisen, each incorporating these agile principles
in their own way. The most remarkable new practice is Test-driven development (TDD).
Recall that an integration test is a black-box test and that as such, we can actually con-
struct the test case in advance and write the implementation afterwards. This concept
is also prevalent in TDD. This practice depicts that if we want to extend the functional-
ity of the application, we should first modify the test cases (or add new test cases) and
then modify the application code until every test case is passing [3].

2.3.2   The need for Agile

In the wake of the world economic crisis, software companies have been forced to
devote efforts into researching how they can reduce their overall expenses. This re-
search has concluded that in order to cut financial risks, developers should reduce the
time-to-market of their applications [18]. As a result of this, the agile methodologies
have received increased attention in scientific literature since this philosophy strives
to deliver a minimal version as soon as possible. Afterwards, we can incrementally
add additional features. This practice indeed results in a shorter time-to-market and
lower costs, since one can decide to cancel the project much earlier in the process.

Additionally, maintaining an agile workflow has also proven beneficial to the success rate of development. A study performed by The Standish Group revealed that the success rate of agile projects is more than three times higher compared to traditional methodologies (Figure 2.4).
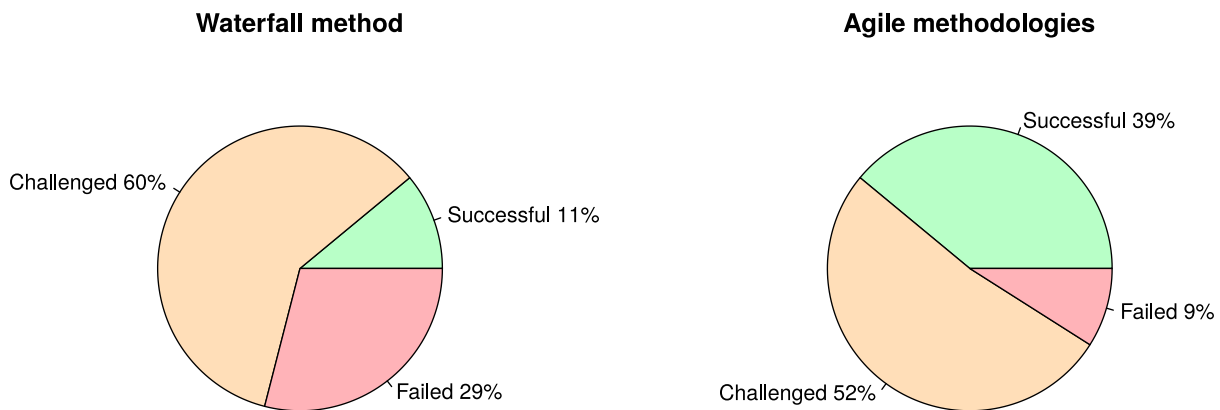
Waterfall method   **Agile methodologies**

Challenged 60%   Successful 11%   Successful 39%

                                              Failed 9%

Failed 29%   Challenged 52%

Figure 2.4: Success rate of Agile methodologies [16].

2.3.3 Continuous Integration

In traditional software development, the design phase usually leads to a representation of the required functionality in multiple, stand-alone modules. These modules are then implemented separately by the individual developers and are afterwards integrated into one monolithic application. This operation can prove to be very complicated since every developer can make their assumptions at the start of the project. Ultimately, these assumptions may render the components mutually incompatible. Furthermore, since it can take several weeks to months before this integration takes place, the developers often need to rewrite old code. Eventually, this will lead to unanticipated delays and costs [33].

Contrarily, agile development methodologies advocate the idea of frequent, yet small deliverables. In order to obtain frequent deliverables, we require to build the code often and integrate the modules multiple times, *continuously*, rather than just once at the end. Another advantage of frequent deliverables is the early identification of problems [14]. We refer to this practice as Continuous Integration (CI) [33]. Note that this idea has existed before the creation of the Agile manifesto. The first notorious software company that has adopted CI is Microsoft in 1989 [8]. Cusumano reports that Microsoft typically builds the entire application at least once per day and as such requires developers to integrate and test their changes multiple times per day.

The introduction of Continuous Integration in software development has significant consequences on the life cycle. Where the waterfall model used a cascading life cycle, CI employs a circular, repetitive structure consisting of three phases, as visualised in Figure 2.5.

1. **Implementation:** In the first phase, every developer writes code individually for their assigned module. At a regular interval, the code is committed to the remote repository.

2. **Integration:** When the developer commits their changes, they simultaneously fetch the changes other developers have made to their modules. Afterwards, the developer must integrate these remote changes with their local module to ensure these remain compatible. In case of a conflict, the developer is responsible for resolving this locally [27].

3. **Test:** After the developer has successfully integrated the remote modules to the locale module, the test suite must be executed to verify that no regressions have been introduced.
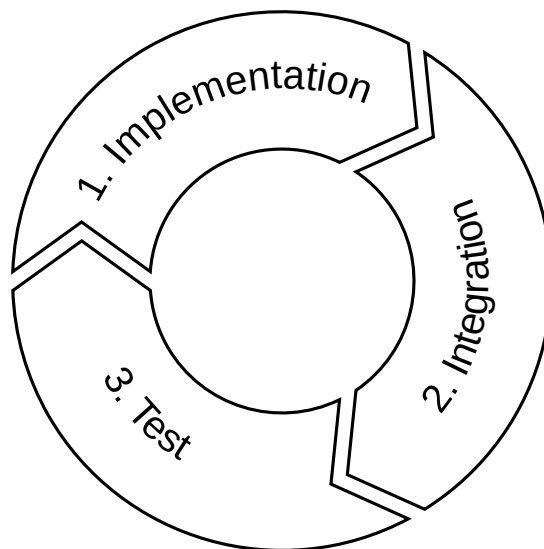
Figure 2.5: Development Life Cycle with Continuous Integration.

Adopting Continuous Integration can prove to be a lengthy and repetitive task. Luckily, a variety of tools and frameworks exist to automate this process. These tools are typically attached to a version control system (e.g. Git, Mercurial, . . .) using a `post-receive` hook [33]. Every time the developers push a commit to the repository, the hook will notify the CI system. The CI system will respond accordingly by automatically building the code and executing the test cases. Optionally, we can configure the CI system to automatically publish successful builds to the end-users, a process referred to as *Continuous Delivery*. This paper will now proceed by discussing four prominent CI systems.

Jenkins

Jenkins CI[4], "the leading open source automation server", was started in 2004 by Kohsuke Kawaguchi as a hobby project. Initially launched as Hudson, the project was later renamed to Jenkins due to trademark issues [33]. Jenkins is programmed in Java and is currently maintained by volunteers. As of today, Jenkins is still widely used for many reasons. Since it is open source and its source code is located on GitHub, it is free to use and can be self-hosted in a private environment. Furthermore, Jenkins provides an open ecosystem that encourages developers to create new plugins and extend its functionality. Market research conducted by ZeroTurnaround in 2016 revealed that Jenkins is the preferred CI tool by 60 % of the developers [25].

Figure 2.6: Logo of Jenkins CI (https://jenkins.io/).

GitHub Actions

Following the successful beta of GitHub Actions which had started in August 2019, GitHublaunched its Continuous Integration system later that year in November[5]. GitHub Actions executes builds in the cloud on servers owned by GitHub, restricting its use to GitHub repositories. Support for GitHub Enterprise repositories is not currently available. Developers can define builds using workflows, which can run both on Linux, Windows as well as macOS hosts. Private repositories are allowed a fixed amount of free build minutes per month, while builds of public repositories are always free of charges [11]. Similar to Jenkins, we can extend GitHub Actions with custom plugins. These plugins can be composed either using a Docker container or using native JavaScript [1]. However, since this service is recent, it does not yet offer many plugins.

Figure 2.7: Logo of GitHub Actions (https://github.com/features/actions).

---

[4]https://jenkins.io/
[5]https://github.blog/2019-08-08-github-actions-now-supports-ci-cd/

GitLab CI

GitLab, the main competitor to GitHub, announced its own Continuous Integration service in late 2012 named GitLab CI[6]. The build configuration is specified in a *pipeline* and is executed by *GitLab Runners*. Developers may host these runners by themselves, or use shared runners hosted by GitLab [2]. Equivalent to the previously mentioned GitHub Actions, shared runners can be used for free by public repositories and are bound by quota for private repositories [13]. A downside of using GitLab CI is the absence of a community-driven plugin system, but support for plugins is planned[7].

Figure 2.8: Logo of GitLab CI (https://gitlab.com/).

Travis CI

The final CI platform that will be discussed is Travis CI. This CI system was launched in 2011 and is only compatible with GitHub repositories. Travis CI build tasks are configured similarly to GitLab CI but lack support for self-hosted runners. In addition to commit-triggered builds, we can also schedule daily, weekly or monthly builds using *cronjobs*. Similar to GitHub Actions, open-source projects can use this service at zero cost, and a paid plan exists for private repositories [10]. It is not possible to create custom plugins, but Travis CI already features built-in support for a variety of programming languages. In 2020, almost one million projects are using Travis CI [34].

Figure 2.9: Logo of Travis CI (https://travis-ci.com/).

---

[6]https://about.gitlab.com/blog/2012/11/13/continuous-integration-server-from-gitlab/
[7]https://gitlab.com/gitlab-org/gitlab/issues/15067

Chapter 3

Related work

In the previous chapter, we have stressed the paramount importance of frequently integrating one's changes into the upstream repository. This process can prove to be a complex and lengthy operation. As a result, software engineers have sought and found ways to automate this task. These solutions and practices embody Continuous Integration (CI). However, CI is not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, we must execute the entire test suite to ensure that we have not introduced any regressions. As the project evolves and the size of the codebase increases, the number of test cases will increase accordingly to preserve a sufficiently high coverage level [30]. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a project consisting of 20 000 lines of code, whose test suite requires up to seven weeks to complete [35].

Fortunately, developers and researchers have found multiple techniques to address the scalability issues of ever-growing test suites. We can classify the techniques currently known in literature into three categories [30]. These categories are Test Suite Minimisation (TSM), Test Case Selection (TCS) or Test Case Prioritisation (TCP). We can apply each technique to every test suite, but the outcome will be different. TSM and TCS will have an impact on the execution time of the test suite, at the cost of a reduced test coverage level. In contrast, TCP will have a weaker impact on the execution time but will not affect the test adequacy.

The following sections will discuss these three approaches in more detail and provide accompanying algorithms. Because the techniques are very similar, the corresponding algorithms can (albeit with minor modifications) be used interchangeably for every approach. The final section of this chapter will investigate the adoption and integration of these techniques in modern software testing frameworks.

3.1   Classification of approaches

3.1.1   Test Suite Minimisation

The first technique is called Test Suite Minimisation (TSM), also referred to as *Test Suite Reduction* in literature. This technique will try to reduce the size of the test suite by permanently removing redundant test cases. This problem has been formally defined by Rothermel [37] in definition 1 and illustrated in Figure 3.1.

Definition 1 (Test Suite Minimisation)**.**
Given:

- $T = \{t_1, \ldots, t_n\}$ *a test suite consisting of test cases t_j.*

- $R = \{r_1, \ldots, r_m\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*

- $\{T_1, \ldots, T_m\}$ *subsets of test cases in T, one associated with each of the requirements r_i, such that any one of the test cases $t_j \in T_i$ can be used to satisfy requirement r_i.*

Subsequently, we can define Test Suite Minimisation as the task of finding a subset T' of test cases $t_j \in T$ that satisfies every requirement r_i.

If we apply the concepts of the previous chapter to the above definition, we can interpret the set of requirements R as source code lines that must be covered. A requirement r_i can subsequently be satisfied by any test case $t_j \in T$ that belongs to the subset T_i. Observe that the problem of finding T' is closely related to the *hitting set problem* (definition 2) [37].

Definition 2 (Hitting Set Problem)**.**
Given:

- $S = \{s_1, \ldots, s_n\}$ *a finite set of elements.*

- $C = \{c_1, \ldots, c_n\}$ *a collection of sets, with $\forall c_i \in C : c_i \subseteq S$.*

- K *a positive integer, $K \leq |S|$.*

The hitting set is a subset $S' \subseteq S$ such that S' contains at least one element from each subset in C.

In the context of Test Suite Minimisation, T' corresponds to the hitting set of T_is. In order to effectively minimise the amount of tests in the test suite, T' should be the minimal hitting set [37]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [12].
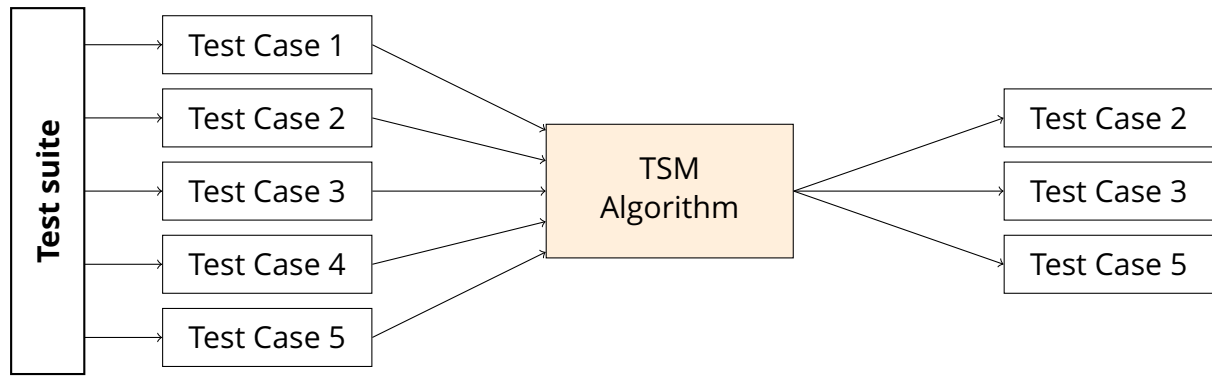
Figure 3.1: Test Suite Minimisation.

3.1.2 Test Case Selection

The second approach closely resembles the previous one. However, instead of permanently removing redundant test cases, Test Case Selection (TCS) has a notion of context. In this algorithm, we will not calculate the minimal hitting set at runtime, but before executing the test suite, we will perform a *white-box static analysis* of the source code. This analysis identifies which parts of the source code have been changed and executes only the corresponding test cases. Subsequent executions of the test suite will require a new analysis, thus making the selection temporary (Figure 3.2) and modification-aware [37]. Rothermel and Harrold define this formally in definition 3.

Definition 3 (Test Case Selection)**.**
Given:

- P *the previous version of the codebase*

- P' *the current (modified) version of the codebase*

- T *the test suite*

Test Case Selection aims to find a subset $T' \subseteq T$ that is used to test P'.
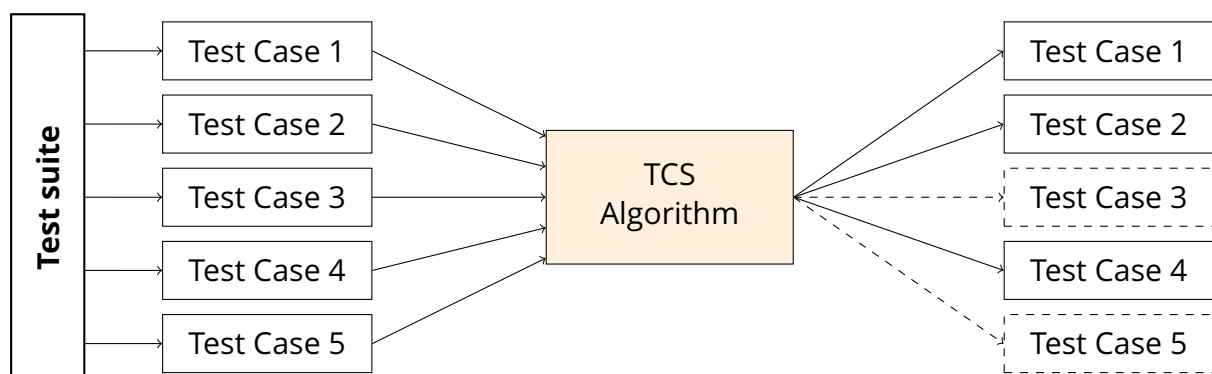

Figure 3.2: Test Case Selection.

3.1.3   Test Case Prioritisation

Both TSM and TCS attempt to execute as few tests as possible to reduce the execution time of the test suite. Nevertheless, in some cases, we may require to execute every test case to guarantee correctness. In this situation, we can still optimise the test suite. Test Case Prioritisation (TCP) aims to find a permutation of the sequence of test cases, rather than eliminating specific tests from being executed (Figure 3.3). We choose the order of the permutation in such a way that we can complete a predefined objective as soon as possible. Once we have achieved our objective, we can early terminate the execution of the test suite. In the worst-case scenario, we will still execute every test case. Some examples of objectives include covering as many lines of code as fast as possible or executing tests ordered on their probability of failure [37]. Definition 4 provides a formal definition of this approach.

Definition 4 (Test Case Prioritisation)**.**
Given:

- T *the test suite*

- PT *the set of permutations of* T

- $f : PT \mapsto \mathbb{R}$ *a function from a subset to a real number, this function is used to compare sequences of test cases to find the optimal permutation.*

Test Case Prioritisation finds a permutation $T' \in PT$ *such that* $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$
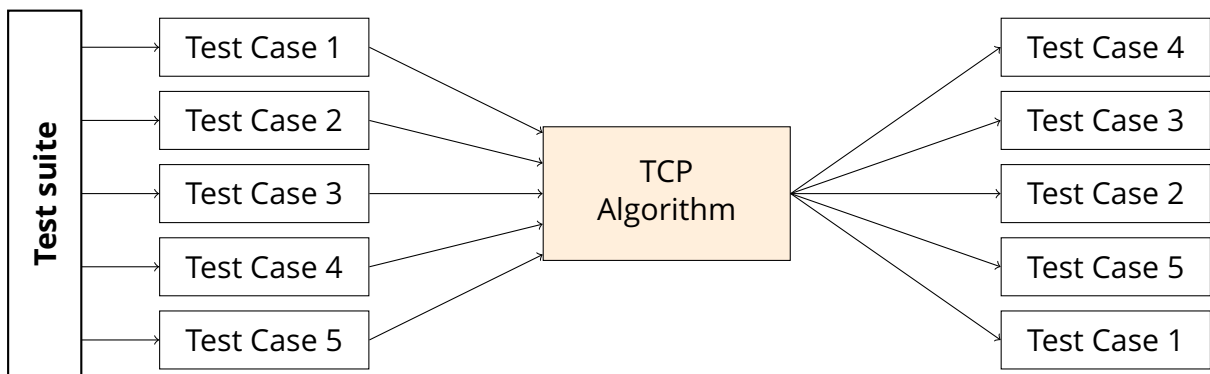

Figure 3.3: Test Case Prioritisation.

3.2 Algorithms

TCP is essentially an extended version of TSM since we can first execute the minimised test suite and afterwards the remaining test cases. Additionally, section 3.1.1 has explained that TSM is an instance of the minimal hitting set problem, which is an NP-complete problem. Consequently, we know that both TSM and TCP are NP-complete problems as well and therefore, we require the use of *heuristics*. A heuristic is an experience-based method that can be applied to solve a hard to compute problem by finding a fast approximation [19]. However, the found solution will mostly be sub-optimal, or sometimes the algorithm might even fail to find any solution at all. Given the relation between TSM and the minimal hitting set problem, we can implement an optimisation algorithm by modifying any known heuristic that finds the minimal hitting set. This paper will now proceed by discussing a selection of these heuristics. The used terminology and the names of the variables have been changed to ensure mutual consistency between the algorithms. Every algorithm has been adapted to adhere to the conventions provided in definitions 5 and 6.

Definition 5 (Naming convention)**.**

- $TS = \{T_1, \ldots, T_n\}$: *the set of all test cases t in the test suite.*

- $RS = \{T_1, \ldots, T_n\} \subseteq TS$: *the representative set of test cases t that have been selected by the algorithm.*

- $C = \{c_1, \ldots, c_m\}$: *the set of all source code lines in the application, that are covered by at least one test case $T \in TS$.*

- $CT = \begin{bmatrix} CT_1 & \ldots & CT_m \end{bmatrix}$: *the list of test groups.*

  - $CT_c = \{T_1, \ldots, T_n\} \subseteq TS$: *the test group c, which corresponds to the set of all test cases $T \in TS$ that cover the source code line $c \in C$.*

- $TL = \begin{bmatrix} TL_1 & \ldots & TL_n \end{bmatrix}$: *the list of coverage groups.*

  - $TL_t = \{c_1, \ldots, c_m\} \subseteq C$: *the set of all source code lines $c \in C$ that are covered by test case $t \in TS$.*

Definition 6 (Cardinality)**.** *For a finite set S, the cardinality $|S|$ is defined as the number of elements in S. In case of potential confusion, we can use $Card(S)$ to denote the cardinality of S.*

3.2.1   Greedy algorithm

The first algorithm is a *greedy* heuristic, which was initially designed by Chvatal to find an approximation for the set-covering problem [30]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [7]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The objective of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few test cases as possible.

Initially, the algorithm starts with an empty representative set RS, the set TS of all test cases and the set C of all coverable source code lines. Furthermore, TL denotes the set of coverage groups as specified in the definition. In essence, the algorithm will iteratively select test cases from TS and add them to RS. The locally optimal choice is always to select the test case that will contribute the most still uncovered lines, ergo the test case t for which the cardinality of the intersection between C and TL_t is maximal. After every iteration, we remove the code lines TL_t from C, since these are now covered. We repeat this selection process until C is empty, which indicates that we have covered every source code line. Afterwards, when we execute the test suite, we only need to execute test cases in RS. We can apply this algorithm to Test Case Prioritisation as well, by changing the type of RS to a list instead. We require a list to maintain the insertion order since this is equivalent to the ideal order of execution.

Algorithm 1 Greedy algorithm for Test Suite Minimisation.

Input: the test suite TS, all coverable lines C, the list of coverage groups TL
Output: representative set $RS \subseteq TS$ of test cases to execute

```
 1:  procedure GREEDYTSM(TS, C, TL)
 2:      RS ← ∅
 3:      while C ≠ ∅ do
 4:          t_max ← 0
 5:          tl_max ← ∅
 6:          for all t ∈ TS do
 7:              tl_current ← C ∩ TL[t]
 8:              if |tl_current| > |tl_max| then
 9:                  t_max ← t
10:                  tl_max ← tl_current
11:          RS ← RS ∪ {t_max}
12:          C ← C \ tl_max
13:      return RS
```

3.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [17]. Similar to the Greedy algorithm (section 3.2.1), this algorithm will also iteratively construct the minimal hitting set. However, instead of considering the coverage groups TL, the algorithm uses the test groups CT. More specifically, we will use the distinct test groups, denoted as CTD. We consider two test groups CT_i and CT_j as distinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

The algorithm consists of two main phases. The first phase begins by constructing an empty representative set RS in which we will store the selected test cases. Subsequently, we iterate over every source code line $c \in C$ to create the corresponding test groups CT. As mentioned before, we will reduce this set to CTD for performance reasons and as such, only retain the distinct test groups. Next, we select every test group of which the cardinality is equal to 1 and add these to RS. The representative set will now contain every test case that covers precisely one line of code, which is exclusively covered by that single test case. Afterwards, we remove every covered line from C. The next phase consists of repeating this process for increasing cardinalities until C is empty. However, since the test groups will now contain more than one test case, we need to make a choice on which test case to select. The authors prefer the test case that covers the most remaining lines. In the event of a tie, we defer the choice until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [17]. In addition to the naming convention introduced in definition 5, let n denote the number of distinct test groups CTD, nt the number of test cases $t \in TS$ and MAX_CARD the cardinality of the test group with the most test cases. In the HGS algorithm we need to perform two steps repeatedly. The first step involves computing the number of occurrences of every test case t in each test group. Given that there are n distinct test groups and, in the worst-case scenario, each test group can contain MAX_CARD test cases which we all need to examine once, the computational cost of this step is equal to $O(n * MAX\_CARD)$. For the next step, in order to determine which test case we should include in the representative set RS, we need to find all test cases for which the number of occurrences in all test groups is maximal, which requires at most $O(nt * MAX\_CARD)$. Since every repetition of these two steps adds a test case that belongs to at least one out of n test groups to the representative set, the overall runtime of the algorithm is $O(n*(n+nt)*MAX\_CARD)$.

Algorithm 2 HGS algorithm ([17]).

Input: distinct test groups CTD, total amount of test cases $nt = Card(TS)$
Output: representative set $RS \subseteq TS$ of test cases to execute
 1: **function** SELECTTEST($CTD, nt, MAX\_CARD, size, list, marked$)
 2:     $count \leftarrow array[1 \dots nt]$ ▷ initially 0
 3:     **for all** $t \in list$ **do**
 4:         **for all** $group \in CTD$ **do**
 5:             **if** $t \in group \land \neg marked[group] \land Card(group) = size$ **then**
 6:                 $count[t] \leftarrow count[t] + 1$
 7:     $max\_count \leftarrow$ MAX($count$)
 8:     $tests \leftarrow \{t | t \in list \land count[t] = max\_count\}$
 9:     **if** $|tests| = 1$ **then return** $tests[0]$
10:     **else if** $|tests| = MAX\_CARD$ **then return** $tests[0]$
11:     **else return** SELECTTEST($CTD, nt, MAX\_CARD, size + 1, tests, marked$)
12: **procedure** HGSTSM(CTD, nt)
13:     $n \leftarrow Card(CTD)$
14:     $marked \leftarrow array[1 \dots n]$ ▷ initially $false$
15:     $MAX\_CARD \leftarrow$ MAX($\{Card(group) | group \in CTD\}$)
16:     $RS \leftarrow \bigcup \{singleton | singleton \in CTD \land Card(singleton) = 1\}$
17:     **for all** $group \in CTD$ **do**
18:         **if** $group \cap RS \neq \emptyset$ **then**
19:             $marked[group] \leftarrow true$
20:     $current \leftarrow 1$
21:     **while** $current < MAX\_CARD$ **do**
22:         $current \leftarrow current + 1$
23:         $list \leftarrow \{t | t \in grp \land grp \in CTD \land Card(grp) = current \land \neg marked[grp]\}$
24:         **while** $list \neq \emptyset$ **do**
25:             $next \leftarrow$ SELECTTEST($current, list$)
26:             $reduce \leftarrow false$
27:             **for all** $group \in CTD$ **do**
28:                 **if** $next \in group$ **then**
29:                     $marked[group] = true$
30:                     **if** $Card(group) = MAX\_CARD$ **then**
31:                         $reduce \leftarrow true$
32:             **if** $reduce$ **then**
33:                 $MAX\_CARD \leftarrow$ MAX($\{Card(grp) | grp \in CTD \land \neg marked[grp]\}$)
34:             $RS \leftarrow RS \cup \{next\}$
35:             $list \leftarrow \{t | t \in grp \land grp \in CTD \land Card(grp) = current \land \neg marked[grp]\}$
36:     **return** RS

3.2.3 ROCKET algorithm

The third and final algorithm is the ROCKET algorithm. This algorithm has been presented by Marijan, Gotlieb and Sen [26] as part of a case study to improve the testing efficiency of industrial video conferencing software. Contrarily to the previous algorithms, which attempted to execute as few test cases as possible, this algorithm does execute the entire test suite. Unlike the previous algorithms that only take code coverage into account, this algorithm also considers historical failure data and test execution time. The objective of this algorithm is twofold: select the test cases with the highest successive failure rate, while also maximising the number of executed test cases in a limited time frame. In the implementation below, we will consider an infinite time frame as this is a domain-specific constraint and irrelevant for this thesis. This algorithm will yield a total ordering of all the test cases in the test suite, ordered using a weighted function.

The modified version of the algorithm (of which the pseudocode is provided in Algorithm 3) takes three inputs:

- $TS = \{T_1, \ldots, T_n\}$: the set of test cases to prioritise.

- $E = \begin{bmatrix} E_1 & \ldots & E_n \end{bmatrix}$: the execution time of each test case.

- $F = \begin{bmatrix} F_1 & \ldots & F_n \end{bmatrix}$: the failure statuses of each test case.

    - $F_t = \begin{bmatrix} f_1 & \ldots & f_m \end{bmatrix}$: the failure status of test case t over the previous m successive executions. $F_{ij} = 1$ if test case i has failed in execution $(current - j)$, 0 if it has passed.

The algorithm starts by creating an array P of length n, which contains the priority of each test case. The priority of each test case is initialised at zero. Next, we construct an $m \times n$ failure matrix MF and fill it using the following formula.

$$MF[i,j] = \begin{cases} 1 & \text{if } F_{ji} = 1 \\ -1 & \text{otherwise} \end{cases}$$

Table 3.1 contains an example of this matrix MF. In this table, we consider the hypothetical failure rates of the last two executions of six test cases.

run	T_1	T_2	T_3	T_4	T_5	T_6
$current - 1$	1	1	1	1	-1	-1
$current - 2$	-1	1	-1	-1	1	-1

Table 3.1: Example of the failure matrix MF.

Afterwards, we fill P with the cumulative priority of each test case. We can calculate the priority of a test case by multiplying its failure rate with a domain-specific weight heuristic ω. This heuristic reflects the probability of repeated failures of a test case, given earlier failures. In their paper [26], the authors apply the following weights:

$$\omega_i = \begin{cases} 0.7 & \text{if } i = 1 \\ 0.2 & \text{if } i = 2 \\ 0.1 & \text{if } i >= 3 \end{cases}$$

$$P_j = \sum_{i=1...m} MF[i, j] * \omega_i$$

Finally, the algorithm groups test cases based on their calculated priority in P. Every test case that belongs to the same group is equally relevant for execution in the current test run. However, within every test group, the test cases will differ in execution time E. The final step is to reorder test cases that belong to the same group in such a way that test cases with a shorter duration are executed earlier in the group.

Algorithm 3 ROCKET algorithm for Test Case Prioritisation.

Input: the test suite TS, the execution times of the test cases E, the amount of previous executions to consider m, the failure statuses F for each test case over the previous m executions

Output: priority P of the test cases

1: **procedure** ROCKETTCP(TS, E, m, F)
2:     $n \leftarrow Card(TS)$
3:     $P \leftarrow array[1 \ldots n]$ ▷ initially 0
4:     $MF \leftarrow array[1 \ldots m]$
5:     **for all** $i \in 1 \ldots m$ **do**
6:         $MF[i] \leftarrow array[1 \ldots n]$
7:         **for all** $j \in 1 \ldots n$ **do**
8:             **if** $F[j][i] = 1$ **then** $MF[i][j] \leftarrow -1$
9:             **else** $MF[i][j] \leftarrow 1$
10:     **for all** $j \in 1 \ldots n$ **do**
11:         **for all** $i \in 1 \ldots m$ **do**
12:             **if** $i = 1$ **then** $P[j] \leftarrow P[j] + (MF[i][j] * 0.7)$
13:             **else if** $i = 2$ **then** $P[j] \leftarrow P[j] + (MF[i][j] * 0.2)$
14:             **else** $P[j] + (MF[i][j] * 0.1)$
15:     $Q \leftarrow \{P[j] | j \in 1 \ldots n\}$ ▷ distinct priorities
16:     $G \leftarrow array[1 \ldots Card(Q)]$ ▷ initially empty sets
17:     **for all** $j \in 1 \ldots n$ **do**
18:         $p \leftarrow P[j]$
19:         $G[p] \leftarrow G[p] \cup \{j\}$
20:     Sort every group in G based on ascending execution time in E.
21:     Sort P according to which group it belongs and its position within that group.
22:     **return** P

3.3 Adoption in testing frameworks

In the final section of this chapter, we will investigate how existing software testing frameworks have implemented these and other optimisation techniques.

3.3.1 Gradle and JUnit

Gradle[1] is a dependency manager and development suite for Java, Groovy and Kotlin projects. It supports multiple plugins to automate tedious tasks, such as configuration management, testing and deploying. One of the supported testing integrations is JUnit[2], which is the most widely used testing framework by Java developers. JUnit 5 is the newest version which is still under active development as of today. Several prominent Java libraries and frameworks, such as Android and Spring have integrated JUnit as the preferred testing framework. The testing framework offers mediocre support for features that optimise the execution of the test suite, primarily when used in conjunction with Gradle. The following three key elements are available:

1. **Parallel test execution:** The Gradle implementation of JUnit features multiple *test class processors*. A test class processor is a component which processes Java classes to find all the test cases, and eventually to execute them. One of these processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. Concurrently executing the test cases results in a significant speed-up of the overall test suite execution.

2. **Prioritise failed test cases:** Gradle provides a second useful test class processor: the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run. This practice is similar to the ROCKET-algorithm (section 3.2.3), but the processor does not take into account the duration of the test cases.

3. **Test order specification:** JUnit allows us to specify the sequence in which it will execute the test cases. By default, it uses a random yet deterministic order[3]. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by applying the `@Order(int)`-annotation to an individual test case. However, we can only use this feature to alter the order of test cases within the same test class. JUnit does not support inter-test class reordering. We could use this feature to (locally) sort test cases based on their execution time.

---

[1] https://gradle.org
[2] https://junit.org
[3] https://junit.org/junit5/docs/current/user-guide/

3.3.2 Maven Surefire

A commonly used alternative to Gradle is Apache Maven[4]. This framework also supports executing JUnit test cases using the Surefire plugin. As opposed to Gradle, Surefire does offer multiple options to specify the order in which the test cases will be executed using the `runOrder` property. Without any configuration, Maven will run the test cases in alphabetical order. By switching the `runOrder` property to `failedFirst`, we can tell Maven to prioritise the previously failed test cases. Another supported value is `balanced`, which orders test cases based on their duration. Finally, we can choose to implement a custom ordering scheme for absolute control.

3.3.3 OpenClover

OpenClover[5] is a code coverage tool for Java and Groovy projects. It was created by Atlassian and open-sourced in 2017. OpenClover profiles itself as "the most sophisticated code coverage tool", by extracting useful metrics from the coverage results and by providing features that can optimise the test suite. These features include powerful integrations with development software and prominent Continuous Integration systems. Furthermore, OpenClover can automatically analyse the coverage results to detect relations between the application source code and the test cases. This feature allows OpenClover to predict which test cases will have been affected, given a set of modifications to the source code. Subsequently, we can interpret these predictions to implement Test Case Selection and therefore reduce the test suite execution time.

---

[4]http://maven.apache.org/
[5]https://openclover.org

Chapter 4

Proposed framework: VeloCIty

The implementation part of this thesis will provide a framework and a set of tools, tailored at optimising the test suite as well as providing accompanying metrics and insights. The framework was named VeloCIty to reflect its purpose of enhancing the efficiency and speed of Continuous Integration. This paper will now proceed by first describing the design goals of the framework, after which a high-level schematic overview of the implemented architecture will be provided. The architecture consists of a seven-step pipeline, divided into three individual components. These steps will be elaborated in more detail in section 4.3. Subsequently, the next section will present the *Alpha* algorithm as as a novel prioritisation algorithm, and this chapter will be concluded with an overview of the analytical features.

4.1 Design goals

VeloCIty has been implemented with four design goals in mind:

1. **Extensibility:** It should be possible and straightforward to support additional CI systems, programming languages and test frameworks. Similarly, a clear interface must be provided to integrate new prioritisation algorithms.

2. **Minimally invasive:** Integrating VeloCIty into an existing test suite should not require drastic changes to any of the test cases.

3. **Language agnosticism:** This design goal is related to the extensibility of the framework. The implemented tools should not need to be aware of the programming language of the source code, nor the used test framework.

4. **Self-improvement:** The prioritisation framework must support multiple algorithms. However, the performance of an algorithm might be dependent on the nature of the source code. An algorithm may offer a very high accuracy on one project but fall short on another. The framework should decide by itself which algorithm it should prefer, by measuring the performance of a prediction and subsequently infer which algorithm to use for future predictions.

4.2   Architecture

4.2.1   Agent

The first component that we will consider is the agent. The agent interacts directly with the source code and the test suite and is, therefore, the only component that is specific to the programming language and the test framework. Every programming language and test framework requires a different implementation of the agent, although these implementations are strongly related. This thesis provides a Java agent, which is available as a plugin for the Gradle and JUnit test framework, a combination which has previously been described in section 3.3.1. When the test suite is started, the plugin will contact the controller (section 4.2.2) to obtain the prioritised test case order and subsequently execute the test cases in that order. Afterwards, the plugin will send a feedback report to the controller, where it is analysed.

4.2.2   Controller

The second component is the core of the framework, acting as an intermediary between the agent on one side and the predictor (section 4.2.3) on the other side. In order to satisfy the second design goal and as such allow language agnosticism, the controller exposes a REST-interface, to which the agent can communicate using the `HTTP` protocol. On the other side, the controller does not communicate directly with the predictor but stores prediction requests in a shared database instead. The predictor will periodically poll this database and update the request with the predicted order. Besides providing routing functionality between the agent and the predictor, the controller is additionally responsible for updating the meta predictor (section 4.3.4) by evaluating the accuracy of earlier predictions.

4.2.3   Predictor and Metrics

The final component is the predictor. The predictor is responsible for applying the prioritisation algorithms to predict the optimal execution order of the test cases. This order is calculated by first executing ten prioritisation algorithms and subsequently consulting the meta predictor to determine the preferred sequence. The predictor has been implemented in Python, because of its accessibility and compatibility with various existing libraries such as NumPy[1] and TensorFlow[2], to allow advanced prioritisation algorithms.

---

[1] https://numpy.org/
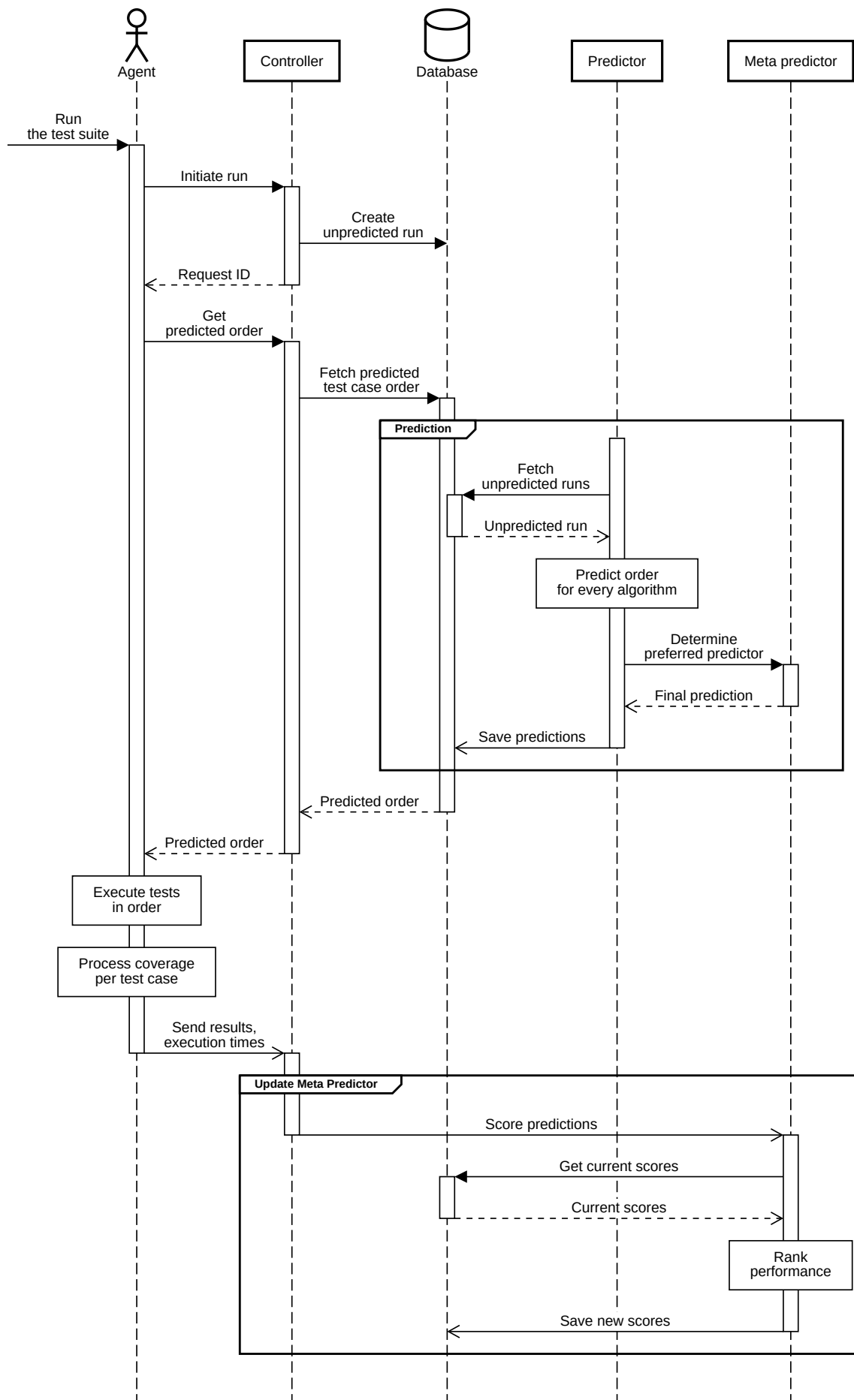[2] https://www.tensorflow.org/

Figure 4.1: Sequence diagram of VeloCIty.

4.3   Pipeline

This section will elaborate on the individual steps of the pipeline. We will review these steps by manually applying the pipeline on a hypothetical Java project. The very first prediction of a project will not use the meta predictor since this requires a previous prediction. Therefore, for the sake of simplicity, we will assume a steady-state situation, which implies that we have previously executed the prediction at least once.

4.3.1   Initialisation

In order to integrate VeloCIty in an existing Gradle project, we must modify the build script (`build.gradle`) in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, we must configure the following three properties:

- `base`: the path to the Java source files, relative to the location of the build script. This path will typically resemble `src/main/java`.

- `repository`: the URL to the git repository that hosts the project. This is required in subsequent steps of the pipeline, to detect which code lines have been changed in the commit currently being analysed.

- `server`: the URL to the controller.

Example 4.1 contains a minimal integration of the agent in a Gradle build script. The controller is hosted at the same machine as the agent and is reachable at port `8080`.

```
1  buildscript {
2      dependencies {
3          classpath 'io.github.thepieterdc.velocity:velocity−junit
               :0.0.1−SNAPSHOT'
4      }
5  }
6
7  apply plugin: 'velocity−junit'
8
9  velocity {
10     base 'src/main/java/'
11     repository 'https://github.com/user/my−java−project'
12     server 'http://localhost:8080'
13 }
```

Example 4.1: Minimal Gradle buildscript.

After we have configured the build script, we can execute the test suite. For the Gradle agent, this involves executing the `velocity` task, which commences the pipeline. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed CI system, this commit hash is available as an environment variable.

The first step in the pipeline is for the agent to initiate a new test run at the controller. The agent can accomplish this by sending a `POST`-request to the `/runs` endpoint of the controller, which will reply with an identifier. On the controller side, this request will enqueue a new prioritisation request in the database. At some point in time, the controller, which is continuously polling the database, will see the request and process it in the next step.

4.3.2 Prediction

The predictor is, as was mentioned before, continuously monitoring the database for unpredicted test runs. When a new test run is detected, the predictor will execute every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

AllInOrder The first algorithm will generate a straightforward execution sequence by ranking every test case alphabetically. We will mainly use this sequence for benchmarking purposes in chapter 5.

AllRandom We will use the second algorithm for benchmarking purposes as well. This algorithm will shuffle the list of test cases and return an arbitrary ranking.

AffectedRandom This algorithm is similar to the previous algorithm, but it will only consider the test cases that cover modified source code lines. These test cases will be ordered arbitrarily, followed by the other test cases in the test suite in no particular order.

GreedyCoverAll This algorithm is the first of three implementations of the Greedy algorithm (section 3.2.1). This variant will execute the standard algorithm on the entire test suite.

GreedyCoverAffected As opposed to the previous greedy algorithm, the second Greedy algorithm will only consider test cases covering changed source code lines. After it has prioritised these test cases, it will order the remaining test cases in the test suite randomly.

GreedyTimeAll   Instead of greedily attempting to cover as many lines of the source code using as few tests as possible, this implementation will attempt to execute as many tests as possible, as soon as possible. In other words, this algorithm will prioritise test cases based on their average execution time.

HGSAll   This algorithm is an implementation of the algorithm presented by Harrold, Gupta and Soffa (section 3.2.2). Similar to the `GreedyCoverAll` algorithm, this algorithm will prioritise every test case in the test suite.

HGSAffected   This algorithm is identical to the previous `HGSAll` algorithm, apart from the fact that it will only prioritise test cases covering changed source code lines.

ROCKET   The penultimate algorithm is a straightforward implementation of the pseudocode provided in section 3.2.3.

Alpha   The final algorithm is a custom algorithm inspired by the other implemented algorithms. Section 4.4 will further elaborate on the details.

Afterwards, the predictor will apply the meta predictor to determine the final prioritisation sequence. In its most primitive form, we can compare the meta predictor to a table which assigns a score to every algorithm. This score reflects the performance of the algorithm on this particular project. Section 4.3.4 will explain how this score is updated. Eventually, the meta predictor will elect the sequence of the algorithm with the highest score as the final prioritised order and persist this to the database.

4.3.3   Test case execution

While the predictor is determining the test execution order, the agent will poll the controller using the previously acquired identifier by sending a `GET` request to the /runs/id endpoint. Should the prioritisation order already be available, the controller will return this. One of the discussed features of Gradle in section 3.3.1 was the possibility to execute test cases in a chosen order by adding annotations. However, we cannot use this feature to implement the Java agent, since it only supports ordering test cases within the same test class. In order to allow complete control over the order of execution, we require a custom `TestProcessor` and `TestListener`.

A `TestProcessor` is responsible for processing every test class in the classpath and forwarding it, along with configurable options, to a delegate processor. The final processor in the chain will eventually perform the actual execution of the test cases. By default, the built-in processors will execute every test case in the test class it receives.

Every built-in processor will, by default, immediately execute every test case in the test class it processes. Since we want to execute test cases across test classes, the custom processor needs to work differently. The provided implementation of the agent will first load every received test class to obtain all test cases in the class using reflection. Afterwards, it will store every test case in a list and iterate this list in the prioritised order. For every test case t in the list, the custom processor will call the delegate processor with a tuple containing the test class and an array of options. This array will exclude every test case in the class except for t. This practice will forward the same test class to the delegate processor multiple times using a different option that restricts test execution to the chosen test case, resulting in the desired behaviour.

Furthermore, the agent calls a custom `TestListener` before and after every executed test case. This listener allows the agent to calculate the duration of the test case, as well as collect the intermediary coverage and save this on a per-test case basis.

4.3.4 Post-processing and analysis

The final step of the pipeline is to provide feedback to the controller to evaluate the accuracy of the predictions and thereby implementing the fourth design goal of self-improvement. After the agent has finished executing all the test cases, it will send the test results, the execution time and the coverage per test case to the controller by issuing a `POST` request to `/runs/id/test-results` and `/runs/id/coverage`.

When the controller receives this feedback information, it will update the meta predictor as follows. If every test case has passed, we do not update the meta predictor. The explanation for this choice is obvious. Since the objective of Test Case Prioritisation is to detect failures as fast as possible, every prioritised sequence is equally good if there are no failures at all. On the other hand, if a test case did fail, the meta predictor will inspect the predicted sequences. For every sequence, the meta predictor will calculate the duration until the first failed test case. Subsequently, it calculates the average of all these durations. Finally, the meta predictor will update the score of every algorithm by comparing its duration until the first failed test case to the average duration. For every algorithm that has a below-average duration, we increase the score and else decrease it. This process will eventually lead to the most accurate algorithms having a higher score, and these will, therefore, be preferred in following test runs.

4.4   Alpha algorithm

Besides the earlier presented Greedy, HGS and ROCKET algorithms (section 3.2), a custom algorithm has been implemented. The *Alpha* algorithm has been constructed by examining the individual strengths of the three preceding algorithms and subsequently combining their philosophies into a novel prioritisation algorithm. This paper will now proceed by providing its specification in accordance with the conventions described in definition 5. The corresponding pseudocode is listed in Algorithm 4.

The algorithm consumes the following inputs:

- $TS = \{T_1, \ldots, T_n\}$: the set of test cases to prioritise.

- $AS = \{T_1, \ldots, T_m\} \subseteq TS$: the set of *affected* test cases. A test case t is considered "affected" if t covers any modified source code line in the current commit.

- $C = \{c_1, \ldots, c_m\}$: the set of all source code lines in the application, that are covered by at least one test case $t \in TS$.

- $F = \begin{bmatrix} F_1 & \ldots & F_n \end{bmatrix}$: the failure statuses of each test case.

  - $F_t = \begin{bmatrix} f_1 & \ldots & f_m \end{bmatrix}$: the failure status of test case t over the previous m successive executions. $F_{ij} = 1$ if test case i has failed in execution $(current - j)$, 0 if it has passed.

- $D = \begin{bmatrix} D_1 & \ldots & D_n \end{bmatrix}$: the execution times of each test case.

  - $D_t = \begin{bmatrix} d_{t1} & \ldots & d_{tm} \end{bmatrix}$: the execution times of test case t. D_{ij} corresponds to the duration (in milliseconds) of test case i in execution $(current - j)$.

- $TL = \begin{bmatrix} TL_1 & \ldots & TL_n \end{bmatrix}$: the list of coverage groups.

  - $TL_t = \{c_1, \ldots, c_o\} \subseteq C$: the set of all source code lines $c \in C$ that are covered by test case $t \in TS$.

The algorithm begins by determining the execution time E_t of every test case t. To calculate the execution time, we distinguish two cases. If t has passed at least once, we calculate the execution time as the average duration of every successful execution of t. In the unlikely event that t has always failed, we compute the average over every execution of t. This distinction is mandatory, since a failed test case might have been aborted prematurely, which introduces a bias in the timings.

$$E_t = \begin{cases} \overline{\{D_{ti}|i \in [1 \ldots k], F_{ti} = 0\}} & \exists j \in [1 \ldots k], F_{tj} = 0 \\ \overline{\{D_{ti}|i \in [1 \ldots k]\}} & \text{otherwise} \end{cases}$$

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. Consecutive failures reflect the behaviour of a developer attempting to resolve the bug that caused the test case failure in the first run of the chain. By particularly executing the affected failing test cases as early as possible, we anticipate a developer that resolves multiple failures one by one. This idea is also used by the ROCKET algorithm (section 3.2.3). If multiple affected test cases are failing, the algorithm sorts those test cases by assigning a higher priority to the test case with the lowest execution time. After every prioritised test case, we update C by subtracting the source code lines that are now covered by that test case.

Afterwards, we repeat the same procedure for every failed (yet unaffected) test case and likewise use the execution time as a tie-breaker. Where the previous phase helps a developer to get fast feedback about whether or not they have resolved the issue, this phase ensures that the other failing test cases are executed early as well. Similar to the previous step, we again update C after every prioritised test case.

Research (section 5.4.1) has indicated that on average, a minor fraction ($10\,\% - 20\,\%$) of all test runs will contain a failed test case. As a result, the previous two phases will often not select any test case at all. Therefore, we will now focus on executing test cases that cover affected code lines. More specifically, the third phase of the algorithm will execute every affected test case, sorted by decreasing cardinality of the intersection between C and the test group of that test case. After every selected test case, we will update C as well. Since this phase uses C in the comparison, every selected test case can influence the next selected test case. The update process of C will ultimately lead to some affected test cases not strictly requiring to be executed, similar to the Greedy algorithm (section 3.2.1).

In the last phase, the algorithm will select the test cases based on the cardinality of the intersection between C and their test group. We repeat this process until C is empty and update C accordingly. When C is empty, the algorithm will yield the remaining test cases. Notice that these test cases will not contribute to the test coverage whatsoever since the previous iteration would already have selected every test case that would incur additional coverage. Subsequently, these test cases are de facto redundant and are therefore candidates for removal by TSM. However, since this is a prioritisation algorithm, these tests will still be executed and prioritised by increasing execution time.

Algorithm 4 Alpha algorithm for Test Case Prioritisation.

Input: the test suite TS, the affected test cases AS, all coverable lines C, the failure
    statuses F for each test case over the previous m executions, the execution times
    D for each test case over the previous m executions, the list of coverage groups TL

Output: ordered list P, sorted by descending priority

  1: **procedure** ALPHATCP(TS, AS, C, F, D, TL)
  2:     Construct E using D as described above.
  3:     $P \leftarrow array[1 \ldots n]$ ▷ initially 0
  4:     $i \leftarrow n$
  5:     $FTS \leftarrow \{t | t \in TS \wedge (F[t][1] = 1 \vee F[t][2] = 1 \vee F[t][3] = 1)\}$
  6:     $AFTS \leftarrow AS \cap FTS$
  7:     **for all** $t \in AFTS$ **do**   ▷ sorted by execution time in E (ascending)
  8:         $C \leftarrow C \setminus TL[t]$
  9:         $P[t] \leftarrow i$
10:         $i \leftarrow i - 1$
11:     $FTS \leftarrow FTS \setminus AFTS$
12:     **for all** $t \in FTS$ **do**   ▷ sorted by execution time in E (ascending)
13:         $C \leftarrow C \setminus TL[t]$
14:         $P[t] \leftarrow i$
15:         $i \leftarrow i - 1$
16:     $AS \leftarrow AS \setminus FTS$
17:     **while** $AS \neq \emptyset$ **do**
18:         $t\_max \leftarrow AS[1]$ ▷ any element from AS
19:         $tl\_max \leftarrow \emptyset$
20:         **for all** $t \in AS$ **do**
21:             $tl\_current \leftarrow C \cap TL_t$
22:             **if** $|tl\_current| > |tl\_max|$ **then**
23:                $t\_max \leftarrow t$
24:                $tl\_max \leftarrow tl\_current$
25:         $C \leftarrow C \setminus tl\_max$
26:         $P[t] \leftarrow i$
27:         $i \leftarrow i - 1$
28:     $TS \leftarrow TS \setminus (AS \cup FTS)$
29:     **while** $TS \neq \emptyset$ **do**
30:         $t\_max \leftarrow TS[1]$ ▷ any element from TS
31:         $tl\_max \leftarrow \emptyset$
32:         **for all** $t \in TS$ **do**
33:             $tl\_current \leftarrow C \cap TL_t$
34:             **if** $|tl\_current| > |tl\_max|$ **then**
35:                $t\_max \leftarrow t$
36:                $tl\_max \leftarrow tl\_current$
37:         $C \leftarrow C \setminus tl\_max$
38:         $P[t] \leftarrow i$
39:         $i \leftarrow i - 1$
40:     **return** P

4.5 Analysis

In this last section, we will take a look at the analytical features of the framework. Since the predictor already generates various statistics about the project which are required by the prioritisation algorithm, we can reuse these. The implementation of the analysis tool comprises a stand-alone version of the predictor daemon and supports the following six commands:

`affected`: The first command will determine which test cases have been affected by the changes in the given commit. This information is calculated based on the coverage information that the predictor has obtained from its last execution. Example 4.2 contains an example output of this command.

```
1  $ predictor affected https://github.com/author/project f5a23e0
2  FooTest.bar
3  FooTest.foo
```

Example 4.2: Output of the affected-command.

`durations`: The second command will compute the mean execution time of every test case in the repository and return the test cases from slowest to fastest.

```
1  $ predictor durations https://github.com/author/project
2  FooTest.foo: 200s
3  OtherBarTest.bar: 100s
```

Example 4.3: Output of the durations-command.

`failures`: Similar to the previous command, this command will determine the failure ratio of every test case in the repository. This ratio is equivalent to the number of failures, divided by the total amount of executions. Note that this denominator is not the same for every test case, since the test suite may be extended with new test cases. The output (Example 4.4) will list the test cases from the highest to the lowest failure rate.

```
1  $ predictor failures https://github.com/author/project
2  HelloWorldTest.hello: 25.00%
3  FooBarTest.bar: 10.00%
```

Example 4.4: Output of the failures-command.

`predict`:   This command allows the user to invoke the predictor by hand for the given test run. We can use this to test new algorithms, as opposed to the usual predictor daemon which does not support repeated predictions of the same test run. The result will contain the prioritised order as predicted by every available algorithm. An example output of this command is listed in Example 4.5.

```
1  $ predictor predict 1
2  HGS: [FooTest.bar, OtherBarTest.bar, HelloWorldTest.hello]
3  Alpha: [HelloWorldTest.hello, FooTest.bar, OtherBarTest.bar]
```

Example 4.5: Output of the predict-command.

`predictions`:   This command allows the user to retrieve historical prediction results. For deterministic algorithms, this will result in the same output as the previous command (which will rerun the algorithms). However, since some algorithms contain a random factor, we do require a separate command that fetches the prediction of the given run from the database.

```
1  $ predictor predictions 3
2  HGS: [FooTest.bar, OtherBarTest.bar, HelloWorldTest.hello]
3  AllRandom: [HelloWorldTest.hello, OtherBarTest.bar, FooTest.bar]
```

Example 4.6: Output of the predictions-command.

`scores`:   The final command yields the current score of every prediction algorithm in the meta predictor table, for the given project. The predictor will always return the test sequence that has been predicted by the algorithm with the current highest score. In Example 4.7 below, this would be the ROCKET algorithm.

```
1  $ predictor scores https://github.com/author/project
2  AllInOrder: −3
3  ROCKET: 7
4  GreedyCoverAffected: 4
```

Example 4.7: Output of the scores-command.

Chapter 5

Evaluation

This chapter will evaluate the performance of the framework presented in the previous chapter. The first section introduces the two test subjects that will be used in subsequent experiments. The next section will restate the research questions formally and extend these. Afterwards, we will elaborate on the procedure of the data collection. The final section will provide answers to the research questions as well as present the results of applying Test Case Prioritisation to the test subjects.

5.1 Test subjects

5.1.1 Dodona

Dodona[1] is an open-source online learning environment created by Ghent University, which allows students from secondary schools and universities in Belgium and South-Korea to submit solutions to programming exercises and receive instant, automated feedback. The application is built on top of the Ruby-on-Rails web framework. To automate the testing process of the application, Dodona employs GitHub Actions (section 2.3.3) which executes the more than 450 test cases in the test suite and performs static code analysis afterwards. The application is tested using the default `MiniTest` testing framework and `SimpleCov`[2] is used to record the coverage of the test suite. Currently, the coverage ratio is approximately 89%. This analysis will consider builds between January 1 and May 17, 2020.

5.1.2 Stratego

The second test subject has been created for the Software Engineering Lab 2 course at Ghent University in 2018. The application was created for a Belgian gas transmission system operator and consists of two main components: a web frontend and a backend. This thesis will test the backend in particular since it is written in Java using the Spring framework. Furthermore, the application uses Gradle and JUnit to execute the $300 - 400$ test cases in the test suite, allowing the Java agent (section 4.2.1) to be applied directly.

---

[1] https://dodona.ugent.be/
[2] https://github.com/colszowka/simplecov

5.2   Research questions

We will answer the following research questions in the subsequent sections:

RQ1: What is the probability that a test run will contain at least one failed test case?   The first research question will provide useful insights into whether a typical test run tends to fail or not. The expectancy is that the probability of failure will be rather low, indicating that it is not strictly necessary to execute every test case and therefore making a case for Test Suite Minimisation.

RQ2: What is the average duration of a test run?   Measuring how long it takes to execute a typical test run is required to estimate the benefit of applying any form of test suite optimisation. We will only consider successful test runs, to reduce bias introduced by prematurely aborting the execution.

RQ3: Suppose that a test run has failed, what is the probability that the next run will fail as well?   The ROCKET algorithm (section 3.2.3) relies on the assumption that if a test case has failed in a given test run, it is likely to fail in the subsequent run as well. This research question will investigate the correctness of this hypothesis.

RQ4: How can Test Case Prioritisation be applied to Dodona and what is the resulting performance benefit?   This research question will investigate the possibility to apply the VeloCIty framework to the Dodona project and analyse how quickly the available predictors can discover a failing test case.

RQ5: Can the Java agent be applied to Stratego?   Since the testing framework used by Stratego should be supported natively by the Java agent, this research question will verify its compatibility. Furthermore, we will analyse the prediction performance, albeit with a small number of relevant test runs.

5.3   Data collection

5.3.1   Travis CI build data

We can answer the first three research questions by analysing data from projects hosted on Travis CI (section 2.3.3). This data has been obtained from two sources.

The first source comprises a database [10] of 35 793 144 log files of executed test runs, which has been contributed by Durieux et al. The magnitude of the dataset (61.11 GiB)

requires a big data approach to parse these log files. Two straightforward MapReduce pipelines (Figure 5.1) have been created using the Apache Spark[3] engine, to provide an answer to the first and second research question.
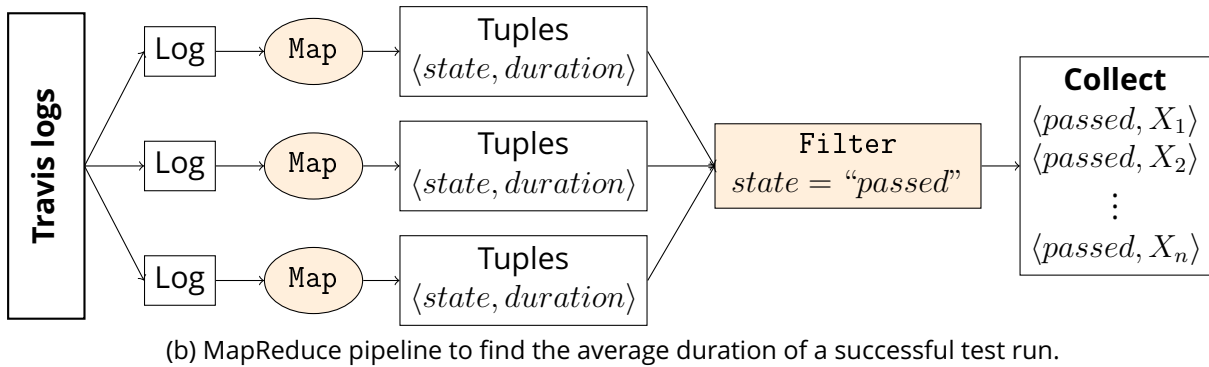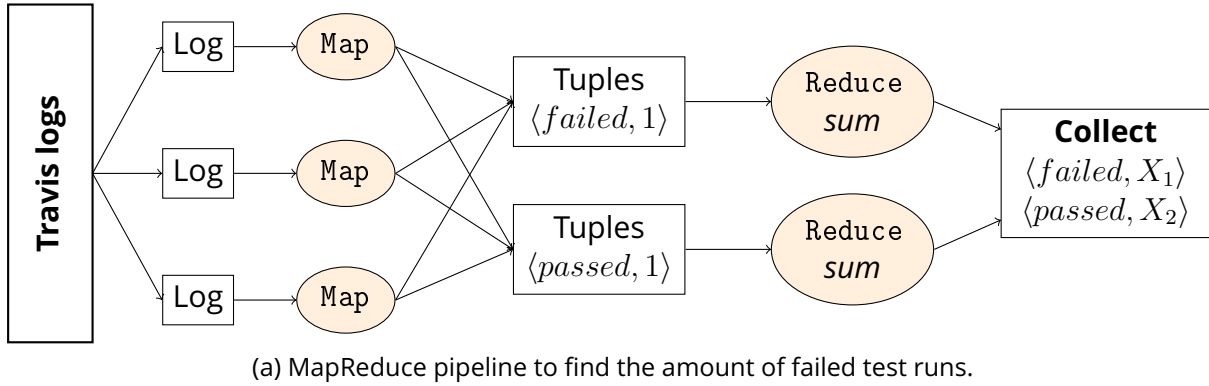

(a) MapReduce pipeline to find the amount of failed test runs.

(b) MapReduce pipeline to find the average duration of a successful test run.

Figure 5.1: MapReduce pipelines for Travis CI data.

In addition to the first source, another $3\,702\,595$ jobs have been analysed from the *TravisTorrent* project [5]. To identify which projects are using Travis CI, the authors have crawled the GitHub API and examined the build status of every commit to retrieve the run identifier. Subsequently, the Travis CI API is used to obtain information about both the build, as well as the project itself. This information includes the programming language, the amount of source code lines and the amount of failed test cases. The latter value provides an accurate answer to the first research question since it indicates why the test run has failed. Without this information, the test suite might have failed to compile as opposed to an actual failure in the test cases. Furthermore, the dataset includes the identifier of the previously executed test run, which we can use to answer the third research question. Additionally, the information contains the build duration. This dataset has been excluded from the second research question however, as the included execution time does not correspond to the actual duration reported on the webpage of Travis CI. The authors have provided a Google BigQuery[4] interface to allow querying the dataset more efficiently. Appendix A contains the executed queries.

---

[3]https://spark.apache.org/
[4]https://bigquery.cloud.google.com/

5.3.2   Dodona data

As mentioned before, Dodona utilises the MiniTest testing framework in conjunction with SimpleCov to calculate the coverage. MiniTest will by default only emit the name of every failed test case, without any further information. Furthermore, SimpleCov can only calculate the coverage for the entire test suite and does not allow us to retrieve the coverage on a per-test basis. To answer the fourth research question and apply the VeloCIty predictors to Dodona, a Python script has been created to reconstruct the conditions of every failed test run. The script first queries the API of GitHub Actions to find which test runs have failed. This thesis will consider 120 failed runs. For every failed commit, the script retrieves the parent commit and calculates the coverage on a per-test basis. This thesis will assume that the coverage of the parent commit resembles the coverage of the failed commit. The coverage is calculated by applying the following two transformations to the parent commits and subsequently rescheduling these in GitHub Actions:

- **Cobertura formatter:** The current SimpleCov reports can only be generated as HTML reports, preventing convenient analysis. We can resolve this problem by using the Cobertura formatter instead, which generates XML reports. The controller already supports the structure of these reports, as this formatter is commonly used by Java testing frameworks as well.

- **Parallel execution:** The Dodona test suite currently executes the test cases by four processes concurrently, to reduce the execution time. Every process individually records the code coverage, and at the end of the test suite, SimpleCov merges these separate reports into one. However, this process is not entirely thread-safe since the test suite requires shared resources. We do not require thread-safety to calculate the total coverage, but we do require this to generate the coverage on a per-test basis. As a result, parallel execution has been disabled in these experiments.

5.3.3   Stratego data

To integrate VeloCIty with the existing Stratego codebase, we can use the instructions described in chapter 4. Afterwards, to analyse the prediction performance, we can take an approach similar to the previous test subject. The GitHub API has been used to identify the failed commits and to find their parent (successful) commits. The parent commits have subsequently been modified to use the VeloCIty Java agent and have been executed using GitHub Actions.

5.4 Results

5.4.1 RQ1: Probability of failure

The two pie charts in Figure 5.2 illustrate the amount of failed and successful test runs. The leftmost chart visualises the failure rate in the dataset [10] by Durieux et al. 4 558 279 test runs out of the 28 882 003 total runs have failed, which corresponds to a failure probability of 18.74 %. The other pie chart uses data from the TravisTorrent [5] project. Since we can infer the cause of failure from this dataset, it is possible to obtain more accurate results. 42.89 % of the failed runs are due to a compilation failure where the test suite did not execute. For the remaining part of the runs, 225 766 out of 2 114 920 executions contain at least one failed test case, corresponding to a failure percentage of 10.67 %.
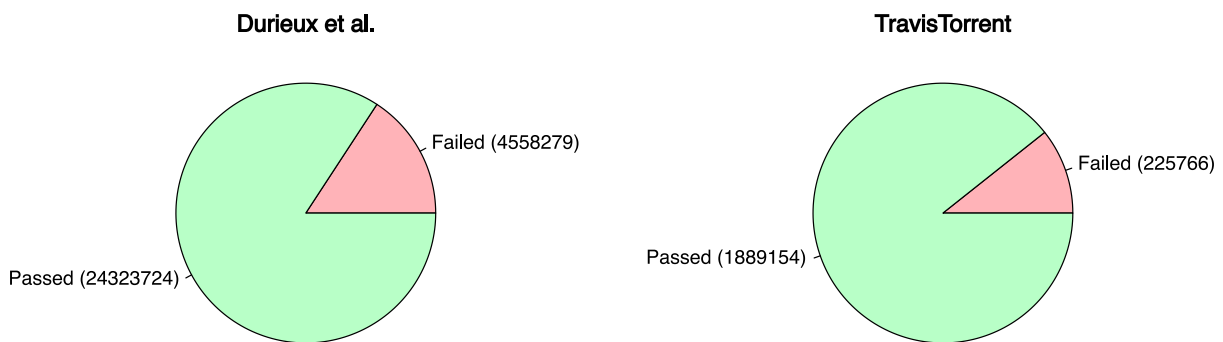
Figure 5.2: Probability of test run failure.

5.4.2 RQ2: Average duration of a test run

The dataset by Durieux et al. [10] has been refined to only include test runs that did not finish within 10 s. A lower execution time generally indicates that the test suite did not execute and that a compilation failure has occurred instead. Table 5.1 contains the characteristics of the remaining 24 320 504 analysed test runs. The median and average execution times suggest that primarily small projects are Travis CI, yet the maximum value is very high. Figure 5.3 confirms that 71 378 test runs have taken longer than one hour to execute. Further investigation has revealed that these are typically projects which are using mutation testing, such as `plexus/yaks`[5].

# runs	Minimum	Mean	Median	Maximum
24 320 504	10 s	385 s	178 s	26 h 11 min 26 s

Table 5.1: Characteristics of the test run durations in [10].

---

[5]A Ruby library for hypermedia (`https://github.com/plexus/yaks`).
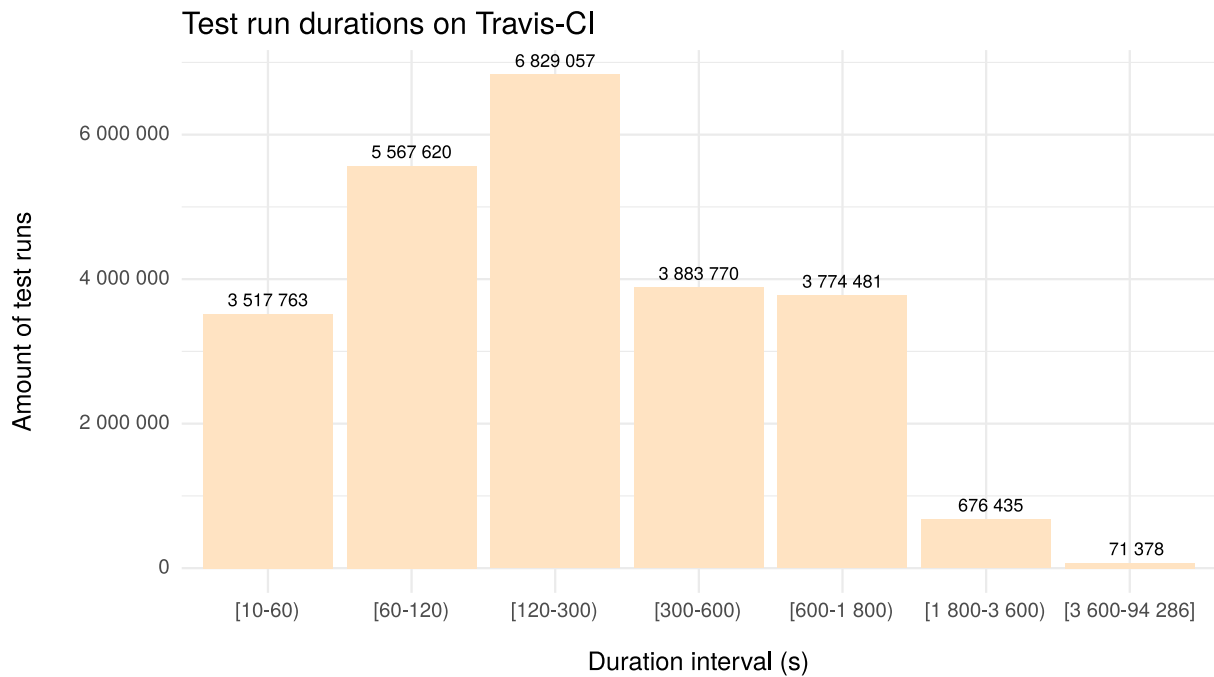
Figure 5.3: Test run durations on Travis CI.

5.4.3 RQ3: Consecutive failure probability

Because the TravisTorrent project is the only dataset that contains the identifier of the previous run, only runs from this project have been used. This dataset consists of 211 040 test runs, immediately following a failed execution. As illustrated in Figure 5.4, 109 224 of these test runs have failed as well, versus 101 816 successful test runs (51.76 %).
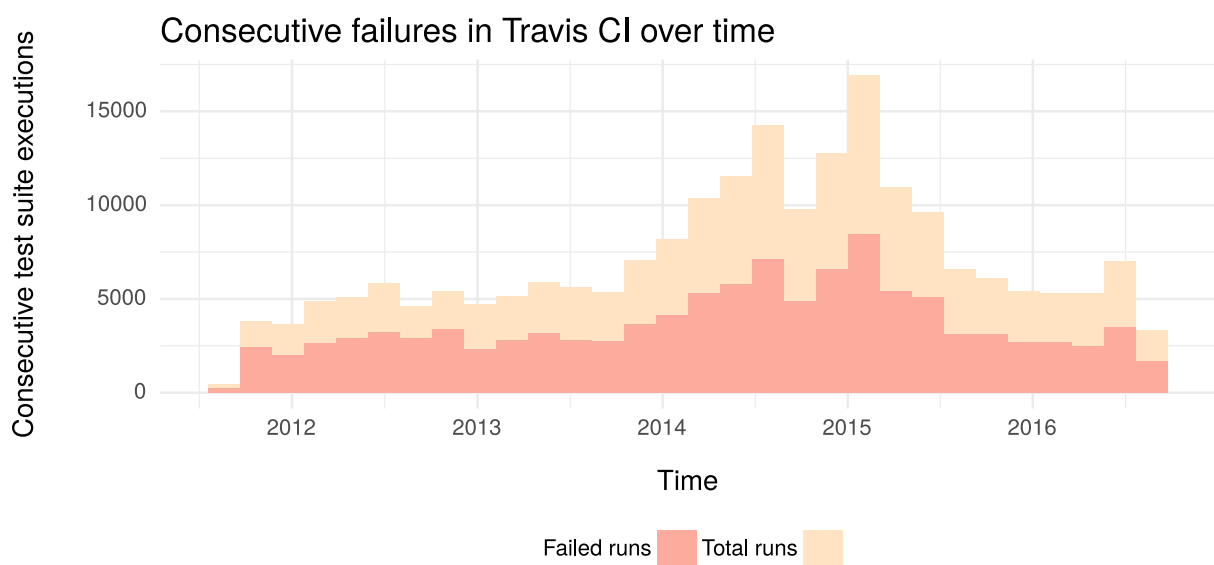

Figure 5.4: Consecutive test run failures on Travis CI.

5.4.4 RQ4: Applying Test Case Prioritisation to Dodona

After executing the 120 failed test runs, the log files have been inspected. These log files have revealed that an error in the configuration was the actual culprit of five failed test runs, rather than a failed test case. These test runs have therefore been omitted from the results because the test suite did not execute. Since configuration-related problems require in-depth contextual information about the project, we cannot automatically predict these.

Table 5.2 contains the amount of executed test cases until we observe the first failure. These results indicate that every predictor is capable of performing at least one successful prediction. Furthermore, the maximum amount of executed test cases is lower than the original value, which means that every algorithm is a valid predictor. The data suggests that the Alpha algorithm and the HGS algorithm are the preferred predictors for the Dodona project. In contrast, the performance of the ROCKET algorithm is rather low.

Algorithm	Minimum	Mean	Median	Maximum
Original	0	155	78	563
Alpha	0	8	3	73
AffectedRandom	0	54	10	428
AllInOrder	0	119	82	460
AllRandom	0	90	27	473
GreedyCoverAffected	0	227	246	494
GreedyCoverAll	0	98	33	514
GreedyTimeAll	0	210	172	482
HGSAffected	0	61	10	511
HGSAll	0	124	54	507
ROCKET	0	210	170	482

Table 5.2: Amount of executed test cases until the first failure.

The previous results have been visualised in Figure 5.5. These charts confirm the low accuracy of the ROCKET algorithm. The Alpha algorithm and the HGS algorithm offer the most accurate predictions, with the former algorithm being the most consistent. Notice the chart of the Greedy algorithm, which succeeds in successfully predicting some of the test runs, while failing to predict others. This behaviour is specific to a greedy heuristic.

(a) Alpha algorithm

(b) Greedy algorithm
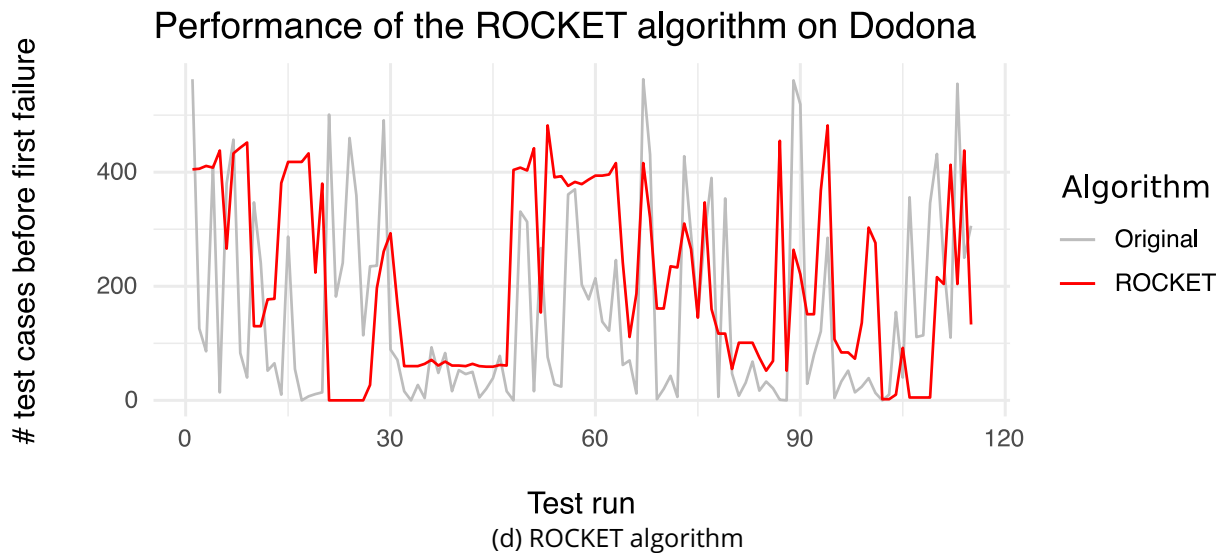

(c) HGS algorithm
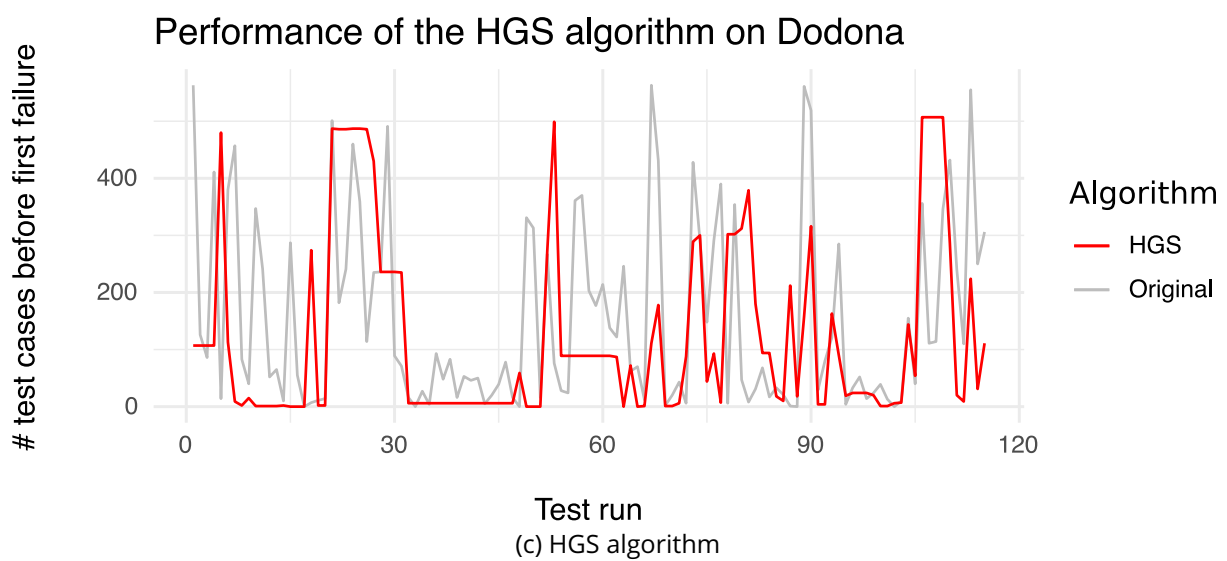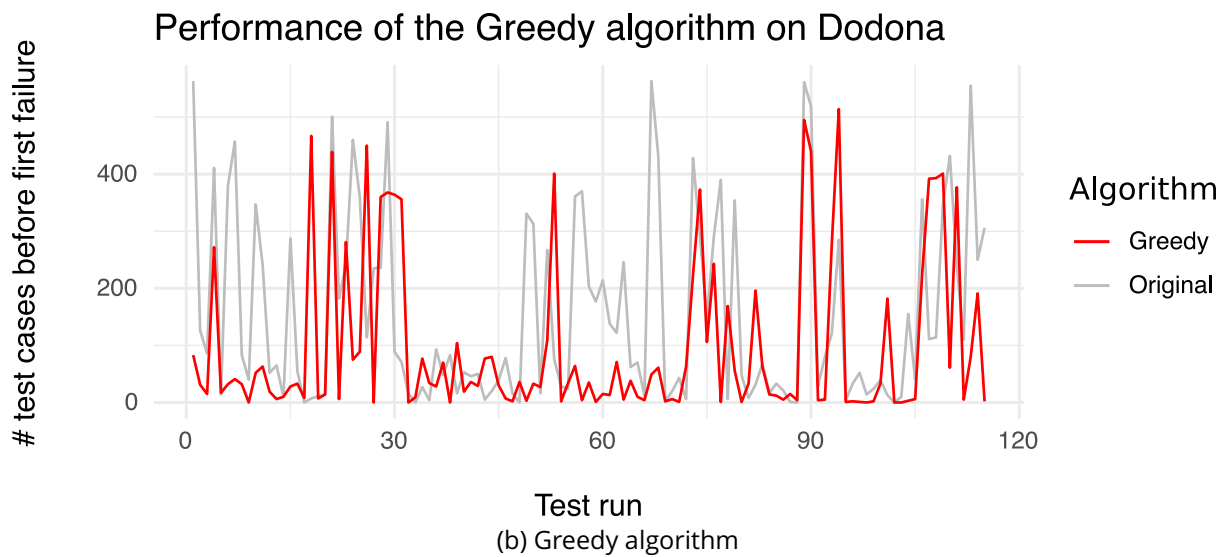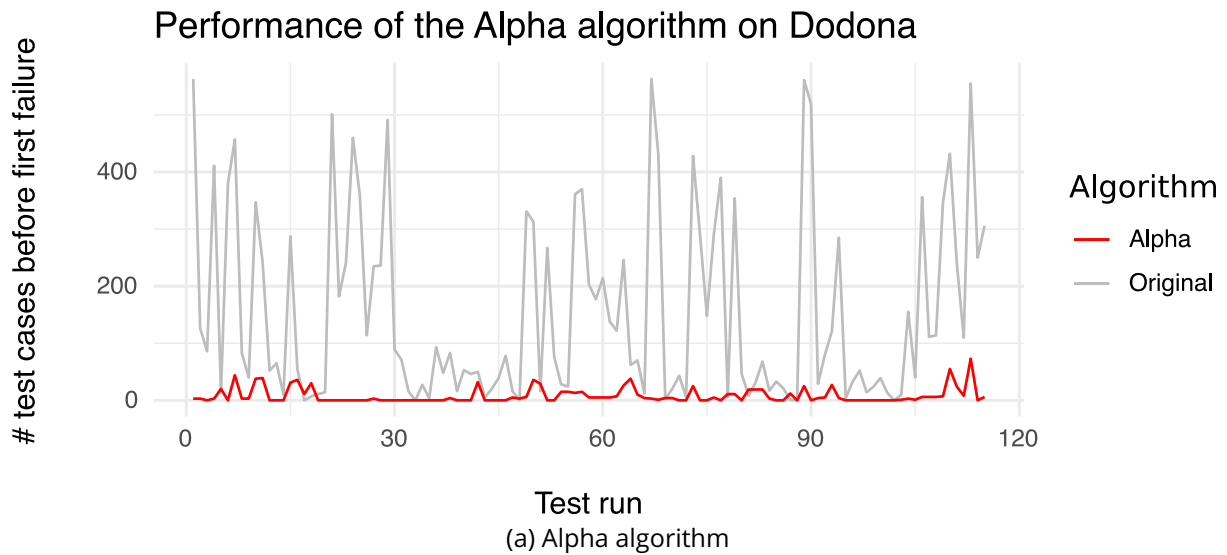
(d) ROCKET algorithm

Figure 5.5: Prediction performance on the Dodona project.

The duration until the first observed failure is reported in Table 5.3. Observe that the previous table indicates that the ROCKET algorithm does not perform well, while this table suggests otherwise. We can explain this behaviour by examining the objective function of this algorithm. This function prioritises cases with a low execution time to be executed first.

Algorithm	Minimum	Mean	Median	Maximum
Original	0 s	135 s	123 s	380 s
Alpha	0 s	3 s	1 s	33 s
AffectedRandom	0 s	28 s	5 s	190 s
AllInOrder	0 s	82 s	71 s	270 s
AllRandom	0 s	43 s	11 s	270 s
GreedyCoverAffected	0 s	88 s	86 s	314 s
GreedyCoverAll	0 s	46 s	12 s	280 s
GreedyTimeAll	0 s	55 s	32 s	175 s
HGSAffected	0 s	35 s	6 s	356 s
HGSAll	0 s	75 s	34 s	377 s
ROCKET	0 s	54 s	32 s	175 s

Table 5.3: Duration until the first failure for the Dodona project.

5.4.5 RQ5: Integrate VeloCIty with Stratego

The data collection phase has already proven that the Java agent is compatible with Stratego. Since VeloCIty is not yet able to predict test cases which have been added in the current commit, we can only use 35 of the 54 failed test runs.

Similar to the previous test subject, Table 5.4 lists how many test cases have been executed before the first observed failure. The table only considers the four main algorithms, since the actual prediction performance was only secondary to this research question, and we have only analysed a small number of test runs. The results suggest that every algorithm except the ROCKET achieves a high prediction accuracy on this project.

Algorithm	Minimum	Mean	Median	Maximum
Original	0	68	2	278
Alpha	0	10	2	57
GreedyCoverAll	0	11	3	57
HGSAll	0	9	4	50
ROCKET	0	42	27	216

Table 5.4: Amount of executed test cases until the first failure.

Even though the performance of the ROCKET algorithm is suboptimal in the previous table, Table 5.5 does indicate that it outperforms every other algorithm time-wise. Notice that the predicted sequence of the HGS algorithm takes the longest to execute, while the previous table suggested a good prediction accuracy. The Alpha and Greedy algorithms seem very similar on both the amount of executed test cases, as well as the execution time.

Algorithm	Minimum	Mean	Median	Maximum
Original	0 s	62 s	8 s	233 s
Alpha	0 s	11 s	2 s	103 s
GreedyCoverAll	0 s	12 s	2 s	103 s
HGSAll	0 s	19 s	1 s	130 s
ROCKET	0 s	6 s	0 s	85 s

Table 5.5: Amount of executed test cases until the first failure.

Figure 5.6 further confirms the above statements. Notice the close resemblance of the charts of the Greedy algorithm and the Alpha algorithm, which indicates that a different failing test case is the cause of every test run failure. The ROCKET algorithm performs better on this project than on Dodona, yet not accurate.
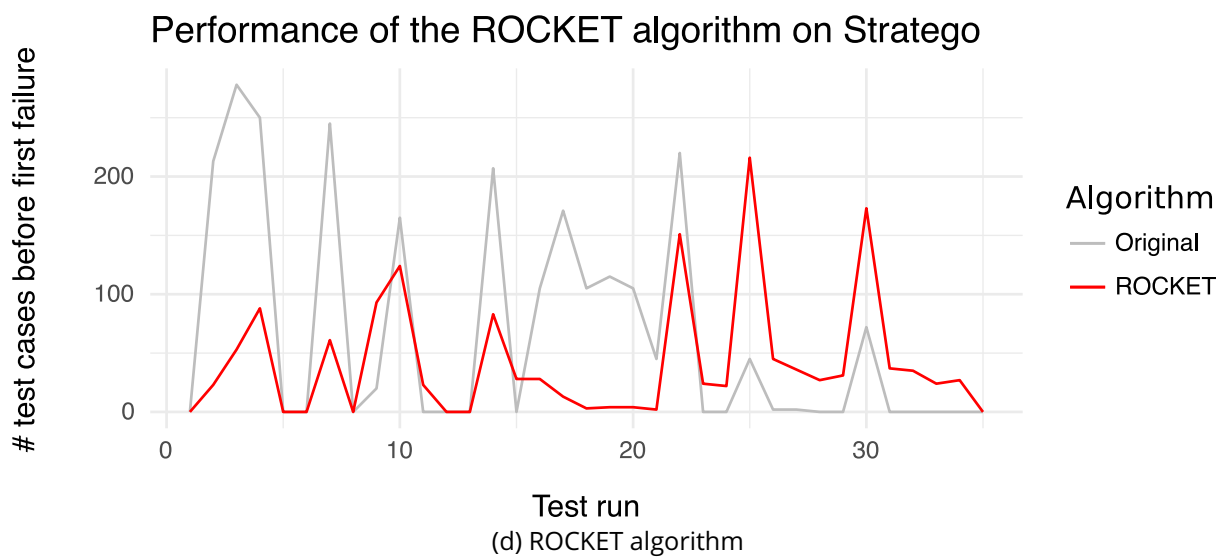
Performance of the Alpha algorithm on Stratego

(a) Alpha algorithm

Performance of the Greedy algorithm on Stratego

(b) Greedy algorithm

Performance of the HGS algorithm on Stratego

(c) HGS algorithm

(d) ROCKET algorithm
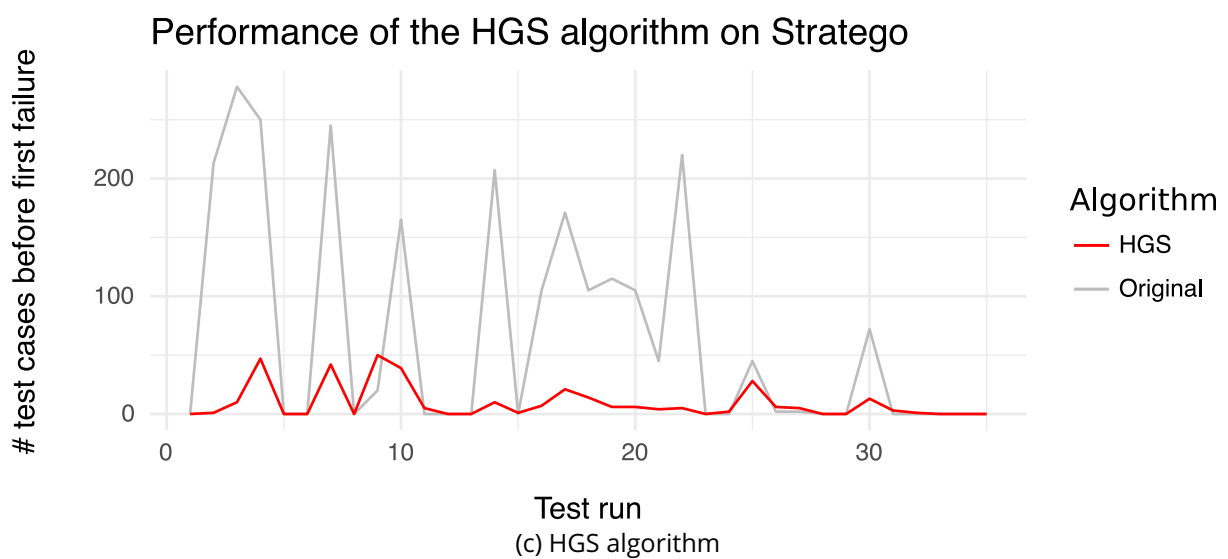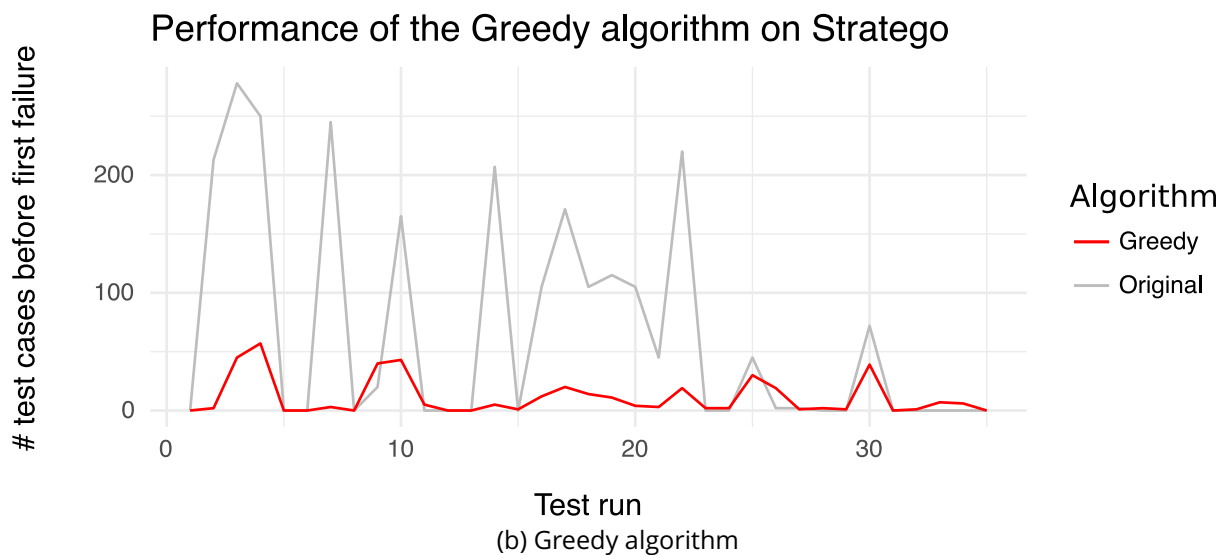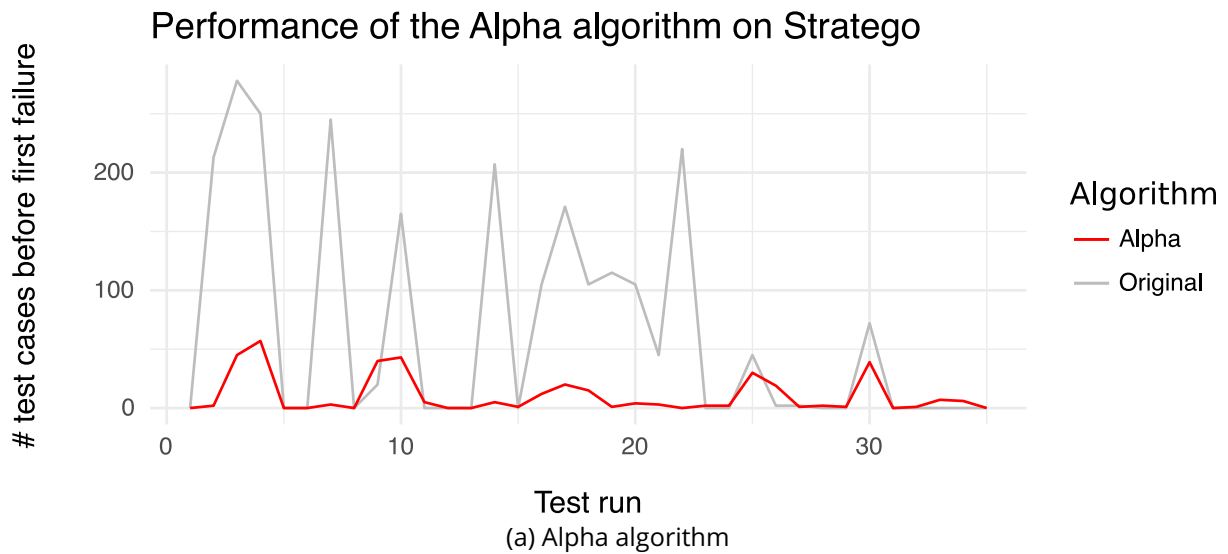
Figure 5.6: Prediction performance on the Stratego project.

Chapter 6

Conclusion

The main objective of this thesis was to study different approaches to optimise the test suite of a typical software project. Three approaches have been introduced to this extend: Test Suite Minimisation, Test Case Selection and Test Case Prioritisation. We have successfully implemented the latter approach using the VeloCIty framework. Furthermore, this framework features the Alpha algorithm as a novel prioritisation algorithm. The performance of the Alpha algorithm has been evaluated, mainly on the UGent Dodona project. The results are promising, resulting in $95\,\%$ fewer executed test cases and $97\,\%$ less time spent waiting for the first test case to fail.

The second purpose of this thesis was to gain useful insights into the characteristics of a regular test suite, formalised into three research questions. The first question was to estimate the expected failure probability of a test run. To answer this question, we have analysed more than 28 million test runs on Travis-CI. This analysis has indicated that $18\,\%$ of those test runs have failed. Additionally, we have used this dataset to answer another question, which was to determine the typical duration of a test run. Statistical analysis has revealed that developers mainly use Travis-CI for small projects, with an average test suite duration of seven minutes. $0.20\,\%$ of the test suites take longer than one hour to execute, and some projects use mutation testing. The final question was to examine the probability of consecutive failing test runs. This probability was estimated at $52\,\%$ using a second Travis-CI dataset from the TravisTorrent project[5].

6.1   Future work

The proposed architecture currently features a Java agent, which supports the prediction of Gradle projects using ten available predictors. However, there is still room for improvements. The paragraphs below will suggest some ideas for possible enhancements.

6.1.1   Java agent

We can extend the functionality of the Java agent in multiple ways. Its current biggest weakness is the lack of support for parallel test case execution. To allow parallel testing, we must first solve a problem related to the scheduling process. Since the execution time of a test case can vary significantly, a coordination mechanism is required to schedule which test case should be executed on which thread. One possibility would be to consider the average execution time per test case, which we can obtain by examining prior runs. Alternatively, the scheduling can be performed at runtime using an existing inter-thread communication paradigm, such as message passing. Specific to the Java agent, implementing parallel execution requires us to modify the current `TestProcessor` to extend the `MaxNParallelTestClassProcessor` instead. A thread pool should ideally be used to diminish the overhead of restarting a new thread for every test case.

6.1.2   Predictions

We can make four different enhancements to the predictors.

For the first enhancement, the predictor should be able to discriminate between a unit test, an integration test or a regression test. Recall that the scope of a unit test is limited to a small fraction of the application code and that its execution time is usually low. Contrarily, an integration test and a regression test usually take much longer to execute and test multiple components of the application at once. The predictor should ideally make use of this distinction and assume that a failing unit test will almost certainly affect a regression or integration test as well. Consequently, the predictor should prioritise unit tests over other types of tests.

Secondly, the prediction algorithms currently take into account which source code lines have either been modified or removed to identify which test cases have been affected. Likewise, a change in the code of the test case itself should also consider that test case affected, as the change might have introduced a bug as well.

A third possible improvement would be to examine the performance of combining multiple prediction algorithms. Currently, the algorithms operate independently from each other, but there might be hidden potential in combining the individual strengths of these algorithms dynamically at runtime. A simple implementation is possible by modifying the existing meta predictor. Instead of assigning a score to the entire prediction, we could combine several predictions using predefined weights from earlier predictions.

Finally, the predictors do not currently consider branch coverage in addition to statement coverage. Not every coverage tool is capable of accurately reporting which branches have been covered, therefore this has not been implemented. Branch coverage can alternatively be supported by instrumenting the source code and rewriting every conditional expression as separate `if`-statements.

6.1.3 Meta predictor

The current implementation of the meta predictor increments the score of the predictor if the prediction was above-average, and decreases the score otherwise. However, a possible problem with this approach is that the nature of the source code might evolve and change as time progresses. As a result, it might take several test suite invocations for the meta predictor to prefer an alternative predictor. We can mitigate this effect if we would use a saturating counter (Figure 6.1) instead. This idea is also used in branch predictors of microprocessors and allows a more versatile meta predictor.
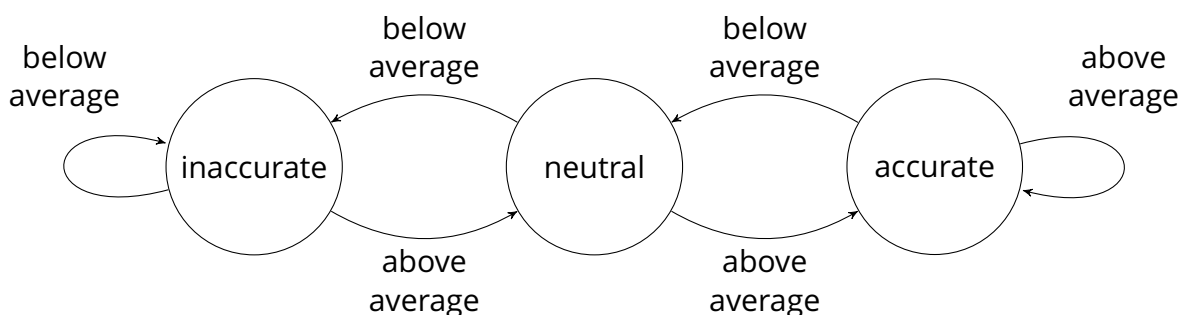
Figure 6.1: Saturating counter with three states.

In addition to implementing a different update strategy, it might be worth to investigate the use of machine learning or linear programming models as a meta predictor, or even as a prediction algorithm.

6.1.4   Final enhancements

Finally, since we can apply every implemented algorithm to Test Suite Minimisation as well, we might extend the architecture to support this technique explicitly. Executing fewer test cases will result in even lower execution times.

Support for new programming languages and frameworks is possible by implementing a new agent. A naive implementation would be to restart the test suite after every executed test case, should test case reordering not be supported natively by the test framework.

Bibliography

[1]   *About GitHub Actions*. URL: https://help.github.com/en/actions/getting-started-with-github-actions/about-github-actions.

[2]   Mohammed Arefeen and Michael Schiller. "Continuous Integration Using Gitlab". In: *Undergraduate Research in Natural and Clinical Science and Technology (URNCST) Journal* 3 (Sept. 2019), pp. 1–6. DOI: 10.26685/urncst.152.

[3]   Beck. *Test Driven Development: By Example*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0321146530.

[4]   Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://www.agilemanifesto.org/.

[5]   Moritz Beller, Georgios Gousios, and Andy Zaidman. "TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration". In: *Proceedings of the 14th working conference on mining software repositories*. 2017.

[6]   H.D. Benington. *Production of large computer programs*. ONR symposium report. Office of Naval Research, Department of the Navy, 1956, pp. 15–27. URL: https://books.google.com/books?id=tLo6AQAAMAAJ.

[7]   Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844.

[8]   Michael Cusumano, Akindutire Michael, and Stanley Smith. "Beyond the waterfall : software development at Microsoft". In: (Feb. 1995).

[9]   Charles-Axel Dein. *dein.fr*. Sept. 2019. URL: https://www.dein.fr/2019-09-06-test-coverage-only-matters-if-at-100-percent.html.

[10]   Thomas Durieux et al. "An Analysis of 35+ Million Jobs of Travis CI". In: (2019). DOI: 10.1109/icsme.2019.00044. eprint: arXiv:1904.09416.

[11]   *Features • GitHub Actions*. URL: https://github.com/features/actions.

[12]   Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman & Co., 1990. ISBN: 0716710455.

[13]   *GitLab CI/CD*. URL: https://docs.gitlab.com/ee/ci/.

[14]   *GitLab Continuous Integration & Delivery*. URL: https://about.gitlab.com/stages-devops-lifecycle/continuous-integration/.

[15]   A. Govardhan. "A Comparison Between Five Models Of Software Engineering". In: *IJCSI International Journal of Computer Science Issues 1694-0814* 7 (Sept. 2010), pp. 94–101.

[16] Standish Group et al. "CHAOS report 2015". In: *The Standish Group International* (2015). URL: https://www.standishgroup.com/sample_research_files/CHAOSReport2015-Final.pdf.

[17] M. Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. "A Methodology for Controlling the Size of a Test Suite". In: *ACM Trans. Softw. Eng. Methodol.* 2.3 (July 1993), pp. 270–285. ISSN: 1049-331X. DOI: 10.1145/152388.152391. URL: https://doi.org/10.1145/152388.152391.

[18] Naftanaila Ionel. "AGILE SOFTWARE DEVELOPMENT METHODOLOGIES: AN OVERVIEW OF THE CURRENT STATE OF RESEARCH". In: *Annals of Faculty of Economics* 4 (May 2009), pp. 381–385.

[19] "ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing –Part 1:Concepts and definitions". In: *ISO/IEC/IEEE 29119-1:2013(E)* (Sept. 2013), pp. 1–64. DOI: 10.1109/IEEESTD.2013.6588537.

[20] "ISO/IEC/IEEE International Standard - Systems and software engineering – System life cycle processes". In: *ISO/IEC/IEEE 15288 First edition 2015-05-15* (May 2015), pp. 1–118. DOI: 10.1109/IEEESTD.2015.7106435.

[21] "ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary". In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[22] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678.

[23] N. Landry. *Iterative and Agile Implementation Methodologies in Business Intelligence Software Development*. Lulu.com, 2011. ISBN: 9780557247585. URL: https://books.google.be/books?id=bUHJAQAAQBAJ.

[24] G. Le Lann. "An analysis of the Ariane 5 flight 501 failure-a system engineering perspective". In: *Proceedings International Conference and Workshop on Engineering of Computer-Based Systems*. Mar. 1997, pp. 339–346. DOI: 10.1109/ECBS.1997.581900.

[25] Simon Maple. *Development Tools in Java: 2016 Landscape*. July 2016. URL: https://www.jrebel.com/blog/java-tools-and-technologies-2016.

[26] D. Marijan, A. Gotlieb, and S. Sen. "Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study". In: *2013 IEEE International Conference on Software Maintenance*. 2013, pp. 540–543.

[27] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C# (Robert C. Martin)*. USA: Prentice Hall PTR, 2006. ISBN: 0131857258.

[28]   Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing*. 3rd. Wiley Publishing, 2011. ISBN: 1118031962.

[29]   Dor Nir, Shmuel Tyszberowicz, and Amiram Yehudai. "Locating Regression Bugs". In: *Hardware and Software: Verification and Testing*. Ed. by Karen Yorav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–234. ISBN: 978-3-540-77966-7.

[30]   Raphael Noemmer and Roman Haas. "An Evaluation of Test Suite Minimization Techniques". In: Dec. 2019, pp. 51–66. ISBN: 978-3-030-35509-8. DOI: `10.1007/978-3-030-35510-4_4`.

[31]   A. Jefferson Offutt and Roland H. Untch. "Mutation 2000: Uniting the Orthogonal". In: *Mutation Testing for the New Century*. Ed. by W. Eric Wong. Boston, MA: Springer US, 2001, pp. 34–44. ISBN: 978-1-4757-5939-6. DOI: `10.1007/978-1-4757-5939-6_7`. URL: `https://doi.org/10.1007/978-1-4757-5939-6_7`.

[32]   W. W. Royce. "Managing the Development of Large Software Systems: Concepts and Techniques". In: *Proceedings of the 9th International Conference on Software Engineering*. ICSE '87. Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338. ISBN: 0-89791-216-0. URL: `http://dl.acm.org/citation.cfm?id=41765.41801`.

[33]   John Ferguson Smart. *Jenkins: The Definitive Guide*. Beijing: O'Reilly, 2011. ISBN: 978-1-4493-0535-2. URL: `https://www.safaribooksonline.com/library/view/jenkins-the-definitive/9781449311155/`.

[34]   Travis. *Travis CI - Test and DeployYour Code with Confidence*. Feb. 2020. URL: `https://travis-ci.org`.

[35]   Kristen R. Walcott et al. "TimeAware Test Suite Prioritization". In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 1–12. ISBN: 1595932631. DOI: `10.1145/1146238.1146240`. URL: `https://doi.org/10.1145/1146238.1146240`.

[36]   James Whittaker. "What is software testing? And why is it so hard?" In: *Software, IEEE* 17 (Feb. 2000), pp. 70–79. DOI: `10.1109/52.819971`.

[37]   S. Yoo and M. Harman. "Regression Testing Minimization, Selection and Prioritization: A Survey". In: *Softw. Test. Verif. Reliab.* 22.2 (Mar. 2012), pp. 67–120. ISSN: 0960-0833. DOI: `10.1002/stv.430`. URL: `https://doi.org/10.1002/stv.430`.

List of Figures

List of Tables

List of Examples

Appendices

Appendix A

TravisTorrent queries

```
1  SELECT
2    COUNTIF(tr_log_bool_tests_failed) as failed,
3    COUNTIF(tr_log_bool_tests_ran) as ran,
4    COUNT(1) as total
5  FROM `travistorrent`
```

Example A.1: TravisTorrent query: Find the amount of failed runs.

```
1  (
2    SELECT gh_build_started_at, true as failed
3    FROM `travistorrent`
4    WHERE
5      tr_build_id IN (
6        SELECT DISTINCT(tr_prev_build)
7        FROM `travistorrent`
8        WHERE tr_log_bool_tests_ran=true AND
            tr_log_bool_tests_failed=true
9      )
10     AND gh_build_started_at IS NOT null AND
          tr_log_bool_tests_failed=true
11 ) UNION ALL (
12   SELECT gh_build_started_at, false as failed
13   FROM `travistorrent`
14   WHERE tr_build_id IN (
15     SELECT DISTINCT(tr_prev_build)
16     FROM `travistorrent`
17     WHERE tr_log_bool_tests_ran=true AND tr_log_bool_tests_failed=
          false
18   )
19   AND gh_build_started_at IS NOT null AND tr_log_bool_tests_failed
        =true
20 )
```

Example A.2: TravisTorrent query: Find the probability of consecutive failures.