Chapter 1

Evaluation

This chapter will evaluate the performance of framework presented in the previous chapter. In the first section, the two test subjects that will be used in the subsequent experiments will be introduced. The next section will restate the research questions formally, and extend these. Afterwards, the procedure of how the data was obtained will be elaborated. 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.

1.1 Test subjects

1.1.1 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 using 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. The application is tested using the default `MiniTest` testing framework and `SimpleCov`[2] is used to record the coverage of the test suite, which is currently approximately 89 %. This analysis will consider builds between January 1, 2020 and May 17, 2020.

1.1.2 Stratego

The second test subject is an application which was 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 (**??**) to be applied directly.

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

1.2 Research questions

The following research questions will be answered 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. It is expected that the probability of failure will be rather low, indicating that it is not strictly necessary to execute every test case and therefore proving the use of 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. Only successful test runs should be considered 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 (**??**) 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 a failing test case can be discovered using the available predictors.

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, the prediction performance will be analysed, albeit with a small number of appropriate test runs.

1.3 Data collection

1.3.1 Travis CI build data

The first three research questions have been answered by analysing data from projects hosted on Travis CI (**??**). This data was obtained from two sources.

The first source comprises a database [2] of 35 793 144 log files of executed test runs, 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 have been created using the Apache Spark[3] engine, to provide an answer to the first and second research question respectively.

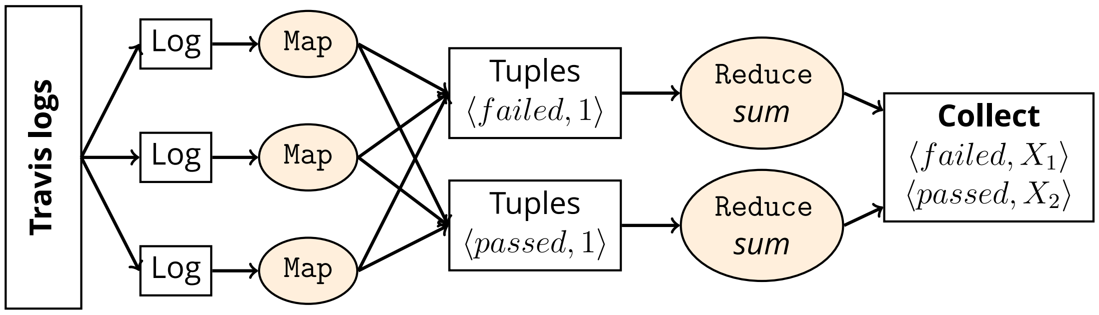

Travis logs
Log
Log
Log
Map
Map
Map
Tuples $\langle failed, 1\rangle$
Tuples $\langle passed, 1\rangle$
Reduce sum
Reduce sum
Collect
$\langle failed, X_1\rangle$
$\langle passed, X_2\rangle$

Figure 1.1: MapReduce pipeline to find the amount of failed test runs.

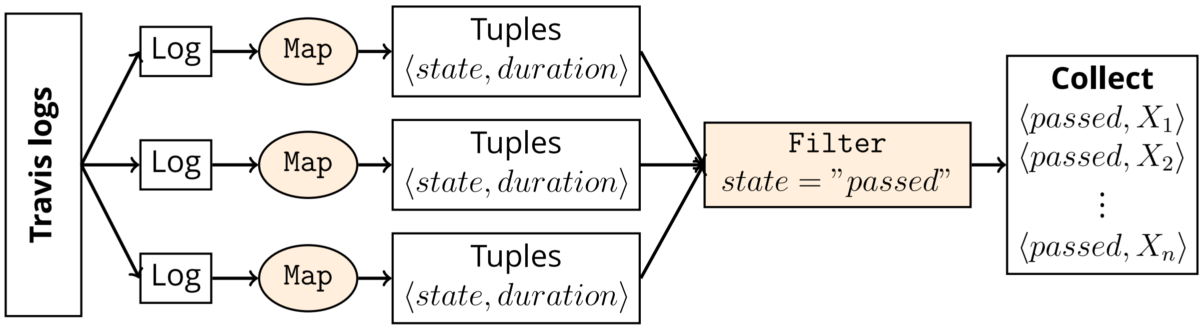

Travis logs
Log
Log
Log
Map
Map
Map
Tuples $\langle state, duration\rangle$
Tuples $\langle state, duration\rangle$
Tuples $\langle state, duration\rangle$
Filter $state = "passed"$
Collect
$\langle passed, X_1\rangle$
$\langle passed, X_2\rangle$
$\vdots$
$\langle passed, X_n\rangle$

Figure 1.2: MapReduce pipeline to find the average duration of a successful test run.

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 the build information, along with other useful statistics about the project. One of these additional values is the identifier of the previously executed run, which can be used to answer the second research question. Another interesting value is the amount of failed test cases. This 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. This dataset has been excluded from the second research question, 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. The executed queries have been listed in **??**.

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

1.3.2 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 is only capable of calculating the coverage for the entire test suite and does not allow 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 every failed test run. This script first queries the API of GitHub Actions to find which test runs have failed. In this thesis, 120 failed runs have been used. 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. This problem is resolved by using the Cobertura formatter instead, which generates XML reports. The structure of these reports is already supported by the Controller.
- **Parallel execution:** To reduce the execution time, the test cases are concurrently executed by four processes. The code coverage is recorded per process and afterwards merged. However, SimpleCov is not entirely thread-safe. This is not a problem if the total coverage is required, but it does prevent the accurate generation of per-test coverage. As a result, parallel execution has been disabled.

1.3.3 Stratego data

To integrate VeloCIty with the existing Stratego codebase, the instructions described in **??** have been used. Afterwards, to analyse the prediction performance, an approach similar to the previous test subject has been taken. 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.

1.4 Results

1.4.1 RQ1: Probability of failure

The two pie charts in Figure 1.3 illustrate the amount of failed and successful test runs. The leftmost chart visualises the failure rate in the dataset [2] 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 the cause of failure can be inferred 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 %.

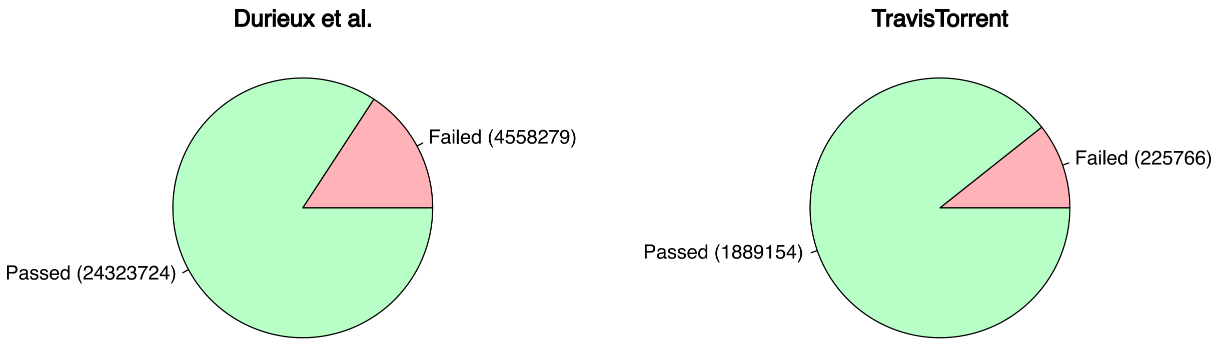


Figure 1.3: Probability of test run failure

1.4.2 RQ2: Average duration of a test run

The dataset by Durieux et al. [2] 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 1.1 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 1.4 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 1.1: Characteristics of the test run durations in [2].

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

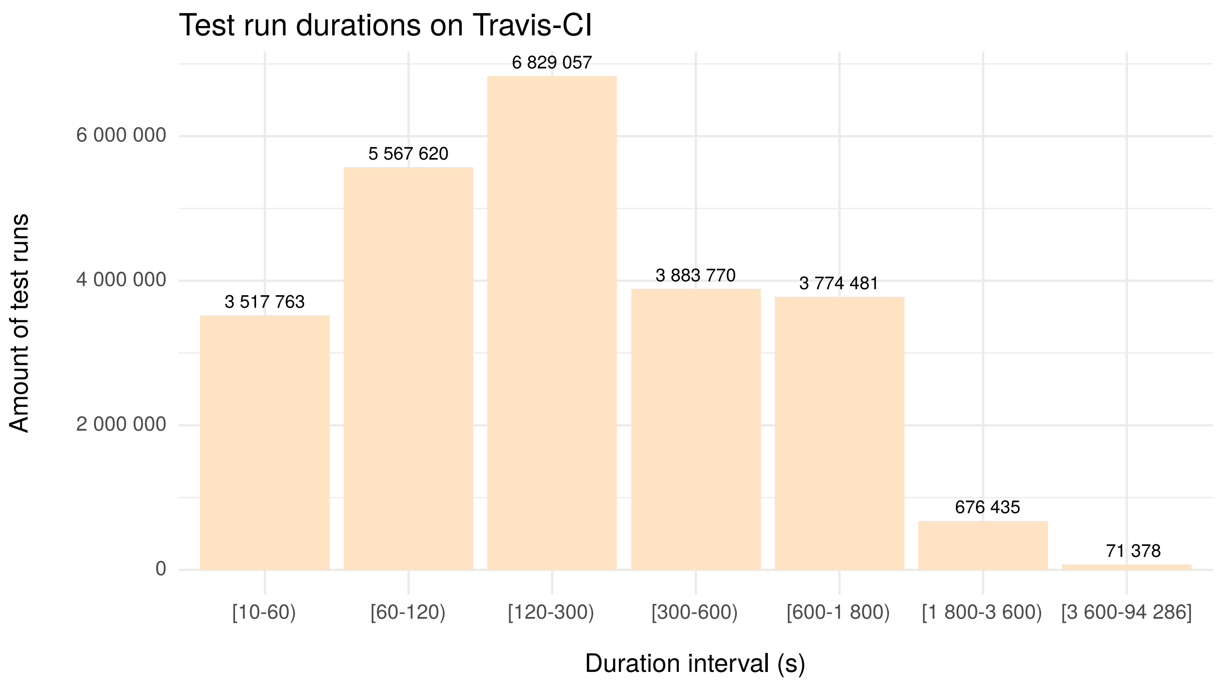


Figure 1.4: Test run durations on Travis CI

1.4.3 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 1.5, 109 224 of these test runs have failed as well, versus 101 816 successful test runs (51.76 %).

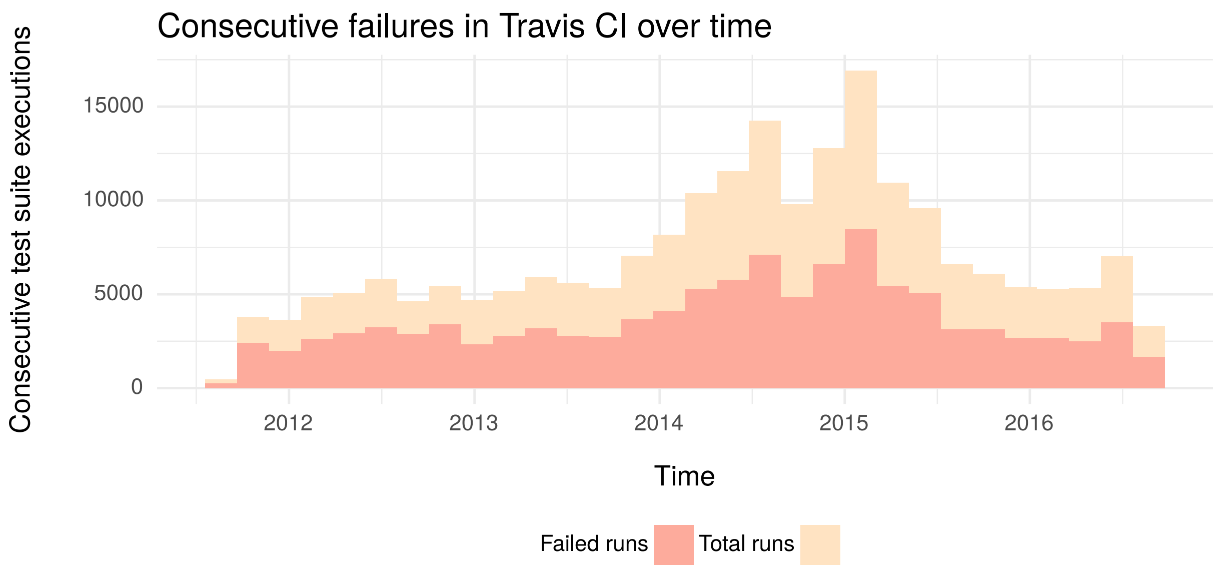


Figure 1.5: Consecutive test run failures on Travis CI

1.4.4 RQ4: Applying Test Case Prioritisation to Dodona

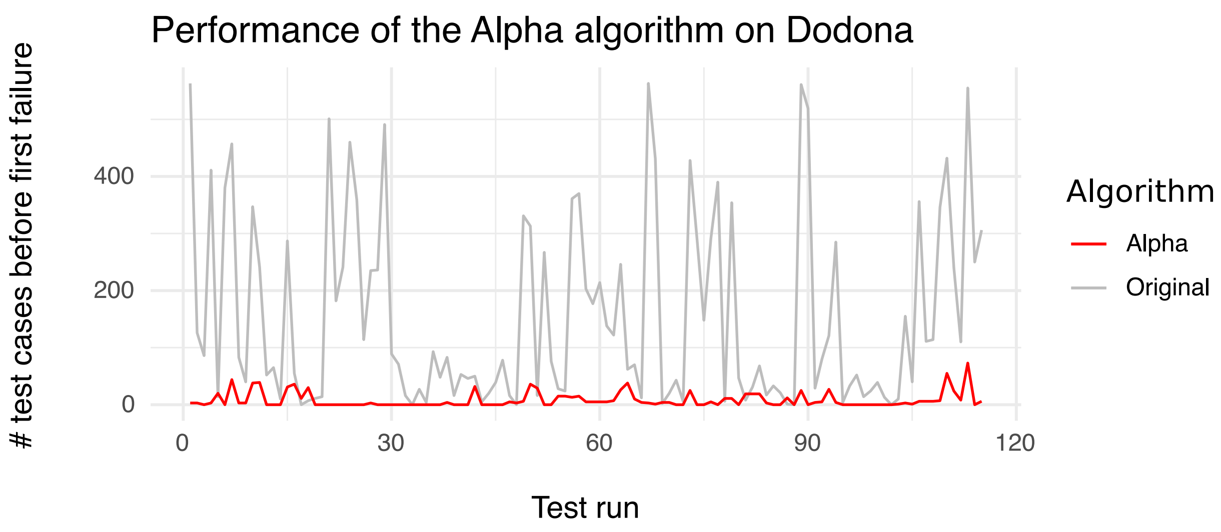
After executing the 120 failed test runs, the log files have been inspected. The log files have revealed that 5 test runs have failed because of an error in the configuration, rather than due to a failing test case. These test runs have therefore been omitted from the results because the test suite did not execute. Configuration problems require in-depth contextual information about the project and can therefore not be predicted.

Table 1.2 contains the amount of executed test cases until the first failure is observed. 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, while the ROCKET algorithm does not achieve a good performance.

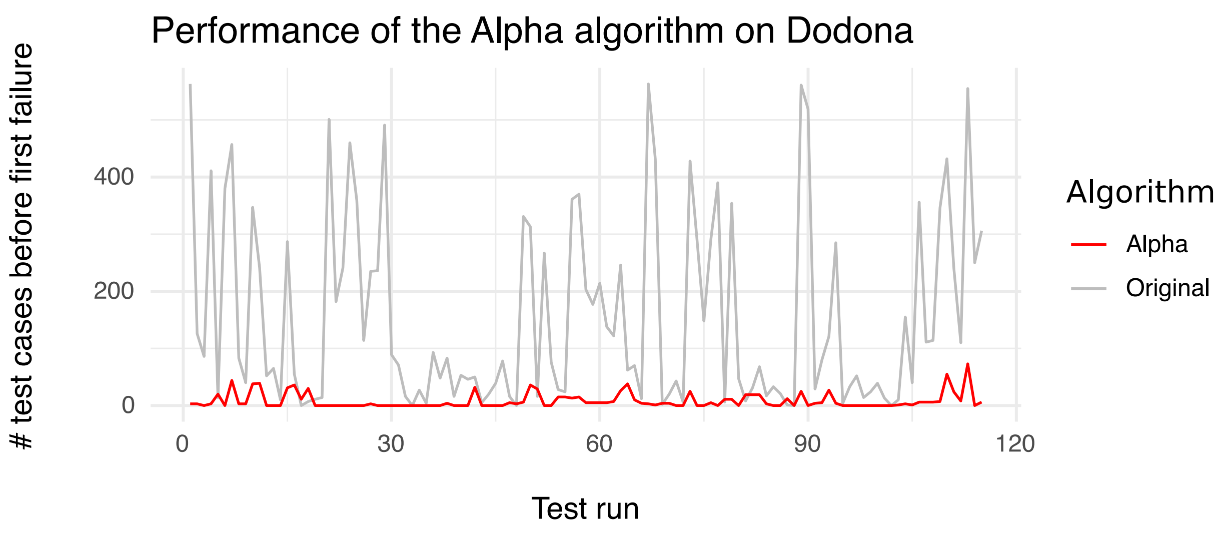
Algorithm	**Minimum**	**Mean**	**Median**	**Maximum**
Original	0	155	78	563
Alpha	0	8	3	73
AffectedRandom	0	54	10	428
AllInOrder	0	119	82	460
AllRandom	0	90	27	473
GreedyCoverAffected	0	227	246	494
GreedyCoverAll	0	98	33	514
GreedyTimeAll	0	210	172	482
HGSAffected	0	61	10	511
HGSAll	0	124	54	507
ROCKET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

Table 1.2: Amount of executed test cases until the first failure.

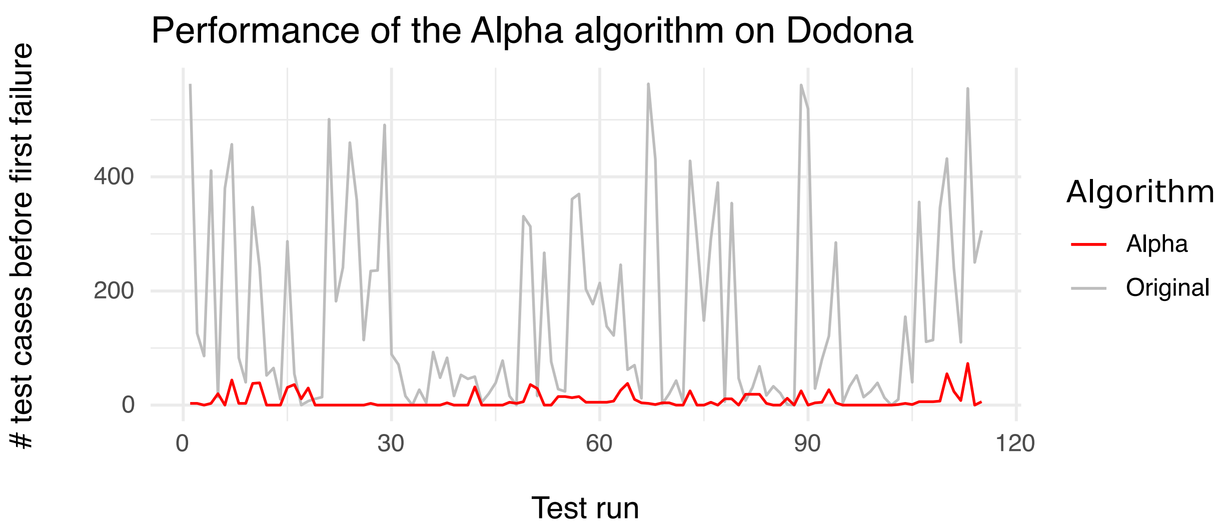
The previous results have been visualised in Figure 1.3. These charts confirm the low accuracy of the ROCKET algorithm. The Alpha algorithm and the HGS algorithm offer the most accurate predictions, with the former algorithm being the most consistent. Notice the chart of the Greedy algorithm, which succeeds in successfully predicting some of the test runs, while failing to predict others. This behaviour is specific to a greedy heuristic.



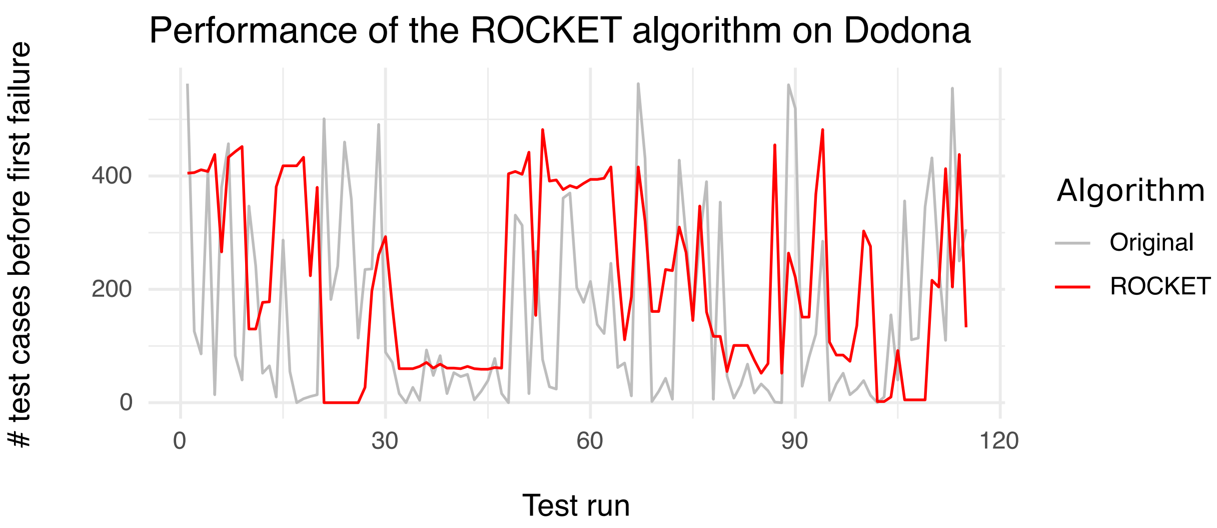
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 1.3: Prediction performance on the Dodona project

The duration until the first observed failure is reported in Table 1.3. Observe that the previous table indicates that the ROCKET algorithm does not perform well, while this table suggests otherwise. This behaviour is explained by the objective function of this algorithm, which prioritises test 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 1.3: Duration until the first failure.

1.4.5 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, only 35 of the 54 failed test runs could be used.

Similar to the previous test subject, Table 1.4 lists how many test cases have been executed before the first failure. The table only considers the four main algorithms, since the actual prediction performance was only secondary to this research question and only a small amount of test runs have been analysed. 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

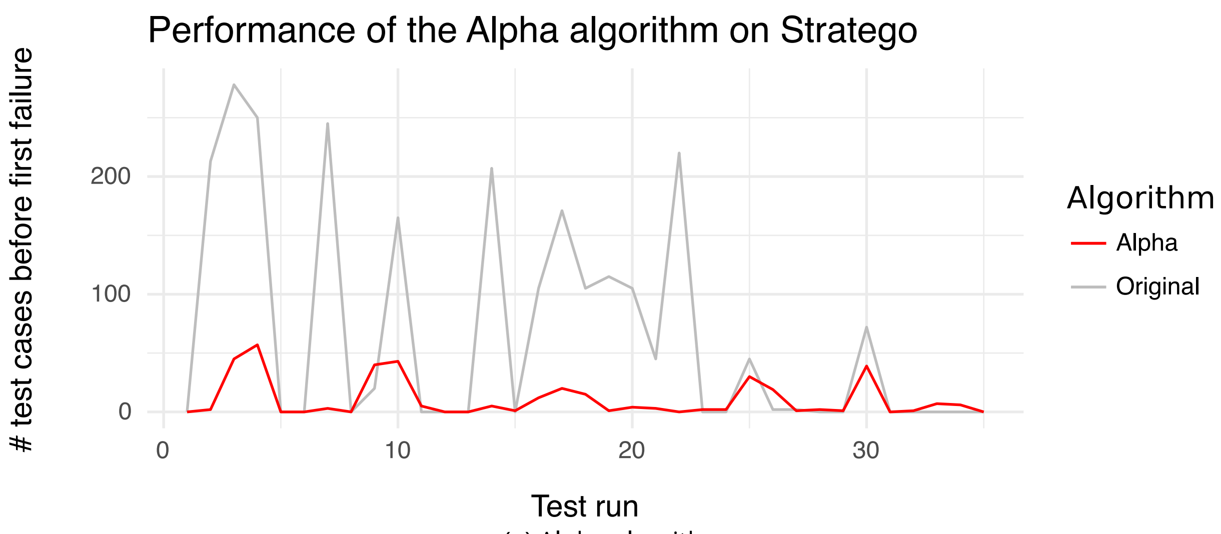
Table 1.4: Amount of executed test cases until the first failure.

Even though the performance of the ROCKET algorithm is suboptimal in the previous table, Table 1.5 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.

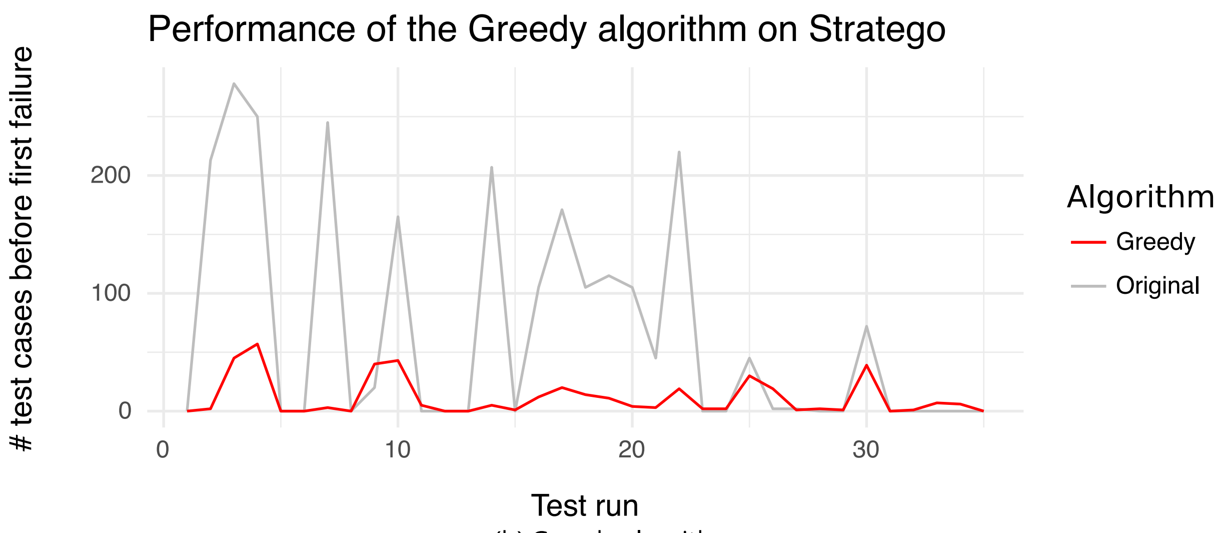
Algorithm	**Minimum**	**Mean**	**Median**	**Maximum**
Original	0 s	62 s	8 s	233 s
Alpha	0 s	11 s	2 s	103 s
GreedyCoverAll	0 s	12 s	2 s	103 s
HGSAll	0 s	19 s	1 s	130 s
ROCKET	0 s	6 s	0 s	85 s

Table 1.5: Amount of executed test cases until the first failure.

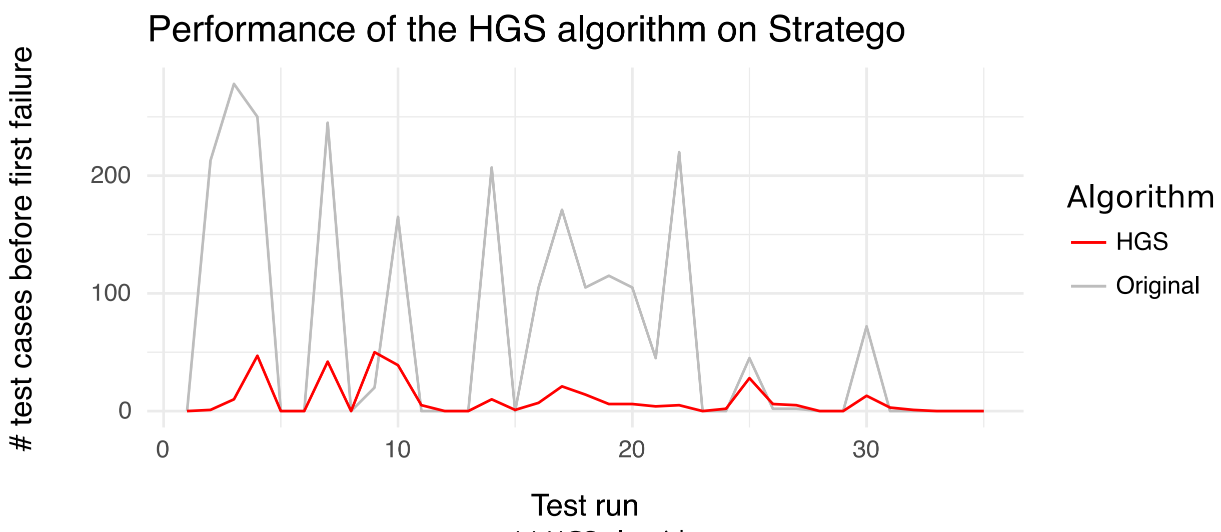
Figure 1.1 further confirms the above statements. Notice the close resemblance of the charts of the Greedy algorithm and the Alpha algorithm, indicating that almost every test run failure was caused by a different failing test case. The ROCKET algorithm performs better on this project than on Dodona, yet not accurate.



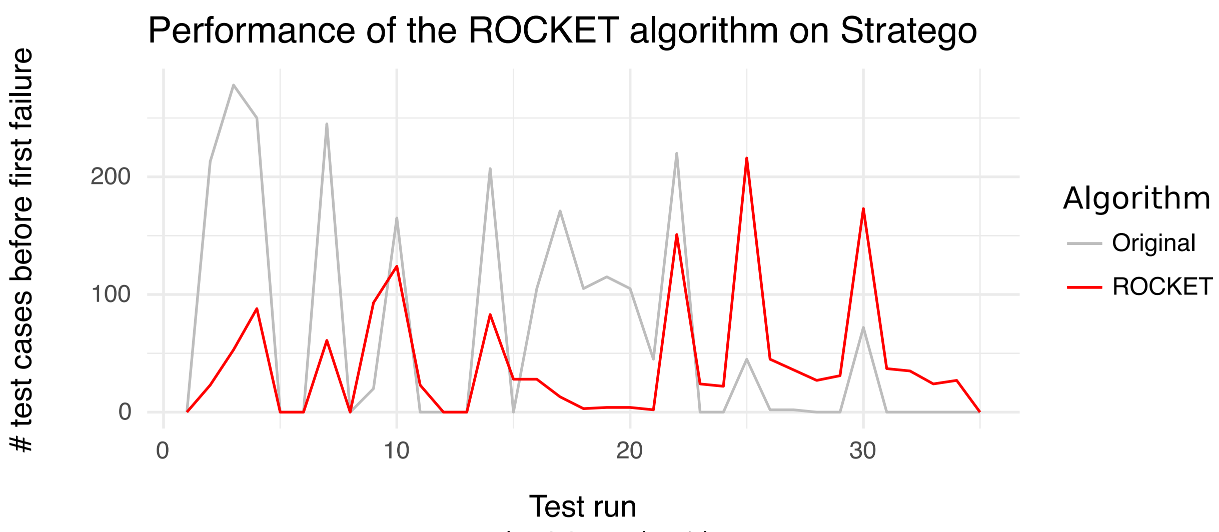
(a) Alpha algorithm



(b) Greedy algorithm



(c) HGS algorithm



(d) ROCKET algorithm

Figure 1.1: Prediction performance on the Stratego project

Chapter 2

Conclusion

The main purpose of this thesis has been to investigate different approaches towards optimising the test suite of a common software project. The concepts of Test Suite Minimisation, Test Case Selection and Test Case Prioritisation have been introduced and accompanying algorithms have been presented. A novel client-server oriented framework for the latter approach has been proposed, as well as a new prioritisation algorithm. Finally, VeloCIty has been applied to the UGent Dodona project, proving its ability to predict test case failure and therefore reduce the execution time of the test suite.

A second purpose of this thesis was to gain useful insights into the behaviour of a typical test suite. These insights have been formulated as three additional research questions, to which answers have been provided in the previous chapter.

2.1 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.

2.1.1 Java agent

The existing Java Agent can be extended in multiple ways. The most prominent addition would be to allow test cases to be executed in parallel. At the moment of writing, this is not possible yet. In order to facilitate parallel testing, one must first decide how to schedule the prioritised test cases across multiple threads, since the execution time of a test case varies strongly. One possibility to perform this scheduling is to use the average execution time per test case, which is obtained from prior runs. Alternatively, this can be performed at runtime by using any existing inter-thread communication paradigm such as message passing. On the implementation side of parallelisation, the current `TestProcessor` should be adapted to inherit from the `MaxNParallelTestClassProcessor`. A thread pool should ideally be used to reduce the overhead of restarting a new thread for every test case.

2.1.2 Predictions

Further research and improvements to the predictors can be made on four different aspects.

The first enhancement is that currently the predictor does not discriminate between a unit test or an integration test. Recall that the scope of a unit test is limited to a small fraction of the application and that its execution time is ideally rather low. An integration test however usually takes longer to execute and tests multiple components of the application at once. The predictor could make use of this distinction and assume that a failure in a unit test has a high probability of resulting in a failed integration test as well, hence prioritising unit tests over integration tests.

Secondly, the prediction algorithms currently take into account which source code lines have either been modified or removed in order to prioritise affected test cases. Likewise, test cases of which the code has been modified should also be considered as candidates for prioritisation, as the changed test case might contain a bug as well.

A third and unexplored research opportunity is to investigate the joint performance of multiple prediction algorithms combined. This could be integrated with the existing meta predictor. Instead of assigning a score to the entire prediction, multiple predictions could be intermingled using predefined weights.

The final improvement is to take into account branch coverage in addition to the statement coverage which is currently used. This is a rather complex feature as not every coverage framework is capable of reporting accurately which branches have been covered and which ones have not. A suggested implementation would be to instrument the source code and rewrite every condition of every branch as separate `if`-statements.

2.1.3 Meta predictor

The proposed meta predictor increases the score of every predictor which predicted an above-average ranking and decreases the score of the other predictors. However, a possible problem with this approach is that the nature of the source code might evolve and change as time progresses. Using the current updating strategy it will take several test suite invocations for an alternative predictor to be preferred by the meta predictor. If a saturating counter would be used instead (Figure 2.1), this would be resolved much more quickly, allowing a more versatile meta predictor.

INSPIRED BY https://en.wikipedia.org/wiki/Branch_predictor

2-LEVEL
Pik een predictor met de hoogste score

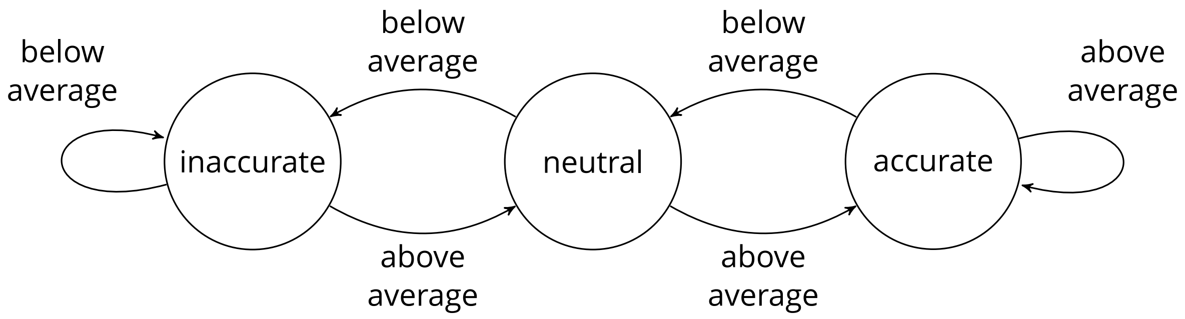


Figure 2.1: Saturating counter

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.

2.1.4 Final enhancements

Finally, since some of the implemented algorithms are inherently Test Suite Minimisation algorithms rather than prioritisation algorithms, the framework might opt to not execute some test cases at all, whereas now the entire test suite is always executed.

Support for other programming languages and frameworks is possible by implementing new agents. The basic implementation is straightforward to restart the test suite after every executed test case, should test case reordering not be supported natively by the test framework.