FACULTY OF SCIENCES

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq
Student number: 01503338

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

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

Academic year 2019-2020

GHENT UNIVERSITY

Acknowledgements

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

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

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

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

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

Thank you.

Pieter – Ghent, 2020

Summary

Summary in English will come here.

Samenvatting

Nederlandse samenvatting komt hier.

Optimising Continuous Integration using Test Case Prioritisation

Pieter De Clercq

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

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

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

I. Introductie

Things will appear here. [1]

References

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

Optimaliseren van Continue Integratie door middel van Test Prioritering

Pieter De Clercq

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

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

Trefwoorden—**woorden, komen, hier**

I. Introductie

Dingen komen hier. [1]

Referenties

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

Lay summary

Lay summary will come here.

Contents

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 [28]. 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 [33].

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 [28]. 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 [35] in definition 1 and illustrated in Figure 1.1.

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

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

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

If we apply the concepts of the previous chapter to the above definition, we can interpret the set of requirements R as source code lines that must be covered. A requirement r_i can subsequently be satisfied by any test case $t_j \in T$ that belongs to the subset T_i. Observe that the problem of finding T' is closely related to the *hitting set problem* (definition 2) [35].

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

- $S = \{s_1, \ldots, s_n\}$ *a finite set of elements.*
- $C = \{c_1, \ldots, c_n\}$ *a collection of sets, with* $\forall c_i \in C : c_i \subseteq S$*.*
- K *a positive integer,* $K \leq |S|$*.*

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

In the context of Test Suite Minimisation, T' corresponds to the hitting set of T_is. In order to effectively minimise the amount of tests in the test suite, T' should be the minimal hitting set [35]. Since we can reduce this problem to the NP-complete *Vertex Cover*-problem, we know that this problem is NP-complete as well [9].

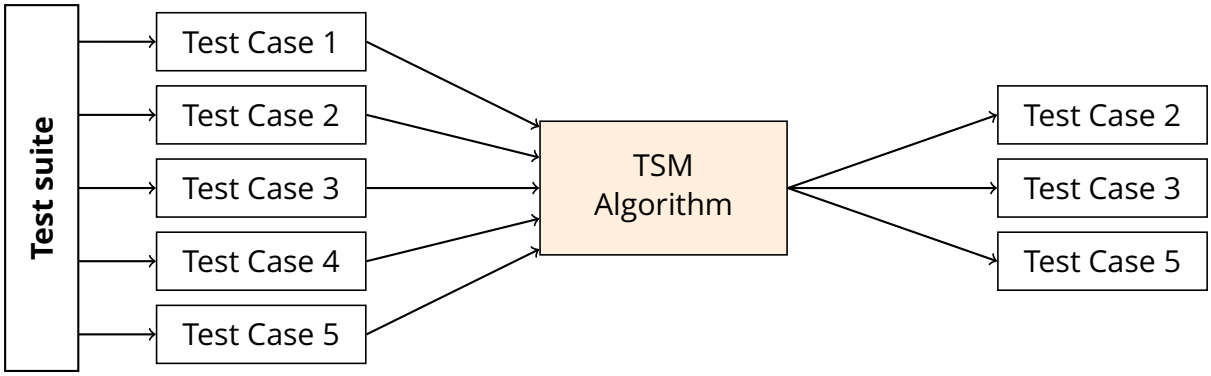

Figure 1.1: Test Suite Minimisation

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 [35]. 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'*.*

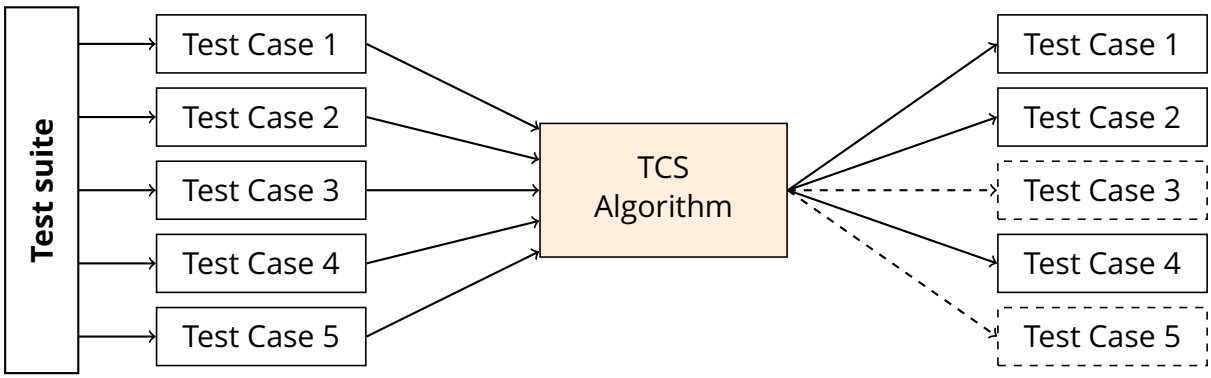

Figure 1.2: Test Case Selection

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

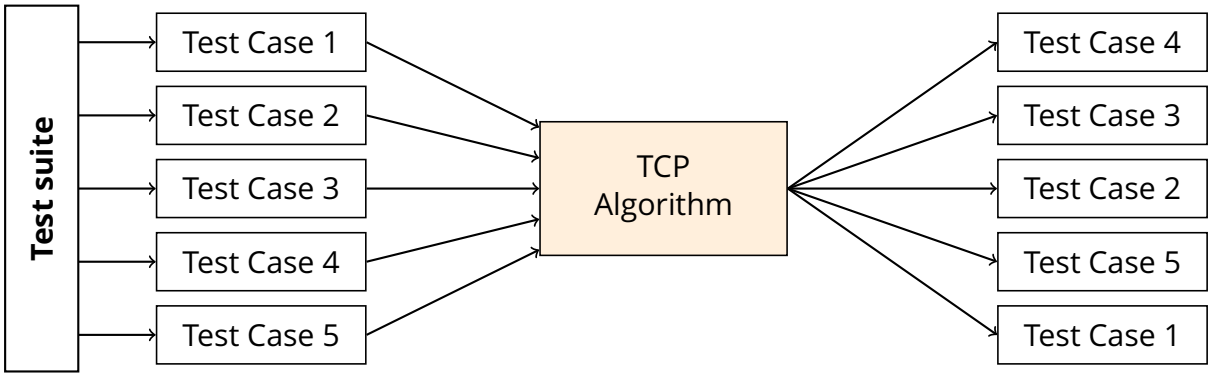

Figure 1.3: Test Case Prioritisation

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 [16]. 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 [28]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [4]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The objective of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few test cases as possible.

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

Algorithm 1 Greedy algorithm for Test Suite Minimisation

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

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

1.2.2 HGS

The second algorithm is the HGS algorithm. The algorithm was named after its creators Harrold, Gupta and Soffa [14]. 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 [14]. 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 ([14])

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** $t \in group \wedge \neg marked[group] \wedge Card(group) = size$ **then**
6: $count[t] \leftarrow count[t] + 1$
7: $max\_count \leftarrow \text{MAX}(count)$
8: $tests \leftarrow \{t | t \in list \wedge count[t] = max\_count\}$
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: $MAX\_CARD \leftarrow \text{MAX}(\{Card(group) | group \in CTD\})$
16: $RS \leftarrow \bigcup \{singleton | singleton \in CTD \wedge Card(singleton) = 1\}$
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: $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$
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: $MAX\_CARD \leftarrow \text{MAX}(\{Card(grp) | grp \in CTD \wedge \neg marked[grp]\})$
34: $RS \leftarrow RS \cup \{next\}$
35: $list \leftarrow \{t | t \in grp \wedge grp \in CTD \wedge Card(grp) = current \wedge \neg marked[grp]\}$
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 [23] 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 [23], the authors apply the following weights:

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

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

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

Algorithm 3 ROCKET algorithm

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

Output: priority P of the test cases

1: **procedure** ROCKETTCP(TS, E, m, F)
2: $n \leftarrow Card(TS)$
3: $P \leftarrow array[1 \dots n]$ ▷ initially 0
4: $MF \leftarrow array[1 \dots m]$
5: **for all** $i \in 1 \dots m$ **do**
6: $MF[i] \leftarrow array[1 \dots n]$
7: **for all** $j \in 1 \dots n$ **do**
8: **if** $F[j][i] = 1$ **then** $MF[i][j] \leftarrow -1$
9: **else** $MF[i][j] \leftarrow 1$
10: **for all** $j \in 1 \dots n$ **do**
11: **for all** $i \in 1 \dots m$ **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: $Q \leftarrow \{P[j] | j \in 1 \dots n\}$ ▷ distinct priorities
16: $G \leftarrow array[1 \dots Card(Q)]$ ▷ initially empty sets
17: **for all** $j \in 1 \dots n$ **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/#writing-tests-test-execution-order`

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 [TODO REVISE]

The implementation part of this thesis consists of 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 speed at which Continuous Integration is practised. This paper will now proceed by describing the design goals of the framework. Afterwards, a high-level schematic overview of the implemented architecture will be provided, followed by a more in-depth explanation of every pipeline step. In the final section of this chapter, the *Alpha* algorithm will be presented.

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 Continuous Integration systems, programming languages and test frameworks. Subsequently, a clear interface should be provided to integrate additional 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 framework being extensible. 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 supports all of the algorithms presented in section 1.2. It is possible that the performance of a given algorithm is strongly dependent on the nature of the project it is being applied to. In order to facilitate this behaviour, the framework should be able to measure the performance of every algorithm and "learn" which algorithm offers the best prediction, given a set of source code.

2.2 Architecture

The architecture of the VeloCIty framework consists of seven steps that are performed sequentially in a pipeline fashion, as illustrated in the sequence diagram (Figure 2.1). Every step is executed by one of three individual components, which will now be introduced briefly.

2.2.1 Agent

The first component that will be discussed is the agent. This is the only component that depends actively on both the programming language, as well as the used test framework, since it must interact directly with the source code and test suite. For every programming language or test framework that needs to be supported, a different implementation of an agent must be provided. This implementations are however strongly related, so much code can be reused or even shared. In this thesis, an agent was implemented in Java, more specifically as a plugin for the widely used Gradle and JUnit test framework. This combination was previously described in subsection 1.3.1. This plugin is responsible for running the test suite in a certain prioritised order, which is obtained by communicating with the controller (subsection 2.2.2). After the test cases have been executed, the plugin sends 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 the left side and the predictor (subsection 2.2.3) on the right side. In order to satisfy the second design goal and allow language agnosticism, the agent communicates with the controller using the `HTTP` protocol by exposing a *REST*-interface. Representational State Transfer `[REST]` is a software architecture used by modern web applications that allows standardised communication using existing HTTP methods. On the right side, the controller does not communicate directly with the predictor, but rather stores prediction requests in a shared database which is periodically polled by the predictor. Besides routing prediction requests from the agent to the predictor, the controller will also update the meta predictor by evaluating the accuracy of earlier predictions of this project.

2.2.3 Predictor and Metrics

The final component is twofold. Its main responsibility is to apply the prioritisation algorithms and predict an order in which the test cases should be executed. This order

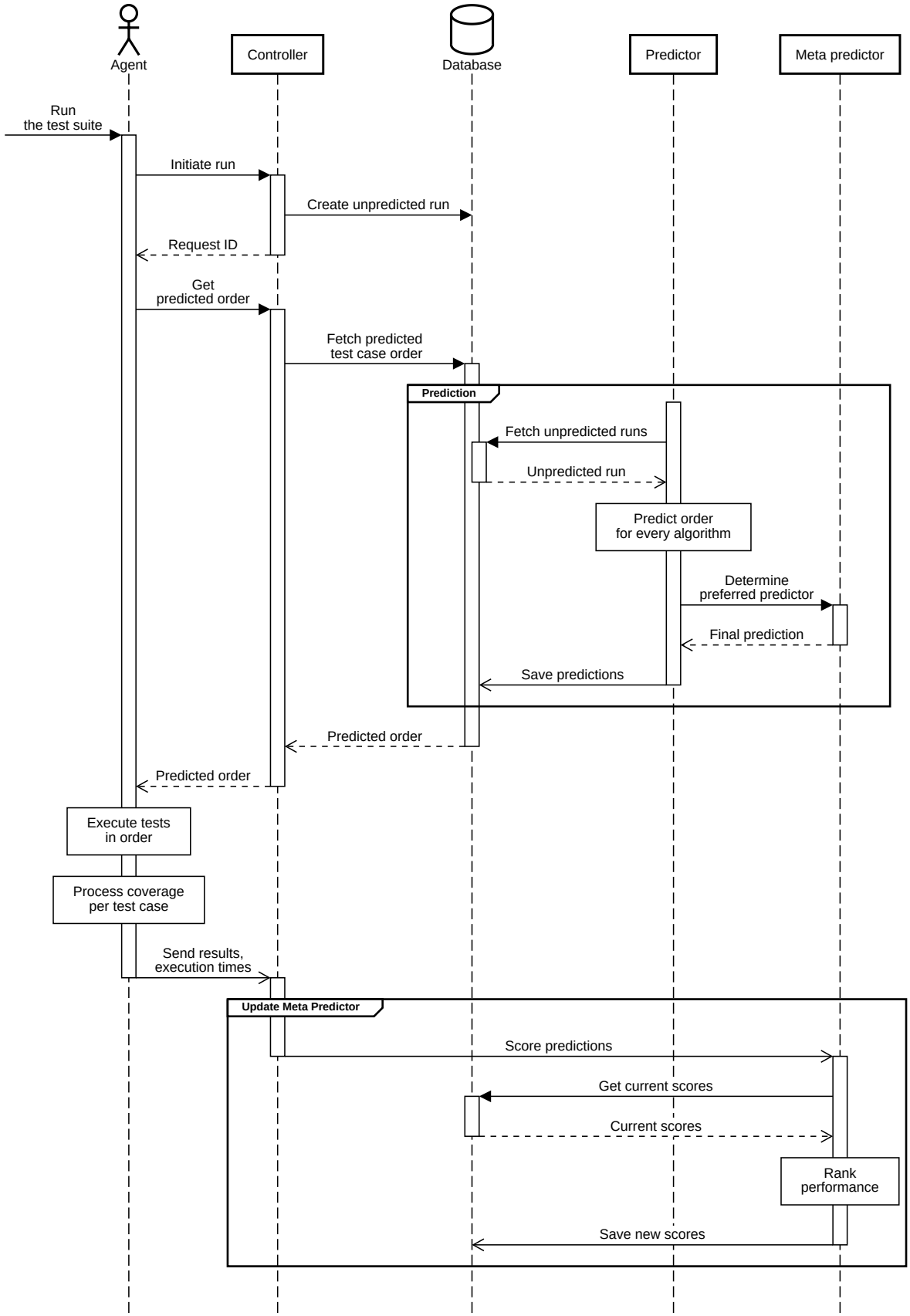

Figure 2.1: Sequence diagram of VeloCIty

is calculated by first executing ten algorithms and subsequently picking the algorithm that has been preferred by the meta predictor. Additionally, this component is able to provide metrics about the test suite, such as identifying superfluous test cases by applying Test Suite Minimisation. More specifically, this redundancy is obtained using the greedy algorithm (subsection 1.2.1). Both of these scripts have been implemented in Python, because of its simplicity and existing libraries for many common operations, such as numerical calculations (NumPy[1]) and machine learning (TensorFlow[2]).

2.3 Pipeline

This section will elaborate on the individual steps of the pipeline. The steps will be discussed by manually executing the pipeline that has hypothetically been implemented on a Java project. For the sake of simplicity, this explanation will assume a steady-state situation, ensuring the existence of at least one completed run of this project in the database at the controller side.

2.3.1 Initialisation

As was explained before, the provided Java implementation of the agent was designed to be used in conjunction with Gradle. In order to integrate VeloCIty into a Gradle project, the build script (`build.gradle`) should be modified in two places. The first change is to include and apply the plugin in the header of the file. Afterwards, the plugin requires three properties to be configured:

- `base` the path to the Java source files, relative to the location of the build script. This will typically resemble `src/main/java`.
- `repository` the url to the git repository at which the project is hosted. 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 at which the controller can be reached.

Listing 2.1 contains a minimal integration of the agent in a Gradle build script, applied to a library for generating random numbers[3]. The controller is hosted at the same host as the agent and is accessible at port 8080.

```
1 buildscript {
2     dependencies {
```

[1] https://numpy.org/
[2] https://www.tensorflow.org/
[3] https://github.com/thepieterdc/random-java

```
3         classpath 'io.github.thepieterdc.velocity:velocity-
              junit:0.0.1-SNAPSHOT'
4     }
5 }
6
7 plugins {
8     id 'java'
9 }
10
11 apply plugin: 'velocity-junit'
12
13 velocity {
14     base 'src/main/java/'
15     repository 'https://github.com/thepieterdc/random-java'
16     server 'http://localhost:8080'
17 }
```

Listing 2.1: Minimal Gradle buildscript

After the project has been configured, the test suite must be executed. For the Gradle agent, this involves executing the built-in `test` task. This task requires an additional argument to be passed, which is the commit hash of the changeset to prioritise. In every discussed Continuous Integration system, this commit hash is available as an environment variable.

The first step is for the agent to initiate a new test run in the controller. This is accomplished 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 result in a new prioritisation request being enqueued in the database that will asynchronously be processed by the predictor daemon in the next step.

2.3.2 Prediction

The prediction of the test execution order is performed by the predictor daemon. This daemon continuously polls the database to fetch new test runs that need to be predicted. When a new test run is detected, the predictor executes every available prediction algorithm in order to obtain multiple prioritised test sequences. The following algorithms are available:

AllInOrder The first algorithm will simply prioritise every test case alphabetically and will be used for for benchmarking purposes in **??**.

AllRandom The second algorithm has also been implemented for benchmarking purposes. This algorithm will "prioritise" every test case arbitrarily.

AffectedRandom This algorithm will only consider the test cases that cover source code lines which have been modified in the current commit. These test cases will be ordered randomly, followed by the other test cases in the test suite in no particular order.

GreedyCoverAll The first of three implementations of the Greedy algorithm (subsection 1.2.1) will execute the algorithm to prioritise 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 to be prioritised. After these test cases, the remaining test cases in the test suite will be ordered 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 (subsection 1.2.2). It is executed for every test case in the test suite.

HGSAffected Similar to the *GreedyAffected* algorithm, this algorithm is identical to the previous *HGSAll* algorithm besides 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 subsection 1.2.3.

Alpha The final algorithm has been inspired by the other implemented algorithms. section 2.4 will further elaborate on the details.

Subsequently, the final prioritisation order is determined by applying the meta predictor. Essentially, the meta predictor can be seen as a table which assigns a score to every algorithm, indicating its performance on this codebase. subsection 2.3.4 will explain later how this score is updated. The predicted order by the algorithm with the highest score is eventually elected by the meta predictor as the final prioritisation order, and saved to the database.

2.3.3 Test case execution

Regarding the agent, the identifier obtained in subsection 2.3.1 is used to poll the controller by sending a `GET` request to `/runs/id`, which will reply with the test execution order if this has already been determined. One of the discussed features of Gradle in subsection 1.3.1 was the possibility to execute test cases in a chosen order by adding annotations. However, this feature cannot be used to implement the Java agent, since it only supports ordering test cases within the same test class. In order to facilitate complete control over the order of execution, a custom `TestProcessor` and `TestListener` have been implemented.

The `TestProcessor` is responsible for processing every test class in the classpath and forward it along with configurable options to a delegate processor. The final processor in this chain will eventually perform the actual execution of the test class. Since the delegate processors that are built into Gradle will by default execute every method in the test class, the custom processor needs to work differently. The implemented agent will first store every received test class into a list and load the class to obtain all test cases in the class using reflection. After all classes have been processed, the processor will iterate over the prioritised order. For every test case t in the order, the delegate processor is called with a tuple of the corresponding test class and an options array which excludes every test case except t. This will effectively forward the same test class multiple times to the delegate processor, but each time with an option that restricts test execution to the prioritised test case, resulting in the desired behaviour.

Subsequently, the `TestListener` is a method that is called before and after every invocation of a test case. This listener allows the agent to calculate the duration of every 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 executing all test cases, the agent sends the test case 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`.

Upon receiving this data, the controller will update the meta predictor using the following procedure. The meta predictor is only updated if at least one of the test cases has failed, since the objective of Test Case Prioritisation is to detect failures as fast as

possible, thus every prioritised order is equally good if there are no failures at all. If however a test case did fail, the predicted orders are inspected to calculate the duration until the first failed test case for every order. Subsequently, the average of all these durations is calculated. Finally, the score of every algorithm that predicted a below average duration until the first failure is increased, otherwise it is decreased. This will eventually lead to the most accurate algorithms being preferred in subsequent test runs.

2.4 Alpha algorithm

Besides the earlier presented Greedy, HGS and ROCKET algorithms (section 1.2), VeloCIty features an additional algorithm. The *Alpha* algorithm has been constructed by examining the individual strengths and weaknesses 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:

- the set of all n test cases: $TS = \{T_1, \ldots, T_n\}$
- the set of m *affected* test cases: $AS = \{T_1, \ldots, T_m\} \subseteq TS$. A test case t is considered "affected" if any source code line which is covered by t has been modified or removed in the commit that is being predicted.
- C: the set of all lines in the application source code, for which a test case $t \in TS$ exists that covers this line and that has not yet been prioritised. Initially, this set contains every covered source code line.
- the failure status of every test case, for every past execution out of k executions of that test case: $F = \{F_1, \ldots, F_n\}$, where $F_i = \{f_1, \ldots, f_k\}$. $F_{tj} = 1$ implies that test case t has failed in execution $current - j$.
- the execution time of test case $t \in TS$ for run $r \in [1 \ldots k]$, in milliseconds: D_{tr}.
- for every test case $t \in TS$, the set TL_t is composed of all source code lines that are covered by test case t.

The first step of the algorithm is to determine the execution time E_t of every test case t. This execution time is calculated as the average of the durations of every successful (i.e.) execution of t, since a test case will be prematurely aborted upon the first failed

assertion, which introduces bias in the duration timings. In case t has never been executed successfully, the average is computed over every execution of t.

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

Next, the algorithm executes every affected test case that has also failed at least once in its three previous executions. This reflects the behaviour of a developer attempting to resolve the bug that caused the test case to fail. Specifically executing *affected* failing test cases first is required in case multiple test cases are failing and the developer is resolving these one by one, an idea which was extracted from the ROCKET algorithm (subsection 1.2.3). In case there are multiple affected failing test cases, the test cases are prioritised by increasing execution time. After every selected test case, C is updated by subtracting the code lines that have been covered by at least one of these test cases.

Afterwards, the same operation is repeated for every failed but unaffected test case, likewise ordered by increasing execution time. Where the previous step helps developers to get fast feedback about whether or not the specific failing test case they were working on has been resolved, this step ensures that other failing test cases are not forgotten and are executed early in the run as well. Similar to the previous step, C is again updated after every prioritised test case.

Research (**??**) has indicated that on average, only a small fraction ($10\,\% - 20\,\%$) of all test runs will contain failed tests, resulting in the previous two steps not being executed at all. Therefore, the most time should be dedicated to executing test cases that cover affected code lines. More specifically, the next step of the algorithm executes every affected test case, sorted by decreasing cardinality of the intersection between C and the lines which are covered by the test case. Conforming to the prior two steps, C is also updated to reflect the selected test case. As a consequence of these updates, the cardinalities of these intersections change after every update, which will ultimately lead to affected tests not strictly requiring to be executed. This idea has been adopted from the Greedy algorithm subsection 1.2.1.

In the penultimate step, the previous operation is repeated in an identical fashion for the remaining test cases, similarly ordered by the cardinality of the intersection with the remaining uncovered lines in C.

Finally, the algorithm selects every test case which had not yet been prioritised. No-

tice that these test cases do not contribute to the test coverage, as every test case that would incur additional coverage would have been prioritised already in the previous step. Subsequently, these test cases are actually redundant and are therefore candidates for removal by Test Suite Minimisation. 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 testcases $AS \subseteq TS$, all coverable lines C
Execution times D_{tr} of every test case t, over all k runs r of that test case,
Failure status FS for each test case over the previous m successive iterations,
Sets $TL = \{TL_1, \ldots, TL_n\}$ of all source code lines that are covered by test case $t \in TS$.
Output: ordered list P, sorted by descending priority

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

Bibliography

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

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

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

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

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

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

[7] 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.

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

[9] 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.

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

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

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

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

[14] 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.

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

[16] “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.

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

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

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

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

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

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

[23] 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.

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

[25] Bertrand Meyer. “Overview”. In: *Agile!: The Good, the Hype and the Ugly*. Cham: Springer International Publishing, 2014, pp. 1–15. ISBN: 978-3-319-05155-0. DOI: 10.1007/978-3-319-05155-0_1. URL: https://doi.org/10.1007/978-3-319-05155-0_1.

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

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

[28] 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.

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

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

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

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

[33] 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.

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

[35] 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.