Chapter 3

Related work

In the previous chapter, the paramount importance of frequently integrating one's changes into the upstream repository was emphasised. Additionally, Continuous Integration was introduced as both a practice and a tool to facilitate this often complex and time-consuming process. Continuous Integration is, however, not the golden bullet for software engineering, as there is a flip side to applying this practice. After every integration, all of the unit and regression tests in the test suite must be executed to ensure that the integration was successful and that no new bugs have been introduced. As the project evolves, the size of the codebase increases and consequently the amount of tests will increase as well in order to maintain a sufficiently high coverage level. An increase in the size of the test suite will inevitably lead to an increase in test duration [34], which imposes an issue of scaling. Walcott, Soffa and Kapfhammer illustrate the magnitude of this problem by providing an example of a codebase consisting of 20.000 lines, for which the tests require up to seven weeks to complete [39].

Fortunately, multiple developers and researchers have found some techniques that can be used to address the scalability issues of growing test suites. The techniques currently known to literature can be classified in three categories. Developers can either apply *Test Suite Minimisation*, *Test Case Selection* or *Test Case Prioritisation* [34]. All three techniques are applicable to any test suite, however there is a trade-off to be made. Depending on which technique is chosen, it will either have a major impact on the duration of the test suite execution in exchange for a reduced test coverage level, or it will result in a higher test adequacy.

In the following sections, the details of these three approaches will be discussed and accompanying algorithms will be provided. Since the approaches share common ideas, the algorithms can (albeit with minor modifications) be applied to all approaches. The final section will investigate the adoption and integration of these techniques and algorithms in existing prominent software testing frameworks.

3.1 Classification of approaches

3.1.1 Test Suite Minimisation

Test Suite Minimisation, also referred to as *Test Suite Reduction*, aims to reduce the size of the test suite by permanently removing redundant tests. This problem is formally defined by Rothermel in definition 1 [41] and illustrated in Figure 3.1.

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

- $T = \{t_1, \ldots, t_n\}$ *a test suite consisting of tests* t_j*.*
- $R = \{r_1, \ldots, r_n\}$ *a set of requirements that must be satisfied in order to provide the desired "adequate" testing of the program.*
- $\{T_1, \ldots, T_n\} \subseteq T$ *subsets of test cases, 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*.*

Test Suite Minimisation is then defined as the task of finding a set T' *of test cases* $t_j \in T$ *that satisfies all requirements* r_i*.*

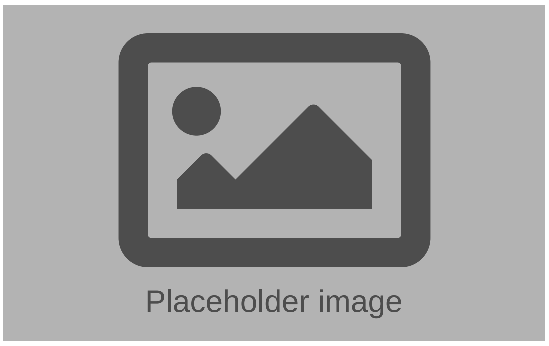

Figure 3.1: Test Suite Minimisation

If we apply this definition to the concepts introduced in chapter 2, the requirements R can be interpreted as lines in the codebase that must be covered. With respect to the definition, a requirement can be satisfied by any test t_j that belongs to subset T_i of T. Observe that the problem of finding T' is closely related to the *hitting set problem* (definition 2) [41].

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' is precisely 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 [41], which is an NP-complete problem as it can be reduced to the *Vertex Cover*-problem [11].

3.1.2 Test Case Selection

The second algorithm closely resembles the previous one. Instead of determining the minimal hitting set of the test suite in order to permanently remove tests, this algorithm has a notion of context. Prior to the execution of the tests, the algorithm performs a *white-box static analysis* of the codebase to identify which parts have been changed. Subsequently, only the tests regarding modified parts are executed, making the selection temporary (Figure 3.2) and modification-aware [41]. Rothermel and Harrold define this formally in definition 3.

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

- *P the previous version of the codebase*
- *P' the current (modified) version of the codebase*
- *T the test suite*

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

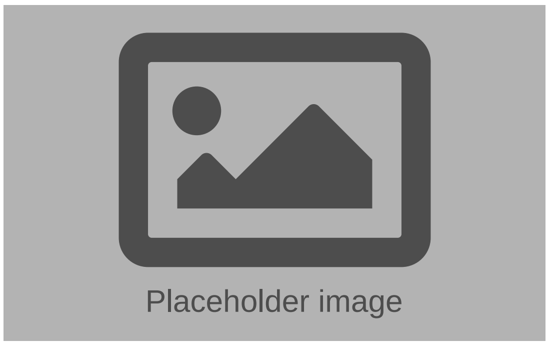

Figure 3.2: Test Case Selection

3.1.3 Test Case Prioritisation

Where the previous algorithms both attempted to execute as few tests as possible, it might sometimes be desired or even required that all tests pass. In this case, the previous ideas can be used as well. In Test Case Prioritisation, we want to find a permutation of the sequence of all tests instead of eliminating certain tests. The order of the permutation is chosen specifically to achieve a given goal as soon as possible, allowing for early termination of the test suite upon failure [41]. Some examples of goals include covering as many lines of code as fast as possible, or early execution of tests with a high probability of failure. A formal definition of this algorithm is provided in definition 4.

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 tests to find the optimal permutation.*

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

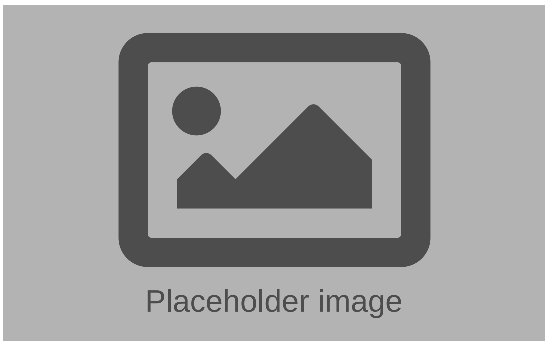

Figure 3.3: Test Case Prioritisation

3.2 Algorithms

In subsection 3.1.1 the relation was explained between applying Test Suite Minimisation and finding the minimal hitting set of the test suite and the set of requirements, which is an NP-complete problem. Therefore, the use of *heuristics* is required. A heuristic is an experience-based method that can be used to solve a hard to compute problem by finding a fast approximation [21]. However, the found solution will mostly be suboptimal or might sometimes even fail to find any solution at all. Considering its relation to the minimal hitting set problem, heuristics that are known to literature for solving this problem can also be used to implement Test Suite Minimisation. A selection of these heuristics will be discussed below. It should be noted however that the used terminology and naming of the variables might have been changed to ensure mutual consistency. Every algorithm has been modified to adhere to the conventions provided in definition 5 and definition 6.

Definition 5 (Naming convention)**.**

- C*: the set of all lines in the application source code that are covered by at least one test case* $t \in TS$*.*
  - CT_l *denotes the test group* l*, which corresponds to the set of all tests* $t \in TS$ *that cover source code line* $l \in C$*.*
- RS*: the representative set of test cases, these are the test cases that have been selected by the algorithm.*
- TS*: the set of all test cases in the test suite.*
  - TL_t *denotes the set of all source code lines that are covered by test* $t \in TS$*.*

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, the cardinality of* S *can also be denoted as* $Card(S)$*.*

3.2.1 Greedy algorithm

The first algorithm is a *greedy* heuristic, which was originally designed by Chvatal to find an approximation for the set-covering problem [34]. A greedy algorithm always makes a locally optimal choice, assuming that this will eventually lead to a globally optimal solution [5]. Algorithm 1 presents the Greedy algorithm for Test Suite Minimisation. The goal of the algorithm is to construct a set of test cases that cover every line in the code, by requiring as few tests as possible.

Initially, the algorithm starts with an empty result set RS, the set TS of all test cases and the set C of all coverable source code lines. Furthermore, TL_t denotes the set of source code lines in C that are covered by test case $t \in TS$. Subsequently, the algorithm iteratively selects test cases from TS and adds them to RS. The locally optimal choice is to always select the test case that will contribute the most still uncovered lines, ergo the test t for which the cardinality of the intersection between C and TL_t is maximal. After every iteration, the now covered lines TL_t are removed from C and the selection process is repeated until C is empty. Upon running the tests, only the tests in RS must be executed. This algorithm can be converted to make it applicable to Test Case Prioritisation by converting the set RS to a list to maintain the order in which the test cases were selected, which is equivalent to the prioritised order of execution.

Algorithm 1 Greedy algorithm for Test Suite Minimisation

```
1:  Input: Set $TS$ of all test cases,
        Set $C$ of all source code lines that are covered by any $t \in TS$,
        $TL_t$ the set of all lines are covered by test case $t \in TS$.
2:  Output: Subset $RS \subseteq TS$ of tests to execute.
3:  $RS \leftarrow \emptyset$
4:  while $C \neq \emptyset$ do
5:      $t\_max \leftarrow 0$
6:      $tl\_max \leftarrow \emptyset$
7:      for all $t \in TS$ do
8:          $tl\_current \leftarrow C \cap TL_t$
9:          if $|tl\_current| > |tl\_max|$ then
10:             $t\_max \leftarrow t$
11:             $tl\_max \leftarrow tl\_current$
12:     $RS \leftarrow RS \cup \{t\_max\}$
13:     $C \leftarrow C \setminus tl\_max$
```

3.2.2 HGS

The second algorithm was created by Harrold, Gupta and Soffa [17]. This algorithm constructs the minimal hitting set of the test suite in an iterative fashion. As opposed to the greedy algorithm (subsection 3.2.1), the HGS algorithm considers the test groups CT instead of the set TLt to obtain a list of test cases that cover all source code lines. More specifically, this algorithm considers the distinct test groups, denoted as CTD. Two test groups are considered indistinct if they differ in at least one test case. The pseudocode for this algorithm is provided in Algorithm 2.

Similar to the previous algorithm, an empty representative set RS is constructed in which the selected test cases will be stored. The algorithm begins by iterating over every source code line $l \in C$ and constructing the corresponding set of test groups CT_l. As mentioned before, for performance reasons this set is reduced to CTD, only retaining distinct test groups. Next, the algorithm selects every test group of which the cardinality is equal to 1 and adds these to RS. This corresponds to every test case that covers a line of code, which is exclusively covered by that single test case. Subsequently, the lines that are covered by any of the selected test cases are removed from C. This process is repeated for an incremented cardinality, until every line in C is covered. Since the remaining test groups will now contain more than one test case, the algorithm needs to make a choice on which test case to select. The authors have chosen that the test case that occurs in the most test groups is preferred. In the event of a tie, this choice is deferred until the next iteration.

The authors have provided an accompanying calculation of the computational time complexity of this algorithm [17]. With respect to the naming convention introduced in definition 5, additionally 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 largest test group. The HGS algorithm consists of two steps which are performed 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 all need to be examined once, the computational cost of this step is equal to $O(n * MAX\_CARD)$. In order to determine which test case should be included in the representative set RS, the algorithm needs to find all test cases for which the number of occurrences in all test groups is maximal, which requires at most $O(nt * MAX\_CARD)$. Since every repetition of these two steps adds a test case that belongs to at least one out of n test groups to the representative set, the overall runtime of the algorithm is $O(n * (n + nt) * MAX\_CARD)$.

Algorithm 2 HGS algorithm ([17])

```
1: Input: Distinct test groups $T_1, \dots T_n \in CDT$, containing test cases from $TS$.
2: Output: Subset $RS \subseteq TS$ of tests to execute.
3: $marked \leftarrow array[1 \dots n]$                                ▷ initially $false$
4: $MAX\_CARD \leftarrow max\{Card(T_i) | T_i \in CDT\}$
5: $RS \leftarrow \bigcup\{T_i | Card(T_i) = 1\}$
6: for all $T_i \in CDT$ do
7:     if $T_i \cap RS \neq \emptyset$ then $marked[i] \leftarrow true$
8: $current \leftarrow 1$
9: while $current < MAX\_CARD$ do
10:     $current \leftarrow current + 1$
11:     while $\exists T_i : Card(T_i) = current, marked[i] = false$ do
12:         $list \leftarrow \{t | t \in T_i : Card(T_i) = current, marked[i] = false\}$
13:         $next \leftarrow SelectTest(current, list)$
14:         $reduce \leftarrow false$
15:         for all $T_i \in CDT$ do
16:             if $next \in T_i$ then
17:                 $marked[i] = true$
18:                 if $Card(T_1) = MAX\_CARD$ then $reduce \leftarrow true$
19:         if $reduce$ then
20:             $MAX\_CARD \leftarrow max\{Card(T_i) | marked[i] = false\}$
21:         $RS \leftarrow RS \cup \{next\}$
22: function SelectTest($size$, $list$)
23:     $count \leftarrow array[1 \dots nt]$
24:     for all $t \in list$ do
25:         $count[t] \leftarrow |\{T_j | t \in T_j, marked[T_j] = false, Card(T_j) = size\}|$
26:     $tests \leftarrow \{t | t \in list, count[t] = max(count)\}$
27:     if $|tests| = 1$ then return $tests[0]$
28:     else if $|tests| = MAX\_CARD$ then return $tests[0]$
29:     else return $SelectTest(size + 1, tests)$
```

3.2.3 ROCKET algorithm

In contrast to the previously discussed algorithms which focused on Test Suite Minimisation, the ROCKET algorithm is tailored for Test Case Prioritisation. This algorithm was presented by Marijan, Gotlieb and Sen [29] as part of a case study to improve the testing efficiency of industrial video conferencing software. 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 consecutive failure rate, whilst also maximising the number of executed test cases in a limited time frame. The below algorithm has been modified slightly, since the time frame is a domain-specific constraint for this particular industry case and irrelevant for this thesis. Since this is a prioritisation algorithm rather than a minimisation algorithm, it yields a total ordering of all the test cases in the test suite, ordered using a weighted function.

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

- the set of test cases to prioritise $TS = \{T_1, \ldots, T_n\}$
- the execution time for each test case $E = \{E_1, \ldots, E_n\}$
- the failure status for each test case over the previous m successive executions $F = \{F_1, \ldots, F_n\}$, where $F_i = \{f_1, \ldots, f_m\}$

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, an $m \times n$ failure matrix MF is constructed and filled using the following formula.

$$MF[i,j] = \begin{cases} 1 & \text{if test case } T_j \text{ passed in execution } (current - i) \\ -1 & \text{if test case } T_j \text{ failed in execution } (current - i) \end{cases}$$

This matrix MF is visualised in Table 3.1. This table contains the hypothetical failure rates of the last three 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
$current - 3$	1	1	-1	1	1	-1

Table 3.1: Visualisation of the failure matrix MF.

Afterwards, P is filled with the cumulative priority of each test case, which is calculated by multiplying its failure rate with a domain-specific weight heuristic ω. This heuristic is used to derive the probability of repeated failures of the same test, given earlier failures. In their paper [29], the authors apply the following weights:

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

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

Finally, the algorithm groups test cases based on their calculated priority in P. Every test case that belongs to the same group is equally relevant for execution in the current test run. However, within every test group the tests 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

```
1: **Input:** Set $TS = \{T_1, \dots, T_n\}$ of all test cases,
       Execution time $E$ of every test case,
       Failure status $FS$ for each test case over the previous $m$ successive iterations.
2: **Output:** Priority of test cases $P$.
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** test $T_j$ failed in run $(current - i)$ **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.
```

3.3 Adoption in testing frameworks

Some of the approaches discussed above have been integrated in existing software testing frameworks. This paper will now proceed by conducting an analysis of these frameworks and tools to analyse which optimisation features are available and how they were implemented.

3.3.1 Gradle and JUnit

Gradle[1] is a dependency manager and application framework for Java, Groovy and Kotlin projects. Gradle 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 unit testing framework by Java developers. JUnit 5 is the newest version which is still actively being developed as of today. The framework is integrated as the testing framework of choice in several other Java libraries and frameworks, such as Android and Spring. JUnit offers mediocre support for features that optimise the execution of the test cases, especially when used in conjunction with Gradle. The following three key features are available:

1. **Parallel test execution:** JUnit comes bundled with multiple test processors that are responsible for processing test classes and to execute the test cases. One of these test processors is the `MaxNParallelTestClassProcessor`, which is capable of running a configurable amount of test cases in parallel. This results in a major speed-up of the overall test suite execution.

2. **Prioritise failed test cases:** Another test class processor which is provided by Gradle, is the `RunPreviousFailedFirstTestClassProcessor`. This processor will prioritise test cases that have failed in the previous run, similar to the idea of the ROCKET-algorithm (subsection 3.2.3), albeit without taking into account the duration of these test cases.

3. **Test order specification:** JUnit allows the user to specify the order in which test cases will be executed[3]. By default, a random yet deterministic order is used. The order can be manipulated by annotating the test class with the `@TestMethodOrder`-annotation, or by annotating individual test cases with the `@Order(int)`-annotation. This feature can only be used to alter the order of test cases within the same test class, it is not possible to perform inter-test class reordering. This feature could be used to 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 3.4: Logo of Gradle

Figure 3.5: Logo of JUnit 5

3.3.2 OpenClover

OpenClover[4] is a code coverage framework for Java and Groovy projects. The framework was created by Atlassian and open-sourced in 2017. It 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. Among these features are powerful integrations with development software and prominent Continuous Integration services. Furthermore, OpenClover offers the automatic analysis of 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, these predictions can be interpreted to implement Test Suite Minimisation. This results in a reduced test suite execution time.

Figure 3.6: Logo of Atlassian Clover

[4] `https://openclover.org`