Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

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

Abstract—**Ever since the introduction of traditional software development models in the previous century, the complexity and magnitude of today's software have vastly increased. This evolution has led to the adoption of Agile software development approaches, which pose the need for frequent integration and automated testing. Eventually, this increase in size will also negatively affect the size of the testing suites, resulting in severe scalability issues. This thesis proposes a framework and a novel algorithm for test suite optimisation by prioritising test cases which are likely to fail. The performance has been evaluated on two existing applications, and the results are auspicious.**

Keywords—**Continuous Integration, test suite, performance, optimisation, prioritisation**

I. Introduction

The most characteristic trait of modern times is the astonishing speed at which everything in this world is evolving. This statement holds in particular for the field of computer science, where new technologies emerge almost every single day. While these inventions often seem primarily hardware-related, such as smartwatches, self-driving cars, or even biological technology, they cannot function without an even more sophisticated software component that controls them. As such, both the complexity and size of software have grown exponentially over the last decades.

Software engineers have experienced that the traditional software methodologies are no longer sufficient to handle this fast-paced development and, as a result, have shifted towards other strategies. Instead of implementing the entire application at once, developers now prefer the Agile approaches [**?**]. The Agile methodologies depict that an initial version of the application should be released as soon as possible and extend its functionality incrementally, to reduce the financial risks taken. A report of The Standish Group confirms that the adoption of these new approaches has led to a decreasing failure rate for new projects [**?**].

However, this evolution is not necessarily a good unfolding. Managing to build a project is one thing, guaranteeing it is built reliably is a different matter. An increase in the complexity of software inevitably makes the software more error-prone. As an attempted solution, the Agile approaches propose Continuous Integration (CI) [**?**]. This practice requires that the test suite of the application is executed after every code change (Figure 1). Multiple CI-services have been created to assist in this process by means of automation. Optionally, every passed version can automatically be released to the end-users (Continuous Deployment).

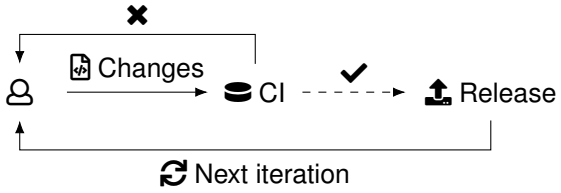


Figure 1: Continuous Integration.

Yet, in the long run, a new problem will emerge. As mentioned before, the size of applications increases exponentially. Every change in the code must be followed by the introduction of one or more test cases to guarantee the correctness. Therefore, the size of the test suite will increase even faster than the size of the project itself, which leads to severe scalability issues.

II. Techniques

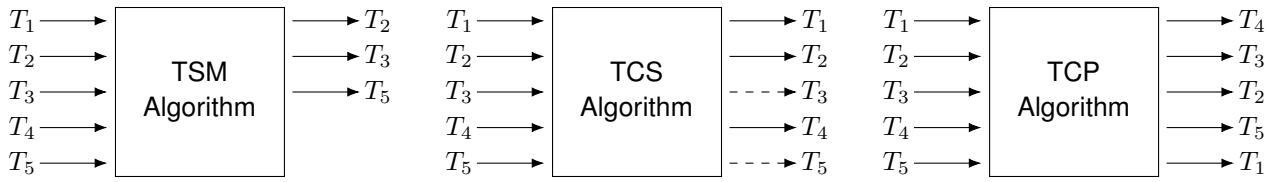
This thesis presents three techniques to tackle this problem.

A. Test Suite Minimisation (TSM)

The first technique is called Test Suite Minimisation (TSM) [**?**]. TSM attempts to reduce the size of the test suite by permanently removing redundant test cases according to the following definition:

Given:
- $T = \{t_1, \ldots, t_n\}$ a test suite consisting of test cases t_j.
- $R = \{r_1, \ldots, r_m\}$ a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.
- $\{T_1, \ldots, T_m\}$ subsets of test cases in T, such that any one of the test cases $t_j \in T_i$ can be used to satisfy requirement r_i.

TSM is then defined as the task of finding a subset T' of test cases $t_j \in T$ that satisfies every requirement r_i.



(a) Test Suite Minimisation (b) Test Case Selection (c) Test Case Prioritisation

Figure 2: Illustration of the techniques.

B. Test Case Selection (TCS)

Instead of permanently removing redundant test cases, it is also possible to analyse the code changes and deduce which test cases should be executed and which might ones be omitted altogether. This technique is referred to as Test Case Selection (TCS) [**?**].

Given:
- T the test suite.
- P the previous version of the codebase.
- P' the current (modified) version of the codebase.

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

C. Test Case Prioritisation (TCP)

While TSM and TCS execute only the relevant test cases, sometimes it might be required that every test case is successfully executed. Consider, for example, critical software for medical purposes. In this case, it is still possible to optimise the test suite by executing all test cases in a specific order. Test Case Prioritisation (TCP) [**?**] constructs an ordered sequence that aims to fulfil a predetermined objective as fast as possible. This thesis primarily uses the detection of the first failed test case as the objective.

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.

TCP finds a permutation $T' \in PT$ such that $\forall T'' \in PT : f(T') \geq f(T'') \Rightarrow (T'' \neq T')$.

III. Algorithms

This thesis prefers to use TCP since this technique does not incur the risk of false-negative failing test cases. In order to determine the optimal order of execution, this thesis presents three existing algorithms.

The input data for these algorithms is threefold:

- **Affected test cases:** By combining previous coverage results and the list of changes that the developer has made to the code, the framework can estimate which test cases are likely affected by those changes and assign a higher priority.
- **Historical data:** Next, historical failure data can be used. Suppose that a test case has failed in its previous execution, then there exists a chance that it will subsequently fail in the current run. Either way, this test case should be executed early to verify that the underlying issue has been resolved.
- **Execution timings:** Finally, if two test cases are considered equally likely to fail, the average duration of the test case can be used as a tie-breaker. Since the objective of TCP is to optimise the test suite, the test case with the lowest duration should be preferred.

A. Greedy algorithm

The first algorithm is a greedy heuristic that was initially designed as an algorithm for the set-covering problem [**?**]. This heuristic starts with an empty set of test cases and the set of all code lines C. Next, the algorithm iteratively selects the test case that contributes the most code lines that are not yet covered, updating C after every selected test case. The algorithm halts when either all test cases are selected or C is empty. In order to modify this algorithm to make it applicable to TCP, the selection order must be preserved and used as the prioritised sequence.

B. HGS algorithm

The second algorithm was created by Harrold, Gupta and Soffa [**?**]. As opposed to the greedy heuristic, this algorithm uses a different perspective. First, the algorithm sorts the code lines increasingly based on the number of test cases that cover them. The motivation for this sorting operation is that some test cases must inevitably be executed, as they are the only test cases that cover a given set of lines. However, these test cases can also cover other lines and therefore make other test cases redundant. The algorithm iterates over the code lines in this order and selects one corresponding test case in each iteration. Afterwards, the order is updated to remove source code lines which are now covered by the selected test case. This process is repeated until there are no lines left to cover.

C. ROCKET algorithm

Finally, this thesis considers the ROCKET algorithm [**?**]. This algorithm prioritises test cases by assigning a score to every test case, which is calculated using historical failure data. Afterwards, the algorithm computes the cumulative score CS_t of every test case t and defines the following objective function g, in which E_t represents the execution time of the test case:

$$g = (maximise(CS_t), minimise(E_t))$$

Finally, the algorithm optimises this function to determine the ideal order of execution S, as follows:

$$(\forall i \in 1 \ldots n)(g(S_i) \geq g(S_{i+1})$$

IV. Framework

This thesis proposes VeloCIty as a language-agnostic framework that enables Test Case Prioritisation on existing software projects. The architecture consists of three main components, a meta predictor, and a novel prioritisation algorithm.

A. Agent

The first component is the agent. This component hooks into the testing framework of the application to execute the test cases in the required order. The agent obtains the optimal execution sequence by communicating with the next component, which is the controller.

B. Controller

The controller is a daemon which performs two tasks. First, the controller listens for requests from agents and acts as a relay to the predictor daemon. Additionally, the controller receives feedback from the agent after every executed test run. This information is used to update the meta predictor, which will be described later.

C. Predictor

The final main component of the architecture is the predictor daemon. This component is responsible for interpreting the code changes and determining the optimal order of execution using ten built-in prediction algorithms. These algorithms are variations of the three discussed algorithms, as well as the Alpha algorithm.

D. Meta predictor

Since the predictor daemon contains multiple prediction algorithms, a small extra component is required to decide which sequence should be preferred as the final execution order. The meta predictor is a table which assigns a score to every prediction algorithm. The final execution order is the one that has been predicted by the prediction algorithm with the highest score. The controller evaluates the performance of every algorithm in the feedback phase, as mentioned before, and updates the scores accordingly.

E. Alpha algorithm

In addition to the Greedy, HGS and ROCKET algorithms, the framework features a custom prioritisation algorithm as well. This algorithm starts by inspecting the changed code lines to obtain the affected test cases ATS. Among ATS, the algorithm selects every test case that has failed at least once in its previous three executions and sorts those by increasing execution time. Next, the algorithm selects the remaining test cases from ATS and sorts those equivalently. After these two steps, the algorithm proceeds like the greedy algorithm until it has processed every test case.

V. Results

The framework has been installed on two existing test subjects. The first project is the Ghent University Dodona project[1], on which the performance of the prediction algorithms has been evaluated. Since the provided version of the agent only supports Java and the Dodona project uses Ruby-on-Rails, a second test subject is required to verify the architecture. For this purpose, the Stratego[2] project was used. Additionally, this thesis answers three research questions to obtain insights into a typical test suite.

A. Performance

Table 1 compares the performance of the four discussed algorithms on two aspects. The first aspect is the amount of executed test cases until the first failure. Secondly, the duration until the first failed test case is measured. These results indicate that the Alpha algorithm executes around 30 times fewer test cases and that the first failure is observed almost instantaneously. The Greedy and HGS algorithms reduce the amount of executed test cases by almost half and offer a significant speed-wise improvement as well. The performance of the ROCKET algorithm is remarkable. The algorithm executes much more test cases than the original median but does detect a failing test case almost four times faster.

B. Research questions

This thesis answers three research questions using data from the Travis CI service. This data has been provided by the TravisTorrent project [**?**] (3 702 595 jobs)

[1] `https://dodona.ugent.be/`

[2] Java Spring application created for the Software Engineering 2 course.

Algorithm	Median (tests)	Median (time)
Original	78	123 s
Alpha	3	1 s
Greedy	33	12 s
HGS	10	6 s
ROCKET	170	32

Table 1: Performance on the Dodona project.

and by Durieux et al. [**?**] (35 793 144 jobs).

RQ1: Probability of failure. The first research question analyses the probability that a test run will fail. In the provided datasets, a combined 15 % of all the runs have failed.

RQ2: Average test run duration. After inspecting the TravisTorrent dataset, the timing information was proven inaccurate. Therefore, only the dataset provided by Durieux et al. has been used to answer this question. This dataset has revealed that Travis CI is used primarily for small projects, with an average execution time of 385 s. The maximum duration is more than 26 h.

RQ3: Consecutive failure. The final research question investigates the probability that a test run will fail multiple times in a row. According to the TravisTorrent dataset, which contains the mandatory identifier of the previous test run of the same project, the probability of consecutive failure is more than 51.76 %.

VI. Conclusion and future work

This thesis has proven that software projects can benefit from an optimised test suite, in particular, using Test Case Prioritisation. The proposed framework has been implemented successfully on two projects, and its results are promising, but it still has some limitations and room for improvements.

Agent. The provided implementation of the Java agent does not support parallel test case execution. Besides the technical difficulties, a coordination mechanism is required to enable this behaviour, to schedule test cases across multiple threads effectively.

Predictor. The prediction algorithms do not currently interact with one another. However, there might be hidden potential in merging the output sequences of multiple algorithms, possibly by using weights.

Meta predictor. The elementary meta predictor could be modified to use a saturating counter or Machine Learning to cope with an evolving codebase. Additionally, Machine Learning algorithms could potentially be used as prediction algorithms.