Glossary

CI	Continuous Integration. 2
MapReduce	a programming paradigm that allows large amounts of data to be processed in a distributed manner. 27
TCP	Test Case Prioritisation. 2, 5, 6
TCS	Test Case Selection. 2, 4, 5
TSM	Test Suite Minimisation. 2, 3, 5, 6

Chapter 1

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 [8]. 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 [9].

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 [8]. 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.

1.1 Classification of approaches

1.1.1 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 [10] in definition 1 and illustrated in Figure 1.1.

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

- $T = \{t_1, \dots, t_n\}$ *a test suite consisting of test cases* t_j*.*
- $R = \{r_1, \dots, r_m\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*
- $\{T_1, \dots, T_m\}$ *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) [10].

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

- $S = \{s_1, \dots, s_n\}$ *a finite set of elements.*
- $C = \{c_1, \dots, c_n\}$ *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 [10]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [4].

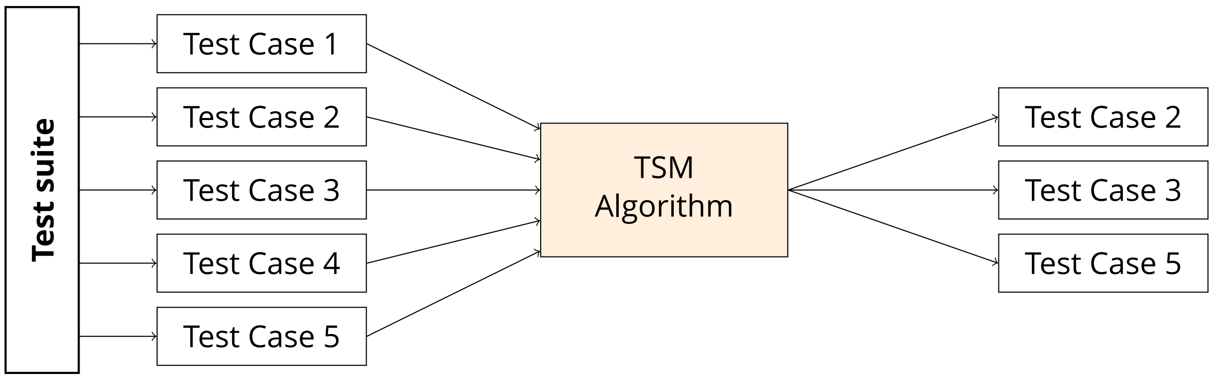


Figure 1.1: Test Suite Minimisation

1.1.2 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 1.2) and modification-aware [10]. 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'*.*

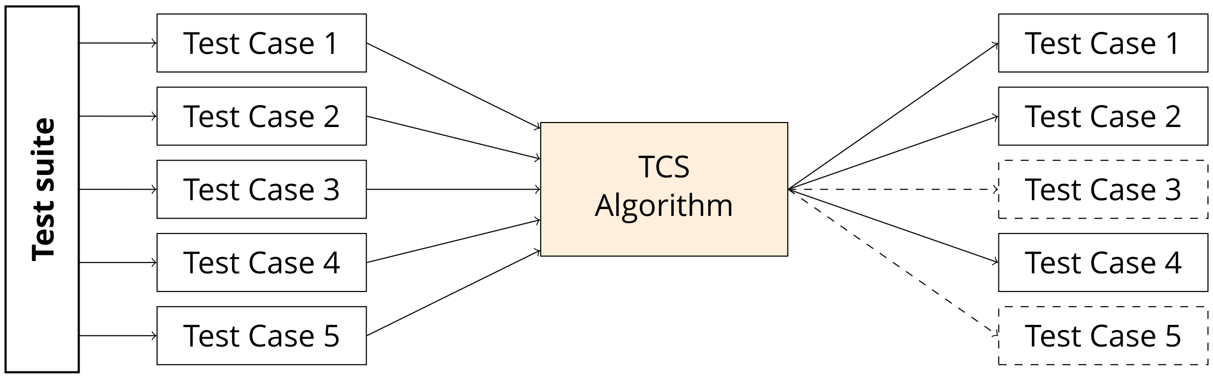


Figure 1.2: Test Case Selection

1.1.3 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 1.3). 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 [10]. 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* $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$

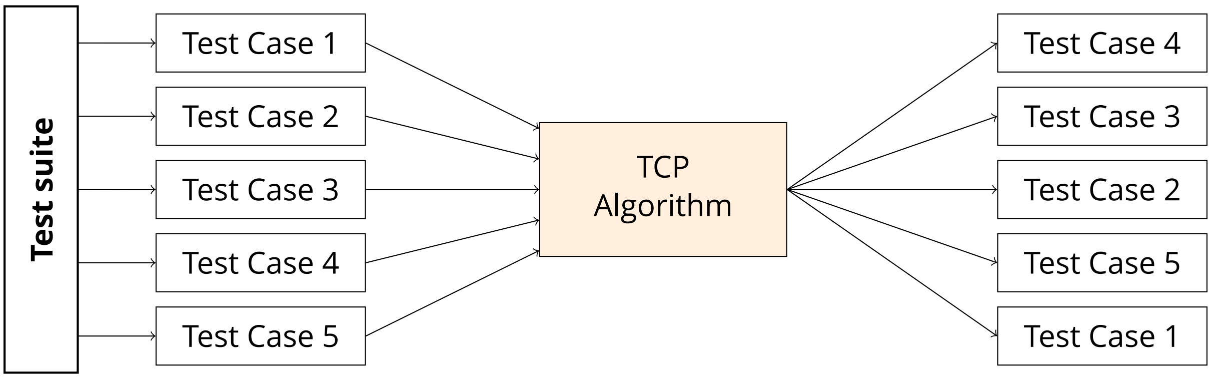


Figure 1.3: Test Case Prioritisation

1.2 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 1.1.1 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 [6]. 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*.*

1.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was initially designed by Chvatal to find an approximation for the set-covering problem [8]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [2]. 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:     $RS \leftarrow \emptyset$
 3:     while $C \neq \emptyset$ do
 4:         $t\_max \leftarrow 0$
 5:         $tl\_max \leftarrow \emptyset$
 6:         for all $t \in TS$ do
 7:             $tl\_current \leftarrow C \cap TL[t]$
 8:             if $|tl\_current| > |tl\_max|$ then
 9:                 $t\_max \leftarrow t$
10:                 $tl\_max \leftarrow tl\_current$
11:         $RS \leftarrow RS \cup \{t\_max\}$
12:         $C \leftarrow C \setminus tl\_max$
13:     return $RS$
```

1.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [5]. Similar to the Greedy algorithm (section 1.2.1), 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 [5]. 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 ([5])

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 $RS$
```

1.2.3 ROCKET algorithm

The third and final algorithm is the ROCKET algorithm. This algorithm has been presented by Marijan, Gotlieb and Sen [7] 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 1.1 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 1.1: 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 [7], the authors apply the following weights:

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

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

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

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 $P$
```

1.3 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.

1.3.1 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 1.2.3), 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/

Figure 1.4: Logo of Gradle

Figure 1.5: Logo of JUnit 5

1.3.2 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.

Figure 1.6: Logo of Maven

1.3.3 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.

Figure 1.7: Logo of Atlassian Clover

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

Chapter 2

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 2.3. 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.

2.1 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.

2.2 Architecture

2.2.1 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 1.3.1. When the test suite is started, the plugin will contact the controller (section 2.2.2) 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.

2.2.2 Controller

The second component is the core of the framework, acting as an intermediary between the agent on one side and the predictor (section 2.2.3) 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 2.3.4) by evaluating the accuracy of earlier predictions.

2.2.3 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/

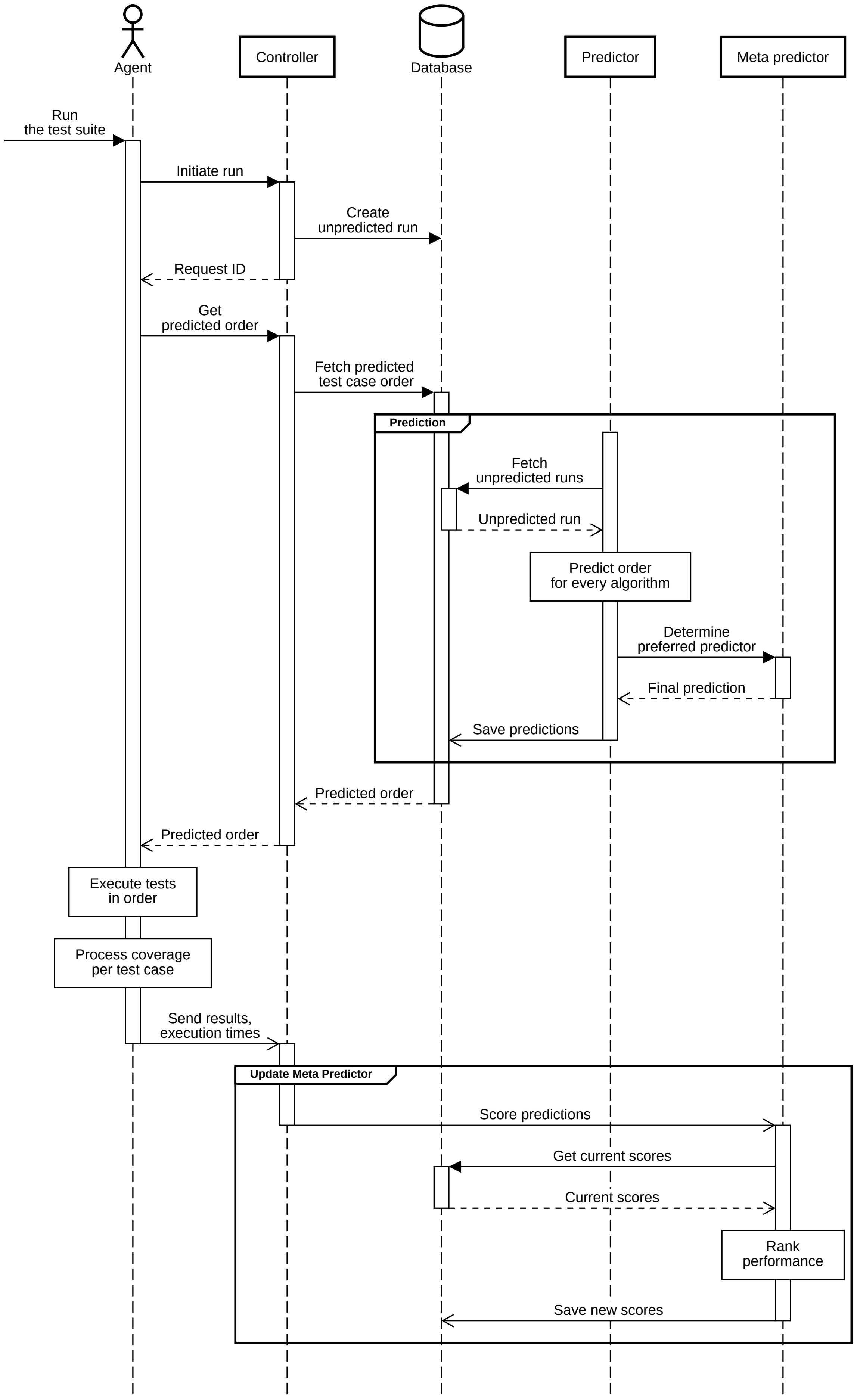


Figure 2.1: Sequence diagram of VeloCIty

2.3 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.

2.3.1 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.

Listing 2.1 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`.

```
buildscript {
    dependencies {
        classpath 'io.github.thepieterdc.velocity:velocity-
           junit:0.0.1-SNAPSHOT'
    }
}

apply plugin: 'velocity-junit'

velocity {
    base 'src/main/java/'
    repository 'https://github.com/user/my-java-project'
    server 'http://localhost:8080'
}
```

Listing 2.1: 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.

2.3.2 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 3.

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 1.2.1). 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 1.2.2). 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 1.2.3.

Alpha The final algorithm is a custom algorithm inspired by the other implemented algorithms. Section 2.4 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 2.3.4 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.

2.3.3 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 1.3.1 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.

2.3.4 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.

2.4 Alpha algorithm

Besides the earlier presented Greedy, HGS and ROCKET algorithms (section 1.2), 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:

- $TS = \{T_1, \dots, T_n\}$: the set of test cases to prioritise.
- $AS = \{T_1, \dots, T_m\} \subseteq TS$: 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.
- $C = \{c_1, \dots, c_m\}$: the set of all source code lines in the application, that are covered by at least one test case $t \in TS$.
- $F = \begin{bmatrix} F_1 & \dots & F_n \end{bmatrix}$: the failure statuses of each test case.
  - $F_t = \begin{bmatrix} f_1 & \dots & f_m \end{bmatrix}$: 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.
- $D = \begin{bmatrix} D_1 & \dots & D_n \end{bmatrix}$: the execution times of each test case.
  - $D_t = \begin{bmatrix} d_{t1} & \dots & d_{tm} \end{bmatrix}$: the execution times of test case t. D_{ij} corresponds to the duration (in milliseconds) of test case i in execution $(current - j)$.
- $TL = \begin{bmatrix} TL_1 & \dots & TL_n \end{bmatrix}$: the list of coverage groups.
  - $TL_t = \{c_1, \dots, c_o\} \subseteq C$: 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 1.2.3). 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 3.4.1) 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 1.2.1).

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:     $P \leftarrow array[1 \dots n]$                        ▷ 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 $P$
```

2.5 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. Listing 2.2 contains an example output of this command.

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

Listing 2.2: Example 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.

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

Listing 2.3: Example 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 (listing 2.4) will list the test cases from the highest to the lowest failure rate.

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

Listing 2.4: Example 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 listing 2.5.

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

Listing 2.5: Example 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, a separate command is required that fetches the prediction of the given run from the database.

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

Listing 2.6: Example 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 listing 2.7 below, this would be the ROCKET algorithm.

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

Listing 2.7: Example output of the scores-command

Chapter 3

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.

3.1 Test subjects

3.1.1 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 (**??**) 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.

3.1.2 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 2.2.1) to be applied directly.

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

3.2 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 1.2.3) 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.

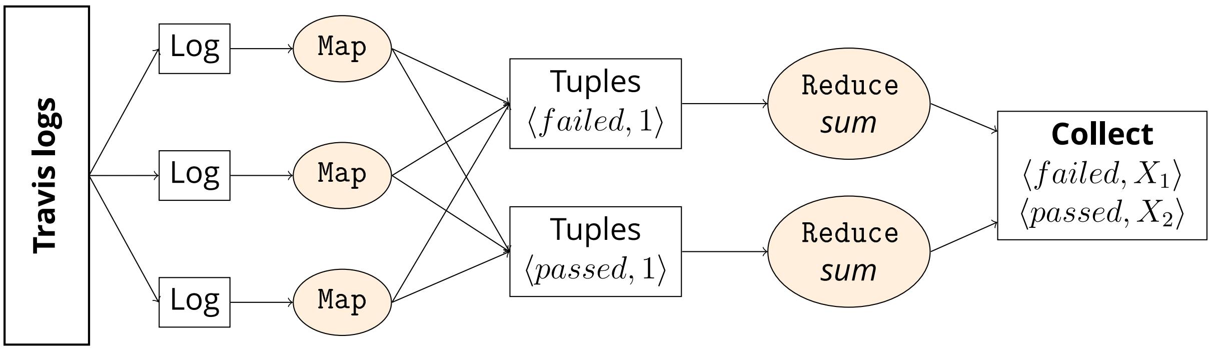
RQ5: Can the Java agent be applied to Stratego? Since the testing framework used by Stratego should be supported natively by the Java agent, this research question will verify its compatibility. Furthermore, we will analyse the prediction performance, albeit with a small number of relevant test runs.

3.3 Data collection

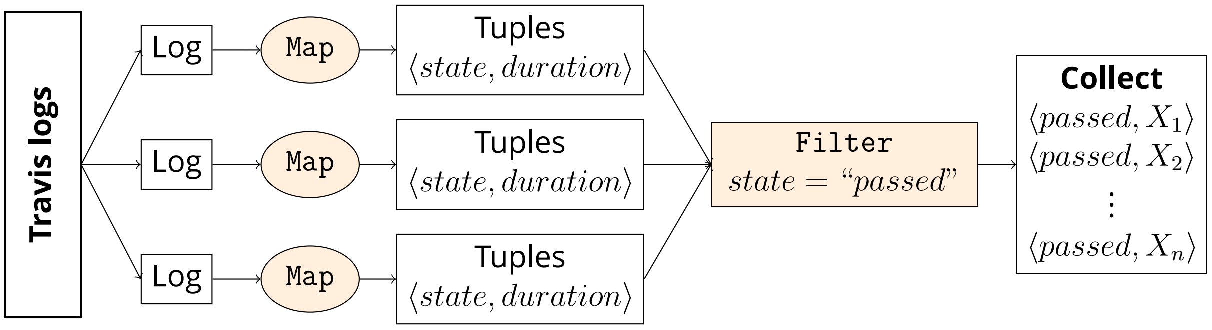
3.3.1 Travis CI build data

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

The first source comprises a database [3] 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 3.1) have been created using the Apache Spark[3] engine, to provide an answer to the first and second research question.


Travis logs
Log
Log
Log
Map
Map
Map
Tuples $\langle failed, 1\rangle$
Tuples $\langle passed, 1\rangle$
Reduce sum
Reduce sum
Collect $\langle failed, X_1\rangle$ $\langle passed, X_2\rangle$

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


Travis logs
Log
Log
Log
Map
Map
Map
Tuples $\langle state, duration\rangle$
Tuples $\langle state, duration\rangle$
Tuples $\langle state, duration\rangle$
Filter $state = \text{"passed"}$
Collect $\langle passed, X_1\rangle$ $\langle passed, X_2\rangle$ $\vdots$ $\langle passed, X_n\rangle$

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

Figure 3.1: MapReduce pipelines for Travis CI data

In addition to the first source, another 3 702 595 jobs have been analysed from the *TravisTorrent* project [1]. 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. **??** contains the executed queries.

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

3.3.2 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.

3.3.3 Stratego data

To integrate VeloCIty with the existing Stratego codebase, we can use the instructions described in chapter 2. 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.

3.4 Results

3.4.1 RQ1: Probability of failure

The two pie charts in Figure 3.2 illustrate the amount of failed and successful test runs. The leftmost chart visualises the failure rate in the dataset [3] 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 [1] 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 %.

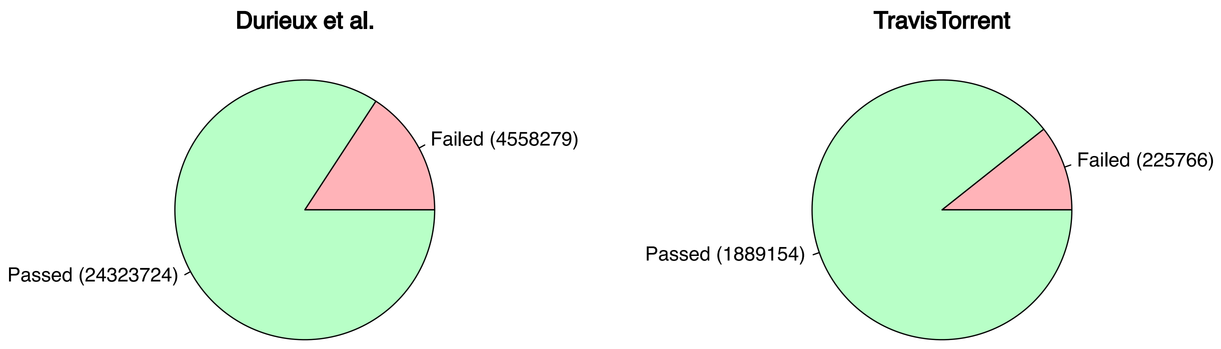


Figure 3.2: Probability of test run failure

3.4.2 RQ2: Average duration of a test run

The dataset by Durieux et al. [3] 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 3.1 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 3.3 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 h11 min26 s

Table 3.1: Characteristics of the test run durations in [3].

[5]A Ruby library for hypermedia (https://github.com/plexus/yaks).

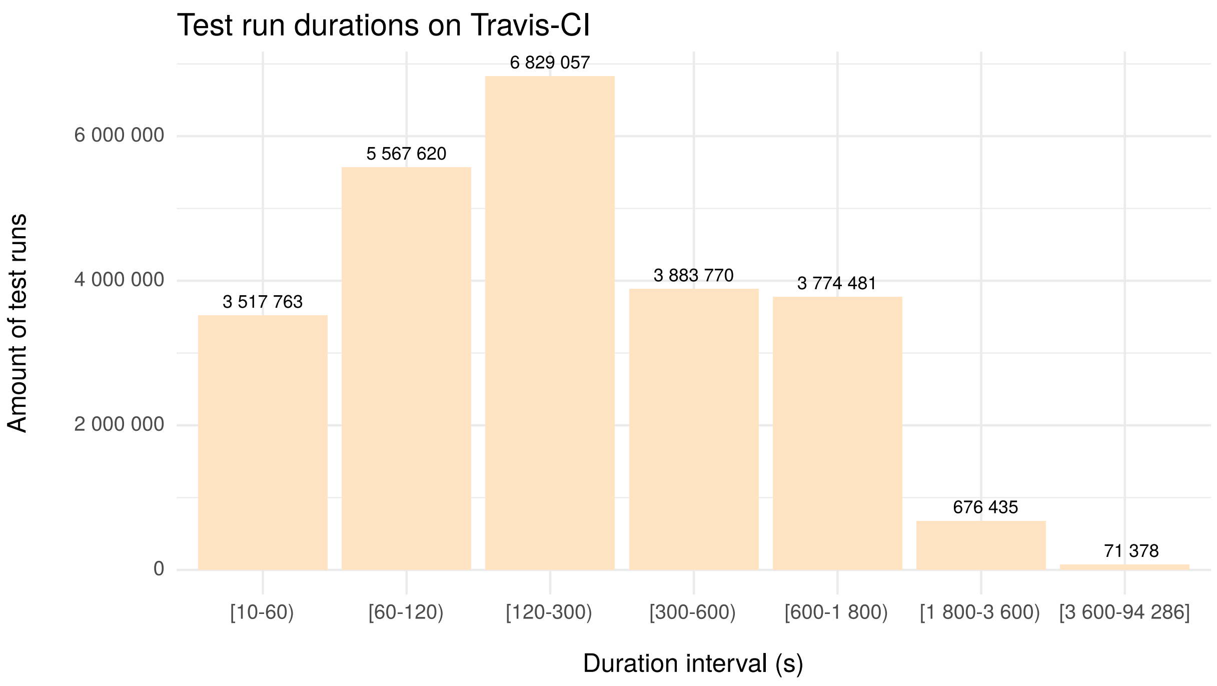


Figure 3.3: Test run durations on Travis CI

3.4.3 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 3.4, 109 224 of these test runs have failed as well, versus 101 816 successful test runs (51.76 %).

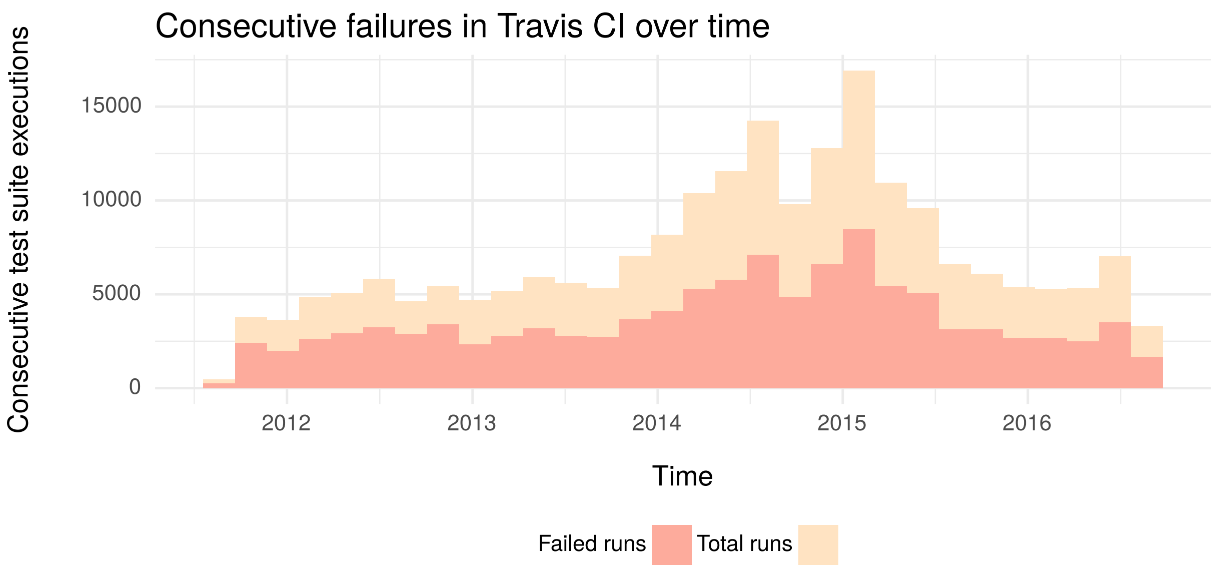


Figure 3.4: Consecutive test run failures on Travis CI

3.4.4 RQ4: Applying Test Case Prioritisation to Dodona

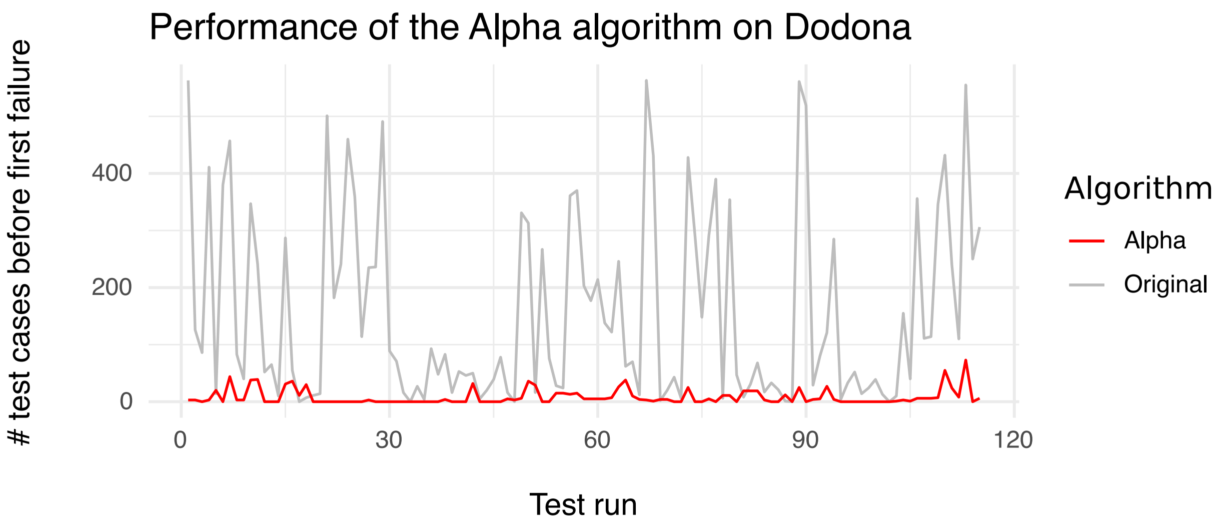
After executing the 120 failed test runs, the log files have been inspected. These log files have revealed that an error in the configuration was the actual culprit of five failed test runs, rather than a failed test case. These test runs have therefore been omitted from the results because the test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

Table 3.2 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.

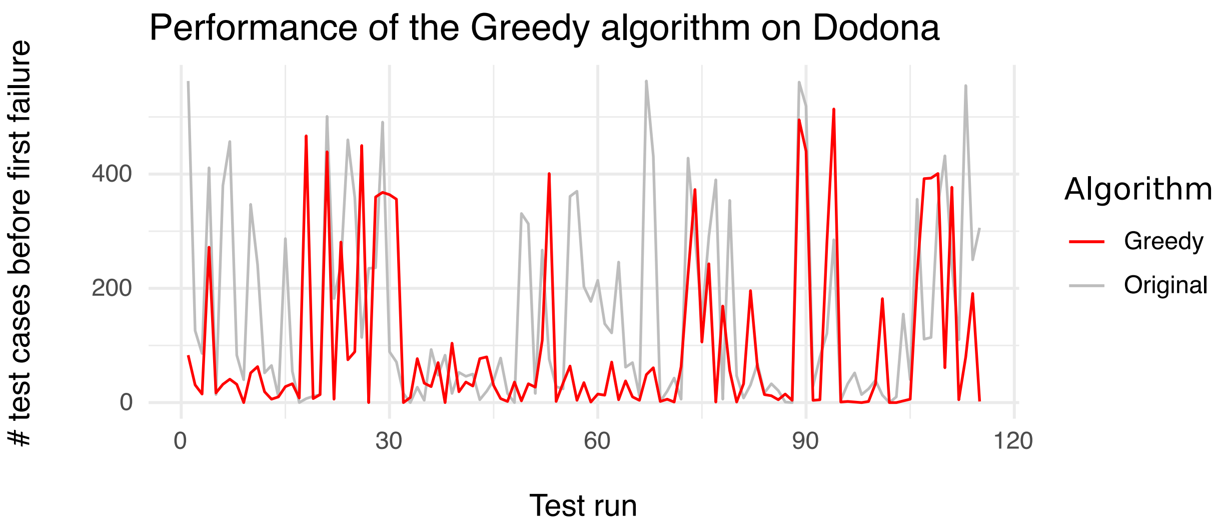
Algorithm	**Minimum**	**Mean**	**Median**	**Maximum**
Original	0	155	78	563
Alpha	0	8	3	73
AffectedRandom	0	54	10	428
AllInOrder	0	119	82	460
AllRandom	0	90	27	473
GreedyCoverAffected	0	227	246	494
GreedyCoverAll	0	98	33	514
GreedyTimeAll	0	210	172	482
HGSAffected	0	61	10	511
HGSAll	0	124	54	507
ROCKET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

Table 3.2: Amount of executed test cases until the first failure.

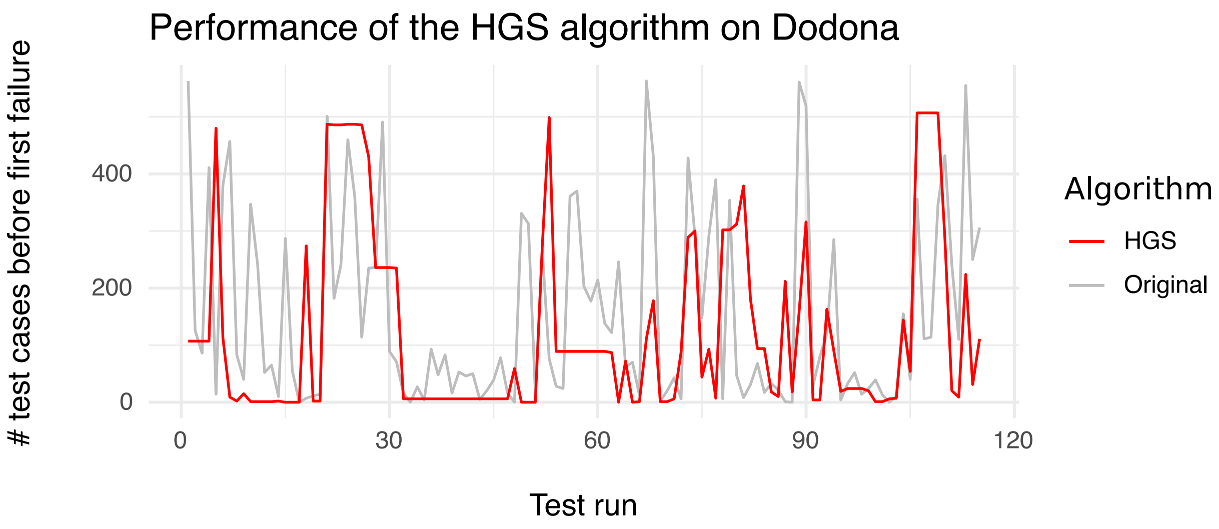
The previous results have been visualised in Figure 3.2. These charts confirm the low accuracy of the ROCKET algorithm. The Alpha algorithm and the HGS algorithm offer the most accurate predictions, with the former algorithm being the most consistent. Notice the chart of the Greedy algorithm, which succeeds in successfully predicting some of the test runs, while failing to predict others. This behaviour is specific to a greedy heuristic.



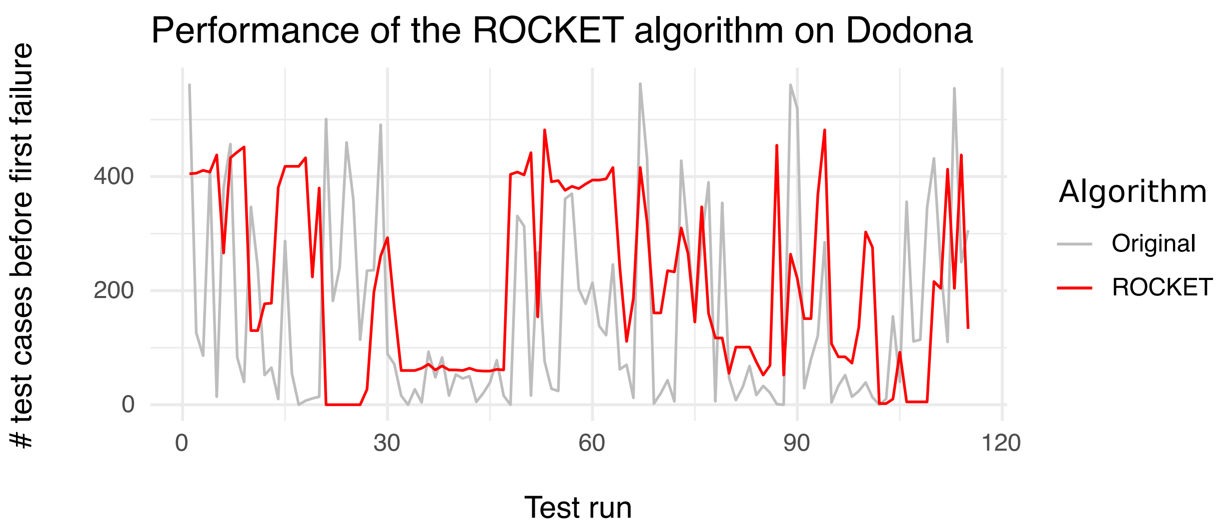
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 3.2: Prediction performance on the Dodona project.

The duration until the first observed failure is reported in Table 3.3. 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 3.3: Duration until the first failure for the Dodona project.

3.4.5 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 3.4 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

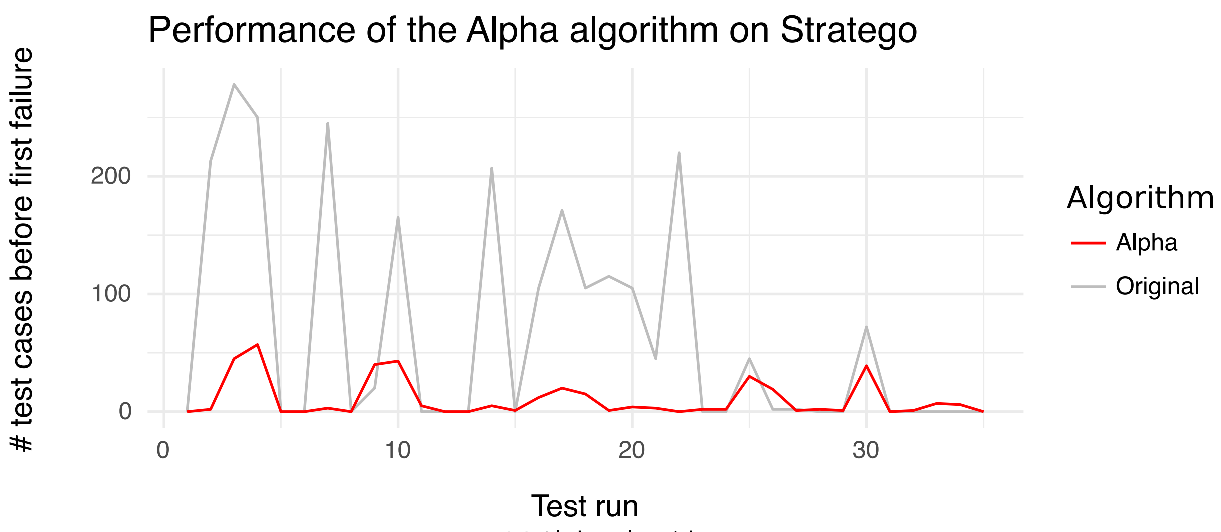
Table 3.4: Amount of executed test cases until the first failure.

Even though the performance of the ROCKET algorithm is suboptimal in the previous table, Table 3.5 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.

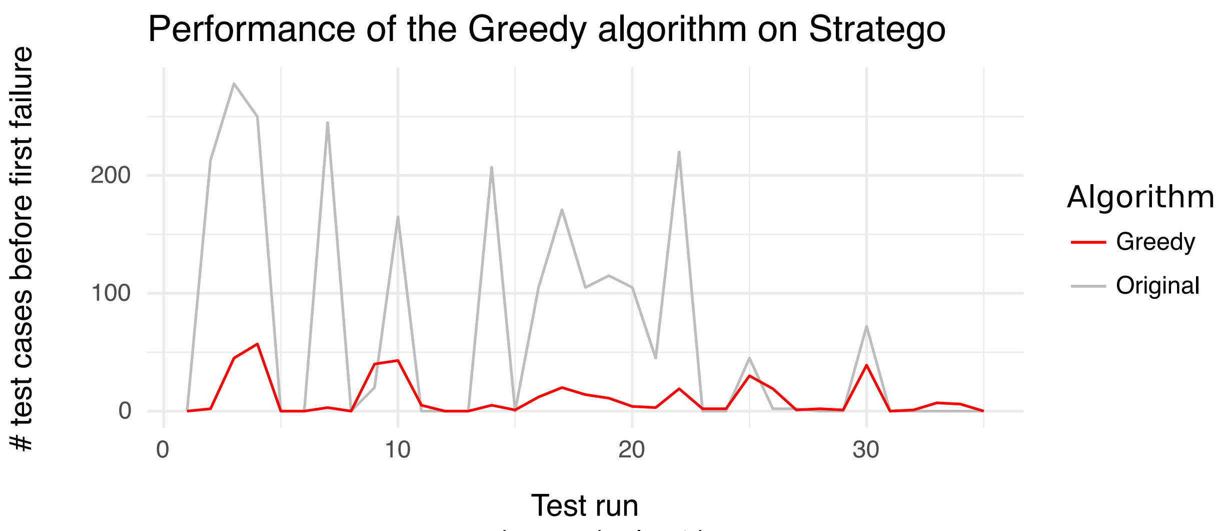
Algorithm	**Minimum**	**Mean**	**Median**	**Maximum**
Original	0 s	62 s	8 s	233 s
Alpha	0 s	11 s	2 s	103 s
GreedyCoverAll	0 s	12 s	2 s	103 s
HGSAll	0 s	19 s	1 s	130 s
ROCKET	0 s	6 s	0 s	85 s

Table 3.5: Amount of executed test cases until the first failure.

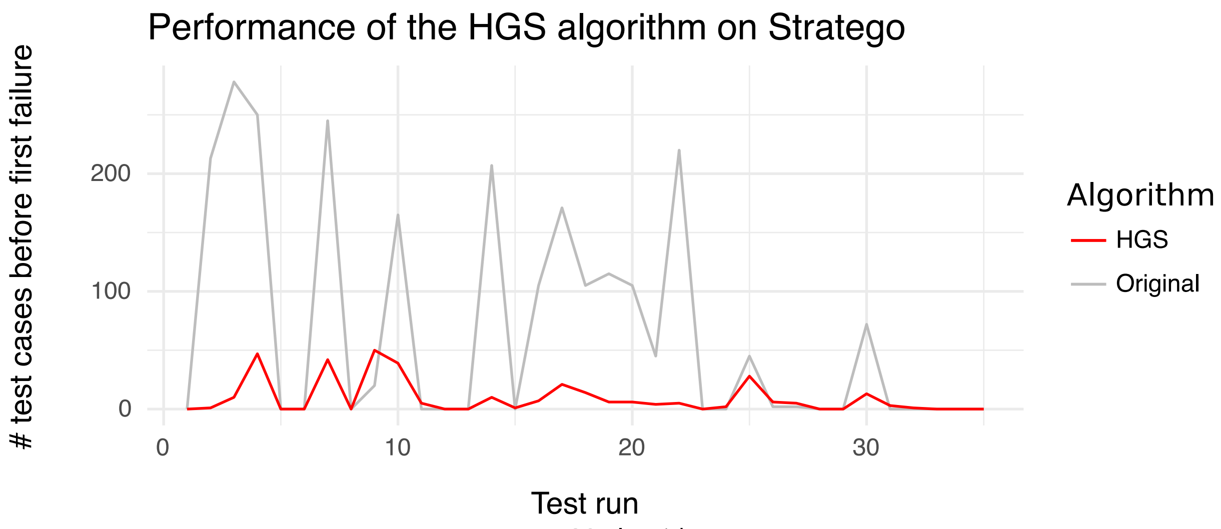
Figure 3.0 further confirms the above statements. Notice the close resemblance of the charts of the Greedy algorithm and the Alpha algorithm, which indicates that a different failing test case is the cause of every test run failure. The ROCKET algorithm performs better on this project than on Dodona, yet not accurate.



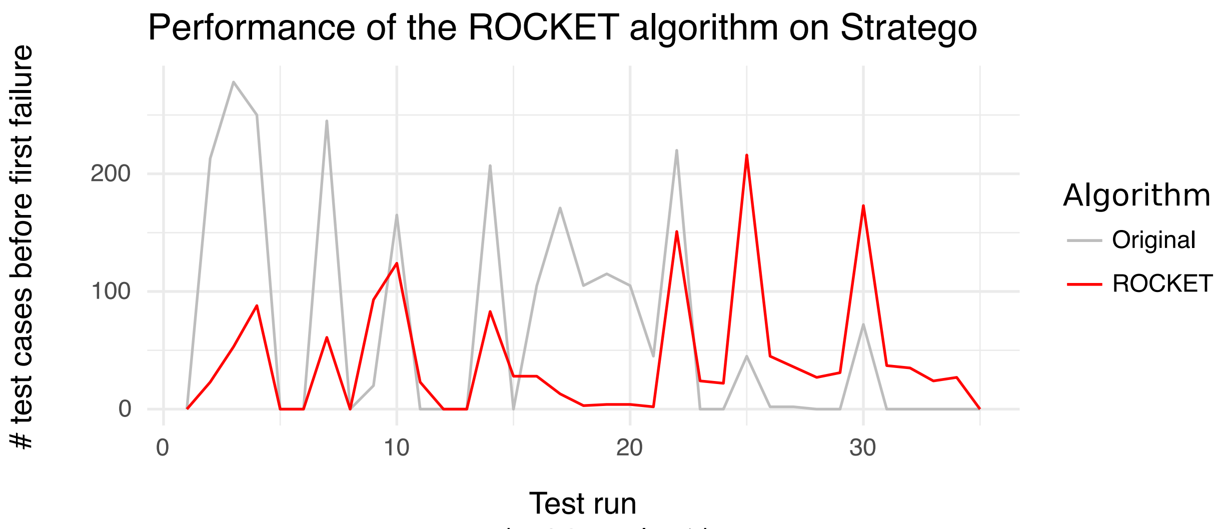
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 3.0: Prediction performance on the Stratego project.

Bibliography

[1] 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.

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

[3] 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.

[4] 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.

[5] 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.

[6] “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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.